aws

Developer Guide

AWS Lambda



Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Lambda: Developer Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is AWS Lambda? 1
How Lambda works 2
Key features 2
Related information 3
How it works 3
Lambda functions and function handlers 4
Lambda execution environment and runtimes 5
Events and triggers 5
Lambda permissions and roles 6
Running code 9
Creating event-driven architectures 25
Designing an application 38
Create your first function 46
Prerequisites 46
Create the function 48
Invoke the function 54
Clean up 57
Next steps 58
Example apps and patterns 60
File Processing 60
Database Integration 60
Scheduled Tasks 60
Long-running Workflows 61
Additional resources 61
File-processing app 61
Create the source code files 63
Deploy the app 66
Test the app 77
Next steps 84
Scheduled-maintenance app 85
Prerequisites 86
Downloading the example app files 87
Creating and populating the example DynamoDB table 97
Creating the scheduled-maintenance app 100

Testing the app ... 104
Next steps ... 105
Creating an Order Processing System with Lambda Durable Functions ... 105
Prerequisites ... 106
Create the Source Code Files ... 106
Deploy the App ... 107
Test the App ... 108
Next Steps ... 108
Development tools ... 110
Local development tools ... 110
Infrastructure as Code (IaC) tools ... 111
GitHub Actions tools ... 112
Powertools for AWS Lambda ... 112
Workflow and event management tools ... 112
Local development ... 113
Key benefits of local development ... 113
Prerequisites ... 113
Authentication and access control ... 114
Moving from console to local development ... 117
Working with functions locally ... 117
Convert your function to an AWS SAM template and use IaC tools ... 120
Next steps ... 120
GitHub Actions ... 121
Example workflow ... 121
Additional resources ... 122
Infrastructure as code (IaC) ... 122
IaC tools for Lambda ... 122
Using AWS SAM and Infrastructure Composer ... 124
Using AWS CDK ... 135
Powertools for AWS Lambda ... 146
Key benefits of Powertools for AWS ... 146
Integrating Powertools with AWS ... 146
Next steps ... 120
Workflows and events ... 147
Code-first orchestration with durable functions ... 147
Orchestrating workflows with Step Functions ... 148

Managing events with EventBridge and EventBridge Scheduler ... 149
Lambda durable functions ... 151
Key benefits ... 151
When to use durable functions ... 151
How durable functions compare to Step Functions ... 152
How it works ... 152
Next steps ... 154
Basic concepts ... 155
Durable execution ... 155
DurableContext ... 156
Steps ... 157
Wait States ... 157
Invoking other functions ... 158
Durable function configuration ... 159
See also ... 161
Creating durable functions ... 161
Prerequisites ... 162
Create the durable function ... 164
Invoke the durable function ... 170
Clean up ... 173
Next steps ... 173
Deploy with CLI ... 174
Using Infrastructure as Code ... 181
Configure durable functions ... 191
Enable durable execution ... 191
Configuration best practices ... 192
Durable functions or Step Functions ... 193
When to use durable functions ... 193
When to use Step Functions ... 193
Decision framework ... 194
Feature comparison ... 194
Hybrid architectures ... 195
Migration considerations ... 195
Related resources ... 195
Examples ... 196
Short-lived fault-tolerant processes ... 196

Long-running processes ... 201
Advanced patterns ... 208
Next steps ... 221
Security and permissions ... 221
Execution role permissions ... 221
State encryption ... 223
CloudTrail logging ... 223
Cross-account considerations ... 224
Inherited Lambda security features ... 224
Durable execution SDK ... 225
DurableContext ... 225
What the SDK does ... 226
How checkpointing works ... 227
Replay behavior ... 227
Available durable operations ... 228
How durable operations are metered ... 238
Supported runtimes ... 241
Invoking durable functions ... 246
Invocation methods ... 246
Qualified ARNs requirement ... 247
Understanding execution lifecycle ... 248
Invoking from application code ... 249
Chained invocations ... 250
Event source mappings ... 252
Retries ... 259
Idempotency ... 266
Testing durable functions ... 271
Local testing ... 272
Cloud testing ... 274
What to test ... 276
Testing strategy ... 276
Debugging failures ... 277
Monitoring durable functions ... 277
CloudWatch metrics ... 278
EventBridge events ... 279
AWS X-Ray tracing ... 282

Best practices ... 282
Write deterministic code ... 283
Design for idempotency ... 286
Manage state efficiently ... 289
Design effective steps ... 290
Use wait operations efficiently ... 291
Additional considerations ... 292
Additional resources ... 292
Lambda Managed Instances ... 294
Key capabilities ... 294
When to use Lambda Managed Instances ... 294
How it works ... 295
Concurrency model ... 295
Tenancy and isolation ... 295
Understanding managed instances ... 296
Pricing ... 296
How Lambda Managed Instances differs from the Lambda (default) compute type ... 296
Next steps ... 297
Getting started ... 298
Creating a Lambda Managed Instance function (console) ... 298
Creating a Lambda Managed Instance function (AWS CLI) ... 299
Core concepts ... 305
Capacity providers ... 306
Scaling ... 310
Security ... 313
Operator role ... 315
Execution environment ... 318
Version publishing ... 320
Runtimes ... 321
Supported languages ... 321
Language-specific considerations ... 321
Next steps ... 322
Java runtime ... 322
Node.js runtime ... 327
Python runtime ... 332
.NET runtime ... 334

Rust ... 339
Networking ... 342
Connectivity options ... 342
Public subnet with an internet gateway ... 342
VPC endpoints ... 343
Private subnet with NAT gateway ... 344
Choosing a connectivity option ... 344
Next steps ... 345
Monitoring ... 345
Available metrics ... 346
Metric frequency and retention ... 347
Viewing metrics in CloudWatch ... 347
Using metrics to optimize performance ... 348
Next steps ... 348
Quotas ... 348
Lambda API request quotas ... 348
Lambda Managed Instances resource quotas ... 349
Event source mapping quotas ... 349
Requesting a quota increase ... 350
Next steps ... 350
Best practices ... 351
Capacity provider configuration ... 351
Instance type selection ... 351
Function configuration ... 351
Scaling configuration ... 352
Security ... 352
Cost optimization ... 353
Monitoring and observability ... 353
Language-specific considerations ... 354
Next steps ... 354
Troubleshooting ... 354
Throttling and scaling issues ... 354
Concurrency issues ... 355
Performance issues ... 357
Capacity provider issues ... 358
Monitoring and observability issues ... 359

Getting additional help ... 361
Next steps ... 361
Lambda runtimes ... 363
Supported runtimes ... 363
New runtime releases ... 367
Runtime deprecation policy ... 368
Shared responsibility model ... 368
Runtime use after deprecation ... 370
Receiving runtime deprecation notifications ... 372
Deprecated runtimes ... 373
Runtime version updates ... 376
Backward compatibility ... 377
Runtime update modes ... 378
Two-phase runtime version rollout ... 379
Configuring runtime management ... 380
Runtime version roll-back ... 381
Runtime version updates ... 383
Shared responsibility model ... 385
Permissions ... 387
Get data about functions by runtime ... 388
Listing function versions that use a particular runtime ... 388
Identifying most commonly and most recently invoked functions ... 390
Runtime modifications ... 395
Language-specific environment variables ... 395
Wrapper scripts ... 395
Runtime API ... 399
Next invocation ... 400
Invocation response ... 401
Initialization error ... 401
Invocation error ... 403
OS-only runtimes ... 406
Building a custom runtime ... 407
Custom runtime tutorial ... 412
Open source repositories ... 420
Runtime Interface Clients ... 420
Event libraries ... 421

Container base images ... 422
Development tools ... 422
Sample projects ... 423
Configuring functions ... 424
.zip file archives ... 426
Creating the function ... 426
Using the console code editor ... 428
Updating function code ... 428
Changing the runtime ... 429
Changing the architecture ... 429
Using the Lambda API ... 430
Downloading your function code ... 430
CloudFormation ... 431
Encryption ... 431
Container images ... 438
Requirements ... 439
Using an AWS base image ... 440
Using an AWS OS-only base image ... 441
Using a non-AWS base image ... 442
Runtime interface clients ... 442
Amazon ECR permissions ... 443
Function lifecycle ... 446
Memory ... 447
When to increase memory ... 447
Using the console ... 448
Using the AWS CLI ... 448
Using AWS SAM ... 449
Accepting function memory recommendations (console) ... 449
Ephemeral storage ... 450
Use cases ... 450
Using the console ... 451
Using the AWS CLI ... 451
Using AWS SAM ... 451
Instruction sets (ARM/x86) ... 453
Advantages of using arm64 architecture ... 453
Requirements for migration to arm64 architecture ... 454

Function code compatibility with arm64 architecture 454
How to migrate to arm64 architecture 454
Configuring the instruction set architecture 455
Timeout 457
When to increase timeout 457
Using the console 457
Using the AWS CLI 458
Using AWS SAM 458
Environment variables 460
Create environment variables 460
Example scenario for environment variables 464
Retrieve environment variables 466
Defined runtime environment variables 467
Securing environment variables 469
Attaching functions to a VPC 473
Required IAM permissions 473
Attaching Lambda functions to an Amazon VPC in your AWS account 475
Internet access when attached to a VPC 478
IPv6 support 479
Best practices for using Lambda with Amazon VPCs 480
Understanding Hyperplane Elastic Network Interfaces (ENIs) 481
Using IAM condition keys for VPC settings 482
VPC tutorials 487
Attaching functions to resources in another account 488
Prerequisites 488
Create an Amazon VPC in your function's account 489
Grant VPC permissions to your function's execution role 489
Create a VPC peering connection request 490
Prepare your resource's account 491
Update VPC configuration in your function's account 492
Test your function 493
Internet access for VPC functions 495
Inbound networking 520
Considerations for Lambda interface endpoints 520
Creating an interface endpoint for Lambda 521
Creating an interface endpoint policy for Lambda 522

File system ... 524
Supported Regions ... 524
Execution role and user permissions ... 524
Configuring a file system and access point ... 525
Connecting to a file system (console) ... 526
Aliases ... 528
Using aliases ... 530
Weighted aliases ... 531
Versions ... 536
Creating function versions ... 537
Using versions ... 538
Granting permissions ... 539
Tags ... 540
Permissions required for working with tags ... 540
Using tags with the console ... 540
Using tags with the AWS CLI ... 542
Response streaming ... 544
Bandwidth limits for response streaming ... 545
VPC compatibility with response streaming ... 545
Writing functions ... 546
Invoking functions ... 548
Tutorial: Creating a response streaming function with a function URL ... 549
Metadata endpoint ... 555
Getting started ... 555
Understanding Availability Zone IDs ... 558
API reference ... 558
Invoking functions ... 561
Invoke a function synchronously ... 563
Asynchronous invocation ... 567
Error handling ... 568
Configuration ... 569
Retaining records ... 571
Invocation ... 581
Synchronous invocation limits ... 581
Asynchronous invocation for long-running workflows ... 581
Execution management APIs ... 582

Event source mappings ... 584
Event source mappings and triggers ... 584
Batching behavior ... 585
Provisioned mode ... 588
Event source mapping API ... 590
Event source mapping tags ... 590
Event filtering ... 595
Understanding event filtering basics ... 596
Handling records that don't meet filter criteria ... 598
Filter rule syntax ... 599
Attaching filter criteria to an event source mapping (console) ... 600
Attaching filter criteria to an event source mapping (AWS CLI) ... 601
Attaching filter criteria to an event source mapping (AWS SAM) ... 603
Encryption of filter criteria ... 603
Using filters with different AWS services ... 609
Testing in console ... 611
Invoking functions with test events ... 611
Creating private test events ... 612
Creating shareable test events ... 612
Deleting shareable test event schemas ... 614
Function states ... 615
Function states during updates ... 616
Retries ... 618
Recursive loop detection ... 620
Understanding recursive loop detection ... 620
Supported AWS services and SDKs ... 622
Recursive loop notifications ... 624
Responding to recursive loop detection notifications ... 625
Allowing a Lambda function to run in a recursive loop ... 626
Supported Regions for Lambda recursive loop detection ... 628
Function URLs ... 629
Creating a function URL (console) ... 630
Creating a function URL (AWS CLI) ... 632
Adding a function URL to a CloudFormation template ... 633
Cross-origin resource sharing (CORS) ... 634
Throttling function URLs ... 636

Deactivating function URLs ... 636
Deleting function URLs ... 636
Access control ... 637
Invoking function URLs ... 648
Monitoring function URLs ... 660
Function URLs vs Amazon API Gateway ... 661
Tutorial: Creating a webhook endpoint ... 667
Function scaling ... 681
Understanding and visualizing concurrency ... 681
Calculating concurrency for a function ... 686
Understanding reserved concurrency and provisioned concurrency ... 687
Reserved concurrency ... 688
Provisioned concurrency ... 690
How Lambda allocates provisioned concurrency ... 694
Comparing reserved concurrency and provisioned concurrency ... 695
Understanding concurrency and requests per second ... 696
Concurrency quotas ... 698
Configuring reserved concurrency ... 700
Configuring reserved concurrency ... 701
Accurately estimating required reserved concurrency for a function ... 703
Configuring provisioned concurrency ... 704
Configuring provisioned concurrency ... 705
Accurately estimating required provisioned concurrency for a function ... 707
Optimizing function code when using provisioned concurrency ... 708
Using environment variables to view and control provisioned concurrency behavior ... 709
Understanding logging and billing behavior with provisioned concurrency ... 709
Using Application Auto Scaling to automate provisioned concurrency management ... 710
Scaling behavior ... 715
Concurrency scaling rate ... 715
Monitoring concurrency ... 717
General concurrency metrics ... 717
Provisioned concurrency metrics ... 717
Working with the `ClaimedAccountConcurrency` metric ... 720
Building with Node.js ... 723
Runtime-included SDK versions ... 725
Using keep-alive ... 725

CA certificate loading ... 725
Experimental Node.js features ... 726
Handler ... 728
Set up your project ... 728
Example function ... 729
CommonJS and ES Modules ... 731
Node.js initialization ... 732
Handler naming conventions ... 733
Input event object ... 733
Valid handler patterns ... 734
Using the SDK for JavaScript ... 736
Accessing environment variables ... 737
Using global state ... 737
Best practices ... 737
Deploy .zip file archives ... 740
Runtime dependencies in Node.js ... 740
Creating a .zip deployment package with no dependencies ... 741
Creating a .zip deployment package with dependencies ... 741
Creating a Node.js layer for your dependencies ... 742
Dependency search path and runtime-included libraries ... 743
Creating and updating Node.js Lambda functions using .zip files ... 744
Deploy container images ... 751
AWS base images for Node.js ... 752
Using an AWS base image ... 753
Using a non-AWS base image ... 759
Layers ... 769
Package your layer content ... 769
Create the layer in Lambda ... 774
Add the layer to your function ... 775
Sample app ... 776
Context ... 777
Logging ... 779
Creating a function that returns logs ... 779
Using Lambda advanced logging controls with Node.js ... 781
Viewing logs in the Lambda console ... 787
Viewing logs in the CloudWatch console ... 787

Viewing logs using the AWS Command Line Interface (AWS CLI) ... 788
Deleting logs ... 791
Tracing ... 792
Using ADOT to instrument your Node.js functions ... 793
Using the X-Ray SDK to instrument your Node.js functions ... 793
Activating tracing with the Lambda console ... 794
Activating tracing with the Lambda API ... 795
Activating tracing with CloudFormation ... 795
Interpreting an X-Ray trace ... 796
Storing runtime dependencies in a layer (X-Ray SDK) ... 799
Building with TypeScript ... 800
Development environment ... 801
Type definitions for Lambda ... 802
Handler ... 804
Set up your project ... 804
Example function ... 805
CommonJS and ES Modules ... 807
Node.js initialization ... 808
Handler naming conventions ... 808
Input event object ... 809
Valid handler patterns ... 809
Using the SDK for JavaScript ... 812
Accessing environment variables ... 813
Using global state ... 813
Best practices ... 814
Deploy .zip file archives ... 816
Using AWS SAM ... 816
Using the AWS CDK ... 818
Using the AWS CLI and esbuild ... 821
Deploy container images ... 824
Using a Node.js base image to build and package TypeScript function code ... 824
Context ... 832
Logging ... 835
Tools and libraries ... 835
Using Powertools for AWS Lambda (TypeScript) and AWS SAM for structured logging ... 836

Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for structured logging ... 838
Viewing logs in the Lambda console ... 842
Viewing logs in the CloudWatch console ... 842
Tracing ... 844
Using Powertools for AWS Lambda (TypeScript) and AWS SAM for tracing ... 845
Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for tracing ... 847
Interpreting an X-Ray trace ... 851
Building with Python ... 852
Runtime-included SDK versions ... 853
Disabled Python features ... 854
Response format ... 854
Graceful shutdown for extensions ... 855
Handler ... 856
Example Python Lambda function code ... 856
Handler naming conventions ... 858
Using the Lambda event object ... 859
Accessing and using the Lambda context object ... 860
Valid handler signatures for Python handlers ... 860
Returning a value ... 861
Using the AWS SDK for Python (Boto3) in your handler ... 862
Accessing environment variables ... 863
Code best practices for Python Lambda functions ... 863
Deploy .zip file archives ... 866
Runtime dependencies in Python ... 866
Creating a .zip deployment package with no dependencies ... 867
Creating a .zip deployment package with dependencies ... 867
Dependency search path and runtime-included libraries ... 870
Using __pycache__ folders ... 872
Creating .zip deployment packages with native libraries ... 872
Creating and updating Python Lambda functions using .zip files ... 873
Deploy container images ... 881
AWS base images for Python ... 882
Using an AWS base image ... 883
Using a non-AWS base image ... 890
Layers ... 900

Package your layer content ... 900
Create the layer in Lambda ... 774
Add the layer to your function ... 906
Sample app ... 907
Context ... 908
Logging ... 910
Printing to the log ... 910
Using a logging library ... 911
Using Lambda advanced logging controls with Python ... 913
Viewing logs in Lambda console ... 917
Viewing logs in CloudWatch console ... 918
Viewing logs with AWS CLI ... 918
Deleting logs ... 921
Tools and libraries ... 921
Using Powertools for AWS Lambda (Python) and AWS SAM for structured logging ... 922
Using Powertools for AWS Lambda (Python) and AWS CDK for structured logging ... 926
Testing ... 933
Testing your serverless applications ... 934
Tracing ... 936
Using Powertools for AWS Lambda (Python) and AWS SAM for tracing ... 937
Using Powertools for AWS Lambda (Python) and the AWS CDK for tracing ... 939
Using ADOT to instrument your Python functions ... 944
Using the X-Ray SDK to instrument your Python functions ... 945
Activating tracing with the Lambda console ... 946
Activating tracing with the Lambda API ... 946
Activating tracing with CloudFormation ... 946
Interpreting an X-Ray trace ... 947
Storing runtime dependencies in a layer (X-Ray SDK) ... 950
Building with Ruby ... 952
Runtime-included SDK versions ... 953
Enabling Yet Another Ruby JIT (YJIT) ... 954
Handler ... 955
Ruby handler basics ... 955
Code best practices for Ruby Lambda functions ... 956
Deploy .zip file archives ... 959
Dependencies in Ruby ... 959

Creating a .zip deployment package with no dependencies ... 960
Creating a .zip deployment package with dependencies ... 960
Creating a Ruby layer for your dependencies ... 962
Creating .zip deployment packages with native libraries ... 962
Creating and updating Ruby Lambda functions using .zip files ... 964
Deploy container images ... 971
AWS base images for Ruby ... 972
Using an AWS base image ... 972
Using a non-AWS base image ... 979
Layers ... 989
Package your layer content ... 989
Create the layer in Lambda ... 774
Using gems from layers in a function ... 996
Add the layer to your function ... 997
Sample app ... 998
Context ... 999
Logging ... 1000
Creating a function that returns logs ... 1000
Viewing logs in the Lambda console ... 1001
Viewing logs in the CloudWatch console ... 1002
Viewing logs using the AWS Command Line Interface (AWS CLI) ... 1002
Deleting logs ... 1005
Working with the Ruby logger library ... 1005
Tracing ... 1007
Enabling active tracing with the Lambda API ... 1012
Enabling active tracing with CloudFormation ... 1013
Storing runtime dependencies in a layer ... 1013
Building with Java ... 1015
Handler ... 1019
Setting up your Java handler project ... 1019
Example Java Lambda function code ... 1020
Valid class definitions for Java handlers ... 1025
Handler naming conventions ... 1026
Defining and accessing the input event object ... 1027
Accessing and using the Lambda context object ... 1028
Using the AWS SDK for Java v2 in your handler ... 1029

Accessing environment variables 1030
Using global state 1031
Code best practices for Java Lambda functions 1031
Deploy .zip file archives 1034
Prerequisites 1034
Tools and libraries 1035
Building a deployment package with Gradle 1036
Using layers for dependencies 1037
Building a deployment package with Maven 1037
Uploading a deployment package with the Lambda console 1039
Uploading a deployment package with the AWS CLI 1041
Uploading a deployment package with AWS SAM 1042
Deploy container images 1045
AWS base images for Java 1046
Using an AWS base image 1047
Using a non-AWS base image 1056
Layers 1068
Package your layer content 1068
Create the layer in Lambda 774
Add the layer to your function 1071
Custom serialization 1073
When to use custom serialization 1073
Implementing custom serialization 1074
Testing custom serialization 1075
Custom startup behavior 1076
Understanding the JAVA_TOOL_OPTIONS environment variable 1076
Log4j patch for Log4Shell 1078
Ahead-of-Time (AOT) and CDS caches 1079
Context 1080
Context in sample applications 1082
Logging 1084
Creating a function that returns logs 1084
Using Lambda advanced logging controls with Java 1086
Implementing advanced logging with Log4j2 and SLF4J 1089
Tools and libraries 1092
Using Powertools for AWS Lambda (Java) and AWS SAM for structured logging 1093

Viewing logs in the Lambda console ... 1097
Viewing logs in the CloudWatch console ... 1097
Viewing logs using the AWS Command Line Interface (AWS CLI) ... 1098
Deleting logs ... 1101
Sample logging code ... 1101
Tracing ... 1103
Using Powertools for AWS Lambda (Java) and AWS SAM for tracing ... 1104
Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing ... 1106
Using ADOT to instrument your Java functions ... 1118
Using the X-Ray SDK to instrument your Java functions ... 1118
Activating tracing with the Lambda console ... 1119
Activating tracing with the Lambda API ... 1119
Activating tracing with CloudFormation ... 1120
Interpreting an X-Ray trace ... 1120
Storing runtime dependencies in a layer (X-Ray SDK) ... 1123
X-Ray tracing in sample applications (X-Ray SDK) ... 1124
Sample apps ... 1126
Building with Go ... 1128
Go runtime support ... 1128
Tools and libraries ... 1129
Handler ... 1130
Setting up your Go handler project ... 1130
Example Go Lambda function code ... 1131
Handler naming conventions ... 1134
Defining and accessing the input event object ... 1134
Accessing and using the Lambda context object ... 1135
Valid handler signatures for Go handlers ... 1136
Using the AWS SDK for Go v2 in your handler ... 1137
Accessing environment variables ... 1138
Using global state ... 1138
Code best practices for Go Lambda functions ... 1139
Context ... 1141
Supported variables, methods, and properties in the context object ... 1141
Accessing invoke context information ... 1142
Using the context in AWS SDK client initializations and calls ... 1144
Deploy .zip file archives ... 1145

Creating a .zip file on macOS and Linux ... 1145
Creating a .zip file on Windows ... 1147
Creating and updating Go Lambda functions using .zip files ... 1150
Deploy container images ... 1157
AWS base images for deploying Go functions ... 1157
Go runtime interface client ... 1158
Using an AWS OS-only base image ... 1158
Using a non-AWS base image ... 1165
Layers ... 1174
Logging ... 1175
Creating a function that returns logs ... 1175
Viewing logs in the Lambda console ... 1177
Viewing logs in the CloudWatch console ... 1177
Viewing logs using the AWS Command Line Interface (AWS CLI) ... 1177
Deleting logs ... 1181
Tracing ... 1182
Using ADOT to instrument your Go functions ... 1183
Using the X-Ray SDK to instrument your Go functions ... 1183
Activating tracing with the Lambda console ... 1183
Activating tracing with the Lambda API ... 1184
Activating tracing with CloudFormation ... 1184
Interpreting an X-Ray trace ... 1185
Building with C# ... 1188
Development environment ... 1188
Installing the .NET project templates ... 1188
Installing and updating the CLI tools ... 1189
Handler ... 1190
Setting up your C# handler project ... 1190
Example C# Lambda function code ... 1191
Class library handlers ... 1195
Executable assembly handlers ... 1196
Valid handler signatures for C# functions ... 1197
Handler naming conventions ... 1197
Serialization in C# Lambda functions ... 1198
File-based functions ... 1201
Accessing and using the Lambda context object ... 1201

Using the SDK for .NET v3 in your handler ... 1202
Accessing environment variables ... 1203
Using global state ... 1203
Simplify function code with the Lambda Annotations framework ... 1204
Code best practices for C# Lambda functions ... 1205
Deployment package ... 1207
NET Lambda Global CLI ... 1208
AWS SAM ... 1214
AWS CDK ... 1217
ASP.NET ... 1221
Layers ... 1226
Deploy container images ... 1227
AWS base images for .NET ... 1228
Using an AWS base image ... 1228
Using a non-AWS base image ... 1230
Native AOT compilation ... 1235
Lambda runtime ... 1235
Prerequisites ... 1236
Getting started ... 1236
Serialization ... 1239
Trimming ... 1240
Troubleshooting ... 1241
Context ... 1242
Logging ... 1244
Creating a function that returns logs ... 1244
Using Lambda advanced logging controls with .NET ... 1245
Tools and libraries ... 1252
Using Powertools for AWS Lambda (.NET) and AWS SAM for structured logging ... 1253
Viewing logs in the Lambda console ... 1256
Viewing logs in the CloudWatch console ... 1256
Viewing logs using the AWS Command Line Interface (AWS CLI) ... 1256
Deleting logs ... 1260
Tracing ... 1261
Using Powertools for AWS Lambda (.NET) and AWS SAM for tracing ... 1262
Using the X-Ray SDK to instrument your .NET functions ... 1265
Activating tracing with the Lambda console ... 1266

Activating tracing with the Lambda API ... 1267
Activating tracing with CloudFormation ... 1267
Interpreting an X-Ray trace ... 1268
Testing ... 1271
Testing your serverless applications ... 1272
Building with PowerShell ... 1275
Development Environment ... 1277
Deployment package ... 1278
Creating a Lambda function ... 1278
Handler ... 1280
Returning data ... 1281
Context ... 1282
Logging ... 1283
Creating a function that returns logs ... 1283
Viewing logs in the Lambda console ... 1285
Viewing logs in the CloudWatch console ... 1285
Viewing logs using the AWS Command Line Interface (AWS CLI) ... 1285
Deleting logs ... 1289
Building with Rust ... 1290
Handler ... 1292
Setting up your Rust handler project ... 1292
Example Rust Lambda function code ... 1293
Valid class definitions for Rust handlers ... 1296
Handler naming conventions ... 1297
Defining and accessing the input event object ... 1297
Accessing and using the Lambda context object ... 1299
Using the AWS SDK for Rust in your handler ... 1299
Accessing environment variables ... 1300
Using shared state ... 1300
Code best practices for Rust Lambda functions ... 1301
Context ... 1303
Accessing invoke context information ... 1303
HTTP events ... 1305
Deploy .zip file archives ... 1307
Prerequisites ... 1307
Building the function ... 1307

Deploying the function ... 1308
Invoking the function ... 1310
Layers ... 1311
Logging ... 1312
Creating a function that writes logs ... 1312
Implementing advanced logging with the Tracing crate ... 1312
Best practices ... 1315
Function code ... 1315
Function configuration ... 1317
Function scalability ... 1318
Metrics and alarms ... 1318
Working with streams ... 1319
Security best practices ... 1321
Testing serverless functions ... 1322
Targeted business outcomes ... 1323
What to test ... 1323
How to test serverless ... 1324
Testing techniques ... 1325
Testing in the cloud ... 1325
Testing with mocks ... 1328
Testing locally using AWS SAM CLI ... 1329
Testing with emulation ... 1330
Best practices ... 1331
Prioritize testing in the cloud ... 1331
Structure your code for testability ... 1331
Accelerate development feedback loops ... 1331
Focus on integration tests ... 1332
Create isolated test environments ... 1332
Use mocks for isolated business logic ... 1333
Use emulators sparingly ... 1334
Challenges testing locally ... 1334
Example: Lambda function creates an S3 bucket ... 1335
Example: Lambda function processes messages from an Amazon SQS queue ... 1335
FAQ ... 1336
Next steps and resources ... 1337
Lambda SnapStart ... 1339

Use cases ... 1340
Supported features and limitations ... 1340
Supported Regions ... 1341
Compatibility considerations ... 1341
Pricing ... 1342
Activating SnapStart ... 1343
Activating SnapStart (console) ... 1343
Activating SnapStart (AWS CLI) ... 1344
Activating SnapStart (API) ... 1346
Function states ... 1347
Updating a snapshot ... 1347
Using SnapStart with AWS SDKs ... 1347
Using SnapStart with CloudFormation, AWS SAM, and AWS CDK ... 1348
Deleting snapshots ... 1348
Handling uniqueness ... 1349
Avoid saving state ... 1349
Use CSPRNGs ... 1351
Scanning tool (Java) ... 1354
Runtime hooks ... 1355
Java ... 1355
Python ... 1359
.NET ... 1361
Monitoring ... 1364
CloudWatch logs ... 1364
AWS X-Ray ... 1365
Telemetry API ... 1365
API Gateway and function URL metrics ... 1366
Security model ... 1367
Best practices ... 1368
Performance tuning ... 1368
Networking best practices ... 1372
Troubleshooting ... 1374
SnapStartNotReadyException ... 1374
SnapStartTimeoutException ... 1374
500 Internal Service Error ... 1375
401 Unauthorized ... 1375

UnknownHostException (Java) 1375
Snapshot creation failures 1376
Snapshot creation latency 1376
Tenant isolation 1377
When to use tenant isolation mode 1378
Supported features and limitations 1378
Supported AWS Regions 1378
Considerations 1378
Pricing 1379
Isolation mode 1379
Enabling tenant isolation 1380
Console 1380
AWS CLI 1380
API 1381
CloudFormation 1381
Invoking functions with tenant isolation 1383
AWS CLI 1383
API 1383
API Gateway 1384
SDK 1385
Accessing in code 1387
Access 1387
Usage patterns 1388
Monitoring 1389
Monitoring 1391
Understanding logging for tenant isolated mode 1391
Troubleshooting 1393
InvalidParameterValueException 1393
TooManyRequestsException 1393
Integrating other services 1394
Creating a trigger 1394
Services list 1395
Apache Kafka 1397
MSK 1399
Self-managed Apache Kafka 1447
Event poller scaling 1466

Polling and stream positions ... 1473
Consumer group ID ... 1474
Event filtering ... 1475
Schema registries with event sources ... 1480
Low latency Apache Kafka ... 1508
Retry configurations ... 1510
Retain failed invocations ... 1516
Kafka on-failure destination ... 1523
Kafka ESM logging ... 1528
Troubleshooting ... 1533
API Gateway ... 1540
Choosing an API type ... 1541
Adding an endpoint to your Lambda function ... 1542
Proxy integration ... 1542
Event format ... 1543
Response format ... 1543
Permissions ... 1544
Sample application ... 1546
The event handler from Powertools for AWS Lambda ... 1546
Tutorial ... 1548
Errors ... 1564
API Gateway vs function URLs ... 1565
Infrastructure Composer ... 1570
Exporting a Lambda function to Infrastructure Composer ... 1570
Other resources ... 1572
CloudFormation ... 1573
Amazon DocumentDB ... 1576
Example Amazon DocumentDB event ... 1577
Prerequisites and permissions ... 1578
Configure network security ... 1579
Creating an Amazon DocumentDB event source mapping (console) ... 1583
Creating an Amazon DocumentDB event source mapping (SDK or CLI) ... 1584
Polling and stream starting positions ... 1587
Monitoring your Amazon DocumentDB event source ... 1587
Tutorial ... 1588
DynamoDB ... 1614

Polling and batching streams ... 1615
Polling and stream starting positions ... 1616
Simultaneous readers ... 1616
Example event ... 1616
Create mapping ... 1618
Batch item failures ... 1621
Error handling ... 1637
Stateful processing ... 1644
Parameters ... 1649
Event filtering ... 1651
Tutorial ... 1660
EC2 ... 1677
Granting permissions to EventBridge (CloudWatch Events) ... 1677
Elastic Load Balancing (Application Load Balancer) ... 1679
Event Handler from Powertools for AWS Lambda ... 1681
Invoke using an EventBridge Scheduler ... 1682
Set up the execution role ... 1682
Create a schedule ... 1682
Related resources ... 1687
IoT ... 1688
Kinesis Data Streams ... 1690
Polling and batching streams ... 1690
Example event ... 1692
Create mapping ... 1693
Batch item failures ... 1698
Error handling ... 1717
Stateful processing ... 1723
Parameters ... 1727
Event filtering ... 1730
Tutorial ... 1734
Kubernetes ... 1751
AWS Controllers for Kubernetes (ACK) ... 1751
Crossplane ... 1751
MQ ... 1753
Understanding the Lambda consumer group for Amazon MQ ... 1755
Configure event source ... 1759

Parameters 1765
Event filtering 1766
Troubleshoot 1772
RDS 1774
Configuring your function to work with RDS resources 1774
Connecting to an Amazon RDS database in a Lambda function 1780
Processing event notifications from Amazon RDS 1800
Complete Lambda and Amazon RDS tutorial 1801
Amazon RDS vs DynamoDB 1801
S3 1805
Tutorial: Use an S3 trigger 1806
Tutorial: Use an Amazon S3 trigger to create thumbnails 1832
Secrets Manager 1861
Choosing an approach 1861
When to use Secrets Manager 1861
Using the AWS parameters and secrets Lambda extension 1862
Using the parameters utility from Powertools for AWS Lambda 1873
SQS 1879
Understanding polling and batching behavior for Amazon SQS event source mappings .. 1880
Using provisioned mode with Amazon SQS event source mappings 1880
Configuring provisioned mode for Amazon SQS event source mapping 1882
Example standard queue message event 1883
Example FIFO queue message event 1885
Create mapping 1886
Scaling behavior 1890
Error handling 1893
Parameters 1907
Event filtering 1909
Tutorial 1914
SQS cross-account tutorial 1932
Step Functions 1939
When to use Step Functions 1939
When not to use Step Functions 1946
S3 Batch 1949
Invoking Lambda functions from Amazon S3 batch operations 1950
SNS 1952

Idempotency utility from Powertools for AWS Lambda 1952
Adding an Amazon SNS topic trigger for a Lambda function using the console 1953
Manually adding an Amazon SNS topic trigger for a Lambda function 1953
Sample SNS event shape 1954
Tutorial 1955
Lambda permissions 1975
Execution role (permissions for functions to access other resources) 1977
Creating an execution role in the IAM console 1977
Creating and managing roles with the AWS CLI 1978
Grant least privilege access to your Lambda execution role 1979
Update execution role 1980
AWS managed policies 1981
Source function ARN 1984
Access permissions (permissions for other entities to access your functions) 1989
Identity-based policies 1989
Resource-based policies 1996
Attribute-based access control 2004
Resources and Conditions 2012
Security, governance, and compliance 2019
Data protection 2020
Encryption in transit 2021
Encryption at rest 2021
Using service-linked roles 2026
Service-linked role permissions for Lambda 2027
Creating a service-linked role for Lambda 2027
Editing a service-linked role for Lambda 2028
Deleting a service-linked role for Lambda 2028
Supported Regions for Lambda service-linked roles 2028
Identity and Access Management 2030
Audience 2030
Authenticating with identities 2031
Managing access using policies 2032
How AWS Lambda works with IAM 2034
Identity-based policy examples 2039
AWS managed policies 2042
Troubleshooting 2050

Governance ... 2052
Proactive controls with Guard ... 2055
Proactive controls with AWS Config ... 2059
Detective controls with AWS Config ... 2066
Code signing ... 2070
Code scanning ... 2073
Observability ... 2078
Compliance validation ... 2085
Resilience ... 2085
Infrastructure security ... 2086
Securing workloads with public endpoints ... 2087
Authentication and authorization ... 2087
Protecting API endpoints ... 2087
Code signing ... 2089
Signature validation ... 2089
Create configuration ... 2090
Permissions ... 2092
Code signing configuration tags ... 2093
Monitoring and debugging functions ... 2097
Pricing ... 2097
Function metrics ... 2098
View function metrics ... 2098
Metric types ... 2099
Function logs ... 2110
Choosing a service destination to send logs to ... 2110
Configuring log destinations ... 2111
Configuring advanced logging controls for Lambda functions ... 2111
Log formats ... 2112
Log-level filtering ... 2118
Log with CloudWatch Logs ... 2124
Log with Firehose ... 2142
Log with Amazon S3 ... 2144
CloudTrail logs ... 2149
Lambda data events in CloudTrail ... 2150
Lambda management events in CloudTrail ... 2152
Using CloudTrail to troubleshoot disabled Lambda event sources ... 2154

Lambda event examples ... 2155
AWS X-Ray ... 2157
Understanding X-Ray traces ... 2158
Default tracing behavior in Lambda ... 2162
Execution role permissions ... 2163
Enabling `Active` tracing with the Lambda API ... 2163
Enabling `Active` tracing with CloudFormation ... 2164
Function insights ... 2165
How it works ... 2165
Pricing ... 2166
Supported runtimes ... 2166
Enabling Lambda Insights in the console ... 2166
Enabling Lambda Insights programmatically ... 2166
Using the Lambda Insights dashboard ... 2167
Detecting function anomalies ... 2168
Troubleshooting a function ... 2170
What's next? ... 2172
View application metrics ... 2173
Application Signals ... 2175
How Application Signals integrates with Lambda ... 2175
Pricing ... 2176
Supported runtimes ... 2176
Enabling Application Signals in the Lambda console ... 2176
Using the Application Signals dashboard ... 2177
Debug with VS Code ... 2179
Supported runtimes ... 2179
Security and remote debugging ... 2179
Prerequisites ... 2180
Remotely debug Lambda functions ... 2180
Disable remote debugging ... 2181
Additional information ... 2182
Lambda layers ... 2183
How to use layers ... 2185
Layers and layer versions ... 2185
Packaging layers ... 2186
Layer paths for each Lambda runtime ... 2186

Creating and deleting layers ... 2190
Creating a layer ... 2190
Deleting a layer version ... 2191
Adding layers ... 2192
Finding layer information ... 2193
Layers with CloudFormation ... 2196
Layers with AWS SAM ... 2197
Lambda extensions ... 2198
Execution environment ... 2199
Impact on performance and resources ... 2200
Permissions ... 2200
Configuring extensions ... 2201
Configuring extensions (.zip file archive) ... 2201
Using extensions in container images ... 2201
Next steps ... 2202
Extensions partners ... 2203
AWS managed extensions ... 2204
Extensions API ... 2205
Lambda execution environment lifecycle ... 2206
Extensions API reference ... 2216
Telemetry API ... 2223
Creating extensions using the Telemetry API ... 2225
Registering your extension ... 2226
Creating a telemetry listener ... 2227
Specifying a destination protocol ... 2228
Configuring memory usage and buffering ... 2229
Sending a subscription request to the Telemetry API ... 2231
Inbound Telemetry API messages ... 2232
API reference ... 2235
`Event` schema reference ... 2239
Converting events to OTel Spans ... 2260
Logs API ... 2266
Troubleshooting ... 2279
Configuration ... 2279
Memory configurations ... 2280
CPU-bound configurations ... 2280

Timeouts 2280
Memory leakage between invocations 2281
Asynchronous results returned to a later invocation 2284
Deployment 2288
General: Permission is denied / Cannot load such file 2289
General: Error occurs when calling the UpdateFunctionCode 2290
Amazon S3: Error Code PermanentRedirect. 2290
General: Cannot find, cannot load, unable to import, class not found, no such file or directory 2291
General: Undefined method handler 2291
General: Lambda code storage limit exceeded 2292
Lambda: Layer conversion failed 2292
Lambda: InvalidParameterValueException or RequestEntityTooLargeException 2293
Lambda: InvalidParameterValueException 2294
Lambda: Concurrency and memory quotas 2294
Lambda: Invalid alias configuration for provisioned concurrency 2294
Invocation 2295
Lambda: Function times out during Init phase (Sandbox.Timedout) 2296
IAM: lambda:InvokeFunction not authorized 2297
Lambda: Couldn't find valid bootstrap (Runtime.InvalidEntrypoint) 2297
Lambda: Operation cannot be performed ResourceConflictException 2297
Lambda: Function is stuck in Pending 2298
Lambda: One function is using all concurrency 2298
General: Cannot invoke function with other accounts or services 2298
General: Function invocation is looping 2299
Lambda: Alias routing with provisioned concurrency 2299
Lambda: Cold starts with provisioned concurrency 2299
Lambda: Cold starts with new versions 2300
Lambda: Unexpected Node.js exit in runtime (Runtime.NodejsExit) 2300
EFS: Function could not mount the EFS file system 2301
EFS: Function could not connect to the EFS file system 2301
EFS: Function could not mount the EFS file system due to timeout 2301
Lambda: Lambda detected an IO process that was taking too long 2301
Container: CodeArtifactUserException errors 2302
Container: InvalidEntrypoint errors 2302
Execution 2302

Lambda: Remote debugging with Visual Studio Code ... 2303
Lambda: Execution takes too long ... 2304
Lambda: Unexpected event payload ... 2304
Lambda: Unexpectedly large payload sizes ... 2305
Lambda: JSON encoding and decoding errors ... 2305
Lambda: Logs or traces don't appear ... 2306
Lambda: Not all of my function's logs appear ... 2306
Lambda: The function returns before execution finishes ... 2307
Lambda: Running an unintended function version or alias ... 2308
Lambda: Detecting infinite loops ... 2308
General: Downstream service unavailability ... 2309
AWS SDK: Versions and updates ... 2310
Python: Libraries load incorrectly ... 2311
Java: Your function takes longer to process events after updating to Java 17 from Java 11 ... 2311
Kafka: Error handling and retry configuration issues ... 2312
Event source mapping ... 2312
Identifying and managing throttling ... 2313
Errors in the processing function ... 2315
Identifying and handling backpressure ... 2317
Networking ... 2318
VPC: Function loses internet access or times out ... 2318
VPC: TCP or UDP connection intermittently fails ... 2319
VPC: Function needs access to AWS services without using the internet ... 2319
VPC: Elastic network interface limit reached ... 2319
EC2: Elastic network interface with type of "lambda" ... 2320
DNS: Fail to connect to hosts with UNKNOWNHOSTEXCEPTION ... 2320
Sample applications ... 2321
Working with AWS SDKs ... 2324
Code examples ... 2326
Basics ... 2328
Hello Lambda ... 2329
Learn the basics ... 2339
Actions ... 2475
Scenarios ... 2609
Automatically confirm known users with a Lambda function ... 2610

Automatically migrate known users with a Lambda function ... 2651
Create a REST API to track COVID-19 data ... 2674
Create a lending library REST API ... 2675
Create a messenger application ... 2676
Create a serverless application to manage photos ... 2677
Create a websocket chat application ... 2681
Create an application to analyze customer feedback ... 2682
Invoke a Lambda function from a browser ... 2688
Transform data with S3 Object Lambda ... 2689
Use API Gateway to invoke a Lambda function ... 2690
Use Step Functions to invoke Lambda functions ... 2692
Use scheduled events to invoke a Lambda function ... 2692
Use the Neptune API to query graph data ... 2694
Write custom activity data with a Lambda function after Amazon Cognito user authentication ... 2695
Serverless examples ... 2718
Connecting to an Amazon RDS database in a Lambda function ... 2718
Invoke a Lambda function from a Kinesis trigger ... 2737
Invoke a Lambda function from a DynamoDB trigger ... 2748
Invoke a Lambda function from a Amazon DocumentDB trigger ... 2757
Invoke a Lambda function from an Amazon MSK trigger ... 2770
Invoke a Lambda function from an Amazon S3 trigger ... 2779
Invoke a Lambda function from an Amazon SNS trigger ... 2791
Invoke a Lambda function from an Amazon SQS trigger ... 2800
Reporting batch item failures for Lambda functions with a Kinesis trigger ... 2809
Reporting batch item failures for Lambda functions with a DynamoDB trigger ... 2822
Reporting batch item failures for Lambda functions with an Amazon SQS trigger ... 2833
AWS community contributions ... 2843
Build and test a serverless application ... 2843
Lambda quotas ... 2846
Compute and storage ... 2847
Function configuration, deployment, and execution ... 2849
Lambda API requests ... 2851
Other services ... 2853
Document history ... 2854
Earlier updates ... 2879

What is AWS Lambda?

> **Tip**
>
> Join Serverless experts for free hands-on workshops to learn how to build Serverless applications with best practices. Click here to sign up.

AWS Lambda is a compute service that runs code without the need to manage servers. Your code runs, scaling up and down automatically, with pay-per-use pricing. To get started, see Create your first function.

You can use Lambda for:

- **File processing**: Process files automatically when uploaded to Amazon Simple Storage Service. See file processing examples for details.
- **Long-running workflows:** Use durable Lambda functions to build stateful, multi-step workflows that can run for up to one year. Perfect for order processing, approval workflows, human-in-the-loop processes, and complex data pipelines that need to remember their progress.
- **Database operations and integration examples**: Respond to database changes and automate data workflows. See database examples for details.
- **Scheduled and periodic tasks**: Run automated operations on a regular schedule using EventBridge. See scheduled task examples for details.
- **Stream processing**: Process real-time data streams for analytics and monitoring. See Kinesis Data Streams for details.
- **Web applications**: Build scalable web apps that automatically adjust to demand.
- **Mobile backends**: Create secure API backends for mobile and web applications.
- **IoT backends**: Handle web, mobile, IoT, and third-party API requests. See IoT for details.

For pricing information, see AWS Lambda Pricing.

How Lambda works

When using Lambda, you are responsible only for your code. Lambda runs your code on a high-availability compute infrastructure and manages all the computing resources, including server and operating system maintenance, capacity provisioning, automatic scaling, and logging.

Because Lambda is a serverless, event-driven compute service, it uses a different programming paradigm than traditional web applications. The following model illustrates how Lambda works:

1. You write and organize your code in Lambda functions, which are the basic building blocks you use to create a Lambda application.
2. You control security and access through Lambda permissions, using execution roles to manage what AWS services your functions can interact with and what resource policies can interact with your code.
3. Event sources and AWS services trigger your Lambda functions, passing event data in JSON format, which your functions process (this includes event source mappings).
4. Lambda runs your code with language-specific runtimes (like Node.js and Python) in execution environments that package your runtime, layers, and extensions.

> **Tip**
>
> To learn how to build **serverless solutions**, check out the Serverless Developer Guide.

Key features

Configure, control, and deploy secure applications:

- Environment variables modify application behavior without new code deployments.
- Versions safely test new features while maintaining stable production environments.
- Lambda layers optimize code reuse and maintenance by sharing common components across multiple functions.
- Code signing enforce security compliance by ensuring only approved code reaches production systems.

Scale and perform reliably:

- Concurrency and scaling controls precisely manage application responsiveness and resource utilization during traffic spikes.
- Lambda SnapStart significantly reduce cold start times. Lambda SnapStart can provide as low as sub-second startup performance, typically with no changes to your function code.
- Response streaming optimize function performance by delivering large payloads incrementally for real-time processing.
- Container images package functions with complex dependencies using container workflows.

Connect and integrate seamlessly:

- VPC networks secure sensitive resources and internal services.
- File system integration that shares persistent data and manage stateful operations across function invocations.
- Function URLs create public-facing APIs and endpoints without additional services.
- Lambda extensions augment functions with monitoring, security, and operational tools.

Related information

- For information on how Lambda works, see How Lambda works.
- To start using Lambda, see Create your first Lambda function.
- For a list of example applications, see Getting started with example applications and patterns.

How Lambda works

Lambda functions are the basic building blocks you use to build Lambda applications. To write functions, it's essential to understand the core concepts and components that make up the Lambda programming model. This section will guide you through the fundamental elements you need to know to start building serverless applications with Lambda.

- **Lambda functions and function handlers** - A Lambda function is a small block of code that runs in response to events. Functions can be standard (up to 15 minutes) or durable (up to one year). Functions are the basic building blocks you use to build applications. Function handlers are the entry point for event objects that your Lambda function code processes.

- **Lambda execution environment and runtimes** - Lambda execution environments manage the resources required to run your function. For durable functions, the execution environment includes automatic state management and checkpointing capabilities. Runtimes are the language-specific environments your functions run in.
- **Events and triggers** - Other AWS services can invoke your functions in response to specific events. For durable functions, events can also trigger resumption of paused workflows.
- **Lambda permissions and roles** - Control who can access your functions and what other AWS services your functions can interact with. Durable functions require additional permissions for state management and extended execution.

> **Tip**
>
> If you want to start by understanding serverless development more generally, see Understanding the difference between traditional and serverless development in the *AWS Serverless Developer Guide*.

Lambda functions and function handlers

In Lambda, **functions** are the fundamental building blocks you use to create applications. A Lambda function is a piece of code that runs in response to events, such as a user clicking a button on a website or a file being uploaded to an Amazon Simple Storage Service (Amazon S3) bucket. With durable functions, your code can pause execution between steps, maintaining state automatically, making them ideal for long-running workflows like order processing or content moderation. You can think of a function as a kind of self-contained program with the following properties.

A Lambda **function handler** is the method in your function code that processes events. When a function runs in response to an event, Lambda runs the function handler. Data about the event that caused the function to run is passed directly to the handler. While the code in a Lambda function can contain more than one method or function, Lambda functions can only have one handler.

To create a Lambda function, you bundle your function code and its dependencies in a deployment package. Lambda supports two types of deployment package, .zip file archives and container images.

- A function has one specific job or purpose
- They run only when needed in response to specific events
- They automatically stop running when finished

Lambda execution environment and runtimes

Lambda functions run inside a secure, isolated *execution environment* which Lambda manages for you. For durable functions, the execution environment includes additional components for state management and workflow coordination. The execution environment manages the processes and resources that are needed to run your function. When a function is first invoked, Lambda creates a new execution environment for the function to run in. After the function has finished running, Lambda doesn't stop the execution environment right away; if the function is invoked again, Lambda can re-use the existing execution environment.

The Lambda execution environment also contains a *runtime*, a language-specific environment that relays event information and responses between Lambda and your function. Lambda provides a number of managed runtimes for the most popular programming languages, or you can create your own.

For managed runtimes, Lambda automatically applies security updates and patches to functions using the runtime.

Events and triggers

You can also invoke a Lambda function directly by using the Lambda console, AWS CLI, or one of the AWS Software Development Kits (SDKs). It's more usual in a production application for your function to be invoked by another AWS service in response to a particular event. For example, you might want a function to run whenever an item is added to an Amazon DynamoDB table.

To make your function respond to events, you set up a **trigger**. A trigger connects your function to an event source, and your function can have multiple triggers. When an event occurs, Lambda receives event data as a JSON document and converts it into an object that your code can process. You might define the following JSON format for your event and the Lambda runtime converts this JSON to an object before passing it to your function's handler.

Example custom Lambda event

```
{
```

```
  "Location": "SEA",
  "WeatherData":{
    "TemperaturesF":{
      "MinTempF": 22,
      "MaxTempF": 78
    },
    "PressuresHPa":{
      "MinPressureHPa": 1015,
      "MaxPressureHPa": 1027
    }
  }
}
```

Stream and queue services like Amazon Kinesis or Amazon SQS use an event source mapping instead of a standard trigger. Event source mappings poll the source for new data, batch records together, and then invoke your function with the batched events. For more information, see How event source mappings differ from direct triggers.

To understand how a trigger works, start by completing the Use an Amazon S3 trigger tutorial, or for a general overview of using triggers and instructions on creating a trigger using the Lambda console, see Integrating other services.

Lambda permissions and roles

For Lambda, there are two main types of permissions that you need to configure:

- Permissions that your function needs to access other AWS services
- Permissions that other users and AWS services need to access your function

The following sections describe both of these permission types and discuss best practices for applying least-privilege permissions.

Permissions for functions to access other AWS resources

Lambda functions often need to access other AWS resources and perform actions on them. For example, a function might read items from a DynamoDB table, store an object in an S3 bucket, or write to an Amazon SQS queue. To give functions the permissions they need to perform these actions, you use an *execution role*.

A Lambda execution role is a special kind of AWS Identity and Access Management (IAM) role, an identity you create in your account that has specific permissions associated with it defined in a *policy*.

Every Lambda function must have an execution role, and a single role can be used by more than one function. When a function is invoked, Lambda assumes the function's execution role and is granted permission to take the actions defined in the role's policy.

When you create a function in the Lambda console, Lambda automatically creates an execution role for your function. The role's policy gives your function basic permissions to write log outputs to Amazon CloudWatch Logs. To give your function permission to perform actions on other AWS resources, you need to edit the role to add the extra permissions. The easiest way to add permissions is to use an AWS managed policy. Managed policies are created and administered by AWS and provide permissions for many common use cases. For example, if your function performs CRUD operations on a DynamoDB table, you can add the AmazonDynamoDBFullAccess policy to your role.

Permissions for other users and resources to access your function

To grant other AWS service permission to access your Lambda function, you use a *resource-based policy*. In IAM, resource-based policies are attached to a resource (in this case, your Lambda function) and define who can access the resource and what actions they are allowed to take.

For another AWS service to invoke your function through a trigger, your function's resource-based policy must grant that service permission to use the `lambda:InvokeFunction` action. If you create the trigger using the console, Lambda automatically adds this permission for you.

To grant permission to other AWS users to access your function, you can define this in your function's resource-based policy in exactly the same way as for another AWS service or resource. You can also use an *identity-based policy* that's associated with the user.

Best practices for Lambda permissions

When you set permissions using IAM policies, security best practice is to grant only the permissions required to perform a task. This is known as the principle of *least privilege*. To get started granting permissions for your function, you might choose to use an AWS managed policy. Managed policies can be the quickest and easiest way to grant permissions to perform a task, but they might also include other permissions you don't need. As you move from early development through test and production, we recommend you reduce permissions to only those needed by defining your own customer-managed policies.

The same principle applies when granting permissions to access your function using a resource-based policy. For example, if you want to give permission to Amazon S3 to invoke your function, best practice is to limit access to individual buckets, or buckets in particular AWS accounts, rather than giving blanket permissions to the S3 service.

Running code with Lambda

When you write a Lambda function, you are creating code that will run in a unique serverless environment. Understanding how Lambda actually runs your code involves two key aspects: the programming model that defines how your code interacts with Lambda, and the execution environment lifecycle that determines how Lambda manages your code's runtime environment.

The Lambda programming model

Programming model functions as a common set of rules for how Lambda works with your code, regardless of whether you're writing in Python, Java, or any other supported language. The programming model includes your runtime and handler.

For standard functions:

1. Lambda receives an event.
2. Lambda uses the runtime to prepare the event in a format your code can use.
3. The runtime sends the formatted event to your handler.
4. Your handler processes the event using the code you've written.

For Durable Functions:

1. Lambda receives an event
2. The runtime prepares both the event and DurableContext
3. Your handler can:
   - Process steps with automatic checkpointing
   - Pause execution without consuming resources
   - Resume from the last successful checkpoint
   - Maintain state between steps

Essential to this model is the *handler*, where Lambda sends events to be processed by your code. Think of it as the entry point to your code. When Lambda receives an event, it passes this event and some context information to your handler. The handler then runs your code to process these events - for example, it might read a file when it's uploaded to Amazon S3, analyze an image, or update a database. Once your code finishes processing an event, the handler is ready to process the next one.

The Lambda execution model

While the programming model defines how Lambda interacts with your code, Execution environment is where Lambda actually runs your function — it's a secure, isolated compute space created specifically for your function.

Each environment follows a lifecycle that varies between standard and durable functions:

Standard Functions (up to 15 minutes):

1. **Initialization:** Environment setup and code loading
2. **Invocation:** Single execution of function code
3. **Shutdown:** Environment cleanup

Durable Functions (up to 1 year):

1. **Initialization:** Environment and durable state setup
2. **Invocation:** Multiple steps with automatic checkpointing
3. **Wait States:** Pause execution without resource consumption
4. **Resume:** Restart from last checkpoint
5. **Shutdown:** Cleanup of durable state

This environment handles important aspects of running your function. It provides your function with memory and a `/tmp` directory for temporary storage. **For Durable Functions, it also manages:**

- Automatic state persistence between steps
- Checkpoint storage and recovery
- Wait state coordination
- Progress tracking across long-running executions

Understanding the Lambda programming model

Lambda offers two programming models: standard functions that run up to 15 minutes, and Durable Functions that can run up to one year. While both share core concepts, Durable Functions add capabilities for long-running, stateful workflows.

Lambda provides a programming model that is common to all of the runtimes. The programming model defines the interface between your code and the Lambda system. You tell Lambda the entry point to your function by defining a *handler* in the function configuration. The runtime passes in objects to the handler that contain the invocation *event* and the *context*, such as the function name and request ID.

For Durable Functions, the handler also receives a DurableContext object that provides:

- Checkpointing capabilities through step()
- Wait state management through wait() and waitForCallback()
- Automatic state persistence between invocations

When the handler finishes processing the first event, the runtime sends it another. For Durable Functions, the handler can pause execution between steps, and Lambda will automatically save and restore state when the function resumes. The function's class stays in memory, so clients and variables that are declared outside of the handler method in *initialization code* can be reused. To save processing time on subsequent events, create reusable resources like AWS SDK clients during initialization. Once initialized, each instance of your function can process thousands of requests.

Your function also has access to local storage in the `/tmp` directory, a transient cache that can be used for multiple invocations. For more information, see Execution environment.

When AWS X-Ray tracing is enabled, the runtime records separate subsegments for initialization and execution.

The runtime captures logging output from your function and sends it to Amazon CloudWatch Logs. In addition to logging your function's output, the runtime also logs entries when function invocation starts and ends. This includes a report log with the request ID, billed duration, initialization duration, and other details. If your function throws an error, the runtime returns that error to the invoker.

> **Note**
>
> Logging is subject to CloudWatch Logs quotas. Log data can be lost due to throttling or, in some cases, when an instance of your function is stopped.

Key differences for Durable Functions:

- State is automatically persisted between steps
- Functions can pause execution without consuming resources
- Steps are automatically retried on failure
- Progress is tracked through checkpoints

Lambda scales your function by running additional instances of it as demand increases, and by stopping instances as demand decreases. This model leads to variations in application architecture, such as:

- Unless noted otherwise, incoming requests might be processed out of order or concurrently.
- Do not rely on instances of your function being long lived, instead store your application's state elsewhere.
- Use local storage and class-level objects to increase performance, but keep to a minimum the size of your deployment package and the amount of data that you transfer onto the execution environment.

For a hands-on introduction to the programming model in your preferred programming language, see the following chapters.

- Building Lambda functions with Node.js
- Building Lambda functions with Python
- Building Lambda functions with Ruby
- Building Lambda functions with Java
- Building Lambda functions with Go
- Building Lambda functions with C#
- Building Lambda functions with PowerShell

Understanding the Lambda execution environment lifecycle

Lambda execution environments support both standard functions (up to 15 minutes) and Durable Functions (up to one year). While both share the same basic lifecycle, Durable Functions add state management capabilities for long-running workflows.

Lambda invokes your function in an execution environment, which provides a secure and isolated runtime environment. The execution environment manages the resources required to run your

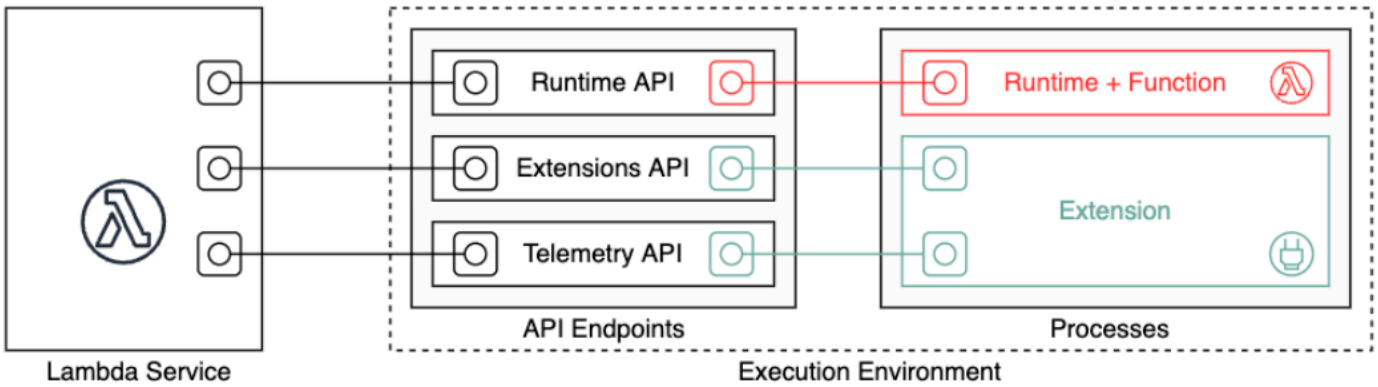
function. The execution environment also provides lifecycle support for the function's runtime and any external extensions associated with your function.

For Durable Functions, the execution environment includes additional components for:

- State persistence between steps
- Checkpointing management
- Wait state coordination
- Progress tracking

> **Lambda Managed Instances execution environment**
>
> If you are using Lambda Managed Instances, the execution environment has important differences compared to Lambda (default) functions. Managed Instances support concurrent invocations, use a different lifecycle model, and run on customer-owned infrastructure. For detailed information about the Managed Instances execution environment, see Understanding the Lambda Managed Instances execution environment.

The function's runtime communicates with Lambda using the Runtime API. Extensions communicate with Lambda using the Extensions API. Extensions can also receive log messages and other telemetry from the function by using the Telemetry API.



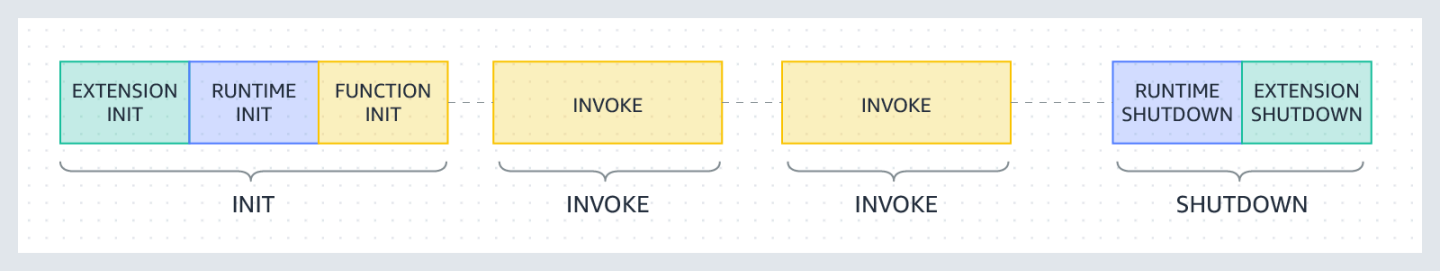
When you create your Lambda function, you specify configuration information, such as the amount of memory available and the maximum execution time allowed for your function. Lambda uses this information to set up the execution environment.

The function's runtime and each external extension are processes that run within the execution environment. Permissions, resources, credentials, and environment variables are shared between the function and the extensions.

Topics

- Lambda execution environment lifecycle
- Cold starts and latency
- Reducing cold starts with Provisioned Concurrency
- Optimizing static initialization

Lambda execution environment lifecycle



Each phase starts with an event that Lambda sends to the runtime and to all registered extensions. The runtime and each extension indicate completion by sending a `Next` API request. Lambda freezes the execution environment when the runtime and each extension have completed and there are no pending events.

The lifecycle phases for Durable Functions include:

- **Init:** Standard initialization plus durable state setup
- **Invoke:** Can include multiple step executions with automatic checkpointing
- **Wait:** Function can pause execution without consuming resources
- **Resume:** Function restarts from last checkpoint
- **Shutdown:** Cleanup of durable state and resources

Topics

- Init phase
- Failures during the Init phase
- Restore phase (Lambda SnapStart only)

- Invoke phase
- Failures during the invoke phase
- Shutdown phase

Init phase

In the `Init` phase, Lambda performs three tasks:

- Start all extensions (`Extension init`)
- Bootstrap the runtime (`Runtime init`)
- Run the function's static code (`Function init`)
- Run any before-checkpoint runtime hooks (Lambda SnapStart only)

The `Init` phase ends when the runtime and all extensions signal that they are ready by sending a `Next` API request. The `Init` phase is limited to 10 seconds. If all three tasks do not complete within 10 seconds, Lambda retries the `Init` phase at the time of the first function invocation with the configured function timeout.

When Lambda SnapStart is activated, the `Init` phase happens when you publish a function version. Lambda saves a snapshot of the memory and disk state of the initialized execution environment, persists the encrypted snapshot, and caches it for low-latency access. If you have a before-checkpoint runtime hook, then the code runs at the end of `Init` phase.

> **Note**
>
> The 10-second timeout doesn't apply to functions that are using provisioned concurrency, SnapStart, or Lambda Managed Instances. For provisioned concurrency, SnapStart, and Managed Instances functions, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the configured function timeout (maximum 900 seconds), whichever is higher.

When you use provisioned concurrency, Lambda initializes the execution environment when you configure the PC settings for a function. Lambda also ensures that initialized execution environments are always available in advance of invocations. You may see gaps between your function's invocation and initialization phases. Depending on your function's runtime and memory

configuration, you may also see variable latency on the first invocation on an initialized execution environment.

For functions using on-demand concurrency, Lambda may occasionally initialize execution environments ahead of invocation requests. When this happens, you may also observe a time gap between your function's initialization and invocation phases. We recommend you to not take a dependency on this behavior.

Failures during the Init phase

If a function crashes or times out during the `Init` phase, Lambda emits error information in the `INIT_REPORT` log.

Example— INIT_REPORT log for timeout

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: timeout
```

Example— INIT_REPORT log for extension failure

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: error Error Type:
 Extension.Crash
```

If the `Init` phase is successful, Lambda doesn't emit the `INIT_REPORT` log unless SnapStart or provisioned concurrency is enabled. SnapStart and provisioned concurrency functions always emit `INIT_REPORT`. For more information, see Monitoring for Lambda SnapStart.

Restore phase (Lambda SnapStart only)

When you first invoke a SnapStart function and as the function scales up, Lambda resumes new execution environments from the persisted snapshot instead of initializing the function from scratch. If you have an after-restore runtime hook, the code runs at the end of the `Restore` phase. You are charged for the duration of after-restore runtime hooks. The runtime must load and after-restore runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException. When the `Restore` phase completes, Lambda invokes the function handler (the Invoke phase).

Failures during the Restore phase

If the `Restore` phase fails, Lambda emits error information in the `RESTORE_REPORT` log.

Example— RESTORE_REPORT log for timeout

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: timeout
```

Example— RESTORE_REPORT log for runtime hook failure

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: error Error Type: Runtime.ExitError
```

For more information about the `RESTORE_REPORT` log, see Monitoring for Lambda SnapStart.

Invoke phase

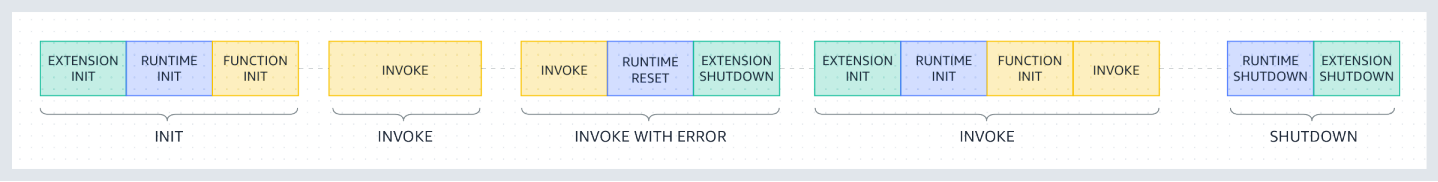
When a Lambda function is invoked in response to a `Next` API request, Lambda sends an `Invoke` event to the runtime and to each extension.

The function's timeout setting limits the duration of the entire `Invoke` phase. For example, if you set the function timeout as 360 seconds, the function and all extensions need to complete within 360 seconds. Note that there is no independent post-invoke phase. The duration is the sum of all invocation time (runtime + extensions) and is not calculated until the function and all extensions have finished executing.

The invoke phase ends after the runtime and all extensions signal that they are done by sending a `Next` API request.

Failures during the invoke phase

If the Lambda function crashes or times out during the `Invoke` phase, Lambda resets the execution environment. The following diagram illustrates Lambda execution environment behavior when there's an invoke failure:



In the previous diagram:

- The first phase is the **INIT** phase, which runs without errors.
- The second phase is the **INVOKE** phase, which runs without errors.
- At some point, suppose your function runs into an invoke failure (common causes include function timeouts, runtime errors, memory exhaustion, VPC connectivity issues, permission

errors, concurrency limits, and various configuration problems). For a complete list of possible invocation failures, see the section called "Invocation". The third phase, labeled **INVOKE WITH ERROR** , illustrates this scenario. When this happens, the Lambda service performs a reset. The reset behaves like a `Shutdown` event. First, Lambda shuts down the runtime, then sends a `Shutdown` event to each registered external extension. The event includes the reason for the shutdown. If this environment is used for a new invocation, Lambda re-initializes the extension and runtime together with the next invocation.

Note that the Lambda reset does not clear the `/tmp` directory content prior to the next init phase. This behavior is consistent with the regular shutdown phase.

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
> If your function's system log configuration is set to plain text, this change affects the log messages captured in CloudWatch Logs when your function experiences an invoke failure. The following examples show log outputs in both old and new formats.
> These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

Example CloudWatch Logs log output (runtime or extension crash) - old style

```
START RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Version: $LATEST
RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Error: Runtime exited without
 providing a reason
Runtime.ExitError
END RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1
REPORT RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Duration: 933.59 ms Billed
 Duration: 934 ms Memory Size: 128 MB Max Memory Used: 9 MB
```

Example CloudWatch Logs log output (function timeout) - old style

```
START RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Version: $LATEST
```

```
2024-03-04T17:22:38.033Z b70435cc-261c-4438-b9b6-efe4c8f04b21 Task timed out after
 3.00 seconds
END RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21
REPORT RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Duration: 3004.92 ms Billed
 Duration: 3117 ms Memory Size: 128 MB Max Memory Used: 33 MB Init Duration: 111.23
 ms
```

The new format for CloudWatch logs includes an additional `status`field in the REPORT line. In the case of a runtime or extension crash, the REPORT line also includes a field `ErrorType`.

Example CloudWatch Logs log output (runtime or extension crash) - new style

```
START RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Version: $LATEST
END RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd
REPORT RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Duration: 133.61 ms Billed
 Duration: 214 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 80.00
 ms Status: error Error Type: Runtime.ExitError
```

Example CloudWatch Logs log output (function timeout) - new style

```
START RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Version: $LATEST
END RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda
REPORT RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Duration: 3016.78 ms Billed
 Duration: 3101 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 84.00
 ms Status: timeout
```

- The fourth phase represents the **INVOKE** phase immediately following an invoke failure. Here, Lambda initializes the environment again by re-running the **INIT** phase. This is called a *suppressed init*. When suppressed inits occur, Lambda doesn't explicitly report an additional **INIT** phase in CloudWatch Logs. Instead, you may notice that the duration in the REPORT line includes an additional **INIT** duration + the **INVOKE** duration. For example, suppose you see the following logs in CloudWatch:

```
2022-12-20T01:00:00.000-08:00 START RequestId: XXX Version: $LATEST
2022-12-20T01:00:02.500-08:00 END RequestId: XXX
2022-12-20T01:00:02.500-08:00 REPORT RequestId: XXX Duration: 3022.91 ms
Billed Duration: 3000 ms Memory Size: 512 MB Max Memory Used: 157 MB
```

In this example, the difference between the REPORT and START timestamps is 2.5 seconds. This doesn't match the reported duration of 3022.91 millseconds, because it doesn't take into account the extra **INIT** (suppressed init) that Lambda performed. In this example, you can infer that the actual **INVOKE** phase took 2.5 seconds.

For more insight into this behavior, you can use the Accessing real-time telemetry data for extensions using the Telemetry API. The Telemetry API emits `INIT_START`, `INIT_RUNTIME_DONE`, and `INIT_REPORT` events with `phase=invoke` whenever suppressed inits occur in between invoke phases.

- The fifth phase represents the **SHUTDOWN** phase, which runs without errors.

Shutdown phase

When Lambda is about to shut down the runtime, it sends a `Shutdown` event to each registered external extension. Extensions can use this time for final cleanup tasks. The `Shutdown` event is a response to a `Next` API request.

Duration limit: The maximum duration of the `Shutdown` phase depends on the configuration of registered extensions:

- 0 ms – A function with no registered extensions
- 500 ms – A function with a registered internal extension
- 2,000 ms – A function with one or more registered external extensions

If the runtime or an extension does not respond to the `Shutdown` event within the limit, Lambda ends the process using a `SIGKILL` signal.

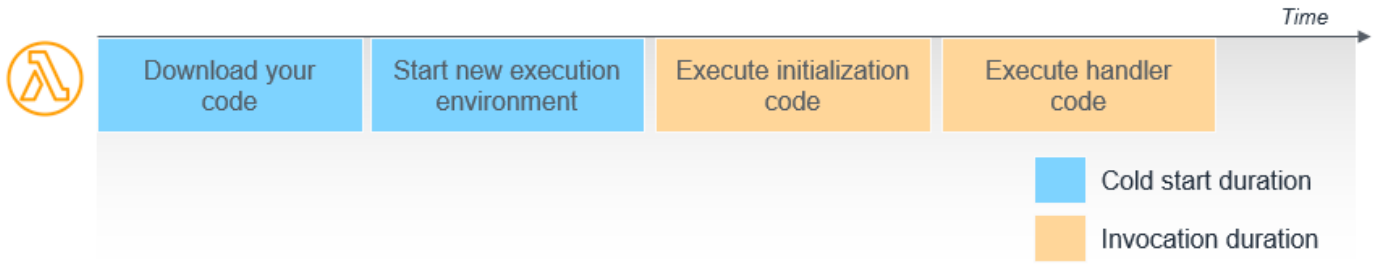
After the function and all extensions have completed, Lambda maintains the execution environment for some time in anticipation of another function invocation. However, Lambda terminates execution environments every few hours to allow for runtime updates and maintenance—even for functions that are invoked continuously. You should not assume that the execution environment will persist indefinitely. For more information, see Implement statelessness in functions.

When the function is invoked again, Lambda thaws the environment for reuse. Reusing the execution environment has the following implications:

- Objects declared outside of the function's handler method remain initialized, providing additional optimization when the function is invoked again. For example, if your Lambda function establishes a database connection, instead of reestablishing the connection, the original connection is used in subsequent invocations. We recommend adding logic in your code to check if a connection exists before creating a new one.
- Each execution environment provides between 512 MB and 10,240 MB, in 1-MB increments, of disk space in the `/tmp` directory. The directory content remains when the execution environment is frozen, providing a transient cache that can be used for multiple invocations. You can add extra code to check if the cache has the data that you stored. For more information on deployment size limits, see Lambda quotas.
- Background processes or callbacks that were initiated by your Lambda function and did not complete when the function ended resume if Lambda reuses the execution environment. Make sure that any background processes or callbacks in your code are complete before the code exits.

Cold starts and latency

When Lambda receives a request to run a function via the Lambda API, the service first prepares an execution environment. During this initialization phase, the service downloads your code, starts the environment, and runs any initialization code outside of the main handler. Finally, Lambda runs the handler code.



In this diagram, the first two steps of downloading the code and setting up the environment are frequently referred to as a “cold start”. You are charged for this time, and it adds latency to your overall invocation duration.

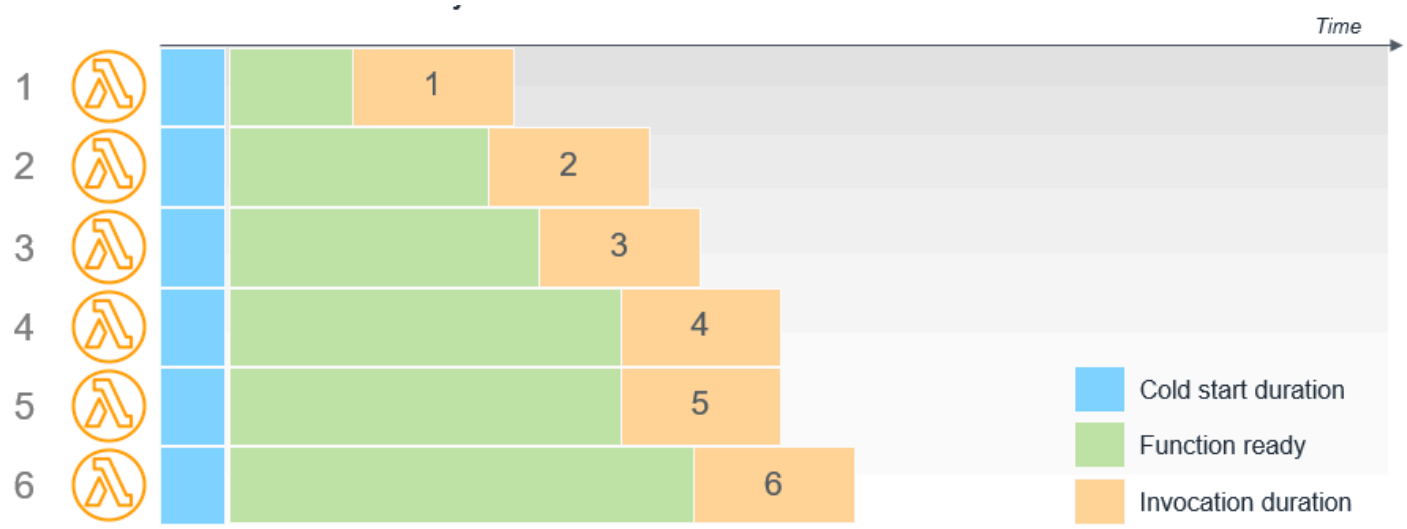
After the invocation completes, the execution environment is frozen. To improve resource management and performance, Lambda retains the execution environment for a period of time. During this time, if another request arrives for the same function, Lambda can reuse the environment. This second request typically finishes more quickly, since the execution environment is already fully set up. This is called a “warm start”.

Cold starts typically occur in under 1% of invocations. The duration of a cold start varies from under 100 ms to over 1 second. In general, cold starts are typically more common in development and test functions than production workloads. This is because development and test functions are usually invoked less frequently.

Reducing cold starts with Provisioned Concurrency

If you need predictable function start times for your workload, provisioned concurrency is the recommended solution to ensure the lowest possible latency. This feature pre-initializes execution environments, reducing cold starts.

For example, a function with a provisioned concurrency of 6 has 6 execution environments pre-warmed.



Optimizing static initialization

Static initialization happens before the handler code starts running in a function. This is the initialization code that you provide, that is outside of the main handler. This code is often used to import libraries and dependencies, set up configurations, and initialize connections to other services.

The following Python example shows importing, and configuring modules, and creating the Amazon S3 client during the initialization phase, before the `lambda_handler` function runs during invoke.

```
import os
import json
```

```
import cv2
import logging
import boto3

s3 = boto3.client('s3')
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

  # Handler logic...
```

The largest contributor of latency before function execution comes from initialization code. This code runs when a new execution environment is created for the first time. The initialization code is not run again if an invocation uses a warm execution environment. Factors that affect initialization code latency include:

- The size of the function package, in terms of imported libraries and dependencies, and Lambda layers.
- The amount of code and initialization work.
- The performance of libraries and other services in setting up connections and other resources.

There are a number of steps that developers can take to optimize static initialization latency. If a function has many objects and connections, you may be able to rearchitect a single function into multiple, specialized functions. These are individually smaller and each have less initialization code.

It's important that functions only import the libraries and dependencies that they need. For example, if you only use Amazon DynamoDB in the AWS SDK, you can require an individual service instead of the entire SDK. Compare the following three examples:

```
// Instead of const AWS = require('aws-sdk'), use:
const DynamoDB = require('aws-sdk/clients/dynamodb')

// Instead of const AWSXRay = require('aws-xray-sdk'), use:
const AWSXRay = require('aws-xray-sdk-core')

// Instead of const AWS = AWSXRay.captureAWS(require('aws-sdk')), use:
const dynamodb = new DynamoDB.DocumentClient()
AWSXRay.captureAWSClient(dynamodb.service)
```

Static initialization is also often the best place to open database connections to allow a function to reuse connections over multiple invocations to the same execution environment. However, you may have large numbers of objects that are only used in certain execution paths in your function. In this case, you can lazily load variables in the global scope to reduce the static initialization duration.

Avoid global variables for context-specific information. If your function has a global variable that is used only for the lifetime of a single invocation and is reset for the next invocation, use a variable scope that is local to the handler. Not only does this prevent global variable leaks across invocations, it also improves the static initialization performance.

Creating event-driven architectures with Lambda

An event is anything that triggers a Lambda function to run. Events can trigger a Lambda function in two ways: through direct invocation (push) and event source mappings (pull).

Many AWS services can directly invoke your Lambda functions. These services *push* events to your Lambda function. Events that trigger functions can be almost anything, from an HTTP request through API Gateway, a schedule managed by an EventBridge rule, an AWS IoT event, or an Amazon S3 event. With event source mapping, Lambda actively fetches (or *pulls*) events from a queue or stream. You configure Lambda to check for events from a supported service, and Lambda handles the polling and invocation of your function.

When passed to your function, events are structured in JSON format. The JSON structure varies depending on the service that generates it and the event type. While standard Lambda function invocations can last up to 15 minutes (or up to one year with durable functions), Lambda is best-suited for short invocations that last one second or less. This is particularly true of event-driven architectures, where each Lambda function is treated as a microservice responsible for performing a narrow set of specific instructions.

> **Note**
>
> Event-driven architectures communicate across different systems using networks, which introduce variable latency. For workloads that require very low latency, such as real-time trading systems, this design might not be the best choice. However, for highly scalable and available workloads, or those with unpredictable traffic patterns, event-driven architectures can provide an effective way to meet these demands.

Topics

- Benefits of event-driven architectures
- Trade-offs of event-driven architectures
- Anti-patterns in Lambda-based event-driven applications

Benefits of event-driven architectures

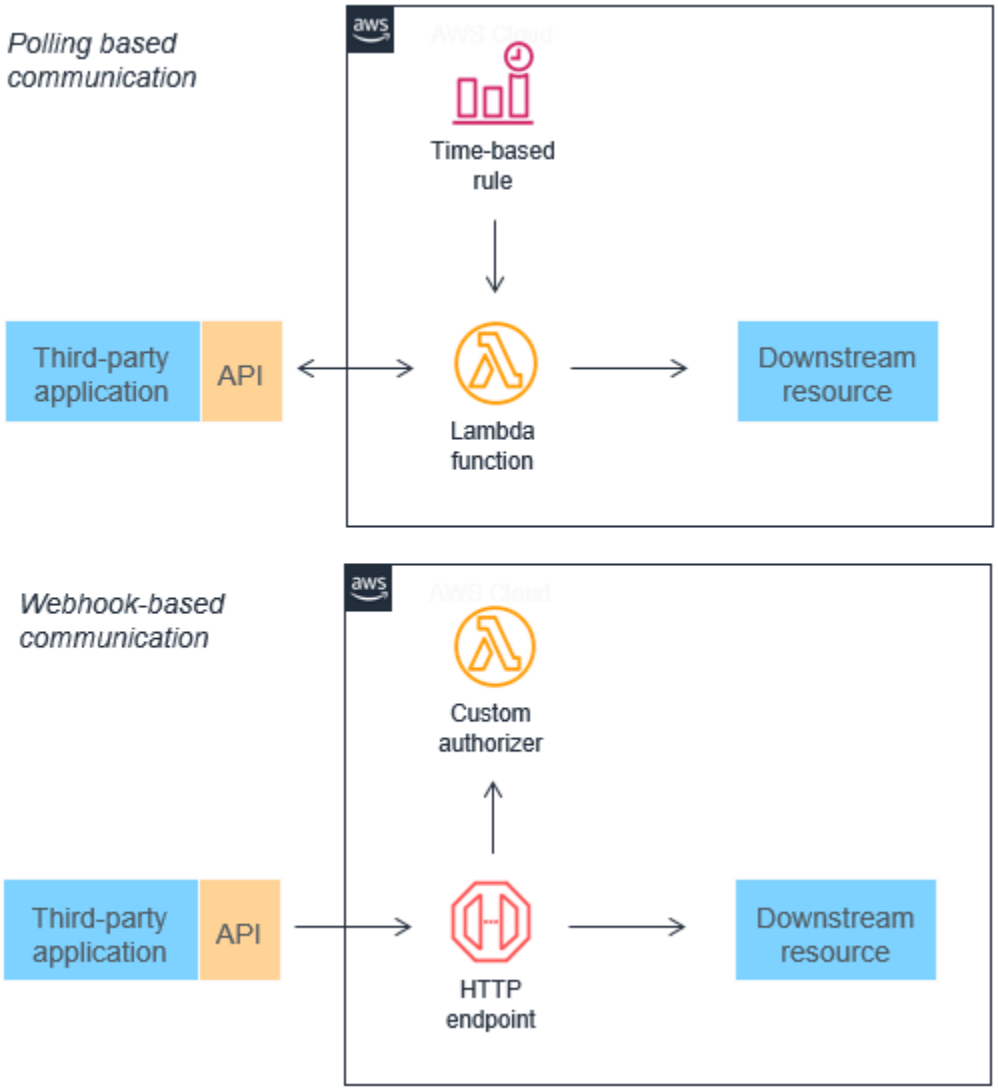
Lambda supports two methods of invocation in event-driven architectures:

1. Direct invocation (push method): AWS services trigger Lambda functions directly. For example:
   - Amazon S3 triggers a function when a file is uploaded
   - API Gateway triggers a function when it receives an HTTP request
2. Event source mapping (pull method): Lambda retrieves events and invokes functions. For example:
   - Lambda retrieves messages from an Amazon SQS queue and invokes a function
   - Lambda reads records from a DynamoDB stream and invokes a function

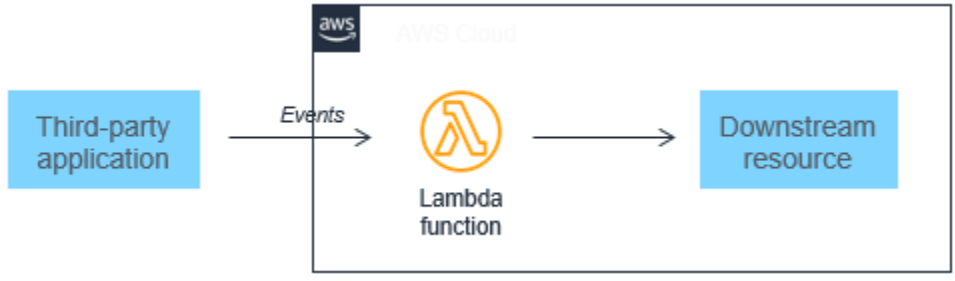
Both methods contribute to the benefits of event-driven architectures, as described below.

Replacing polling and webhooks with events

Many traditional architectures use polling and webhook mechanisms to communicate state between different components. Polling can be highly inefficient for fetching updates since there is a lag between new data becoming available and synchronization with downstream services. Webhooks are not always supported by other microservices that you want to integrate with. They might also require custom authorization and authentication configurations. In both cases, these integration methods are challenging to scale on-demand without additional work by development teams.



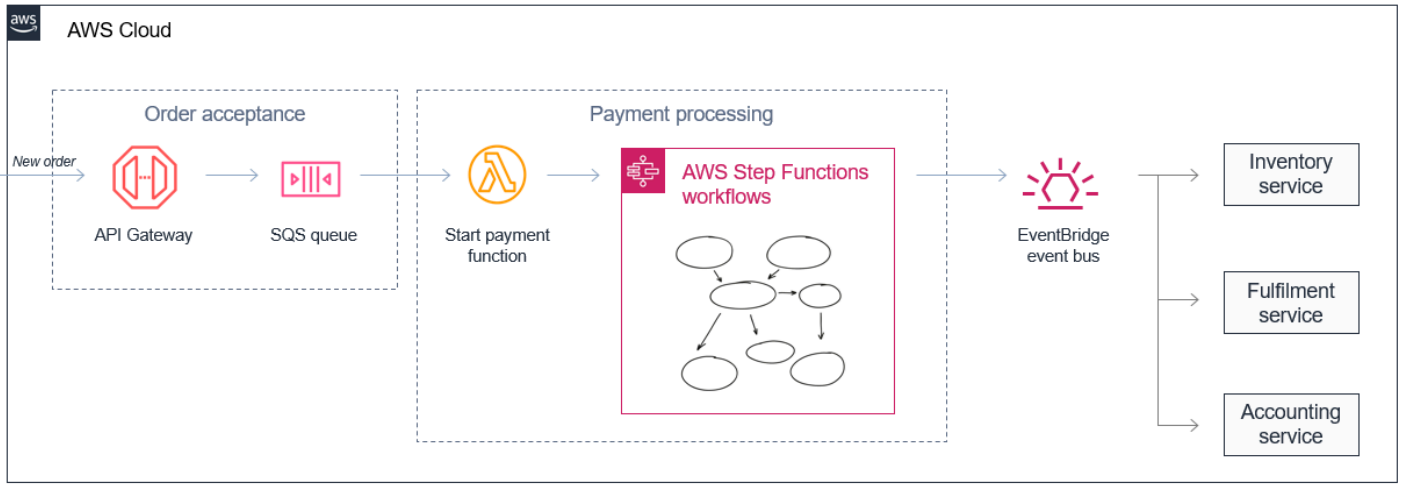
Both of these mechanisms can be replaced by events, which can be filtered, routed, and pushed downstream to consuming microservices. This approach can result in less bandwidth consumption, CPU utilization, and potentially lower cost. These architectures can also reduce complexity, since each functional unit is smaller and there is often less code.



Event-driven architectures can also make it easier to design near-real-time systems, helping organizations move away from batch-based processing. Events are generated at the time when state in the application changes, so the custom code of a microservice should be designed to handle the processing of a single event. Since scaling is handled by the Lambda service, this architecture can handle significant increases in traffic without changing custom code. As events scale up, so does the compute layer that processes events.

Reducing complexity

Microservices enable developers and architects to simplify complex workflows. For example, an ecommerce monolith can be broken down into order acceptance and payment processes with separate inventory, fulfillment and accounting services. What might be complex to manage and orchestrate in a monolith becomes a series of decoupled services that communicate asynchronously with events.



This approach also makes it possible to assemble services that process data at different rates. In this case, an order acceptance microservice can store high volumes of incoming orders by buffering the messages in an Amazon SQS queue.

A payment processing service, which is typically slower due to the complexity of handling payments, can take a steady stream of messages from the Amazon SQS queue. It can orchestrate complex retry and error handling logic using AWS Step Functions, and coordinate active payment workflows for hundreds of thousands of orders.

Alternative approach: For orchestration using standard programming languages, you can use Lambda durable functions. Durable functions let you write the order acceptance, payment processing, and notification logic in code with automatic checkpointing and retry. This approach

works well when the workflow primarily involves Lambda functions and you prefer keeping orchestration logic in code.

Improving scalability and extensibility

Microservices generate events that are typically published to messaging services like Amazon SNS and Amazon SQS. These behave like an elastic buffer between microservices and help handle scaling when traffic increases. Services like Amazon EventBridge can then filter and route messages depending upon the content of the event, as defined in rules. As a result, event-based applications can be more scalable and offer greater redundancy than monolithic applications.

This system is also highly extensible, allowing other teams to extend features and add functionality without impacting the order processing and payment processing microservices. By publishing events using EventBridge, this application integrates with existing systems, such as the inventory microservice, but also enables any future application to integrate as an event consumer. Producers of events have no knowledge of event consumers, which can help simplify the microservice logic.

Trade-offs of event-driven architectures

Variable latency

Unlike monolithic applications, which might process everything within the same memory space on a single device, event-driven applications communicate across networks. This design introduces variable latency. While it's possible to engineer applications to minimize latency, monolithic applications can almost always be optimized for lower latency at the expense of scalability and availability.

Workloads that require consistent low-latency performance, such as high-frequency trading applications in banks or sub-millisecond robotics automation in warehouses, are not good candidates for event-driven architecture.

Eventual consistency

An event represents a change in state, and with many events flowing through different services in an architecture at any given point of time, such workloads are often eventually consistent. This makes it more complex to process transactions, handle duplicates, or determine the exact overall state of a system.

Some workloads contain a combination of requirements that are eventually consistent (for example, total orders in the current hour) or strongly consistent (for example, current inventory).

For workloads needing strong data consistency, there are architecture patterns to support this. For example:

- DynamoDB can provide strongly consistent reads, sometimes at a higher latency, consuming a greater throughput than the default mode. DynamoDB can also support transactions to help maintain data consistency.
- You can use Amazon RDS for features needing ACID properties, though relational databases are generally less scalable than NoSQL databases like DynamoDB. Amazon RDS Proxy can help manage connection pooling and scaling from ephemeral consumers like Lambda functions.

Event-based architectures are usually designed around individual events instead of large batches of data. Generally, workflows are designed to manage the steps of an individual event or execution flow instead of operating on multiple events simultaneously. In serverless, real-time event processing is preferred over batch processing: batches should be replaced with many smaller incremental updates. While this can make workloads more available and scalable, it also makes it more challenging for events to have awareness of other events.

Returning values to callers

In many cases, event-based applications are asynchronous. This means that caller services do not wait for requests from other services before continuing with other work. This is a fundamental characteristic of event-driven architectures that enables scalability and flexibility. This means that passing return values or the result of a workflow is more complex than in synchronous execution flows.

Most Lambda invocations in production systems are asynchronous, responding to events from services like Amazon S3 or Amazon SQS. In these cases, the success or failure of processing an event is often more important than returning a value. Features such as dead letter queues (DLQs) in Lambda are provided to ensure you can identify and retry failed events, without needing to notify the caller.

Debugging across services and functions

Debugging event-driven systems is also different compared to a monolithic application. With different systems and services passing events, it's not possible to record and reproduce the exact state of multiple services when errors occur. Since each service and function invocation has separate log files, it can be more complicated to determine what happened to a specific event that caused an error.

There are three important requirements for building a successful debugging approach in event-driven systems. First, a robust logging system is critical, and this is provided across AWS services and embedded in Lambda functions by Amazon CloudWatch. Second, in these systems, it's important to ensure that every event has a transaction identifier that is logged at each step throughout a transaction, to help when searching for logs.

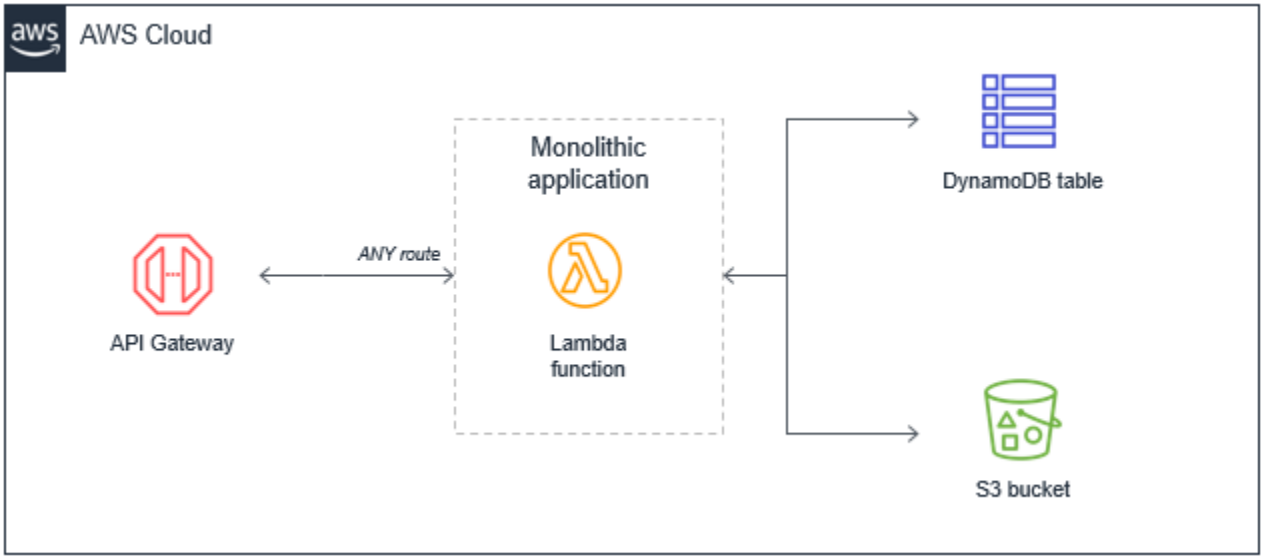
Finally, it's highly recommended to automate the parsing and analysis of logs by using a debugging and monitoring service like AWS X-Ray. This can consume logs across multiple Lambda invocations and services, making it much easier to pinpoint the root cause of issues. See Troubleshooting walkthrough for in-depth coverage of using X-Ray for troubleshooting.

Anti-patterns in Lambda-based event-driven applications

When building event-driven architectures with Lambda, avoid the following common anti-patterns. These patterns work but can increase costs and complexity.

The Lambda monolith

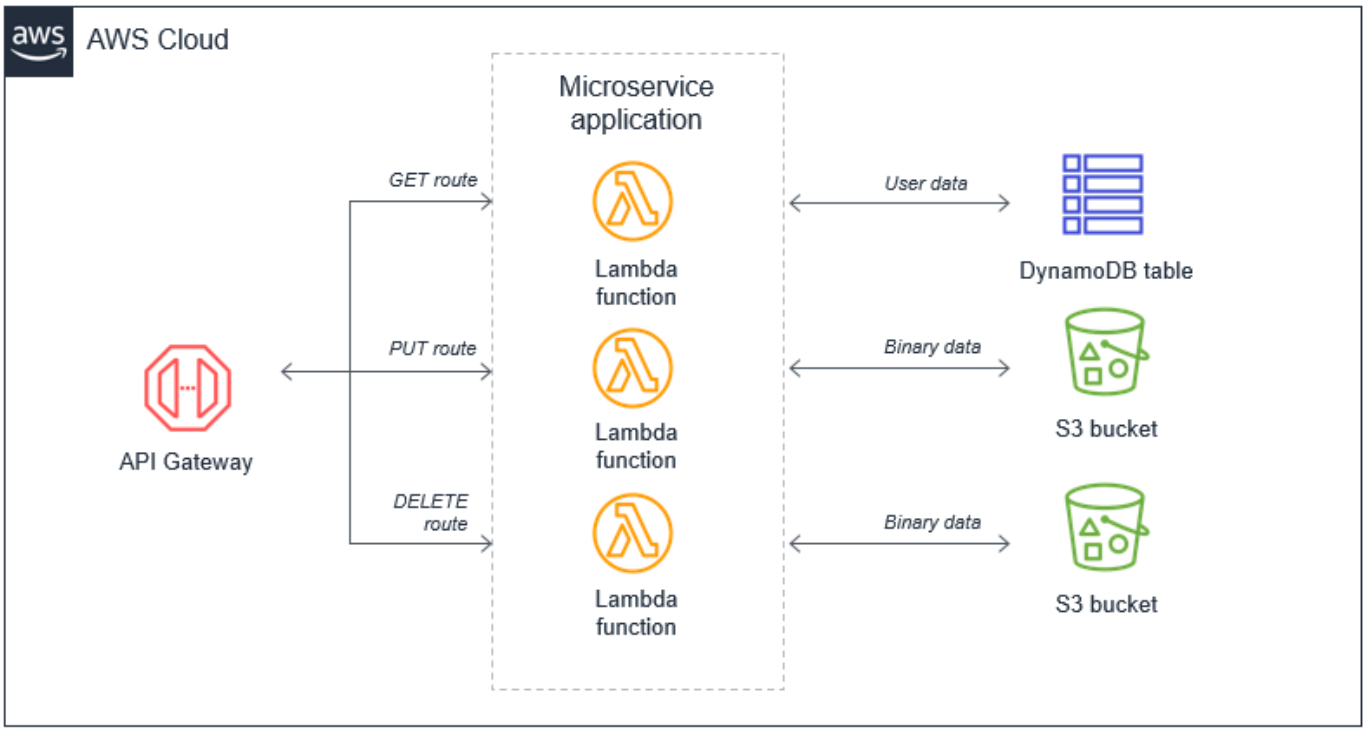
In many applications migrated from traditional servers, such as Amazon EC2 instances or Elastic Beanstalk applications, developers "lift and shift" existing code. Frequently, this results in a single Lambda function that contains all of the application logic that is triggered for all events. For a basic web application, a monolithic Lambda function would handle all API Gateway routes and integrate with all necessary downstream resources.



This approach has several drawbacks:

- **Package size** – The Lambda function might be much larger because it contains all possible code for all paths, which makes it slower for the Lambda service to run.
- **Hard to enforce least privilege** – The function's execution role must allow permissions to all resources needed for all paths, making the permissions very broad. This is a security concern. Many paths in the functional monolith do not need all the permissions that have been granted.
- **Harder to upgrade** – In a production system, any upgrades to the single function are more risky and could break the entire application. Upgrading a single path in the Lambda function is an upgrade to the entire function.
- **Harder to maintain** – It's more difficult to have multiple developers working on the service since it's a monolithic code repository. It also increases the cognitive burden on developers and makes it harder to create appropriate test coverage for code.
- **Harder to reuse code** – It's harder to separate reusable libraries from monoliths, making code reuse more difficult. As you develop and support more projects, this can make it harder to support the code and scale your team's velocity.
- **Harder to test** – As the lines of code increase, it becomes harder to unit test all the possible combinations of inputs and entry points in the code base. It's generally easier to implement unit testing for smaller services with less code.

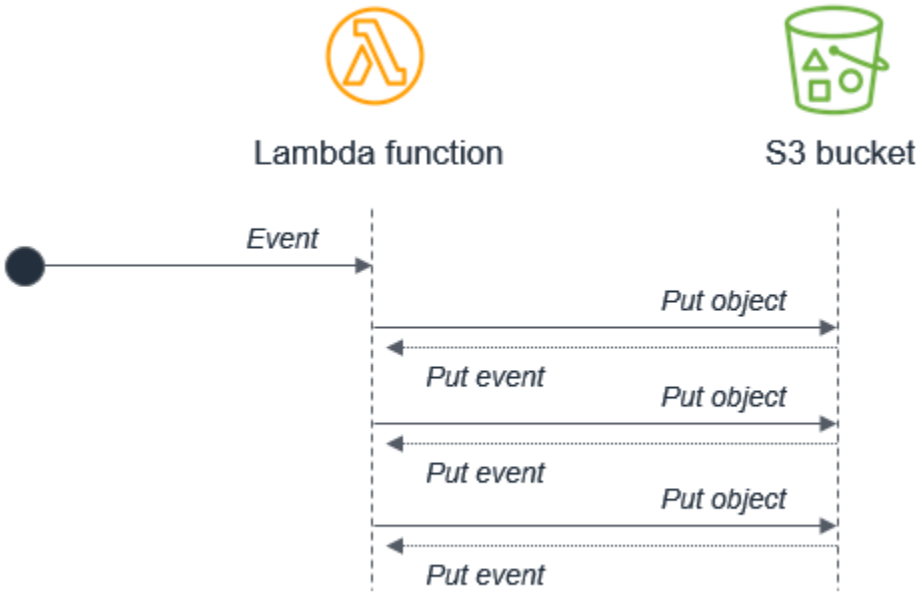
The preferred alternative is to break down the monolithic Lambda function into individual microservices, mapping a single Lambda function to a single, well-defined task. In this simple web application with a few API endpoints, the resulting microservice-based architecture can be based upon the API Gateway routes.



Recursive patterns that cause runaway Lambda functions

AWS services generate events that invoke Lambda functions, and Lambda functions can send messages to AWS services. Generally, the service or resource that invokes a Lambda function should be different to the service or resource that the function outputs to. Failure to manage this can result in infinite loops.

For example, a Lambda function writes an object to an Amazon S3 object, which in turn invokes the same Lambda function via a put event. The invocation causes a second object to be written to the bucket, which invokes the same Lambda function:



While the potential for infinite loops exists in most programming languages, this anti-pattern has the potential to consume more resources in serverless applications. Both Lambda and Amazon S3 automatically scale based upon traffic, so the loop can cause Lambda to scale to consume all available concurrency and Amazon S3 will continue to write objects and generate more events for Lambda.

This example uses S3, but the risk of recursive loops also exists in Amazon SNS, Amazon SQS, DynamoDB, and other services. You can use recursive loop detection to find and avoid this anti-pattern.

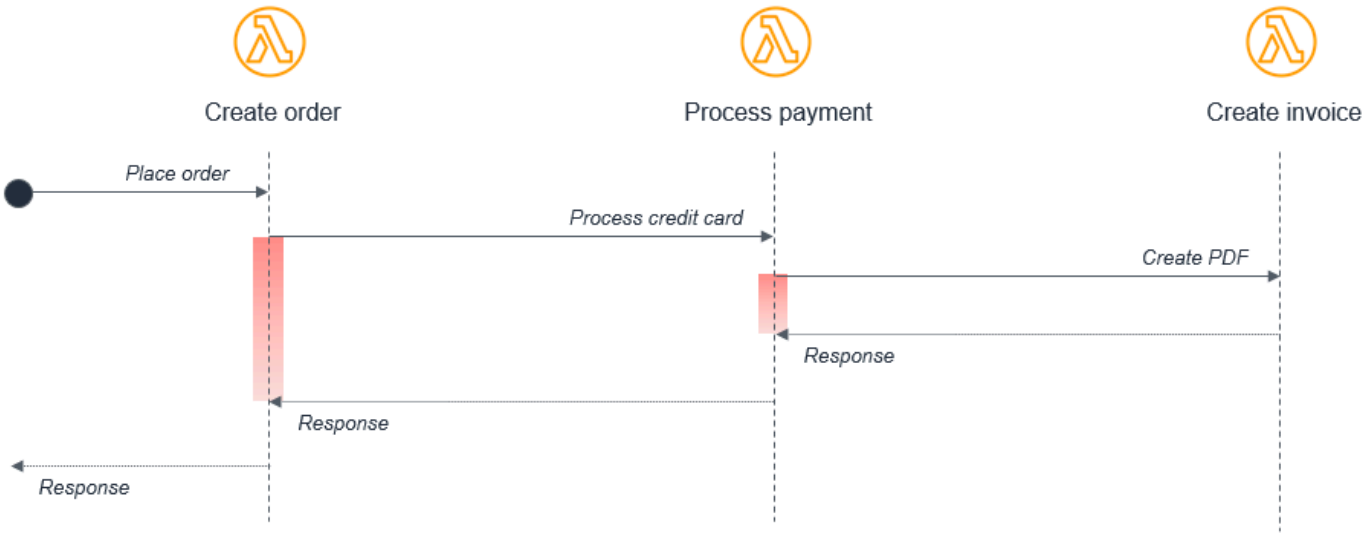
Lambda functions calling Lambda functions

Functions enable encapsulation and code re-use. Most programming languages support the concept of code synchronously calling functions within a code base. In this case, the caller waits until the function returns a response.

> **Note**
>
> While Lambda functions directly calling other Lambda functions is generally an anti-pattern due to cost and complexity concerns, this doesn't apply to durable functions, which are specifically designed to orchestrate multi-step workflows by invoking other functions.

When this happens on a traditional server or virtual instance, the operating system scheduler switches to other available work. Whether the CPU runs at 0% or 100% does not affect the overall cost of the application, since you are paying for the fixed cost of owning and operating a server.

This model often does not adapt well to serverless development. For example, consider a simple ecommerce application consisting of three Lambda functions that process an order:



In this case, the *Create order* function calls the *Process payment* function, which in turn calls the *Create invoice* function. While this synchronous flow might work within a single application on a server, it introduces several avoidable problems in a distributed serverless architecture:

- **Cost** – With Lambda, you pay for the duration of an invocation. In this example, while the *Create invoice* functions runs, two other functions are also running in a wait state, shown in red on the diagram.
- **Error handling** – In nested invocations, error handling can become much more complex. For example, an error in *Create invoice* might require the *Process payment* function to reverse the charge, or it might instead retry the *Create invoice* process.
- **Tight coupling** – Processing a payment typically takes longer than creating an invoice. In this model, the availability of the entire workflow is limited by the slowest function.
- **Scaling** – The concurrency of all three functions must be equal. In a busy system, this uses more concurrency than would otherwise be needed.

In serverless applications, there are two common approaches to avoid this pattern. First, use an Amazon SQS queue between Lambda functions. If a downstream process is slower than an upstream process, the queue durably persists messages and decouples the two functions. In this

example, the *Create order* function would publish a message to an Amazon SQS queue, and the *Process payment* function consumes messages from the queue.

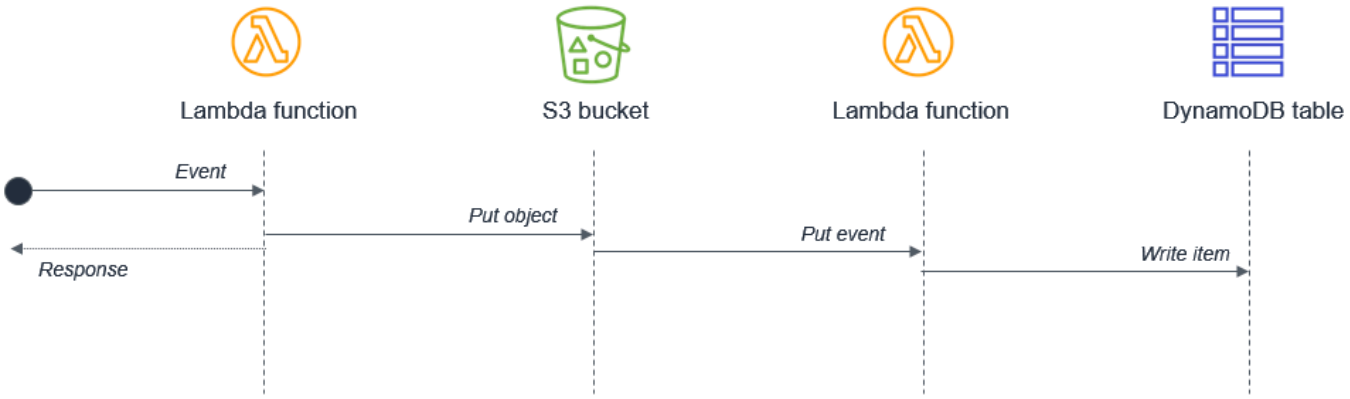
The second approach is to use AWS Step Functions. For complex processes with multiple types of failure and retry logic, Step Functions can help reduce the amount of custom code needed to orchestrate the workflow. As a result, Step Functions orchestrates the work and robustly handles errors and retries, and the Lambda functions contain only business logic.

Synchronous waiting within a single Lambda function

Make sure that any potentially concurrent activities are not scheduled synchronously within a single Lambda function. For example, a Lambda function might write to an S3 bucket and then write to a DynamoDB table:



In this design, wait times are compounded because the activities are sequential. In cases where the second task depends on the completion of the first task, you can reduce the total waiting time and the cost of execution by have two separate Lambda functions:



In this design, the first Lambda function responds immediately after putting the object to the Amazon S3 bucket. The S3 service invokes the second Lambda function, which then writes data to the DynamoDB table. This approach minimizes the total wait time in the Lambda function executions.

Designing Lambda applications

A well-architected event-driven application uses a combination of AWS services and custom code to process and manage requests and data. This chapter focuses on Lambda-specific topics in application design. There are many important considerations for serverless architects when designing applications for busy production systems.

Many of the best practices that apply to software development and distributed systems also apply to serverless application development. The overall goal is to develop workloads that are:

- **Reliable** – offering your end users a high level of availability. AWS serverless services are reliable because they are also designed for failure.
- **Durable** – providing storage options that meet the durability needs of your workload.
- **Secure** – following best practices and using the tools provided to secure access to workloads and limit the blast radius.
- **Performant** – using computing resources efficiently and meeting the performance needs of your end users.
- **Cost-efficient**– designing architectures that avoid unnecessary cost that can scale without overspending, and also be decommissioned without significant overhead.

The following design principles can help you build workloads that meet these goals. Not every principle may apply to every architecture, but they should guide you in general architecture decisions.

Topics

- Use services instead of custom code
- Understand Lambda abstraction levels
- Implement statelessness in functions
- Minimize coupling
- Build for on-demand data instead of batches
- Choose an orchestration option for complex workflows
- Implement idempotency
- Use multiple AWS accounts for managing quotas

Use services instead of custom code

Serverless applications usually comprise several AWS services, integrated with custom code run in Lambda functions. While Lambda can be integrated with most AWS services, the services most commonly used in serverless applications are:

Category	AWS service
Compute	AWS Lambda
Data storage	Amazon S3 Amazon DynamoDB Amazon RDS
API	Amazon API Gateway
Application integration	Amazon EventBridge Amazon SNS Amazon SQS
Orchestration	Lambda durable functions AWS Step Functions
Streaming data and analytics	Amazon Data Firehose

> **Note**
>
> Many serverless services provide replication and support for multiple Regions, including DynamoDB and Amazon S3. Lambda functions can be deployed in multiple Regions as part of a deployment pipeline, and API Gateway can be configured to support this configuration. See this example architecture that shows how this can be achieved.

There are many well-established, common patterns in distributed architectures that you can build yourself or implement using AWS services. For most customers, there is little commercial value in

investing time to develop these patterns from scratch. When your application needs one of these patterns, use the corresponding AWS service:

Pattern	AWS service
Queue	Amazon SQS
Event bus	Amazon EventBridge
Publish/subscribe (fan-out)	Amazon SNS
Orchestration	Lambda durable functions AWS Step Functions
API	Amazon API Gateway
Event streams	Amazon Kinesis

These services are designed to integrate with Lambda and you can use infrastructure as code (IaC) to create and discard resources in the services. You can use any of these services via the AWS SDK without needing to install applications or configure servers. Becoming proficient with using these services via code in your Lambda functions is an important step to producing well-designed serverless applications.

Understand Lambda abstraction levels

The Lambda service limits your access to the underlying operating systems, hypervisors, and hardware running your Lambda functions. The service continuously improves and changes infrastructure to add features, reduce cost and make the service more performant. Your code should assume no knowledge of how Lambda is architected and assume no hardware affinity.

Similarly, Lambda's integrations with other services are managed by AWS, with only a small number of configuration options exposed to you. For example, when API Gateway and Lambda interact, there is no concept of load balancing since it is entirely managed by the services. You also have no direct control over which Availability Zones the services use when invoking functions at any point in time, or how Lambda determines when to scale up or down the number of execution environments.

This abstraction helps you focus on the integration aspects of your application, the flow of data, and the business logic where your workload provides value to your end users. Allowing the services to manage the underlying mechanics helps you develop applications more quickly with less custom code to maintain.

Implement statelessness in functions

For standard Lambda functions, you should assume that the environment exists only for a single invocation. The function should initialize any required state when it is first started. For example, your function may require fetching data from a DynamoDB table. It should commit any permanent data changes to a durable store such as Amazon S3, DynamoDB, or Amazon SQS before exiting. It should not rely on any existing data structures or temporary files, or any internal state that would be managed by multiple invocations.

When using Durable Functions, state is automatically preserved between invocations, eliminating the need to manually persist state to external storage. However, you should still follow stateless principles for any data not explicitly managed through the DurableContext.

To initialize database connections and libraries, or load state, you can take advantage of static initialization. Since execution environments are reused where possible to improve performance, you can amortize the time taken to initialize these resources over multiple invocations. However, you should not store any variables or data used in the function within this global scope.

Minimize coupling

Most architectures should prefer many, shorter functions over fewer, larger ones. The purpose of each function should be to handle the event passed into the function, with no knowledge or expectations of the overall workflow or volume of transactions. This makes the function agnostic to the source of the event with minimal coupling to other services.

Any global-scope constants that change infrequently should be implemented as environment variables to allow updates without deployments. Any secrets or sensitive information should be stored in AWS Systems Manager Parameter Store or AWS Secrets Manager and loaded by the function. Since these resources are account-specific, you can create build pipelines across multiple accounts. The pipelines load the appropriate secrets per environment, without exposing these to developers or requiring any code changes.

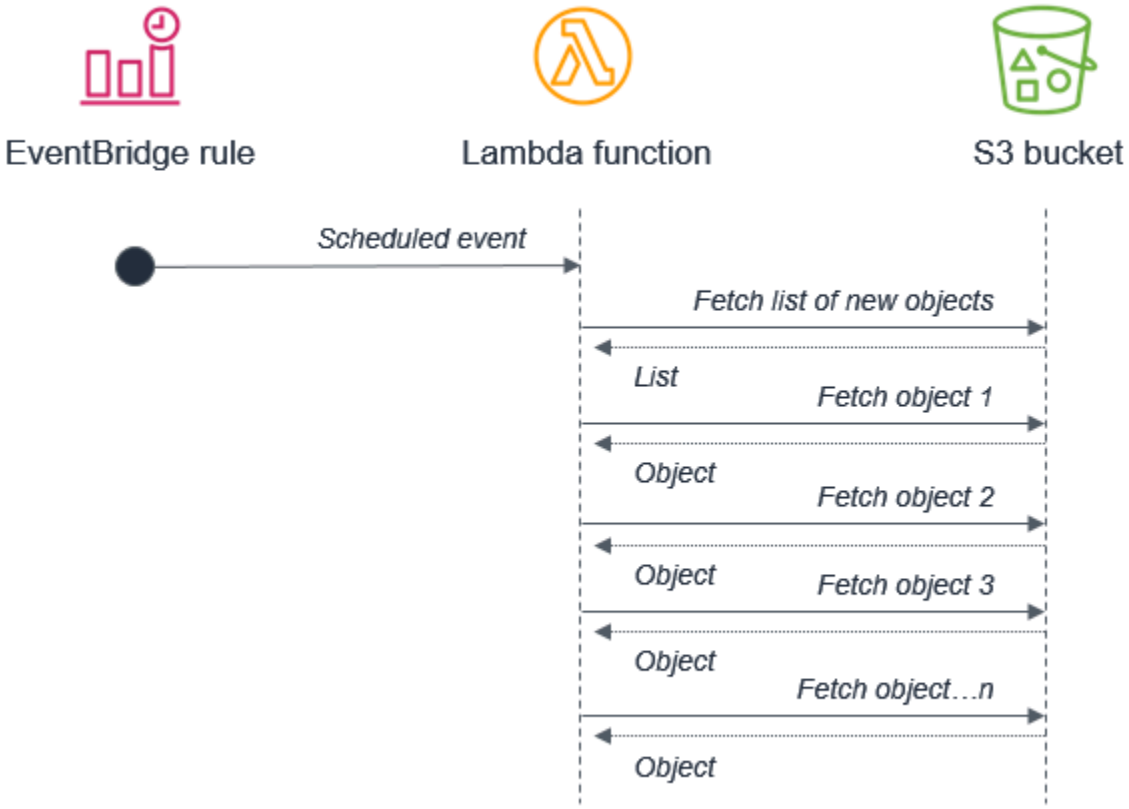
Build for on-demand data instead of batches

Many traditional systems are designed to run periodically and process batches of transactions that have built up over time. For example, a banking application may run every hour to process ATM transactions into central ledgers. In Lambda-based applications, the custom processing should be triggered by every event, allowing the service to scale up concurrency as needed, to provide near-real time processing of transactions.

While standard Lambda functions are limited to 15 minutes of execution time, Durable Functions can run for up to one year, making them suitable for longer-running processing needs. However, you should still prefer event-driven processing over batch processing when possible.

While you can run cron tasks in serverless applications by using scheduled expressions for rules in Amazon EventBridge, these should be used sparingly or as a last-resort. In any scheduled task that processes a batch, there is the potential for the volume of transactions to grow beyond what can be processed within the 15-minute Lambda duration limit. If the limitations of external systems force you to use a scheduler, you should generally schedule for the shortest reasonable recurring time period.

For example, it's not best practice to use a batch process that triggers a Lambda function to fetch a list of new Amazon S3 objects. This is because the service may receive more new objects in between batches than can be processed within a 15-minute Lambda function.



Instead, Amazon S3 should invoke the Lambda function each time a new object is put into the bucket. This approach is significantly more scalable and works in near-real time.



Choose an orchestration option for complex workflows

Workflows that involve branching logic, different types of failure models, and retry logic typically use an orchestrator to keep track of the state of the overall execution. Don't build ad-hoc orchestration in standard Lambda functions. This results in tight coupling, complex routing code, and no automatic state recovery.

Instead, use one of these purpose-built orchestration options:

- **Lambda durable functions:** Application-centric orchestration using standard programming languages with automatic checkpointing, built-in retry, and execution recovery. Ideal for developers who prefer keeping workflow logic in code alongside business logic within Lambda.
- **AWS Step Functions:** Visual workflow orchestration with native integrations to 220+ AWS services. Ideal for multi-service coordination, zero-maintenance infrastructure, and visual workflow design.

For guidance on choosing between these options, see Durable functions or Step Functions.

With Step Functions, you use state machines to manage orchestration. This extracts the error handling, routing, and branching logic from your code, replacing it with state machines declared using JSON. Apart from making workflows more robust and observable, you can also add versioning to workflows and make the state machine a codified resource that you can add to a code repository.

It's common for simpler workflows in Lambda functions to become more complex over time. When operating a production serverless application, it's important to identify when this is happening, so you can migrate this logic to a state machine or durable function.

Implement idempotency

AWS serverless services, including Lambda, are fault-tolerant and designed to handle failures. For example, if a service invokes a Lambda function and there is a service disruption, Lambda invokes your function in a different Availability Zone. If your function throws an error, Lambda retries the invocation.

Since the same event may be received more than once, functions should be designed to be idempotent. This means that receiving the same event multiple times does not change the result beyond the first time the event was received.

You can implement idempotency in Lambda functions by using a DynamoDB table to track recently processed identifiers to determine if the transaction has already been handled previously. The DynamoDB table usually implements a Time To Live (TTL) value to expire items to limit the storage space used.

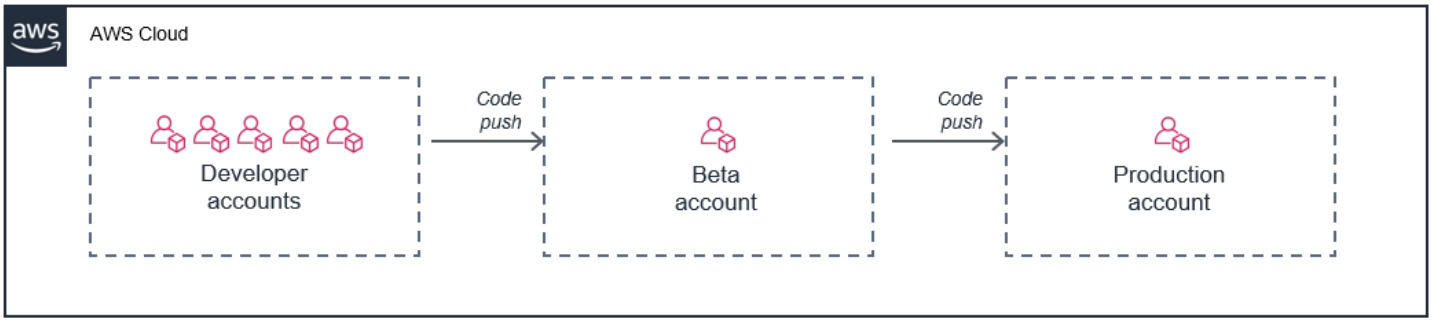
Use multiple AWS accounts for managing quotas

Many service quotas in AWS are set at the account level. This means that as you add more workloads, you can quickly exhaust your limits.

An effective way to solve this issue is to use multiple AWS accounts, dedicating each workload to its own account. This prevents quotas from being shared with other workloads or non-production resources.

In addition, by using AWS Organizations, you can centrally manage the billing, compliance, and security of these accounts. You can attach policies to groups of accounts to avoid custom scripts and manual processes.

One common approach is to provide each developer with an AWS account, and then use separate accounts for a beta deployment stage and production:



In this model, each developer has their own set of limits for the account, so their usage does not impact your production environment. This approach also allows developers to test Lambda functions locally on their development machines against live cloud resources in their individual accounts.

Create your first Lambda function

To get started with Lambda, use the Lambda console to create a function. In a few minutes, you can create and deploy a function and test it in the console.

As you carry out the tutorial, you'll learn some fundamental Lambda concepts, like how to pass arguments to your function using the Lambda *event object*. You'll also learn how to return log outputs from your function, and how to view your function's invocation logs in Amazon CloudWatch Logs.

To keep things simple, you create your function using either the Python or Node.js runtime. With these interpreted languages, you can edit function code directly in the console's built-in code editor. With compiled languages like Java and C#, you must create a deployment package on your local build machine and upload it to Lambda. To learn about deploying functions to Lambda using other runtimes, see the links in the the section called “Next steps” section.

> **Tip**
>
> To learn how to build **serverless solutions**, check out the Serverless Developer Guide.

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Create a Lambda function with the console

In this example, your function takes a JSON object containing two integer values labeled `"length"` and `"width"`. The function multiplies these values to calculate an area and returns this as a JSON string.

Your function also prints the calculated area, along with the name of its CloudWatch log group. Later in the tutorial, you'll learn to use CloudWatch Logs to view records of your functions' invocation.

To create a Hello world Lambda function with the console

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Select **Author from scratch**.
4. In the **Basic information** pane, for **Function name**, enter **`myLambdaFunction`**.
5. For **Runtime**, choose either **Node.js 24** or **Python 3.14**.
6. Leave **architecture** set to **x86_64**, and then choose **Create function**.

In addition to a simple function that returns the message `Hello from Lambda!`, Lambda also creates an execution role for your function. An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and

resources. For your function, the role that Lambda creates grants basic permissions to write to CloudWatch Logs.

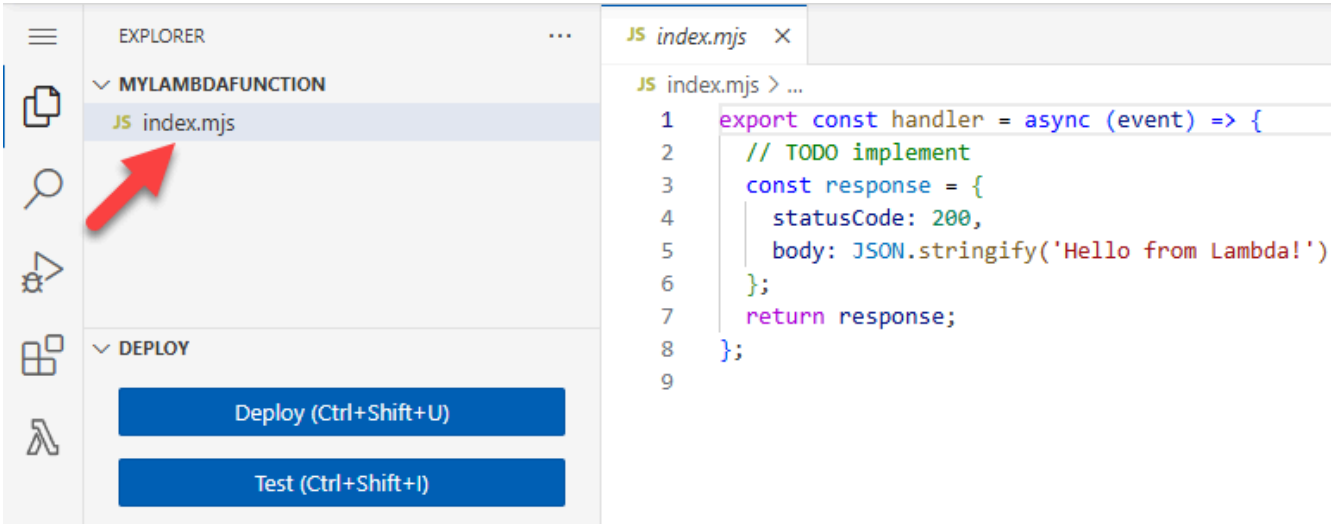
Use the console's built-in code editor to replace the Hello world code that Lambda created with your own function code.

Node.js

To modify the code in the console

1. Choose the **Code** tab.

   In the console's built-in code editor, you should see the function code that Lambda created. If you don't see the **index.mjs** tab in the code editor, select **index.mjs** in the file explorer as shown on the following diagram.



2. Paste the following code into the **index.mjs** tab, replacing the code that Lambda created.

```
export const handler = async (event, context) => {

  const length = event.length;
  const width = event.width;
  let area = calculateArea(length, width);
  console.log(`The area is ${area}`);

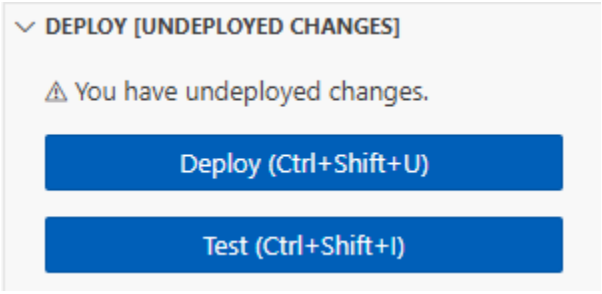
  console.log('CloudWatch log group: ', context.logGroupName);

  let data = {
    "area": area,
  };
```

```
    return JSON.stringify(data);

  function calculateArea(length, width) {
    return length * width;
  }
};
```

3. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Understanding your function code

Before you move to the next step, let's take a moment to look at the function code and understand some key Lambda concepts.

- The Lambda handler:

  Your Lambda function contains a Node.js function named `handler`. A Lambda function in Node.js can contain more than one Node.js function, but the *handler* function is always the entry point to your code. When your function is invoked, Lambda runs this method.

  When you created your Hello world function using the console, Lambda automatically set the name of the handler method for your function to `handler`. Be sure not to edit the name of this Node.js function. If you do, Lambda won't be able to run your code when you invoke your function.

  To learn more about the Lambda handler in Node.js, see the section called "Handler".

- The Lambda event object:

  The function `handler` takes two arguments, `event` and `context`. An *event* in Lambda is a JSON formatted document that contains data for your function to process.

  If your function is invoked by another AWS service, the event object contains information about the event that caused the invocation. For example, if your function is invoked when

an object is uploaded to an Amazon Simple Storage Service (Amazon S3) bucket, the event contains the name of the bucket and the object key.

In this example, you'll create an event in the console by entering a JSON formatted document with two key-value pairs.

- The Lambda context object:

  The second argument that your function takes is `context`. Lambda passes the *context object* to your function automatically. The context object contains information about the function invocation and execution environment.

  You can use the context object to output information about your function's invocation for monitoring purposes. In this example, your function uses the `logGroupName` parameter to output the name of its CloudWatch log group.

  To learn more about the Lambda context object in Node.js, see the section called "Context".

- Logging in Lambda:

  With Node.js, you can use console methods like `console.log` and `console.error` to send information to your function's log. The example code uses `console.log` statements to output the calculated area and the name of the function's CloudWatch Logs group. You can also use any logging library that writes to `stdout` or `stderr`.

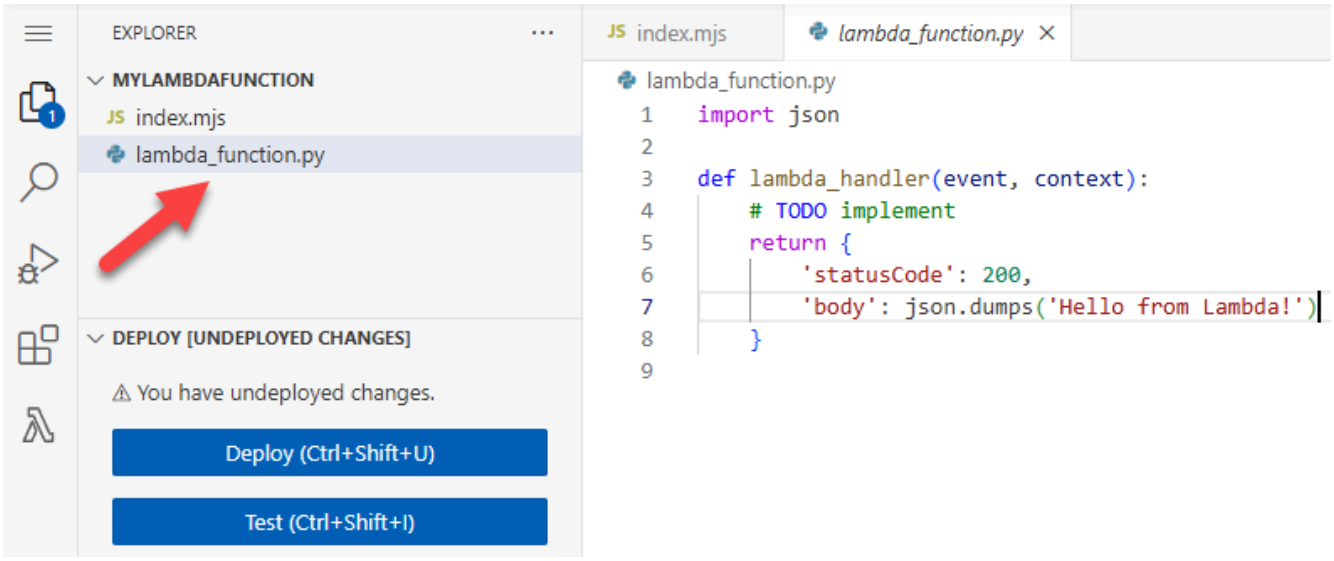
  To learn more, see the section called "Logging". To learn about logging in other runtimes, see the 'Building with' pages for the runtimes you're interested in.

Python

To modify the code in the console

1. Choose the **Code** tab.

   In the console's built-in code editor, you should see the function code that Lambda created. If you don't see the **lambda_function.py** tab in the code editor, select **lambda_function.py** in the file explorer as shown on the following diagram.



2. Paste the following code into the **lambda_function.py** tab, replacing the code that Lambda created.

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length = event['length']
    width = event['width']

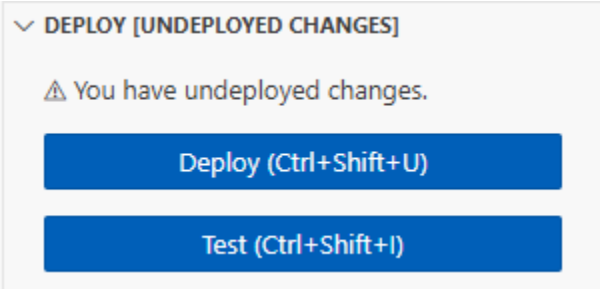
    area = calculate_area(length, width)
    print(f"The area is {area}")

    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width
```

3. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Understanding your function code

Before you move to the next step, let's take a moment to look at the function code and understand some key Lambda concepts.

- The Lambda handler:

  Your Lambda function contains a Python function named `lambda_handler`. A Lambda function in Python can contain more than one Python function, but the *handler* function is always the entry point to your code. When your function is invoked, Lambda runs this method.

  When you created your Hello world function using the console, Lambda automatically set the name of the handler method for your function to `lambda_handler`. Be sure not to edit the name of this Python function. If you do, Lambda won't be able to run your code when you invoke your function.

  To learn more about the Lambda handler in Python, see the section called "Handler".

- The Lambda event object:

  The function `lambda_handler` takes two arguments, `event` and `context`. An *event* in Lambda is a JSON formatted document that contains data for your function to process.

  If your function is invoked by another AWS service, the event object contains information about the event that caused the invocation. For example, if your function is invoked when an object is uploaded to an Amazon Simple Storage Service (Amazon S3) bucket, the event contains the name of the bucket and the object key.

  In this example, you'll create an event in the console by entering a JSON formatted document with two key-value pairs.

- The Lambda context object:

The second argument that your function takes is `context`. Lambda passes the *context object* to your function automatically. The context object contains information about the function invocation and execution environment.

You can use the context object to output information about your function's invocation for monitoring purposes. In this example, your function uses the `log_group_name` parameter to output the name of its CloudWatch log group.

To learn more about the Lambda context object in Python, see the section called "Context".

- Logging in Lambda:

  With Python, you can use either a `print` statement or a Python logging library to send information to your function's log. To illustrate the difference in what's captured, the example code uses both methods. In a production application, we recommend that you use a logging library.

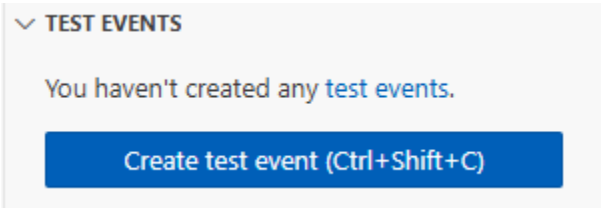
  To learn more, see the section called "Logging". To learn about logging in other runtimes, see the 'Building with' pages for the runtimes you're interested in.

Invoke the Lambda function using the console code editor

To invoke your function using the Lambda console code editor, create a test event to send to your function. The event is a JSON formatted document containing two key-value pairs with the keys `"length"` and `"width"`.

To create the test event

1. In the **TEST EVENTS** section of the console code editor, choose **Create test event**.



2. For **Event Name**, enter **`myTestEvent`**.
3. In the **Event JSON** section, replace the default JSON with the following:

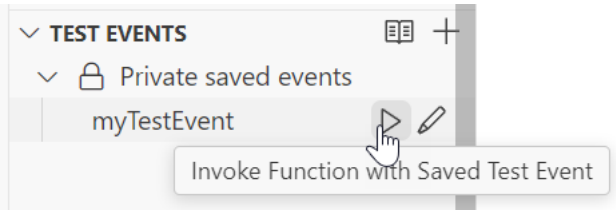
```
{
```

```
  "length": 6,
  "width": 7
}
```

4. Choose **Save**.

To test your function and view invocation records

In the **TEST EVENTS** section of the console code editor, choose the run icon next to your test event:



When your function finishes running, the response and function logs are displayed in the **OUTPUT** tab. You should see results similar to the following:

Node.js

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\":42}"

Function Logs
START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST
2024-08-31T23:39:45.313Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO The area is 42
2024-08-31T23:39:45.331Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO CloudWatch log
 group:  /aws/lambda/myLambdaFunction
END RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a
REPORT RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Duration: 20.67 ms Billed
 Duration: 21 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 163.87 ms

Request ID
5c012b0a-18f7-4805-b2f6-40912935034a
```

Python

```
Status: Succeeded
Test Event Name: myTestEvent
```

```
Response
"{\"area\": 42}"

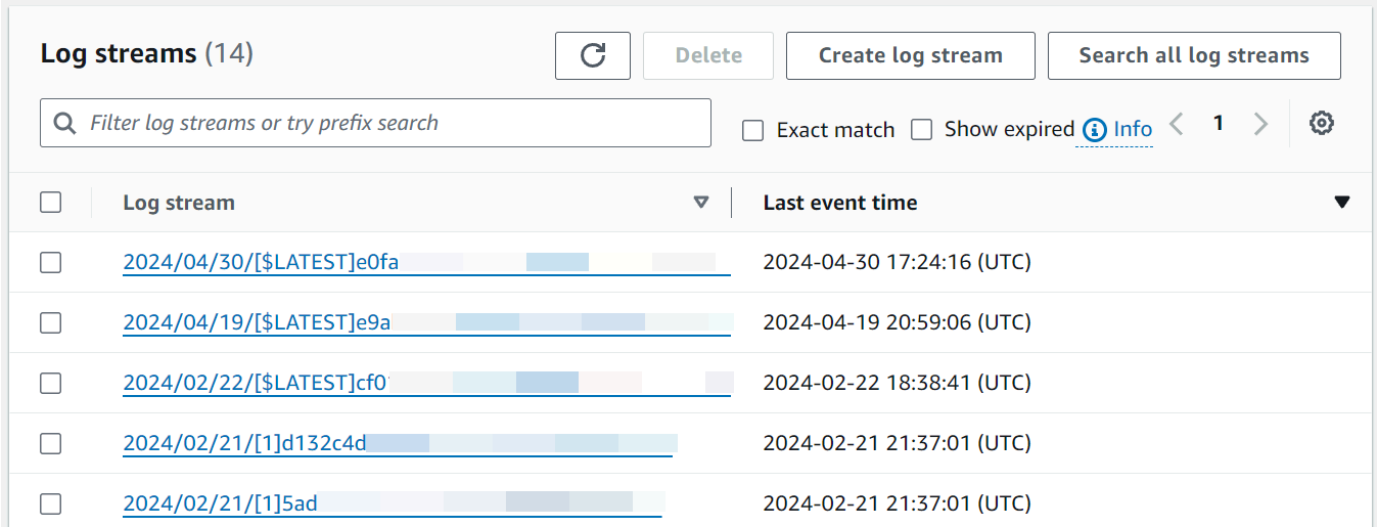
Function Logs
START RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Version: $LATEST
The area is 42
[INFO] 2024-08-31T23:43:26.428Z 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b CloudWatch logs
 group: /aws/lambda/myLambdaFunction
END RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
REPORT RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Duration: 1.42 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 39 MB Init Duration: 123.74 ms

Request ID
2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

When you invoke your function outside of the Lambda console, you must use CloudWatch Logs to view your function's execution results.

To view your function's invocation records in CloudWatch Logs

1. Open the Log groups page of the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/myLambdaFunction`). This is the log group name that your function printed to the console.
3. Scroll down and choose the **Log stream** for the function invocations you want to look at.



You should see output similar to the following:

Node.js

```
INIT_START Runtime Version: nodejs:22.v13    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:e3aaabf6b92ef8755eaae2f4bfdcb7eb8c4536a5e044900570a42bdba7b869d9
START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20 Version: $LATEST
2024-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area
 is 42
2024-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dacd20  INFO
 CloudWatch log group:  /aws/lambda/myLambdaFunction
END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20
REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20    Duration: 17.77 ms
 Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init
 Duration: 178.85 ms
```

Python

```
INIT_START Runtime Version: python:3.13.v16    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3deb1d5acc9aee9e297b072
START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
The area is 42
[INFO] 2024-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
 Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

Clean up

When you're finished working with the example function, delete it. You can also delete the log group that stores the function's logs, and the execution role that the console created.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the log group

1. Open the Log groups page of the CloudWatch console.
2. Select the function's log group (`/aws/lambda/myLambdaFunction`).
3. Choose **Actions**, **Delete log group(s)**.
4. In the **Delete log group(s)** dialog box, choose **Delete**.

To delete the execution role

1. Open the Roles page of the AWS Identity and Access Management (IAM) console.
2. Select the function's execution role (for example, `myLambdaFunction-role-31exxmpl`).
3. Choose **Delete**.
4. In the **Delete role** dialog box, enter the role name, and then choose **Delete**.

Additional resources and next steps

Now that you've created and tested a simple Lambda function using the console, take these next steps:

- Learn to add dependencies to your function and deploy it using a .zip deployment package. Choose your preferred language from the following links.

  Node.js

  the section called "Deploy .zip file archives"

  Typescript

  the section called "Deploy .zip file archives"

  Python

  the section called "Deploy .zip file archives"

  Ruby

  the section called "Deploy .zip file archives"

  Java

  the section called "Deploy .zip file archives"

Go

the section called “Deploy .zip file archives”

C#

the section called “Deployment package”

- To learn how to invoke a Lambda function using another AWS service, see Tutorial: Using an Amazon S3 trigger to invoke a Lambda function.
- Choose one of the following tutorials for more complex examples of using Lambda with other AWS services.
  - Tutorial: Using Lambda with API Gateway: Create an Amazon API Gateway REST API that invokes a Lambda function.
  - Using a Lambda function to access an Amazon RDS database: Use a Lambda function to write data to an Amazon Relational Database Service (Amazon RDS) database through RDS Proxy.
  - Using an Amazon S3 trigger to create thumbnail images: Use a Lambda function to create a thumbnail every time an image file is uploaded to an Amazon S3 bucket.

Getting started with example applications and patterns

The following resources can be used to quickly create and deploy serverless apps that implement some common Lambda uses cases. For each of the example apps, we provide instructions to either create and configure resources manually using the AWS Management Console, or to use the AWS Serverless Application Model to deploy the resources using IaC. Follow the console intructions to learn more about configuring the individual AWS resources for each app, or use to AWS SAM to quickly deploy resources as you would in a production environment.

File Processing

- **PDF Encryption Application**: Create a serverless application that encrypts PDF files when they are uploaded to an Amazon Simple Storage Service bucket and saves them to another bucket, which is useful for securing sensitive documents upon upload.

- **Image Analysis Application**: Create a serverless application that extracts text from images using Amazon Rekognition, which is useful for document processing, content moderation, and automated image analysis.

Database Integration

- **Queue-to-Database Application**: Create a serverless application that writes queue messages to an Amazon RDS database, which is useful for processing user registrations and handling order submissions.

- **Database Event Handler**: Create a serverless application that responds to Amazon DynamoDB table changes, which is useful for audit logging, data replication, and automated workflows.

Scheduled Tasks

- **Database Maintenance Application**: Create a serverless application that automatically deletes entries more than 12 months old from an Amazon DynamoDB table using a cron schedule, which is useful for automated database maintenance and data lifecycle management.

- **Create an EventBridge scheduled rule for Lambda functions**: Use scheduled expressions for rules in EventBridge to trigger a Lambda function on a timed schedule. This format uses cron syntax and can be set with a one-minute granularity.

Long-running Workflows

- **Order Processing Application**: Create a serverless application using Durable Functions that handles complex order fulfillment, including payment processing, inventory checks, and shipping coordination. This example demonstrates how to build workflows that can run for extended periods while maintaining state.

Additional resources

Use the following resources to further explore Lambda and serverless application development:

- **Serverless Land**: a library of ready-to-use patterns for building serverless apps. It helps developers create applications faster using AWS services like Lambda, API Gateway, and EventBridge. The site offers pre-built solutions and best practices, making it easier to develop serverless systems.
- **Lambda sample applications**: Applications that are available in the GitHub repository for this guide. These samples demonstrate the use of various languages and AWS services. Each sample application includes scripts for easy deployment and cleanup and supporting resources.
- **Code examples for Lambda using AWS SDKs**: Examples that show you how to use Lambda with AWS software development kits (SDKs). These examples include basics, actions, scenarios, and AWS community contributions. Examples cover essential operations, individual service functions, and specific tasks using multiple functions or AWS services.

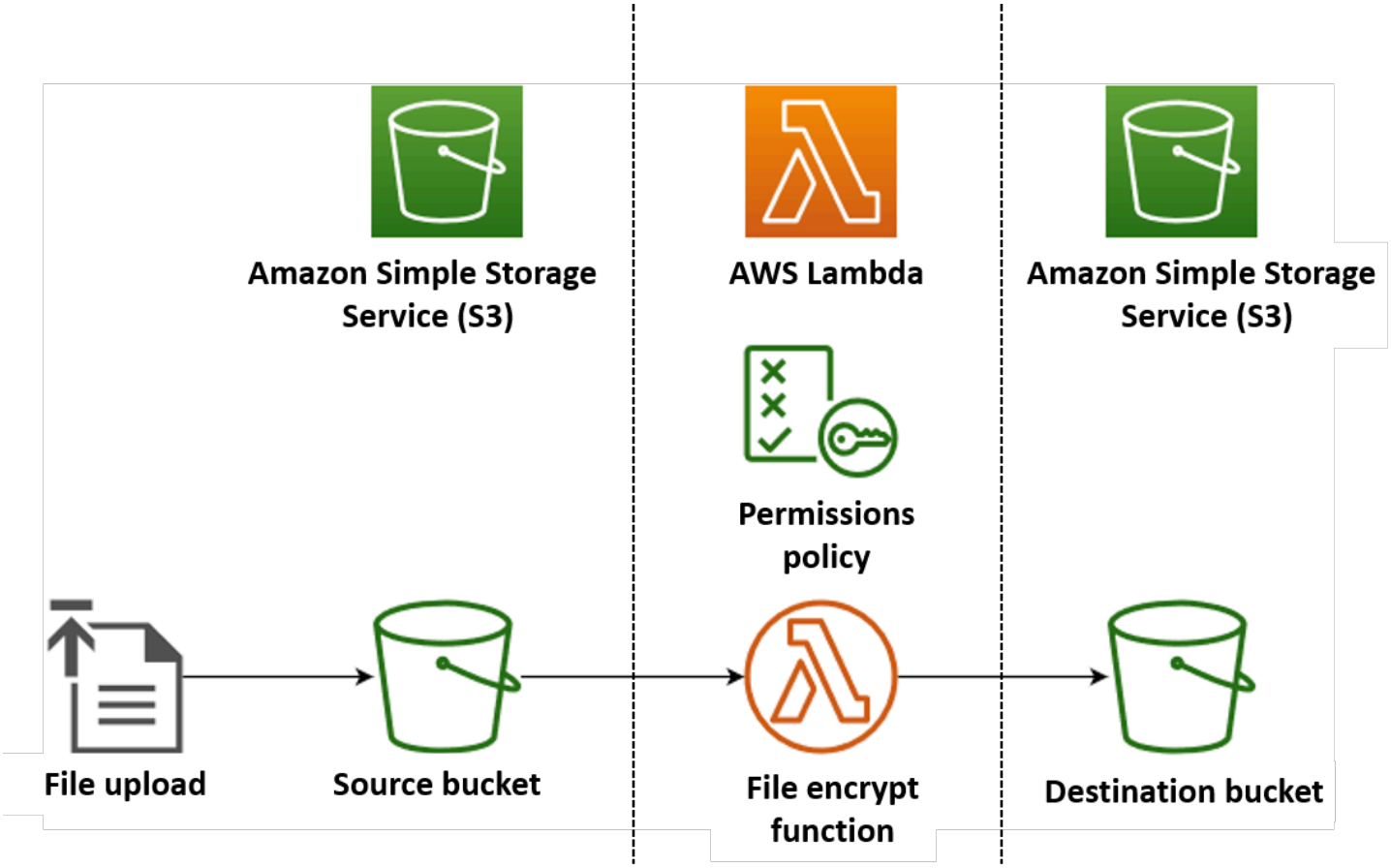
Create a serverless file-processing app

One of the most common use cases for Lambda is to perform file processing tasks. For example, you might use a Lambda function to automatically create PDF files from HTML files or images, or to create thumbnails when a user uploads an image.

In this example, you create an app which automatically encrypts PDF files when they are uploaded to an Amazon Simple Storage Service (Amazon S3) bucket. To implement this app, you create the following resources:

- An S3 bucket for users to upload PDF files to
- A Lambda function in Python which reads the uploaded file and creates an encrypted, password-protected version of it
- A second S3 bucket for Lambda to save the encrypted file in

You also create an AWS Identity and Access Management (IAM) policy to give your Lambda function permission to perform read and write operations on your S3 buckets.



Tip

If you're brand new to Lambda, we recommend that you start with the tutorial *Create your first function* before creating this example app.

You can deploy your app manually by creating and configuring resources with the AWS Management Console or the AWS Command Line Interface (AWS CLI). You can also deploy the app by using the AWS Serverless Application Model (AWS SAM). AWS SAM is an infrastructure as code

(IaC) tool. With IaC, you don't create resources manually, but define them in code and then deploy them automatically.

If you want to learn more about using Lambda with IaC before deploying this example app, see the section called "Infrastructure as code (IaC)".

Create the Lambda function source code files

Create the following files in your project directory:

- `lambda_function.py` - the Python function code for the Lambda function that performs the file encryption
- `requirements.txt` - a manifest file defining the dependencies that your Python function code requires

Expand the following sections to view the code and to learn more about the role of each file. To create the files on your local machine, either copy and paste the code below, or download the files from the aws-lambda-developer-guide GitHub repo.

Python function code

Copy and paste the following code into a file named `lambda_function.py`.

```
from pypdf import PdfReader, PdfWriter
import uuid
import os
from urllib.parse import unquote_plus
import boto3

# Create the S3 client to download and upload objects from S3
s3_client = boto3.client('s3')

def lambda_handler(event, context):
    # Iterate over the S3 event object and get the key for all uploaded files
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = unquote_plus(record['s3']['object']['key']) # Decode the S3 object key to
 remove any URL-encoded characters
        download_path = f'/tmp/{uuid.uuid4()}.pdf' # Create a path in the Lambda tmp
 directory to save the file to
```

```
        upload_path = f'/tmp/converted-{uuid.uuid4()}.pdf' # Create another path to
 save the encrypted file to

        # If the file is a PDF, encrypt it and upload it to the destination S3 bucket
        if key.lower().endswith('.pdf'):
            s3_client.download_file(bucket, key, download_path)
            encrypt_pdf(download_path, upload_path)
            encrypted_key = add_encrypted_suffix(key)
            s3_client.upload_file(upload_path, f'{bucket}-encrypted', encrypted_key)

# Define the function to encrypt the PDF file with a password
def encrypt_pdf(file_path, encrypted_file_path):
    reader = PdfReader(file_path)
    writer = PdfWriter()

    for page in reader.pages:
        writer.add_page(page)

    # Add a password to the new PDF
    writer.encrypt("my-secret-password")

    # Save the new PDF to a file
    with open(encrypted_file_path, "wb") as file:
        writer.write(file)

# Define a function to add a suffix to the original filename after encryption
def add_encrypted_suffix(original_key):
    filename, extension = original_key.rsplit('.', 1)
    return f'{filename}_encrypted.{extension}'
```

Note

In this example code, a password for the encrypted file (`my-secret-password`) is hardcoded into the function code. In a production application, don't include sensitive information like passwords in your function code. Instead, create an AWS Secrets Manager secret and then use the AWS Parameters and Secrets Lambda extension to retrieve your credentials in your Lambda function.

The python function code contains three functions - the handler function that Lambda runs when your function is invoked, and two separate function named `add_encrypted_suffix` and `encrypt_pdf` that the handler calls to perform the PDF encryption.

When your function is invoked by Amazon S3, Lambda passes a JSON formatted *event* argument to the function that contains details about the event that caused the invocation. In this case, the information includes name of the S3 bucket and the object keys for the uploaded files. To learn more about the format of event object for Amazon S3, see the section called "S3".

Your function then uses the AWS SDK for Python (Boto3) to download the PDF files specified in the event object to its local temporary storage directory, before encrypting them using the `pypdf` library.

Finally, the function uses the Boto3 SDK to store the encrypted file in your S3 destination bucket.

`requirements.txt` manifest file

Copy and paste the following code into a file named `requirements.txt`.

```
boto3
pypdf
```

For this example, your function code has only two dependencies that aren't part of the standard Python library - the SDK for Python (Boto3) and the `pypdf` package the function uses to perform the PDF encryption.

> **Note**
>
> A version of the SDK for Python (Boto3) is included as part of the Lambda runtime, so your code would run without adding Boto3 to your function's deployment package. However, to maintain full control of your function's dependencies and avoid possible issues with version misalignment, best practice for Python is to include all function dependencies in your function's deployment package. See the section called "Runtime dependencies in Python" to learn more.

Deploy the app

You can create and deploy the resources for this example app either manually or by using AWS SAM. In a production environment, we recommend that you use an IaC tool like AWS SAM to quickly and repeatably deploy whole serverless applications without using manual processes.

Deploy the resources manually

To deploy your app manually:

- Create source and destination Amazon S3 buckets
- Create a Lambda function that encrypts a PDF file and saves the encrypted version to an S3 bucket
- Configure a Lambda trigger that invokes your function when objects are uploaded to your source bucket

Before you begin, make sure that Python is installed on your build machine.

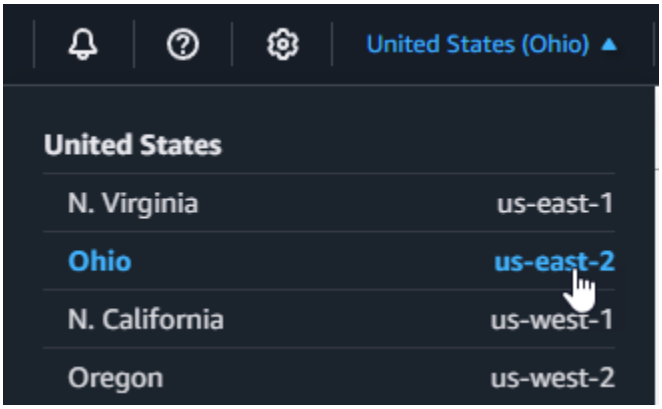
Create two S3 buckets

First create two S3 buckets. The first bucket is the source bucket you will upload your PDF files to. The second bucket is used by Lambda to save the encrypted file when you invoke your function.

Console

To create the S3 buckets (console)

1. Open the General purpose buckets page of the Amazon S3 console.
2. Select the AWS Region closest to your geographical location. You can change your region using the drop-down list at the top of the screen.



3. Choose **Create bucket**.
4. Under **General configuration**, do the following:
   a. For **Bucket type**, ensure **General purpose** is selected.
   b. For **Bucket name**, enter a globally unique name that meets the Amazon S3 bucket naming rules. Bucket names can contain only lower case letters, numbers, dots (.), and hyphens (-).
5. Leave all other options set to their default values and choose **Create bucket**.
6. Repeat steps 1 to 4 to create your destination bucket. For **Bucket name**, enter **`amzn-s3-demo-bucket-encrypted`**, where **`amzn-s3-demo-bucket`** is the name of the source bucket you just created.

AWS CLI

Before you begin, make sure that the AWS CLI is installed on your build machine.

To create the Amazon S3 buckets (AWS CLI)

1. Run the following CLI command to create your source bucket. The name you choose for your bucket must be globally unique and follow the Amazon S3 bucket naming rules. Names can only contain lower case letters, numbers, dots (.), and hyphens (-). For `region` and `LocationConstraint`, choose the AWS Region closest to your geographical location.

```
aws s3api create-bucket --bucket amzn-s3-demo-bucket --region us-east-2 \
--create-bucket-configuration LocationConstraint=us-east-2
```

   Later in the tutorial, you must create your Lambda function in the same AWS Region as your source bucket, so make a note of the region you chose.
2. Run the following command to create your destination bucket. For the bucket name, you must use **`amzn-s3-demo-bucket-encrypted`**, where **`amzn-s3-demo-bucket`** is the name of the source bucket you created in step 1. For `region` and `LocationConstraint`, choose the same AWS Region you used to create your source bucket.

```
aws s3api create-bucket --bucket amzn-s3-demo-bucket-encrypted --region us-
east-2 \
--create-bucket-configuration LocationConstraint=us-east-2
```

Create an execution role

An execution role is an IAM role that grants a Lambda function permission to access AWS services and resources. To give your function read and write access to Amazon S3, you attach the AWS managed policy `AmazonS3FullAccess`.

Console

To create an execution role and attach the `AmazonS3FullAccess` managed policy (console)

1. Open the Roles page in the IAM console.
2. Choose **Create role**.
3. For **Trusted entity type**, select **AWS service**, and for **Use case**, select **Lambda**.
4. Choose **Next**.
5. Add the `AmazonS3FullAccess` managed policy by doing the following:
   a. In **Permissions policies**, enter **`AmazonS3FullAccess`** into the search bar.
   b. Select the checkbox next to the policy.
   c. Choose **Next**.
6. In **Role details**, for **Role name** enter **`LambdaS3Role`**.
7. Choose **Create Role**.

AWS CLI

To create an execution role and attach the `AmazonS3FullAccess` managed policy (AWS CLI)

1. Save the following JSON in a file named `trust-policy.json`. This trust policy allows Lambda to use the role's permissions by giving the service principal `lambda.amazonaws.com` permission to call the AWS Security Token Service (AWS STS) `AssumeRole` action.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
```

```
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. From the directory you saved the JSON trust policy document in, run the following CLI command to create the execution role.

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. To attach the `AmazonS3FullAccess` managed policy, run the following CLI command.

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::aws:policy/AmazonS3FullAccess
```

Create the function deployment package

To create your function, you create a *deployment package* containing your function code and its dependencies. For this application, your function code uses a separate library for the PDF encryption.

To create the deployment package

1. Navigate to the project directory containing the `lambda_function.py` and `requirements.txt` files you created or downloaded from GitHub earlier and create a new directory named `package`.
2. Install the dependencies specified in the `requirements.txt` file in your `package` directory by running the following command.

```
pip install -r requirements.txt --target ./package/
```

3. Create a .zip file containing your application code and its dependencies. In Linux or MacOS, run the following commands from your command line interface.

```
cd package
zip -r ../lambda_function.zip .
cd ..
```

```
zip lambda_function.zip lambda_function.py
```

In Windows, use your preferred zip tool to create the `lambda_function.zip` file. Make sure that your `lambda_function.py` file and the folders containing your dependencies are all at the root of the .zip file.

You can also create your deployment package using a Python virtual environment. See Working with .zip file archives for Python Lambda functions

Create the Lambda function

You now use the deployment package you created in the previous step to deploy your Lambda function.

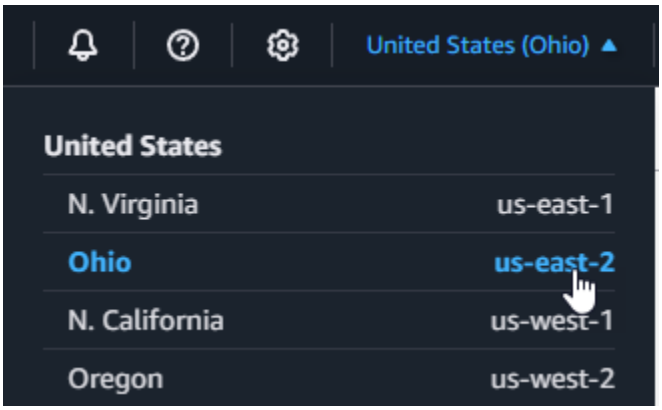
Console

To create the function (console)

To create your Lambda function using the console, you first create a basic function containing some 'Hello world' code. You then replace this code with your own function code by uploading the.zip file you created in the previous step.

To ensure that your function doesn't time out when encrypting large PDF files, you configure the function's memory and timeout settings. You also set the function's log format to JSON. Configuring JSON formatted logs is necessary when using the provided test script so it can read the function's invocation status from CloudWatch Logs to confirm successful invocation.

1. Open the Functions page of the Lambda console.
2. Make sure you're working in the same AWS Region you created your S3 bucket in. You can change your region using the drop-down list at the top of the screen.



3. Choose **Create function**.
4. Choose **Author from scratch**.
5. Under **Basic information**, do the following:
    a. For **Function name**, enter **`EncryptPDF`**.
    b. For **Runtime** choose **Python 3.12**.
    c. For **Architecture**, choose **x86_64**.
6. Attach the execution role you created in the previous step by doing the following:
    a. Expand the **Change default execution role** section.
    b. Select **Use an existing role**.
    c. Under **Existing role**, select your role (`LambdaS3Role`).
7. Choose **Create function**.

To upload the function code (console)

1. In the **Code source** pane, choose **Upload from**.
2. Choose **.zip file**.
3. Choose **Upload**.
4. In the file selector, select your .zip file and choose **Open**.
5. Choose **Save**.

To configure the function memory and timeout (console)

1. Select the **Configuration** tab for your function.
2. In the **General configuration** pane, choose **Edit**.
3. Set **Memory** to 256 MB and **Timeout** to 15 seconds.
4. Choose **Save**.

To configure the log format (console)

1. Select the **Configuration** tab for your function.
2. Select **Monitoring and operations tools**.
3. In the **Logging configuration** pane, choose **Edit**.

4. For **Logging configuration**, select **JSON**.
5. Choose **Save**.

AWS CLI

To create the function (AWS CLI)

- Run the following command from the directory containing your `lambda_function.zip` file.For the `region` parameter, replace `us-east-2` with the region you created your S3 buckets in.

```
aws lambda create-function --function-name EncryptPDF \
--zip-file fileb://lambda_function.zip --handler lambda_function.lambda_handler
 \
--runtime python3.12 --timeout 15 --memory-size 256 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-2 \
--logging-config LogFormat=JSON
```

Configure an Amazon S3 trigger to invoke the function

For your Lambda function to run when you upload a file to your source bucket, you need to configure a trigger for your function. You can configure the Amazon S3 trigger using either the console or the AWS CLI.

> **Important**
>
> This procedure configures the S3 bucket to invoke your function every time that an object is created in the bucket. Be sure to configure this only on the source bucket. If your Lambda function creates objects in the same bucket that invokes it, your function can be invoked continuously in a loop. This can result in un expected charges being billed to your AWS account.

Console

To configure the Amazon S3 trigger (console)

1. Open the Functions page of the Lambda console and choose your function (`EncryptPDF`).

2. Choose **Add trigger**.
3. Select **S3**.
4. Under **Bucket**, select your source bucket.
5. Under **Event types**, select **All object create events**.
6. Under **Recursive invocation**, select the check box to acknowledge that using the same S3 bucket for input and output is not recommended. You can learn more about recursive invocation patterns in Lambda by reading Recursive patterns that cause run-away Lambda functions in Serverless Land.
7. Choose **Add**.

   When you create a trigger using the Lambda console, Lambda automatically creates a resource based policy to give the service you select permission to invoke your function.

AWS CLI

To configure the Amazon S3 trigger (AWS CLI)

1. Add a resource based policy to your function that allows your Amazon S3 source bucket to invoke your function when you add a file. A resource-based policy statement gives other AWS services permission to invoke your function. To give Amazon S3 permission to invoke your function, run the following CLI command. Be sure to replace the `source-account` parameter with your own AWS account ID and to use your own source bucket name.

   ```
   aws lambda add-permission --function-name EncryptPDF \
   --principal s3.amazonaws.com --statement-id s3invoke --action
    "lambda:InvokeFunction" \
   --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
   --source-account 123456789012
   ```

   The policy you define with this command allows Amazon S3 to invoke your function only when an action takes place on your source bucket.

   > **Note**
   >
   > Although S3 bucket names are globally unique, when using resource-based policies it is best practice to specify that the bucket must belong to your account. This is

because if you delete a bucket, it is possible for another AWS account to create a bucket with the same Amazon Resource Name (ARN).

2. Save the following JSON in a file named `notification.json`. When applied to your source bucket, this JSON configures the bucket to send a notification to your Lambda function every time a new object is added. Replace the AWS account number and AWS Region in the Lambda function ARN with your own account number and region.

```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "EncryptPDFEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-2:123456789012:function:EncryptPDF",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. Run the following CLI command to apply the notification settings in the JSON file you created to your source bucket. Replace `amzn-s3-demo-bucket` with the name of your own source bucket.

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-bucket \
--notification-configuration file://notification.json
```

To learn more about the `put-bucket-notification-configuration` command and the `notification-configuration` option, see put-bucket-notification-configuration in the *AWS CLI Command Reference*.

Deploy the resources using AWS SAM

Before you begin, make sure that Docker and the latest version of the AWS SAM CLI are installed on your build machine.

1. In your project directory, copy and paste the following code into a file named `template.yaml`. Replace the placeholder bucket names:

- For the source bucket, replace `amzn-s3-demo-bucket` with any name that complies with the S3 bucket naming rules.
- For the destination bucket, replace `amzn-s3-demo-bucket-encrypted` with `<source-bucket-name>-encrypted`, where `<source-bucket>` is the name you chose for your source bucket.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  EncryptPDFFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: EncryptPDF
      Architectures: [x86_64]
      CodeUri: ./
      Handler: lambda_function.lambda_handler
      Runtime: python3.12
      Timeout: 15
      MemorySize: 256
      LoggingConfig:
        LogFormat: JSON
      Policies:
        - AmazonS3FullAccess
      Events:
        S3Event:
          Type: S3
          Properties:
            Bucket: !Ref PDFSourceBucket
            Events: s3:ObjectCreated:*

  PDFSourceBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: amzn-s3-demo-bucket

  EncryptedPDFBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: amzn-s3-demo-bucket-encrypted
```

The AWS SAM template defines the resources you create for your app. In this example, the template defines a Lambda function using the `AWS::Serverless::Function` type and two S3 buckets using the `AWS::S3::Bucket` type. The bucket names specified in the template are placeholders. Before you deploy the app using AWS SAM, you need to edit the template to rename the buckets with globally unique names that meet the S3 bucket naming rules. This step is explained further in the section called "Deploy the resources using AWS SAM".

The definition of the Lambda function resource configures a trigger for the function using the `S3Event` event property. This trigger causes your function to be invoked whenever an object is created in your source bucket.

The function definition also specifies an AWS Identity and Access Management (IAM) policy to be attached to the function's execution role. The AWS managed policy `AmazonS3FullAccess` gives your function the permissions it needs to read and write objects to Amazon S3.

2. Run the following command from the directory in which you saved your `template.yaml`, `lambda_function.py`, and `requirements.txt`files.

```
sam build --use-container
```

This command gathers the build artifacts for your application and places them in the proper format and location to deploy them. Specifying the `--use-container` option builds your function inside a Lambda-like Docker container. We use it here so you don't need to have Python 3.12 installed on your local machine for the build to work.

During the build process, AWS SAM looks for the Lambda function code in the location you specified with the `CodeUri` property in the template. In this case, we specified the current directory as the location (`./`).

If a `requirements.txt` file is present, AWS SAM uses it to gather the specified dependencies. By default, AWS SAM creates a .zip deployment package with your function code and dependencies. You can also choose to deploy your function as a container image using the PackageType property.

3. To deploy your application and create the Lambda and Amazon S3 resources specified in your AWS SAM template, run the following command.

```
sam deploy --guided
```

Using the `--guided` flag means that AWS SAM will show you prompts to guide you through the deployment process. For this deployment, accept the default options by pressing Enter.

During the deployment process, AWS SAM creates the following resources in your AWS account:

- An CloudFormation stack named `sam-app`
- A Lambda function with the name `EncryptPDF`
- Two S3 buckets with the names you chose when you edited the `template.yaml` AWS SAM template file
- An IAM execution role for your function with the name format `sam-app-EncryptPDFFunctionRole-`*`2qGaapHFW0Q8`*

When AWS SAM finishes creating your resources, you should see the following message:

```
Successfully created/updated stack - sam-app in us-east-2
```

Test the app

To test your app, upload a PDF file to your source bucket, and confirm that Lambda creates an encrypted version of the file in your destination bucket. In this example, you can either test this manually using the console or the AWS CLI, or by using the provided test script.

For production applications, you can use traditional test methods and techniques, such as unit testing, to confirm the correct functioning of your Lambda function code. Best practice is also to conduct tests like those in the provided test script which perform integration testing with real, cloud-based resources. Integration testing in the cloud confirms that your infrastructure has been correctly deployed and that events flow between different services as expected. To learn more, see *Testing serverless functions*.

Testing the app manually

You can test your function manually by adding a PDF file to your Amazon S3 source bucket. When you add your file to the source bucket, your Lambda function should be automatically invoked and should store an encrypted version of the file in your target bucket.

Console

To test your app by uploading a file (console)

1. To upload a PDF file to your S3 bucket, do the following:
   a. Open the Buckets page of the Amazon S3 console and choose your source bucket.
   b. Choose **Upload**.
   c. Choose **Add files** and use the file selector to choose the PDF file you want to upload.
   d. Choose **Open**, then choose **Upload**.
2. Verify that Lambda has saved an encrypted version of your PDF file in your target bucket by doing the following:
   a. Navigate back to the Buckets page of the Amazon S3 console and choose your destination bucket.
   b. In the **Objects** pane, you should now see a file with name format `filename_encrypted.pdf` (where `filename.pdf` was the name of the file you uploaded to your source bucket). To download your encrypted PDF, select the file, then choose **Download**.
   c. Confirm that you can open the downloaded file with the password your Lambda function protected it with (`my-secret-password`).

AWS CLI

To test your app by uploading a file (AWS CLI)

1. From the directory containing the PDF file you want to upload, run the following CLI command. Replace the `--bucket` parameter with the name of your source bucket. For the `--key` and `--body` parameters, use the filename of your test file.

```
aws s3api put-object --bucket amzn-s3-demo-bucket --key test.pdf --body ./
test.pdf
```

2. Verify that your function has created an encrypted version of your file and saved it to your target S3 bucket. Run the following CLI command, replacing `amzn-s3-demo-bucket-encrypted` with the name of your own destination bucket.

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-bucket-encrypted
```

If your function runs successfully, you'll see output similar to the following. Your target bucket should contain a file with the name format *<your_test_file>*_encrypted.pdf, where <your_test_file> is the name of the file you uploaded.

```
{
    "Contents": [
        {
            "Key": "test_encrypted.pdf",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        }
    ]
}
```

3. To download the file that Lambda saved in your destination bucket, run the following CLI command. Replace the --bucket parameter with the name of your destination bucket. For the --key parameter, use the filename *<your_test_file>*_encrypted.pdf, where <your_test_file> is the name of the the test file you uploaded.

```
aws s3api get-object --bucket amzn-s3-demo-bucket-encrypted --
key test_encrypted.pdf my_encrypted_file.pdf
```

This command downloads the file to your current directory and saves it as my_encrypted_file.pdf.

4. Confirm the you can open the downloaded file with the password your Lambda function protected it with (my-secret-password).

Testing the app with the automated script

Create the following files in your project directory:

- test_pdf_encrypt.py - a test script you can use to automatically test your application
- pytest.ini - a configuration file for the the test script

Expand the following sections to view the code and to learn more about the role of each file.

Automated test script

Copy and paste the following code into a file named `test_pdf_encrypt.py`. Be sure to replace the placeholder bucket names:

- In the `test_source_bucket_available` function, replace `amzn-s3-demo-bucket` with the name of your source bucket.
- In the `test_encrypted_file_in_bucket` function, replace `amzn-s3-demo-bucket-encrypted` with *`source-bucket`*`-encrypted`, where `source-bucket>` is the name of your source bucket.
- In the `cleanup` function, replace `amzn-s3-demo-bucket` with the name of your source bucket, and replace `amzn-s3-demo-bucket-encrypted` with the name of your destination bucket.

```
import boto3
import json
import pytest
import time
import os

@pytest.fixture
def lambda_client():
    return boto3.client('lambda')

@pytest.fixture
def s3_client():
    return boto3.client('s3')

@pytest.fixture
def logs_client():
    return boto3.client('logs')

@pytest.fixture(scope='session')
def cleanup():
    # Create a new S3 client for cleanup
    s3_client = boto3.client('s3')

    yield
    # Cleanup code will be executed after all tests have finished
```

```
    # Delete test.pdf from the source bucket
    source_bucket = 'amzn-s3-demo-bucket'
    source_file_key = 'test.pdf'
    s3_client.delete_object(Bucket=source_bucket, Key=source_file_key)
    print(f"\nDeleted {source_file_key} from {source_bucket}")

    # Delete test_encrypted.pdf from the destination bucket
    destination_bucket = 'amzn-s3-demo-bucket-encrypted'
    destination_file_key = 'test_encrypted.pdf'
    s3_client.delete_object(Bucket=destination_bucket, Key=destination_file_key)
    print(f"Deleted {destination_file_key} from {destination_bucket}")


@pytest.mark.order(1)
def test_source_bucket_available(s3_client):
    s3_bucket_name = 'amzn-s3-demo-bucket'
    file_name = 'test.pdf'
    file_path = os.path.join(os.path.dirname(__file__), file_name)

    file_uploaded = False
    try:
        s3_client.upload_file(file_path, s3_bucket_name, file_name)
        file_uploaded = True
    except:
        print("Error: couldn't upload file")

    assert file_uploaded, "Could not upload file to S3 bucket"



@pytest.mark.order(2)
def test_lambda_invoked(logs_client):

    # Wait for a few seconds to make sure the logs are available
    time.sleep(5)

    # Get the latest log stream for the specified log group
    log_streams = logs_client.describe_log_streams(
        logGroupName='/aws/lambda/EncryptPDF',
        orderBy='LastEventTime',
        descending=True,
        limit=1
    )
```

```
    latest_log_stream_name = log_streams['logStreams'][0]['logStreamName']

    # Retrieve the log events from the latest log stream
    log_events = logs_client.get_log_events(
        logGroupName='/aws/lambda/EncryptPDF',
        logStreamName=latest_log_stream_name
    )

    success_found = False
    for event in log_events['events']:
        message = json.loads(event['message'])
        status = message.get('record', {}).get('status')
        if status == 'success':
            success_found = True
            break

    assert success_found, "Lambda function execution did not report 'success' status in
 logs."

@pytest.mark.order(3)
def test_encrypted_file_in_bucket(s3_client):
    # Specify the destination S3 bucket and the expected converted file key
    destination_bucket = 'amzn-s3-demo-bucket-encrypted'
    converted_file_key = 'test_encrypted.pdf'

    try:
        # Attempt to retrieve the metadata of the converted file from the destination
 S3 bucket
        s3_client.head_object(Bucket=destination_bucket, Key=converted_file_key)
    except s3_client.exceptions.ClientError as e:
        # If the file is not found, the test will fail
        pytest.fail(f"Converted file '{converted_file_key}' not found in the
 destination bucket: {str(e)}")

def test_cleanup(cleanup):
    # This test uses the cleanup fixture and will be executed last
    pass
```

The automated test script executes three test functions to confirm correct operation of your app:

- The test `test_source_bucket_available` confirms that your source bucket has been successfully created by uploading a test PDF file to the bucket.

- The test `test_lambda_invoked` interrogates the latest CloudWatch Logs log stream for your function to confirm that when you uploaded the test file, your Lambda function ran and reported success.
- The test `test_encrypted_file_in_bucket` confirms that your destination bucket contains the encrypted `test_encrypted.pdf` file.

After all these tests have run, the script runs an additional cleanup step to delete the `test.pdf` and `test_encrypted.pdf` files from both your source and destination buckets.

As with the AWS SAM template, the bucket names specified in this file are placeholders. Before running the test, you need to edit this file with your app's real bucket names. This step is explained further in the section called “Testing the app with the automated script”

Test script configuration file

Copy and paste the following code into a file named `pytest.ini`.

```
[pytest]
markers =
    order: specify test execution order
```

This is needed to specify the order in which the tests in the `test_pdf_encrypt.py` script run.

To run the tests do the following:

1. Ensure that the `pytest` module is installed in your local environment. You can install `pytest` by running the following command:

   ```
   pip install pytest
   ```

2. Save a PDF file named `test.pdf` in the directory containing the `test_pdf_encrypt.py` and `pytest.ini` files.
3. Open a terminal or shell program and run the following command from the directory containing the test files.

   ```
   pytest -s -v
   ```

   When the test completes, you should see output like the following:

```
============================================================= test session starts
 ==============================================================
platform linux -- Python 3.12.2, pytest-7.2.2, pluggy-1.0.0 -- /usr/bin/python3
cachedir: .pytest_cache
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/
pdf_encrypt_app/.hypothesis/examples')
Test order randomisation NOT enabled. Enable with --random-order or --random-order-
bucket=<bucket_type>
rootdir: /home/pdf_encrypt_app, configfile: pytest.ini
plugins: anyio-3.7.1, hypothesis-6.70.0, localserver-0.7.1, random-order-1.1.0
collected 4 items

test_pdf_encrypt.py::test_source_bucket_available PASSED
test_pdf_encrypt.py::test_lambda_invoked PASSED
test_pdf_encrypt.py::test_encrypted_file_in_bucket PASSED
test_pdf_encrypt.py::test_cleanup PASSED
Deleted test.pdf from amzn-s3-demo-bucket
Deleted test_encrypted.pdf from amzn-s3-demo-bucket-encrypted


============================================================== 4 passed in 7.32s
 ==============================================================
```

Next steps

Now you've created this example app, you can use the provided code as a basis to create other types of file-processing application. Modify the code in the `lambda_function.py` file to implement the file-processing logic for your use case.

Many typical file-processing use cases involve image processing. When using Python, the most popular image-processing libraries like pillow typically contain C or C++ components. In order to ensure that your function's deployment package is compatible with the Lambda execution environment, it's important to use the correct source distribution binary.

When deploying your resources with AWS SAM, you need to take some extra steps to include the right source distribution in your deployment package. Because AWS SAM won't install dependencies for a different platform than your build machine, specifying the correct source distribution (`.whl` file) in your `requirements.txt` file won't work if your build machine uses an

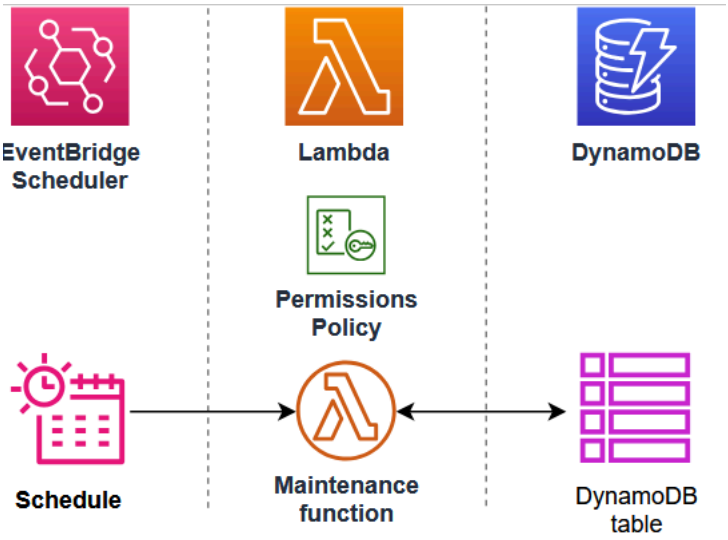
operating system or architecture that's different from the Lambda execution environment. Instead, you should do one of the following:

- Use the `--use-container` option when running `sam build`. When you specify this option, AWS SAM downloads a container base image that's compatible with the Lambda execution environment and builds your function's deployment package in a Docker container using that image. To learn more, see Building a Lambda function inside of a provided container.
- Build your function's .zip deployment package yourself using the correct source distribution binary and save the .zip file in the directory you specify as the `CodeUri` in the AWS SAM template. To learn more about building .zip deployment packages for Python using binary distributions, see the section called “Creating a .zip deployment package with dependencies” and the section called “Creating .zip deployment packages with native libraries”.

Create an app to perform scheduled database maintenance

You can use AWS Lambda to replace scheduled processes such as automated system backups, file conversions, and maintenance tasks. In this example, you create a serverless application that performs regular scheduled maintenance on a DynamoDB table by deleting old entries. The app uses EventBridge Scheduler to invoke a Lambda function on a cron schedule. When invoked, the function queries the table for items older than one year, and deletes them. The function logs each deleted item in CloudWatch Logs.

To implement this example, first create a DynamoDB table and populate it with some test data for your function to query. Then, create a Python Lambda function with an EventBridge Scheduler trigger and an IAM execution role that gives the function permission to read, and delete, items from your table.



> **Tip**
>
> If you're new to Lambda, we recommend that you complete the tutorial *Create your first function* before creating this example app.

You can deploy your app manually by creating and configuring resources with the AWS Management Console. You can also deploy the app by using the AWS Serverless Application Model (AWS SAM). AWS SAM is an infrastructure as code (IaC) tool. With IaC, you don't create resources manually, but define them in code and then deploy them automatically.

If you want to learn more about using Lambda with IaC before deploying this example app, see the section called "Infrastructure as code (IaC)".

Prerequisites

Before you can create the example app, make sure you have the required command line tools and programs installed.

- **Python**

  To populate the DynamoDB table you create to test your app, this example uses a Python script and a CSV file to write data into the table. Make sure you have Python version 3.8 or later installed on your machine.

- **AWS SAM CLI**

If you want to create the DynamoDB table and deploy the example app using AWS SAM, you need to install the AWS SAM CLI. Follow the installation instructions in the *AWS SAM User Guide*.

- **AWS CLI**

  To use the provided Python script to populate your test table, you need to have installed and configured the AWS CLI. This is because the script uses the AWS SDK for Python (Boto3), which needs access to your AWS Identity and Access Management (IAM) credentials. You also need the AWS CLI installed to deploy resources using AWS SAM. Install the CLI by following the installation instructions in the *AWS Command Line Interface User Guide*.

- **Docker**

  To deploy the app using AWS SAM, Docker must also be installed on your build machine. Follow the instructions in Install Docker Engine on the Docker documentation website.

Downloading the example app files

To create the example database and the scheduled-maintenance app, you need to create the following files in your project directory:

Example database files

- `template.yaml` - an AWS SAM template you can use to create the DynamoDB table
- `sample_data.csv` - a CSV file containing sample data to load into your table
- `load_sample_data.py` - a Python script that writes the data in the CSV file into the table

Scheduled-maintenance app files

- `lambda_function.py` - the Python function code for the Lambda function that performs the database maintenance
- `requirements.txt` - a manifest file defining the dependencies that your Python function code requires
- `template.yaml` - an AWS SAM template you can use to deploy the app

Test file

- `test_app.py` - a Python script that scans the table and confirms successful operation of your function by outputting all records older than one year

Expand the following sections to view the code and to learn more about the role of each file in creating and testing your app. To create the files on your local machine, copy and paste the code below.

AWS SAM template (example DynamoDB table)

Copy and paste the following code into a file named `template.yaml`.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for DynamoDB Table with Order_number as Partition Key and
 Date as Sort Key

Resources:
  MyDynamoDBTable:
    Type: AWS::DynamoDB::Table
    DeletionPolicy: Retain
    UpdateReplacePolicy: Retain
    Properties:
      TableName: MyOrderTable
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: Order_number
          AttributeType: S
        - AttributeName: Date
          AttributeType: S
      KeySchema:
        - AttributeName: Order_number
          KeyType: HASH
        - AttributeName: Date
          KeyType: RANGE
      SSESpecification:
        SSEEnabled: true
      GlobalSecondaryIndexes:
        - IndexName: Date-index
          KeySchema:
            - AttributeName: Date
              KeyType: HASH
          Projection:
```

```
            ProjectionType: ALL
      PointInTimeRecoverySpecification:
        PointInTimeRecoveryEnabled: true

Outputs:
  TableName:
    Description: DynamoDB Table Name
    Value: !Ref MyDynamoDBTable
  TableArn:
    Description: DynamoDB Table ARN
    Value: !GetAtt MyDynamoDBTable.Arn
```

Note

AWS SAM templates use a standard naming convention of `template.yaml`. In this example, you have two template files - one to create the example database and another to create the app itself. Save them in separate sub-directories in your project folder.

This AWS SAM template defines the DynamoDB table resource you create to test your app. The table uses a primary key of `Order_number` with a sort key of `Date`. In order for your Lambda function to find items directly by date, we also define a Global Secondary Index named `Date-index`.

To learn more about creating and configuring a DynamoDB table using the `AWS::DynamoDB::Table` resource, see AWS::DynamoDB::Table in the *AWS CloudFormation User Guide*.

Sample database data file

Copy and paste the following code into a file named `sample_data.csv`.

```
Date,Order_number,CustomerName,ProductID,Quantity,TotalAmount
2023-09-01,ORD001,Alejandro Rosalez,PROD123,2,199.98
2023-09-01,ORD002,Akua Mansa,PROD456,1,49.99
2023-09-02,ORD003,Ana Carolina Silva,PROD789,3,149.97
2023-09-03,ORD004,Arnav Desai,PROD123,1,99.99
2023-10-01,ORD005,Carlos Salazar,PROD456,2,99.98
2023-10-02,ORD006,Diego Ramirez,PROD789,1,49.99
2023-10-03,ORD007,Efua Owusu,PROD123,4,399.96
2023-10-04,ORD008,John Stiles,PROD456,2,99.98
2023-10-05,ORD009,Jorge Souza,PROD789,3,149.97
```

```
2023-10-06,ORD010,Kwaku Mensah,PROD123,1,99.99
2023-11-01,ORD011,Li Juan,PROD456,5,249.95
2023-11-02,ORD012,Marcia Oliveria,PROD789,2,99.98
2023-11-03,ORD013,Maria Garcia,PROD123,3,299.97
2023-11-04,ORD014,Martha Rivera,PROD456,1,49.99
2023-11-05,ORD015,Mary Major,PROD789,4,199.96
2023-12-01,ORD016,Mateo Jackson,PROD123,2,199.99
2023-12-02,ORD017,Nikki Wolf,PROD456,3,149.97
2023-12-03,ORD018,Pat Candella,PROD789,1,49.99
2023-12-04,ORD019,Paulo Santos,PROD123,5,499.95
2023-12-05,ORD020,Richard Roe,PROD456,2,99.98
2024-01-01,ORD021,Saanvi Sarkar,PROD789,3,149.97
2024-01-02,ORD022,Shirley Rodriguez,PROD123,1,99.99
2024-01-03,ORD023,Sofia Martinez,PROD456,4,199.96
2024-01-04,ORD024,Terry Whitlock,PROD789,2,99.98
2024-01-05,ORD025,Wang Xiulan,PROD123,3,299.97
```

This file contains some example test data to populate your DynamoDB table with in a standard comma-separated values (CSV) format.

Python script to load sample data

Copy and paste the following code into a file named `load_sample_data.py`.

```
import boto3
import csv
from decimal import Decimal

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('MyOrderTable')
print("DDB client initialized.")

def load_data_from_csv(filename):
    with open(filename, 'r') as file:
        csv_reader = csv.DictReader(file)
        for row in csv_reader:
            item = {
                'Order_number': row['Order_number'],
                'Date': row['Date'],
                'CustomerName': row['CustomerName'],
                'ProductID': row['ProductID'],
                'Quantity': int(row['Quantity']),
```

```
                'TotalAmount': Decimal(str(row['TotalAmount']))
            }
            table.put_item(Item=item)
            print(f"Added item: {item['Order_number']} - {item['Date']}")

if __name__ == "__main__":
    load_data_from_csv('sample_data.csv')
    print("Data loading completed.")
```

This Python script first uses the AWS SDK for Python (Boto3) to create a connection to your DynamoDB table. It then iterates over each row in the example-data CSV file, creates an item from that row, and writes the item to the DynamoDB table using the boto3 SDK.

Python function code

Copy and paste the following code into a file named `lambda_function.py`.

```
import boto3
from datetime import datetime, timedelta
from boto3.dynamodb.conditions import Key, Attr
import logging

logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    # Initialize the DynamoDB client
    dynamodb = boto3.resource('dynamodb')

    # Specify the table name
    table_name = 'MyOrderTable'
    table = dynamodb.Table(table_name)

    # Get today's date
    today = datetime.now()

    # Calculate the date one year ago
    one_year_ago = (today - timedelta(days=365)).strftime('%Y-%m-%d')

    # Scan the table using a global secondary index
    response = table.scan(
        IndexName='Date-index',
        FilterExpression='#date < :one_year_ago',
```

```
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago
        }
    )

     # Delete old items
    with table.batch_writer() as batch:
        for item in response['Items']:
            Order_number = item['Order_number']
            batch.delete_item(
                Key={
                    'Order_number': Order_number,
                    'Date': item['Date']
                }
            )
            logger.info(f'deleted order number {Order_number}')

    # Check if there are more items to scan
    while 'LastEvaluatedKey' in response:
        response = table.scan(
            IndexName='DateIndex',
            FilterExpression='#date < :one_year_ago',
            ExpressionAttributeNames={
                '#date': 'Date'
            },
            ExpressionAttributeValues={
                ':one_year_ago': one_year_ago
            },
            ExclusiveStartKey=response['LastEvaluatedKey']
        )

        # Delete old items
        with table.batch_writer() as batch:
            for item in response['Items']:
                batch.delete_item(
                    Key={
                        'Order_number': item['Order_number'],
                        'Date': item['Date']
                    }
                )
```

```
    return {
        'statusCode': 200,
        'body': 'Cleanup completed successfully'
    }
```

The Python function code contains the handler function (`lambda_handler`) that Lambda runs when your function is invoked.

When the function is invoked by EventBridge Scheduler, it uses the AWS SDK for Python (Boto3) to create a connection to the DynamoDB table on which the scheduled maintenance task is to be performed. It then uses the Python `datetime` library to calculate the date one year ago, before scanning the table for items older than this and deleting them.

Note that responses from DynamoDB query and scan operations are limited to a maximum of 1 MB in size. If the response is larger than 1 MB, DynamoDB paginates the data and returns a `LastEvaluatedKey` element in the response. To ensure that our function processes all the records in the table, we check for the presence of this key and continue performing table scans from the last evaluated position until the whole table has been scanned.

`requirements.txt` manifest file

Copy and paste the following code into a file named `requirements.txt`.

```
boto3
```

For this example, your function code has only one dependency that isn't part of the standard Python library - the SDK for Python (Boto3) that the function uses to scan and delete items from the DynamoDB table.

> **Note**
>
> A version of the SDK for Python (Boto3) is included as part of the Lambda runtime, so your code would run without adding Boto3 to your function's deployment package. However, to maintain full control of your function's dependencies and avoid possible issues with version misalignment, best practice for Python is to include all function dependencies in your function's deployment package. See the section called “Runtime dependencies in Python” to learn more.

AWS SAM template (scheduled-maintenance app)

Copy and paste the following code into a file named `template.yaml`.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for Lambda function and EventBridge Scheduler rule

Resources:
  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: ScheduledDBMaintenance
      CodeUri: ./
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      Events:
        ScheduleEvent:
          Type: ScheduleV2
          Properties:
            ScheduleExpression: cron(0 3 1 * ? *)
            Description: Run on the first day of every month at 03:00 AM
      Policies:
        - CloudWatchLogsFullAccess
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:Scan
                - dynamodb:BatchWriteItem
              Resource: !Sub 'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/
MyOrderTable'

  LambdaLogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub /aws/lambda/${MyLambdaFunction}
      RetentionInDays: 30

Outputs:
  LambdaFunctionName:
    Description: Lambda Function Name
    Value: !Ref MyLambdaFunction
```

```
  LambdaFunctionArn:
    Description: Lambda Function ARN
    Value: !GetAtt MyLambdaFunction.Arn
```

> **Note**
>
> AWS SAM templates use a standard naming convention of `template.yaml`. In this example, you have two template files - one to create the example database and another to create the app itself. Save them in separate sub-directories in your project folder.

This AWS SAM template defines the resources for your app. We define the Lambda function using the `AWS::Serverless::Function` resource. The EventBridge Scheduler schedule and the trigger to invoke the Lambda function are created by using the `Events` property of this resource using a type of `ScheduleV2`. To learn more about defining EventBridge Scheduler schedules in AWS SAM templates, see ScheduleV2 in the *AWS Serverless Application Model Developer Guide*.

In addition to the Lambda function and the EventBridge Scheduler schedule, we also define a CloudWatch log group for your function to send records of deleted items to.

Test script

Copy and paste the following code into a file named `test_app.py`.

```
import boto3
from datetime import datetime, timedelta
import json

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')

# Specify your table name
table_name = 'YourTableName'
table = dynamodb.Table(table_name)

# Get the current date
current_date = datetime.now()

# Calculate the date one year ago
one_year_ago = current_date - timedelta(days=365)
```

```
# Convert the date to string format (assuming the date in DynamoDB is stored as a
 string)
one_year_ago_str = one_year_ago.strftime('%Y-%m-%d')

# Scan the table
response = table.scan(
    FilterExpression='#date < :one_year_ago',
    ExpressionAttributeNames={
        '#date': 'Date'
    },
    ExpressionAttributeValues={
        ':one_year_ago': one_year_ago_str
    }
)

# Process the results
old_records = response['Items']

# Continue scanning if we have more items (pagination)
while 'LastEvaluatedKey' in response:
    response = table.scan(
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago_str
        },
        ExclusiveStartKey=response['LastEvaluatedKey']
    )
    old_records.extend(response['Items'])

for record in old_records:
    print(json.dumps(record))

# The total number of old records should be zero.
print(f"Total number of old records: {len(old_records)}")
```

This test script uses the AWS SDK for Python (Boto3) to create a connection to your DynamoDB table and scan for items older than one year. To confirm if the Lambda function has run successfully, at the end of the test, the function prints the number of records older than one year still in the table. If the Lambda function was successful, the number of old records in the table should be zero.

Creating and populating the example DynamoDB table

To test your scheduled-maintenance app, you first create a DynamoDB table and populate it with some sample data. You can create the table either manually using the AWS Management Console or by using AWS SAM. We recommend that you use AWS SAM to quickly create and configure the table using a few AWS CLI commands.

Console

To create the DynamoDB table

1. Open the Tables page of the DynamoDB console.
2. Choose **Create table**.
3. Create the table by doing the following:
   a. Under **Table details**, for **Table name**, enter **`MyOrderTable`**.
   b. For **Partition key**, enter **`Order_number`** and leave the type as **String**.
   c. For **Sort key**, enter **`Date`** and leave the type as **String**.
   d. Leave **Table settings** set to **Default settings** and choose **Create table**.
4. When your table has finished creating and its **Status** shows as **Active**, create a global secondary index (GSI) by doing the following. Your app will use this GSI to search for items directly by date to determine what to delete.
   a. Choose **MyOrderTable** from the list of tables.
   b. Choose the **Indexes** tab.
   c. Under **Global secondary indexes**, choose **Create index**.
   d. Under **Index details**, enter **`Date`** for the **Partition key** and leave the **Data type** set to **String**.
   e. For **Index name**, enter **`Date-index`**.
   f. Leave all other parameters set to their default values, scroll to the bottom of the page, and choose **Create index**.

AWS SAM

To create the DynamoDB table

1. Navigate to the folder you saved the `template.yaml` file for the DynamoDB table in. Note that this example uses two `template.yaml` files. Make sure they are saved in separate sub-folders and that you are in the correct folder containing the template to create your DynamoDB table.
2. Run the following command.

   ```
   sam build
   ```

   This command gathers the build artifacts for the resources you want to deploy and places them in the proper format and location to deploy them.
3. To create the DynamoDB resource specified in the `template.yaml` file, run the following command.

   ```
   sam deploy --guided
   ```

   Using the `--guided` flag means that AWS SAM will show you prompts to guide you through the deployment process. For this deployment, enter a `Stack name` of **`cron-app-test-db`**, and accept the defaults for all other options by using Enter.

   When AWS SAM has finished creating the DynamoDB resource, you should see the following message.

   ```
   Successfully created/updated stack - cron-app-test-db in us-west-2
   ```
4. You can additionally confirm that the DynamoDB table has been created by opening the Tables page of the DynamoDB console. You should see a table named `MyOrderTable`.

After you've created your table, you next add some sample data to test your app. The CSV file `sample_data.csv` you downloaded earlier contains a number of example entries comprised of order numbers, dates, and customer and order information. Use the provided python script `load_sample_data.py` to add this data to your table.

To add the sample data to the table

1. Navigate to the directory containing the `sample_data.csv` and `load_sample_data.py` files. If these files are in separate directories, move them so they're saved in the same location.
2. Create a Python virtual environment to run the script in by running the following command. We recommend that you use a virtual environment because in a following step you'll need to install the AWS SDK for Python (Boto3).

   ```
   python -m venv venv
   ```

3. Activate the virtual environment by running the following command.

   ```
   source venv/bin/activate
   ```

4. Install the SDK for Python (Boto3) in your virtual environment by running the following command. The script uses this library to connect to your DynamoDB table and add the items.

   ```
   pip install boto3
   ```

5. Run the script to populate the table by running the following command.

   ```
   python load_sample_data.py
   ```

   If the script runs successfully, it should print each item to the console as it loads it and report `Data loading completed`.

6. Deactivate the virtual environment by running the following command.

   ```
   deactivate
   ```

7. You can verify that the data has been loaded to your DynamoDB table by doing the following:

   a. Open the Explore items page of the DynamoDB console and select your table (`MyOrderTable`).

   b. In the **Items returned** pane, you should see the 25 items from the CSV file that the script added to the table.

Creating the scheduled-maintenance app

You can create and deploy the resources for this example app step by step using the AWS Management Console or by using AWS SAM. In a production environment, we recommend that you use an Infrastructure-as-Code (IaC) tool like AWS SAM to repeatably deploy serverless applications without using manual processes.

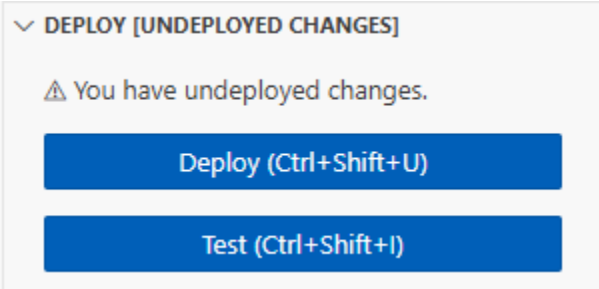
For this example, follow the console instructions to learn how to configure each AWS resource separately, or follow the AWS SAM instructions to quickly deploy the app using AWS CLI commands.

Console

To create the function using the AWS Management Console

First, create a function containing basic starter code. You then replace this code with your own function code by either copying and pasting the code directly in the Lambda code editor, or by uploading your code as a `.zip` package. For this task, we recommend copying and pasting the code.

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch**.
4. Under **Basic information**, do the following:
    a. For **Function name**, enter **`ScheduledDBMaintenance`**.
    b. For **Runtime** choose the latest Python version.
    c. For **Architecture**, choose **x86_64**.
5. Choose **Create function**.
6. After your function is created, you can configure your function with the provided function code.
    a. In the **Code source** pane, replace the Hello world code that Lambda created with the Python function code from the `lambda_function.py` file that you saved earlier.
    b. In the **DEPLOY** section, choose **Deploy** to update your function's code:



To configure the function memory and timeout (console)

1. Select the **Configuration** tab for your function.
2. In the **General configuration** pane, choose **Edit**.
3. Set **Memory** to 256 MB and **Timeout** to 15 seconds. If you are processing a large table with many records, for example in the case of a production environment, you might consider setting **Timeout** to a larger number. This gives your function more time to scan, and clean the database.
4. Choose **Save**.

To configure the log format (console)

You can configure Lambda functions to output logs in either unstructured text or JSON format. We recommend that you use JSON format for logs to make it easier to search and filter log data. To learn more about Lambda log configuration options, see the section called "Configuring advanced logging controls for Lambda functions".

1. Select the **Configuration** tab for your function.
2. Select **Monitoring and operations tools**.
3. In the **Logging configuration** pane, choose **Edit**.
4. For **Logging configuration**, select **JSON**.
5. Choose **Save**.

To set Up IAM permissions

To give your function the permissions it needs to read and delete DynamoDB items, you need to add a policy to your function's execution role defining the necessary permissions.

1. Open the **Configuration** tab, then choose **Permissions** from the left navigation bar.
2. Choose the role name under **Execution role**.
3. In the IAM console, choose **Add permissions**, then **Create inline policy**.
4. Use the JSON editor and enter the following policy:

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:Scan",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/MyOrderTable"
        }
    ]
}
```

5. Name the policy **`DynamoDBCleanupPolicy`**, then create it.

To set up EventBridge Scheduler as a trigger (console)

1. Open the EventBridge console.
2. In the left navigation pane, choose **Schedulers** under the **Scheduler** section.
3. Choose **Create schedule**.
4. Configure the schedule by doing the following:
   a. Under **Schedule name**, enter a name for your schedule (for example, **`DynamoDBCleanupSchedule`**).
   b. Under **Schedule pattern**, choose **Recurring schedule**.
   c. For **Schedule type** leave the default as **Cron-based schedule**, then enter the following schedule details:
      - **Minutes**: **`0`**
      - **Hours**: **`3`**

- **Day of month**: **1**
- **Month**: *****
- **Day of the week**: **?**
- **Year**: *****

When evaluated, this cron expression runs on the first day of every month at 03:00 AM.

d. For **Flexible time window**, select **Off**.

5. Choose **Next**.
6. Configure the trigger for your Lambda function by doing the following:

a. In the **Target detail** pane, leave **Target API** set to **Templated targets**, then select **AWS Lambda Invoke**.

b. Under **Invoke**, select your Lambda function (`ScheduledDBMaintenance`) from the dropdown list.

c. Leave the **Payload** empty and choose **Next**.

d. Scroll down to **Permissions** and select **Create a new role for this schedule**. When you create a new EventBridge Scheduler schedule using the console, EventBridge Scheduler creates a new policy with the required permissions the schedule needs to invoke your function. For more information about managing your schedule permissions, see Cron-based schedules. in the *EventBridge Scheduler User Guide*.

e. Choose **Next**.

7. Review your settings and choose **Create schedule** to complete creation of the schedule and Lambda trigger.

AWS SAM

To deploy the app using AWS SAM

1. Navigate to the folder you saved the `template.yaml` file for the app in. Note that this example uses two `template.yaml` files. Make sure they are saved in separate sub-folders and that you are in the correct folder containing the template to create the app.
2. Copy the `lambda_function.py` and `requirements.txt` files you downloaded earlier to the same folder. The code location specified in the AWS SAM template is `./`, meaning

the current location. AWS SAM will search in this folder for the Lambda function code when you try to deploy the app.

3. Run the following command.

```
sam build --use-container
```

This command gathers the build artifacts for the resources you want to deploy and places them in the proper format and location to deploy them. Specifying the `--use-container` option builds your function inside a Lambda-like Docker container. We use it here so you don't need to have Python 3.12 installed on your local machine for the build to work.

4. To create the Lambda and EventBridge Scheduler resources specified in the `template.yaml` file, run the following command.

```
sam deploy --guided
```

Using the `--guided` flag means that AWS SAM will show you prompts to guide you through the deployment process. For this deployment, enter a `Stack name` of **`cron-maintenance-app`**, and accept the defaults for all other options by using Enter.

When AWS SAM has finished creating the Lambda and EventBridge Scheduler resources, you should see the following message.

```
Successfully created/updated stack - cron-maintenance-app in us-west-2
```

5. You can additionally confirm that the Lambda function has been created by opening the Functions page of the Lambda console. You should see a function named `ScheduledDBMaintenance`.

Testing the app

To test that your schedule correctly triggers your function, and that your function correctly cleans records from the database, you can temporarily modify your schedule to run once at a specific time. You can then run `sam deploy` again to reset your recurrence schedule to run once a month.

To run the application using the AWS Management Console

1. Navigate back to the EventBridge Scheduler console page.

2. Choose your schedule, then choose **Edit**.
3. In the **Schedule pattern** section, under **Recurrence**, choose **One-time schedule**.
4. Set your invocation time to a few minutes from now, review your settings, then choose **Save**.

After the schedule runs and invokes its target, you run the `test_app.py` script to verify that your function successfully removed all old records from the DynamoDB table.

To verify that old records are deleted using a Python script

1. In your command line, navigate to the folder where you saved `test_app.py`.
2. Run the script.

```
python test_app.py
```

If successful, you will see the following output.

```
Total number of old records: 0
```

Next steps

You can now modify the EventBridge Scheduler schedule to meet your particular application requirements. EventBridge Scheduler supports the following schedule expressions: cron, rate, and one-time schedules.

For more information about EventBridge Scheduler schedule expressions, see Schedule types in the *EventBridge Scheduler User Guide*. Access Management in the *IAM User Guide*

Creating an Order Processing System with Lambda Durable Functions

> **Note**
>
> NEED: Add architecture diagram showing API Gateway, Durable Function workflow, and supporting services (DynamoDB, EventBridge)

Prerequisites

- AWS CLI installed and configured
- NEED: Specific Durable Functions requirements

Create the Source Code Files

Create the following files in your project directory:

- `lambda_function.py` - the function code
- `requirements.txt` - dependencies manifest

Function Code

```
# NEED: Verify correct imports
import boto3
import json

def lambda_handler(event, context):
    # NEED: Verify DurableContext syntax
    durable = context.durable

    try:
        # Validate and store order
        order = await durable.step('validate', async () => {
            return validate_order(event['order'])
        })

        # Process payment
        # NEED: Verify wait syntax
        await durable.wait(/* wait configuration */)

        # Additional steps
        # NEED: Complete implementation

    except Exception as e:
        # NEED: Error handling patterns
        raise e

def validate_order(order_data):
```

```
    # NEED: Implementation
    pass
```

Requirements File

```
# NEED: List of required packages
```

Deploy the App

Create a DynamoDB Table for Orders

1. Open the DynamoDB console at https://console.aws.amazon.com/dynamodb/
2. Choose **Create table**
3. For **Table name**, enter **`Orders`**
4. For **Partition key**, enter **`orderId`**
5. Leave other settings as default
6. Choose **Create table**

Create the Lambda Function

1. Open the Lambda console at https://console.aws.amazon.com/lambda/
2. Choose **Create function**
3. Select **Author from scratch**
4. For **Function name**, enter **`ProcessOrder`**
5. For **Runtime**, choose your preferred runtime
6. NEED: Add Durable Functions-specific configuration
7. Choose **Create function**

Create the API Gateway Endpoint

1. Open the API Gateway console at https://console.aws.amazon.com/apigateway/
2. Choose **Create API**
3. Select **HTTP API**
4. Choose **Build**

5. Add an integration with your Lambda function
6. Configure routes for order processing
7. Deploy the API

Test the App

Submit a test order:

```
{
    "orderId": "12345",
    "items": [
        {
            "productId": "ABC123",
            "quantity": 1
        }
    ]
}
```

NEED: Add specific monitoring instructions for Durable Functions

Next Steps

Add Business Logic

Implement inventory management:

```
async def check_inventory(order):
    # Add inventory check logic
    pass
```

Add price calculations:

```
async def calculate_total(order):
    # Add pricing logic
    pass
```

Improve Error Handling

Add compensation logic:

```
async def reverse_payment(order):
    # Add payment reversal logic
    pass
```

Handle order cancellations:

```
async def cancel_order(order):
    # Add cancellation logic
    pass
```

Integrate External Systems

```
async def notify_shipping_provider(order):
    # Add shipping integration
    pass

async def send_customer_notification(order):
    # Add notification logic
    pass
```

Enhance Monitoring

- Create CloudWatch dashboards
- Set up metrics for order processing times
- Configure alerts for delayed orders

Development tools for Lambda

You have access to a variety of tools that increase productivity and ease-of-use throughout the entire development lifecycle. This section provides information on tools that help many Lambda customers design, develop, and manage their applications. From local development in your IDE to deploying and managing complex serverless applications, these tools help you streamline your workflow, improve code quality, and accelerate the development of robust Lambda-based solutions.

- **Local development**—Write and test Lambda functions faster in your preferred development environment. The AWS Toolkit for VS Code enables local function development, debugging, and testing with direct deployment capabilities to Lambda.
- **Infrastructure as Code (IaC)**—Deploy and manage serverless applications consistently from local testing to production. AWS SAM, AWS CDK, and CloudFormation let you define and manage your serverless infrastructure through code for consistent, version-controlled deployments.
- **GitHub Actions**—Automate Lambda deployments directly from your code repository. GitHub Actions allows you to set up workflows that automatically deploy your Lambda functions whenever you push code or configuration changes, simplifying your CI/CD pipeline.
- **Powertools for AWS Lambda**—Build production-ready serverless applications with less custom code. Powertools for AWS Lambda (also referred to as Powertools for AWS) is an open-source developer toolkit that simplifies implementing serverless best practices such as observability, parameter retrieval, and idempotency across Python, TypeScript, Java, and .NET.
- **Workflows and events**—Coordinate Lambda functions with AWS services, APIs, and external systems. Lambda provides two orchestration options: Lambda durable functions for application-centric orchestration using standard programming languages within Lambda, and AWS Step Functions for workflow-centric orchestration with visual design across multiple services. Amazon EventBridge provides event management capabilities for event-driven architectures. For help choosing an orchestration approach, see Durable functions or Step Functions.

Local development tools

Local development environments enable you to work offline and leverage advanced IDE features while iterating quickly on your Lambda functions. These tools help you debug complex functions and develop in environments with limited connectivity. They also support team collaboration and integration with version control systems.

For more information on developing Lambda functions locally, see Developing Lambda functions locally with VS Code. This page describes how to move Lambda function development from the AWS console to Visual Studio Code, which provides a rich development environment with features like debugging and code completion. To make the transition, you need to set up the AWS Toolkit for Visual Studio Code and credentials, after which you can use advanced features in VS Code while maintaining the ability to deploy directly to AWS.

Local development for Lambda provides several key capabilities:

- Use Visual Studio Code integration with the Lambda console
- Configure local Lambda development environments
- Debug and test functions locally
- Apply best practices for local function management

For more information, see Developing Lambda functions locally with VS Code.

Infrastructure as Code (IaC) tools

With Infrastructure as Code (IaC) tools, you can define and manage your serverless architecture using code. This approach helps maintain consistency across environments, lets you control your infrastructure versions, and facilitates DevOps practices. IaC is especially valuable for automating deployments, ensuring consistent environments, and managing multi-region deployments.

Key IaC tools and concepts for Lambda include frameworks for template creation, deployment management, and best practices for serverless infrastructure:

- Core IaC principles for Lambda development
- CloudFormation, AWS SAM, and AWS CDK capabilities
- Tool selection criteria and comparison
- Best practices for Lambda IaC implementation

Whether you're working independently on a small project or as part of a large team managing enterprise-scale serverless applications, these development and deployment tools can help you write, deploy, and manage your Lambda functions more effectively.

For more information, see Using Lambda with infrastructure as code (IaC).

GitHub Actions tools

GitHub Actions provides automated deployment capabilities for your Lambda functions directly from your code repository. By creating workflow files in your repository, you can automatically deploy Lambda functions whenever code or configuration changes are pushed, streamlining your continuous integration and continuous deployment (CI/CD) pipeline. The Deploy Lambda Function action offers a declarative YAML interface that simplifies the deployment process, handles AWS credentials through OpenID Connect (OIDC), and supports various deployment scenarios including code updates, configuration changes, and dry run validations. This integration enables teams to maintain a consistent and automated deployment process while leveraging their existing GitHub workflows.

For more information, see Using GitHub Actions to deploy Lambda functions.

Powertools for AWS Lambda

Powertools for AWS is an open-source developer toolkit that helps you implement serverless best practices with minimal custom code. Available for Python, TypeScript/Node.js, Java, and .NET, it provides utility functions, decorators, and middleware that streamline common Lambda development tasks. The toolkit includes built-in observability features like structured logging, tracing, and metrics collection, such as utilities for parameter retrieval, secrets management, and idempotency patterns. These tools align with AWS well-architected best practices and help developers build production-ready serverless applications more efficiently. By reducing boilerplate code and standardizing common patterns, Powertools for AWS enables teams to focus on business logic while maintaining consistent implementation of serverless best practices across their applications.

For more information, see Powertools for AWS Lambda.

Workflow and event management tools

Lambda applications can be used in orchestration of complex workflows and handling of various events. AWS provides specialized tools to help you manage these aspects of serverless development. Learn about AWS Step Functions for workflow orchestration and Amazon EventBridge for event management, and how to integrate them with your Lambda functions. These tools can significantly enhance the scalability and reliability of your serverless applications by providing robust state management and event-driven architectures. By leveraging these

services, you can build more sophisticated and resilient Lambda-based solutions that can handle complex business processes and react to a wide range of system and application events.

For more information, see Managing Lambda workflows and events.

Developing Lambda functions locally with VS Code

You can move your Lambda functions from the Lambda console to Visual Studio Code, which provides a full development environment and allows you to use other local development options like AWS SAM and AWS CDK.

Key benefits of local development

While the Lambda console provides a quick way to edit and test functions, local development offers more advanced capabilities:

- **Advanced IDE features**: Debugging, code completion, and refactoring tools
- **Offline development**: Work and test changes locally before cloud deployment
- **Infrastructure as code integration**: Seamless use with AWS SAM, AWS CDK, and Infrastructure Composer
- **Dependency management**: Full control over function dependencies

Prerequisites

Before developing Lambda functions locally in VS Code, you must have:

- **VS Code**: For installation instructions, see Download VS Code.
- **AWS Toolkit for Visual Studio Code**: For installation instructions, see Setting up the AWS Toolkit for Visual Studio Code. For an overview, see AWS Toolkit for Visual Studio Code.
- **AWS credentials**: For information about configuring credentials, see Setting up your AWS credentials.
- **AWS SAM CLI**: For installation instructions, see Installing the AWS SAM CLI.
- **Docker installed (optional, but required for local testing)**: For installation instructions, see Get Docker.

> **Note**
>
> If you already have an AWS account and profile configured locally, ensure that the AdministratorAccess managed policy is added to your configured AWS profile.

Authentication and access control

To develop Lambda functions locally, you need AWS credentials to securely access and manage AWS resources on your behalf, just like they would in the cloud. The AWS Toolkit for VS Code supports the following authentication methods:

The AWS Toolkit for VS Code supports the following authentication methods:

- IAM user long-term credentials
- Temporary credentials from assumed roles
- Identity federation
- AWS account root user credentials (not recommended)

This section guides you through obtaining and configuring these credentials using IAM user long-term credentials.

Get IAM Credentials

If you already have an IAM user with access keys, have both the access key ID and secret access key ready for the next section. If you don't have these keys, follow these steps to create them:

> **Note**
>
> You must use both the access key ID and secret access key together to authenticate your requests.

To create an IAM user and access keys:

1. Open the IAM console at https://console.aws.amazon.com/iam/
2. In the navigation pane, choose **Users**.

3. Choose **Create user**.
4. For **User name**, enter a name and choose **Next**.
5. Under **Set permissions**, choose **Attach policies directly**.
6. Select **AdministratorAccess** and choose **Next**.
7. Choose **Create user**.
8. In the success banner, choose **View user**.
9. Choose **Create access key**.
10. For **Use case**, select **Local code**.
11. Select the confirmation check box and choose **Next**.
12. (Optional) Enter a description tag value.
13. Choose **Create access key**.
14. Copy your access key and secret access key immediately. **You won't be able to access the secret access key again after you leave this page.**

> **Important**
>
> Never share your secret key or commit it to source control. Store these keys securely and delete them when no longer needed.

> **Note**
>
> For more information, see Create an IAM user in your AWS account and Manage access keys for IAM users in the *IAM User Guide*.

Configure AWS credentials using the AWS Toolkit

The following table summarizes the credential setup process you will complete in the following procedure.

What to Do	Why?
Open Sign In panel	Start authentication

What to Do	Why?
Use Command Palette, search for AWS Add a New Connection	Access the sign-in UI
Choose IAM Credential	Use your access keys for programmatic access
Enter profile name, access key, secret key	Provide credentials for connection
See AWS Explorer update	Confirm you're connected

Complete the following steps authenticate to your AWS account:

1. Open the Sign In panel in VS Code:
   a. To start the authentication process, select the AWS icon in the left navigation pane or open the Command Palette (Cmd+Shift+P on Mac or Ctrl+Shift+P on Windows/Linux) and search for and select **AWS Add a New Connection**.
2. In the sign in panel, choose **IAM Credentials** and select **Continue**.

> **Note**
>
> To proceed, you will need to allow AWS IDE Extensions for VS Code to access your data.

3. Enter your profile name, access key ID, and secret access key, then select **Continue**.
4. Verify the connection by checking the AWS Explorer in VS Code for your AWS services and resources.

For information on setting up authentication with long-term credentials, see Using long-term credentials to authenticate AWS SDKs and tools.

For information about configuring authentication, see AWS IAM credentials in the AWS Toolkit for Visual Studio Code User Guide.

Moving from console to local development

> **Note**
>
> If you've made changes in the console, make sure you don't have any undeployed changes before transitioning to local development.

To move a Lambda function from the Lambda console to VS Code, complete the following steps:

1. Open the Lambda console.
2. Choose the name of your function.
3. Select the **Code source** tab.
4. Choose **Open in Visual Studio Code**.

   > **Note**
   >
   > The **Open in Visual Studio Code** button is only available in AWS Toolkit version **3.69.0** and later. If you have an earlier version of the AWS Toolkit installed, you may see a `Cannot open the handler` message in VS Code. To resolve this, update your AWS Toolkit to the latest version.

5. When prompted, allow your browser to open VS Code.

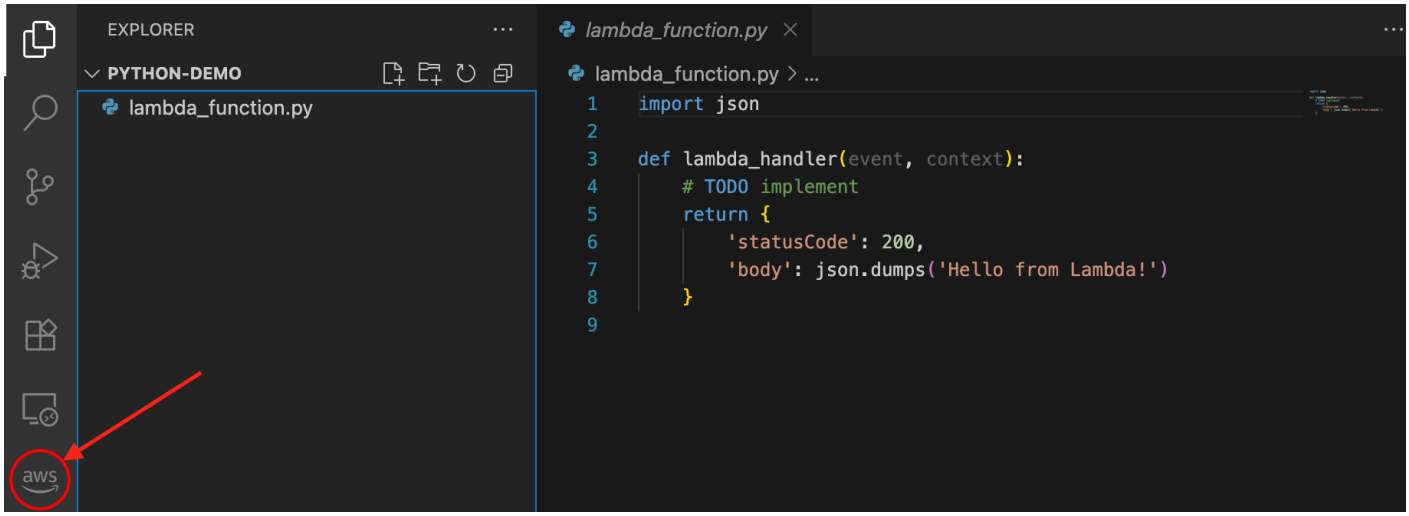
When you open your function in VS Code, Lambda creates a local project with your function code in a temporary location that's designed for quick testing and deployment. This includes the function code, dependencies, and a basic project structure that you can use for local development.

For details on using AWS in VS Code, see the *AWS Toolkit for Visual Studio Code User Guide*.

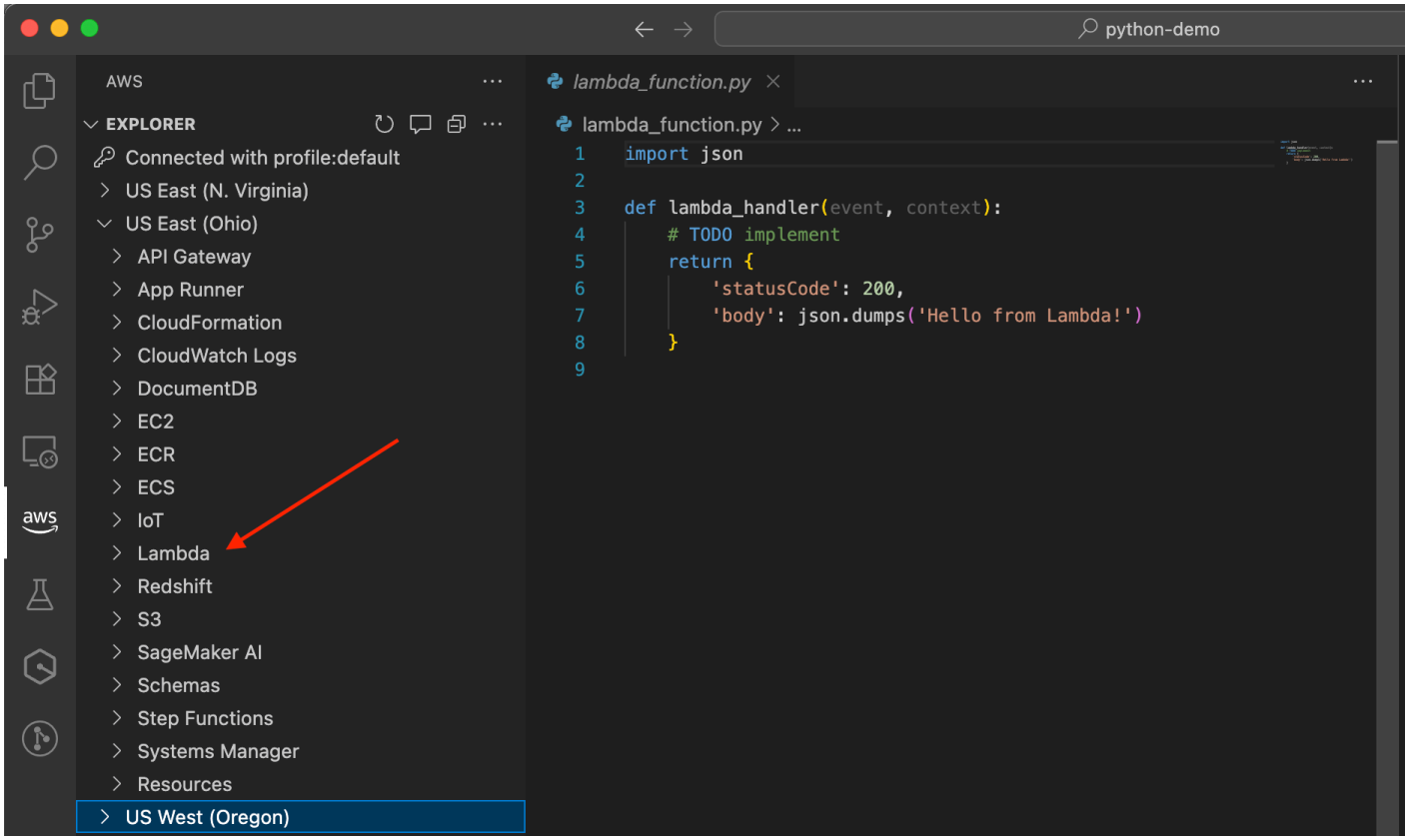
Working with functions locally

After opening your function in VS Code, follow these steps to access and manage your functions:

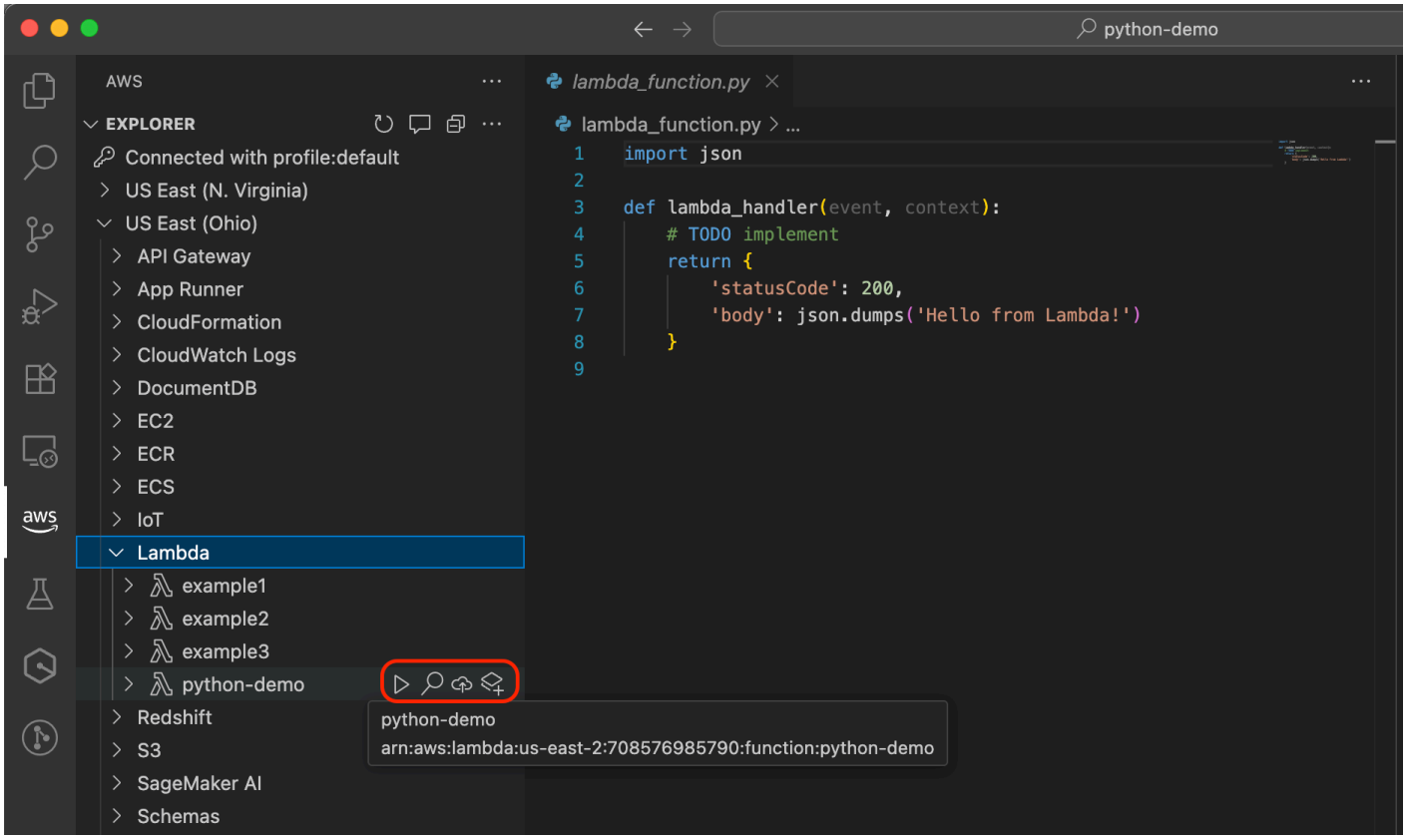
1. Select the AWS icon in the sidebar to open the AWS Explorer:



2. In the AWS Explorer, select the region with your Lambda function:



3. Under your selected region, expand the Lambda section to view and manage your functions:



With your function opened in VS Code, you can:

- Edit function code with full language support and code completion.
- Use the LocalStack integration in VS Code to test Lambda functions that make API calls to other AWS services during execution, such as reading from DynamoDB tables or writing to Amazon S3 buckets. LocalStack is a cloud service emulator that provides a complete local development environment for testing service integrations. You can also use AWS SAM CLI to test your function in a local container. If your function makes API calls to other AWS services those calls will reach real AWS resources, not emulated ones.
- Debug your function with breakpoints and variable inspection. For more information, see Running and debugging Lambda functions directly from code in the *AWS Toolkit for Visual Studio Code User Guide*.
- Deploy your updated function back to AWS using the cloud icon.
- Install and manage dependencies for your function.

For more information, see Working with AWS Lambda functions in the AWS Toolkit for Visual Studio Code User Guide.

Convert your function to an AWS SAM template and use IaC tools

In VS Code, you can convert your Lambda function to an AWS SAM template by choosing the **Convert to AWS SAM Application** icon next to your Lambda function. You will be prompted to select an AWS SAM project location. Once selected, your Lambda function will be converted to a `template.yaml` file that is saved in your new AWS SAM project.

With your function converted to an AWS SAM template, you can:

- Control the versioning of your infrastructure
- Automate deployments
- Remotely debug functions
- Add additional AWS resources to your application
- Maintain consistent environments across your development lifecycle
- Use Infrastructure Composer to visually edit your AWS SAM template

For more information on using IaC tools, refer to the following guides:

- The AWS Serverless Application Model Developer Guide
- The AWS Cloud Development Kit (AWS CDK) Developer Guide
- The Infrastructure Composer Developer Guide
- The AWS CloudFormation User Guide

These tools provide additional capabilities for defining, testing, and deploying your serverless applications.

Next steps

To learn more about working with Lambda functions in VS Code, see the following resources:

- Working with AWS Lambda functions in the AWS Toolkit for VS Code User Guide
- Working with serverless applications in the AWS Toolkit for VS Code User Guide

- Infrastructure as code in the Lambda Developer Guide

Using GitHub Actions to deploy Lambda functions

You can use GitHub Actions to automatically deploy Lambda functions when you push code or configuration changes to your repository. The Deploy Lambda Function action provides a declarative, simple YAML interface that eliminates the complexity of manual deployment steps.

Example workflow

To configure automated Lambda function deployment, create a workflow file in your repository's `.github/workflows/` directory:

Example GitHub Actions workflow for Lambda deployment

```
name: Deploy AWS Lambda

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    permissions:
      id-token: write # Required for OIDC authentication
      contents: read  # Required to check out the repository
    steps:
      - uses: actions/checkout@v4

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/GitHubActionRole
          aws-region: us-east-1

      - name: Deploy Lambda Function
        uses: aws-actions/aws-lambda-deploy@v1
        with:
          function-name: my-lambda-function
```

```
          code-artifacts-dir: ./dist
```

This workflow runs when you push changes to the `main` branch. It checks out your repository, configures AWS credentials using OpenID Connect (OIDC), and deploys your function using the code in the `./dist` directory.

For additional examples including updating function configuration, deploying via S3 buckets, and dry run validation, see the Deploy Lambda Function README.

Additional resources

- Configure AWS Credentials GitHub Action
- Configuring OpenID Connect in AWS

Using Lambda with infrastructure as code (IaC)

Lambda functions rarely run in isolation. Instead, they often form part of a serverless application with other resources such as databases, queues, and storage. With infrastructure as code (IaC), you can automate your deployment processes to quickly and repeatably deploy and update whole serverless applications involving many separate AWS resources. This approach speeds up your development cycle, makes configuration management easier, and ensures that your resources are deployed the same way every time.

IaC tools for Lambda

CloudFormation

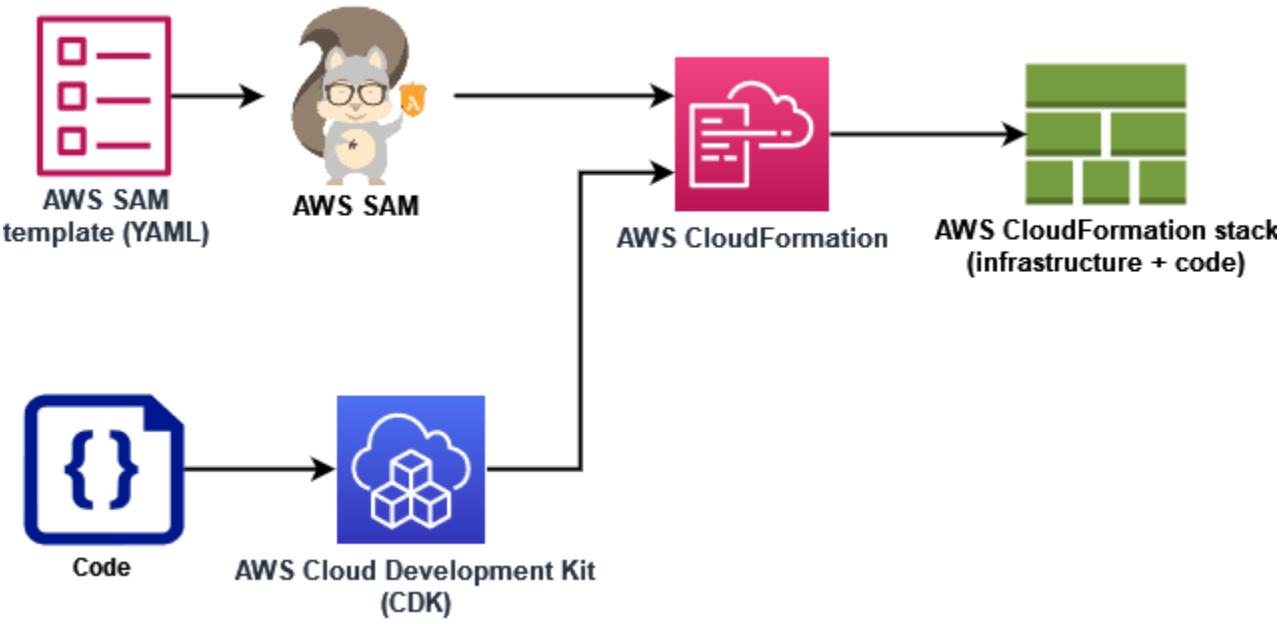
CloudFormation is the foundational IaC service from AWS. You can use YAML or JSON templates to model and provision your entire AWS infrastructure, including Lambda functions. CloudFormation handles the complexities of creating, updating, and deleting your AWS resources.

AWS Serverless Application Model (AWS SAM)

AWS SAM is an open-source framework built on top of CloudFormation. It provides a simplified syntax for defining serverless applications. Use AWS SAM templates to quickly provision Lambda functions, APIs, databases, and event sources with just a few lines of YAML.

AWS Cloud Development Kit (AWS CDK)

The CDK is a code-first approach to IaC. You can define your Lambda-based architecture using TypeScript, JavaScript, Python, Java, C#/.Net, or Go. Choose your preferred language and use programming elements like parameters, conditionals, loops, composition, and inheritance to define the desired outcome of your infrastructure. The CDK then generates the underlying CloudFormation templates for deployment. For an example of how to use Lambda with CDK, see Deploying Lambda functions with AWS CDK.



AWS also provides a service called AWS Infrastructure Composer to develop IaC templates using a simple graphical interface. With Infrastructure Composer, you design an application architecture by dragging, grouping, and connecting AWS services in a visual canvas. Infrastructure Composer then creates an AWS SAM template or an CloudFormation template from your design that you can use to deploy your application.

In the the section called “Using AWS SAM and Infrastructure Composer” section below, you use Infrastructure Composer to develop a template for a serverless application based on an existing Lambda function.

Using Lambda functions in AWS SAM and Infrastructure Composer

In this tutorial, you can get started using IaC with Lambda by creating an AWS SAM template from an existing Lambda function and then building out a serverless application in Infrastructure Composer by adding other AWS resources.

As you carry out this tutorial, you'll learn some fundamental concepts, like how AWS resources are specified in AWS SAM. You'll also learn how to use Infrastructure Composer to build a serverless application you can deploy using AWS SAM or CloudFormation.

To complete this tutorial, you'll carry out the following steps:

- Create an example Lambda function
- Use the Lambda console to view the AWS SAM template for the function
- Export your function's configuration to AWS Infrastructure Composer and design a simple serverless application based on your function's configuration
- Save an updated AWS SAM template you can use as a basis to deploy your serverless application

Prerequisites

In this tutorial, you use Infrastructure Composer's local sync feature to save your template and code files to your local build machine. To use this feature, you need a browser that supports the File System Access API, which allows web applications to read, write, and save files in your local file system . We recommend using either Google Chrome or Microsoft Edge. For more information about the File System Access API, see What is the File System Access API?

Create a Lambda function

In this first step, you create a Lambda function you can use to complete the rest of the tutorial. To keep things simple, you use the Lambda console to create a basic 'Hello world' function using the Python 3.11 runtime.

To create a 'Hello world' Lambda function using the console

1. Open the Lambda console.
2. Choose **Create function**.
3. Leave **Author from scratch** selected, and under **Basic information**, enter **`LambdaIaCDemo`** for **Function name**.

4. For **Runtime**, select **Python 3.11**.
5. Choose **Create function**.

View the AWS SAM template for your function

Before you export your function configuration to Infrastructure Composer, use the Lambda console to view your function's current configuration as an AWS SAM template. By following the steps in this section, you'll learn about the anatomy of an AWS SAM template and how to define resources like Lambda functions to start specifying a serverless application.

To view the AWS SAM template for your function

1. Open the Functions page of the Lambda console.
2. Choose the function you just created (`LambdaIaCDemo`).
3. In the **Function overview** pane, choose **Template**.

   In place of the diagram representing your function's configuration, you'll see an AWS SAM template for your function. The template should look like the following.

```
# This AWS SAM template has been generated from your function's
# configuration. If your function has one or more triggers, note
# that the AWS resources associated with these triggers aren't fully
# specified in this template and include placeholder values.Open this template
# in AWS Application Composer or your favorite IDE and modify
# it to specify a serverless application with other AWS resources.
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
```

```
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        Statement:
          - Effect: Allow
            Action:
              - logs:CreateLogGroup
            Resource: arn:aws:logs:us-east-1:123456789012:*
          - Effect: Allow
            Action:
              - logs:CreateLogStream
              - logs:PutLogEvents
            Resource:
              - >-
                arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/
LambdaIaCDemo:*
```

Let's take a moment to look at the YAML template for your function and understand some key concepts.

The template starts with the declaration `Transform: AWS::Serverless-2016-10-31`. This declaration is required because behind the scenes, AWS SAM templates are deployed through CloudFormation. Using the `Transform` statement identifies the template as an AWS SAM template file.

Following the `Transform` declaration comes the `Resources` section. This is where the AWS resources you want to deploy with your AWS SAM template are defined. AWS SAM templates can contain a combination of AWS SAM resources and CloudFormation resources. This is because during deployment, AWS SAM templates expand to CloudFormation templates, so any valid CloudFormation syntax can be added to an AWS SAM template.

At the moment, there is just one resource defined in the `Resources` section of the template, your Lambda function `LambdaIaCDemo`. To add a Lambda function to an AWS SAM template, you

use the `AWS::Serverless::Function` resource type. The `Properties` of a Lambda function resource define the function's runtime, function handler, and other configuration options. The path to your function's source code that AWS SAM should use to deploy the function is also defined here. To learn more about Lambda function resources in AWS SAM, see AWS::Serverless::Function in the *AWS SAM Developer Guide*.

As well as the function properties and configurations, the template also specifies an AWS Identity and Access Management (IAM) policy for your function. This policy gives your function permission to write logs to Amazon CloudWatch Logs. When you create a function in the Lambda console, Lambda automatically attaches this policy to your function. To learn more about specifying an IAM policy for a function in an AWS SAM template, see the `policies` property on the AWS::Serverless::Function page of the *AWS SAM Developer Guide*.

To learn more about the structure of AWS SAM templates, see AWS SAM template anatomy.

Use AWS Infrastructure Composer to design a serverless application

To start building out a simple serverless application using your function's AWS SAM template as a starting point, you export your function configuration to Infrastructure Composer and activate Infrastructure Composer's local sync mode. Local sync automatically saves your function's code and your AWS SAM template to your local build machine and keeps your saved template synced as you add other AWS resources in Infrastructure Composer.

To export your function to Infrastructure Composer

1. In the **Function Overview** pane, choose **Export to Application Composer**.

   To export your function's configuration and code to Infrastructure Composer, Lambda creates an Amazon S3 bucket in your account to temporarily store this data.
2. In the dialog box, choose **Confirm and create project** to accept the default name for this bucket and export your function's configuration and code to Infrastructure Composer.
3. (Optional) To choose another name for the Amazon S3 bucket that Lambda creates, enter a new name and choose **Confirm and create project**. Amazon S3 bucket names must be globally unique and follow the bucket naming rules.

   Selecting **Confirm and create project** opens the Infrastructure Composer console. On the *canvas*, you'll see your Lambda function.
4. From the **Menu** dropdown, choose **Activate local sync**.

5. In the dialog box that opens, choose **Select folder** and select a folder on your local build machine.
6. Choose **Activate** to activate local sync.

To export your function to Infrastructure Composer, you need permission to use certain API actions. If you're unable to export your function, see the section called "Required permissions" and make sure you have the permissions you need.

> **Note**
>
> Standard Amazon S3 pricing applies for the bucket Lambda creates when you export a function to Infrastructure Composer. The objects that Lambda puts into the bucket are automatically deleted after 10 days, but Lambda doesn't delete the bucket itself.
> To avoid additional charges being added to your AWS account, follow the instructions in Deleting a bucket after you have exported your function to Infrastructure Composer. For more information about the Amazon S3 bucket Lambda creates, see the section called "Infrastructure Composer".

To design your serverless application in Infrastructure Composer

After activating local sync, changes you make in Infrastructure Composer will be reflected in the AWS SAM template saved on your local build machine. You can now drag and drop additional AWS resources onto the Infrastructure Composer canvas to build out your application. In this example, you add an Amazon SQS simple queue as a trigger for your Lambda function and a DynamoDB table for the function to write data to.

1. Add an Amazon SQS trigger to your Lambda function by doing the following:
   a. In the search field in the **Resources** palette, enter **SQS**.
   b. Drag the **SQS Queue** resource onto your canvas and position it to the left of your Lambda function.
   c. Choose **Details**, and for **Logical ID** enter **`LambdaIaCQueue`**.
   d. Choose **Save**.
   e. Connect your Amazon SQS and Lambda resources by clicking on the **Subscription** port on the SQS queue card and dragging it to the left hand port on the Lambda function card. The appearance of a line between the two resources indicates a successful connection.

Infrastructure Composer also displays a message at the bottom of the canvas indicating that the two resources are successfully connected.

2. Add an Amazon DynamoDB table for your Lambda function to write data to by doing the following:

   a. In the search field in the **Resources** palette, enter **`DynamoDB`**.

   b. Drag the **DynamoDB Table** resource onto your canvas and position it to the right of your Lambda function.

   c. Choose **Details**, and for **Logical ID** enter **`LambdaIaCTable`**.

   d. Choose **Save**.

   e. Connect the DynamoDB table to your Lambda function by clicking on the right hand port of the Lambda function card and dragging it to the left hand port on the DynamoDB card.

Now that you've added these extra resources, let's take a look at the updated AWS SAM template Infrastructure Composer has created.

To view your updated AWS SAM template

- On the Infrastructure Composer canvas, choose **Template** to switch from the canvas view to the template view.

Your AWS SAM template should now contain the following additional resources and properties:

- An Amazon SQS queue with the identifier `LambdaIaCQueue`

```
LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
```

  When you add an Amazon SQS queue using Infrastructure Composer, Infrastructure Composer sets the `MessageRetentionPeriod` property. You can also set the `FifoQueue` property by selecting **Details** on the SQS Queue card and checking or unchecking **Fifo queue**.

  To set other properties for your queue, you can manually edit the template to add them. To learn more about the `AWS::SQS::Queue` resource and its available properties, see AWS::SQS::Queue in the *CloudFormation User Guide*.

- An `Events` property in your Lambda function definition that specifies the Amazon SQS queue as a trigger for the function

```
Events:
  LambdaIaCQueue:
    Type: SQS
    Properties:
      Queue: !GetAtt LambdaIaCQueue.Arn
      BatchSize: 1
```

  The `Events` property consists of an event type and a set of properties that depend on the type. To learn about the different AWS services you can configure to trigger a Lambda function and the properties you can set, see EventSource in the *AWS SAM Developer Guide*.

- A DynamoDB table with the identifier `LambdaIaCTable`

```
LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

  When you add a DynamoDB table using Infrastructure Composer, you can set your table's keys by choosing **Details** on the DynamoDB table card and editing the key values. Infrastructure Composer also sets default values for a number of other properties including `BillingMode` and `StreamViewType`.

  To learn more about these properties and other properties you can add to your AWS SAM template, see AWS::DynamoDB::Table in the *CloudFormation User Guide*.

- A new IAM policy that gives your function permission to perform CRUD operations on the DynamoDB table you added.

```
Policies:
```

```
...
  - DynamoDBCrudPolicy:
    TableName: !Ref LambdaIaCTable
```

The complete final AWS SAM template should look like the following.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
              Resource: arn:aws:logs:us-east-1:594035263019:*
            - Effect: Allow
              Action:
                - logs:CreateLogStream
                - logs:PutLogEvents
              Resource:
```

```
                - arn:aws:logs:us-east-1:594035263019:log-group:/aws/lambda/
LambdaIaCDemo:*
        - DynamoDBCrudPolicy:
            TableName: !Ref LambdaIaCTable
      Events:
        LambdaIaCQueue:
          Type: SQS
          Properties:
            Queue: !GetAtt LambdaIaCQueue.Arn
            BatchSize: 1
      Environment:
        Variables:
          LAMBDAIACTABLE_TABLE_NAME: !Ref LambdaIaCTable
          LAMBDAIACTABLE_TABLE_ARN: !GetAtt LambdaIaCTable.Arn
  LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
  LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

Deploy your serverless application using AWS SAM (optional)

If you want to use AWS SAM to deploy a serverless application using the template you just created in Infrastructure Composer, you first need to install the AWS SAM CLI. To do this, follow the instructions in Installing the AWS SAM CLI.

Before you deploy your application, you also need to update the function code that Infrastructure Composer saved along with your template. At the moment, the `lambda_function.py` file that Infrastructure Composer saved contains only the basic 'Hello world' code that Lambda provided when you created the function.

To update your function code, copy the following code and paste it into the `lambda_function.py` file Infrastructure Composer saved to your local build machine. You specified the directory for Infrastructure Composer to save this file to when you activated Local Sync mode.

This code accepts a key value pair in a message from the Amazon SQS queue you created in Infrastructure Composer. If both the key and value are strings, the code then uses them to write an item to the DynamoDB table defined in your template.

Updated Python function code

```
import boto3
import os
import json

# define the DynamoDB table that Lambda will connect to
tablename = os.environ['LAMBDAIACTABLE_TABLE_NAME']

# create the DynamoDB resource
dynamo = boto3.client('dynamodb')

def lambda_handler(event, context):
    # get the message out of the SQS event
    message = event['Records'][0]['body']
    data = json.loads(message)
    # write event data to DDB table
    if check_message_format(data):
        key = next(iter(data))
        value = data[key]
        dynamo.put_item(
            TableName=tablename,
            Item={
                'id': {'S': key},
                'Value': {'S': value}
            }
        )
    else:
        raise ValueError("Input data not in the correct format")

# check that the event object contains a single key value
# pair that can be written to the database
def check_message_format(message):
    if len(message) != 1:
```

```
        return False

    key, value = next(iter(message.items()))

    if not (isinstance(key, str) and isinstance(value, str)):
        return False

    else:
        return True
```

To deploy your serverless application

To deploy your application using the AWS SAM CLI, carry out the following steps. For your function to build and deploy correctly, Python version 3.11 must be installed on your build machine and on your PATH.

1. Run the following command from the directory in which Infrastructure Composer saved your `template.yaml` and `lambda_function.py` files.

   ```
   sam build
   ```

   This command gathers the build artifacts for your application and places them in the proper format and location to deploy them.

2. To deploy your application and create the Lambda, Amazon SQS, and DynamoDB resources specified in your AWS SAM template, run the following command.

   ```
   sam deploy --guided
   ```

   Using the `--guided` flag means that AWS SAM will show you prompts to guide you through the deployment process. For this deployment, accept the default options by pressing Enter.

During the deployment process, AWS SAM creates the following resources in your AWS account:

- An CloudFormation stack named `sam-app`
- A Lambda function with the name format `sam-app-LambdaIaCDemo-`*`99VXPpYQVv1M`*
- An Amazon SQS queue with the name format `sam-app-LambdaIaCQueue-`*`xL87VeKsGiIo`*
- A DynamoDB table with the name format `sam-app-LambdaIaCTable-`*`CN0S66C0VLNV`*

AWS SAM also creates the necessary IAM roles and policies so that your Lambda function can read messages from the Amazon SQS queue and perform CRUD operations on the DynamoDB table.

Testing your deployed application (optional)

To confirm that your serverless application deployed correctly, send a message to your Amazon SQS queue containing a key value pair and check that Lambda writes an item into your DynamoDB table using these values.

To test your serverless application

1. Open the Queues page of the Amazon SQS console and select the queue that AWS SAM created from your template. The name has the format `sam-app-LambdaIaCQueue-xL87VeKsGiIo`.
2. Choose **Send and receive messages** and paste the following JSON into the **Message body** in the **Send message** section.

   ```
   {
       "myKey": "myValue"
   }
   ```

3. Choose **Send message**.

   Sending your message to the queue causes Lambda to invoke your function through the event source mapping defined in your AWS SAM template. To confirm that Lambda has invoked your function as expected, confirm that an item has been added to your DynamoDB table.

4. Open the Tables page of the DynamoDB console and select your table. The name has the format `sam-app-LambdaIaCTable-CN0S66C0VLNV`.
5. Choose **Explore table items**. In the **Items returned** pane, you should see an item with the **id** `myKey` and the **Value** `myValue`.

Deploying Lambda functions with AWS CDK

The AWS Cloud Development Kit (AWS CDK) is an infrastructure as code (IaC) framework that you can use to define AWS cloud infrastructure by using a programming language of your choosing. To define your own cloud infrastructure, you first write an app (in one of the CDK's supported languages) that contains one or more stacks. Then, you synthesize it to an CloudFormation

template and deploy your resources to your AWS account. Follow the steps in this topic to deploy a Lambda function that returns an event from an Amazon API Gateway endpoint.

The AWS Construct Library, included with the CDK, provides modules that you can use to model the resources that AWS services provide. For popular services, the library provides curated constructs with smart defaults and best practices. You can use the aws_lambda module to define your function and supporting resources with just a few lines of code.

Prerequisites

Before starting this tutorial, install the AWS CDK by running the following command.

```
npm install -g aws-cdk
```

Step 1: Set up your AWS CDK project

Create a directory for your new AWS CDK app and initialize the project.

JavaScript

```
mkdir hello-lambda
cd hello-lambda
cdk init --language javascript
```

TypeScript

```
mkdir hello-lambda
cd hello-lambda
cdk init --language typescript
```

Python

```
mkdir hello-lambda
cd hello-lambda
cdk init --language python
```

After the project starts, activate the project's virtual environment and install the baseline dependencies for AWS CDK.

```
source .venv/bin/activate
```

```
python -m pip install -r requirements.txt
```

Java

```
mkdir hello-lambda
cd hello-lambda
cdk init --language java
```

Import this Maven project to your Java integrated development environment (IDE). For example, in Eclipse, choose **File**, **Import**, **Maven**, **Existing Maven Projects**.

C#

```
mkdir hello-lambda
cd hello-lambda
cdk init --language csharp
```

> **Note**
>
> The AWS CDK application template uses the name of the project directory to generate names for source files and classes. In this example, the directory is named `hello-lambda`. If you use a different project directory name, your app won't match these instructions.

AWS CDK v2 includes stable constructs for all AWS services in a single package that's called `aws-cdk-lib`. This package is installed as a dependency when you initialize the project. When working with certain programming languages, the package is installed when you build the project for the first time.

Step 2: Define the AWS CDK stack

A CDK *stack* is a collection of one or more constructs, which define AWS resources. Each CDK stack represents an CloudFormation stack in your CDK app.

To define your CDK stack, follow the instructions for your preferred programming language. This stack defines the following:

- The function's logical name: `MyFunction`

- The location of the function code, specified in the `code` property. For more information, see Handler code in the *AWS Cloud Development Kit (AWS CDK) API Reference*.
- The REST API's logical name: `HelloApi`
- The API Gateway endpoint's logical name: `ApiGwEndpoint`

Note that all of the CDK stacks in this tutorial use the Node.js runtime for the Lambda function. You can use different programming languages for the CDK stack and the Lambda function to leverage the strengths of each language. For example, you can use TypeScript for the CDK stack to leverage the benefits of static typing for your infrastructure code. You can use JavaScript for the Lambda function to take advantage of the flexibility and rapid development of a dynamically typed language.

JavaScript

Open the `lib/hello-lambda-stack.js` file and replace the contents with the following.

```
const { Stack } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigw = require('aws-cdk-lib/aws-apigateway');

class HelloLambdaStack extends Stack {
  /**
   *
   * @param {Construct} scope
   * @param {string} id
   * @param {StackProps=} props
   */
  constructor(scope, id, props) {
    super(scope, id, props);
    const fn = new lambda.Function(this, 'MyFunction', {
      code: lambda.Code.fromAsset('lib/lambda-handler'),
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler'
    });

    const endpoint = new apigw.LambdaRestApi(this, 'MyEndpoint', {
      handler: fn,
      restApiName: "HelloApi"
    });

  }
```

```
}

module.exports = { HelloLambdaStack }
```

TypeScript

Open the `lib/hello-lambda-stack.ts` file and replace the contents with the following.

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as apigw from "aws-cdk-lib/aws-apigateway";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as path from 'node:path';

export class HelloLambdaStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps){
    super(scope, id, props)
    const fn = new lambda.Function(this, 'MyFunction', {
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler',
      code: lambda.Code.fromAsset(path.join(__dirname, 'lambda-handler')),
    });

    const endpoint = new apigw.LambdaRestApi(this, `ApiGwEndpoint`, {
      handler: fn,
      restApiName: `HelloApi`,
    });

  }
}
```

Python

Open the `/hello-lambda/hello_lambda/hello_lambda_stack.py` file and replace the contents with the following.

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigw,
    aws_lambda as _lambda
)
from constructs import Construct
```

```
class HelloLambdaStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        fn = _lambda.Function(
            self,
            "MyFunction",
            runtime=_lambda.Runtime.NODEJS_LATEST,
            handler="index.handler",
            code=_lambda.Code.from_asset("lib/lambda-handler")
        )

        endpoint = apigw.LambdaRestApi(
            self,
            "ApiGwEndpoint",
            handler=fn,
            rest_api_name="HelloApi"
        )
```

Java

Open the /hello-lambda/src/main/java/com/myorg/HelloLambdaStack.java file and replace the contents with the following.

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.lambda.Function;

public class HelloLambdaStack extends Stack {
    public HelloLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloLambdaStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Function hello = Function.Builder.create(this, "MyFunction")
```

```
 .runtime(software.amazon.awscdk.services.lambda.Runtime.NODEJS_LATEST)

 .code(software.amazon.awscdk.services.lambda.Code.fromAsset("lib/lambda-handler"))
                            .handler("index.handler")
                            .build();

        LambdaRestApi api = LambdaRestApi.Builder.create(this, "ApiGwEndpoint")
                                .restApiName("HelloApi")
                                .handler(hello)
                                .build();

    }
}
```

C#

Open the `src/HelloLambda/HelloLambdaStack.cs` file and replace the contents with the following.

```
using Amazon.CDK;
using Amazon.CDK.AWS.APIGateway;
using Amazon.CDK.AWS.Lambda;
using Constructs;

namespace HelloLambda
{
    public class HelloLambdaStack : Stack
    {
        internal HelloLambdaStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var fn = new Function(this, "MyFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_LATEST,
                Code = Code.FromAsset("lib/lambda-handler"),
                Handler = "index.handler"
            });

            var api = new LambdaRestApi(this, "ApiGwEndpoint", new
 LambdaRestApiProps
            {
                Handler = fn
```

```
            });
        }
    }
}
```

Step 3: Create the Lambda function code

1. From the root of your project (`hello-lambda`), create the `/lib/lambda-handler` directory for the Lambda function code. This directory is specified in the `code` property of your AWS CDK stack.
2. Create a new file called `index.js` in the `/lib/lambda-handler` directory. Paste the following code into the file. The function extracts specific properties from the API request and returns them as a JSON response.

```
exports.handler = async (event) => {
  // Extract specific properties from the event object
  const { resource, path, httpMethod, headers, queryStringParameters, body } =
 event;
  const response = {
    resource,
    path,
    httpMethod,
    headers,
    queryStringParameters,
    body,
  };
  return {
    body: JSON.stringify(response, null, 2),
    statusCode: 200,
  };
};
```

Step 4: Deploy the AWS CDK stack

1. From the root of your project, run the cdk synth command:

```
cdk synth
```

This command synthesizes an AWS CloudFormation template from your CDK stack. The template is an approximately 400-line YAML file, similar to the following.

> **Note**
>
> If you get the following error, make sure that you are in the root of your project directory.
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

Example CloudFormation template

```
Resources:
  MyFunctionServiceRole3C357FF2:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: HelloLambdaStack/MyFunction/ServiceRole/Resource
  MyFunction1BAA52E7:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket:
          Fn::Sub: cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}
        S3Key:
 ab1111111cd32708dc4b83e81a21c296d607ff2cdef00f1d7f48338782f92l3901.zip
```

```
      Handler: index.handler
      Role:
        Fn::GetAtt:
          - MyFunctionServiceRole3C357FF2
          - Arn
      Runtime: nodejs24.x
      ...
```

2. Run the cdk deploy command:

```
cdk deploy
```

Wait while your resources are created. The final output includes the URL for your API Gateway endpoint. Example:

```
Outputs:
HelloLambdaStack.ApiGwEndpoint77F417B1 = https://abcd1234.execute-api.us-
east-1.amazonaws.com/prod/
```

Step 5: Test the function

To invoke the Lambda function, copy the API Gateway endpoint and paste it into a web browser or run a `curl` command:

```
curl -s https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/
```

The response is a JSON representation of selected properties from the original event object, which contains information about the request made to the API Gateway endpoint. Example:

```
{
  "resource": "/",
  "path": "/",
  "httpMethod": "GET",
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
    "Accept-Encoding": "gzip, deflate, br, zstd",
    "Accept-Language": "en-US,en;q=0.9",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
```

```
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-ASN": "16509",
    "CloudFront-Viewer-Country": "US",
    "Host": "abcd1234.execute-api.us-east-1.amazonaws.com",
     ...
```

Step 6: Clean up your resources

The API Gateway endpoint is publicly accessible. To prevent unexpected charges, run the cdk destroy command to delete the stack and all associated resources.

```
cdk destroy
```

Next steps

For information about writing AWS CDK apps in your language of choice, see the following:

TypeScript

- Working with the AWS CDK in TypeScript

JavaScript

- Working with the AWS CDK in JavaScript

Python

- Working with the AWS CDK in Python

Java

- Working with the AWS CDK in Java

C#

- Working with the AWS CDK in C#

Go

- Working with the AWS CDK in Go

Powertools for AWS Lambda

Powertools for AWS Lambda (also referred to as Powertools for AWS) provides utility functions, decorators, and middleware that handle common Lambda tasks like structured logging, tracing, metrics collection, and input validation. Use Powertools for AWS Lambda to implement serverless best practices and accelerate development across multiple Lambda functions. Doing this simplifies common development tasks in your Lambda functions.

Key benefits of Powertools for AWS

While Lambda development is possible without Powertools for AWS, using it offers several advantages:

- Built-in observability: Structured logging, tracing, and custom metrics
- Secrets management: Parameter retrieval, secrets handling, and idempotency
- Progressive Enhancement: Choose the utilities that best suit your needs
- Accelerated development: Event parsing, validation, and batch processing
- Best practices: Implementation of AWS Well-Architected serverless patterns

Integrating Powertools with AWS

Powertools for AWS helps you build production-ready serverless applications with less custom code. Available in Python, TypeScript/Node.js, .NET, and Java, Powertools for AWS can be included through Lambda Layers, or using the language package manager. Each language implementation provides core features like structured logging, tracing, metrics collection, and event handling, while maintaining idioms natural to each programming language. These implementations are complemented by specialized components for AWS service integration, supporting parameter retrieval, batch processing, and API handling, along with best practices like correlation ID propagation, error handling, and idempotency patterns. Together, these features enable developers to build robust, maintainable serverless applications while reducing custom code overhead.

- Powertools for AWS Lambda (Python)
- Powertools for AWS Lambda (TypeScript)
- Powertools for AWS Lambda (Java)
- Powertools for AWS Lambda (.NET)

Next steps

To learn more about working with Powertools for AWS, see the following resources:

- Powertools for AWS Lambda workshop
- Serverless patterns that use Powertools for AWS
- AWS well-architected serverless lens
- Building Serverless APIs with Powertools for AWS Lambda

Managing Lambda workflows and events

When building serverless applications with Lambda, you often need ways to orchestrate function execution and handle events. AWS provides several approaches for coordinating Lambda functions:

- Lambda durable functions for code-first workflow orchestration within Lambda
- AWS Step Functions for visual workflow orchestration across multiple services
- Amazon EventBridge Scheduler and Amazon EventBridge for event-driven architectures and scheduling

You can also integrate these approaches together. For example, you might use EventBridge Scheduler to trigger durable functions or Step Functions workflows when specific events occur, or configure workflows to publish events to EventBridge Scheduler at defined execution points. The following topics in this section provide more information on how you can use these orchestration options.

Code-first orchestration with durable functions

Lambda durable functions provide a code-first approach to workflow orchestration, allowing you to build stateful, long-running workflows directly within your Lambda functions. Unlike external orchestration services, durable functions keep your workflow logic in code, making it easier to version, test, and maintain alongside your business logic.

Durable functions are ideal when you need:

- **Use standard programming languages:** Define workflows using familiar programming languages like JavaScript and Python

- **Long-running processes:** Execute workflows that can run for up to one year, far beyond the 15-minute limit of standard Lambda functions
- **Simplified development:** Keep workflow logic and business logic in the same codebase for easier maintenance and testing
- **Cost-effective waiting:** Pause execution during wait states without consuming compute resources
- **Built-in state management:** Automatic checkpointing and state persistence without external storage configuration

For help choosing between durable functions and Step Functions, see Durable functions or Step Functions.

For more information on durable functions, see Lambda durable functions.

Orchestrating workflows with Step Functions

AWS Step Functions is a workflow orchestration service that helps you coordinate multiple Lambda functions and other AWS services into structured workflows. These workflows can maintain state, handle errors with sophisticated retry mechanisms, and process data at scale.

Step Functions offers two types of workflows to meet different orchestration needs:

Standard workflows

Ideal for long-running, auditable workflows that require exactly-once execution semantics. Standard workflows can run for up to one year, provide detailed execution history, and support visual debugging. They are suitable for processes like order fulfillment, data processing pipelines, or multi-step analytics jobs.

Express workflows

Designed for high-event-rate, short-duration workloads with at-least-once execution semantics. Express workflows can run for up to five minutes and are ideal for high-volume event processing, streaming data transformations, or IoT data ingestion scenarios. They offer higher throughput and potentially lower cost compared to Standard workflows.

> **Note**
>
> For more information on Step Functions workflow types, see Choosing workflow type in Step Functions.

Within these workflows, Step Functions provides two types of Map states for parallel processing:

Inline Map

Processes items from a JSON array within the execution history of the parent workflow. Inline Map supports up to 40 concurrent iterations and is suitable for smaller datasets or when you need to keep all processing within a single execution. For more information, see Using Map state in Inline mode.

Distributed Map

Enables processing of large-scale parallel workloads by iterating over datasets that exceed 256 KiB or require more than 40 concurrent iterations. With support for up to 10,000 parallel child workflow executions, Distributed Map excels at processing semi-structured data stored in Amazon S3, such as JSON or CSV files, making it ideal for batch processing and ETL operations. For more information, see Using Map state in Distributed mode.

By combining these workflow types and Map states, Step Functions provides a flexible and powerful toolset for orchestrating complex serverless applications, from small-scale operations to large-scale data processing pipelines.

To get started with using Lambda with Step Functions, see Orchestrating Lambda functions with Step Functions.

Managing events with EventBridge and EventBridge Scheduler

Amazon EventBridge is an event bus service that helps you build event-driven architectures. It routes events between AWS services, integrated applications, and software as a service (SaaS) applications. EventBridge Scheduler is a serverless scheduler that enables you to create, run, and manage tasks from one central service, allowing you to invoke Lambda functions on a schedule using cron and rate expressions, or configure one-time invocations.

Amazon EventBridge and EventBridge Scheduler help you build event-driven architectures with Lambda. EventBridge routes events between AWS services, integrated applications, and SaaS

applications, while EventBridge Scheduler provides specific scheduling capabilities for invoking Lambda functions on a recurring or one-time basis.

These services provide several key capabilities for working with Lambda functions:

- Create rules that match and route events to Lambda functions using EventBridge
- Set up recurring function invocations using cron and rate expressions with EventBridge Scheduler
- Configure one-time function invocations at specific dates and times
- Define flexible time windows and retry policies for scheduled invocations

For more information, see Invoke a Lambda function on a schedule.

Lambda durable functions

Lambda durable functions enable you to build resilient multi-step applications and AI workflows that can execute for up to one year while maintaining reliable progress despite interruptions. When a durable function runs, this complete lifecycle is called a durable execution, which uses checkpoints to track progress and automatically recover from failures through replay, re-executing from the beginning while skipping completed work.

Within each function, you use durable operations as fundamental building blocks. Steps execute business logic with built-in retries and progress tracking, while waits suspend execution without incurring compute charges, making them ideal for long-running processes like human-in-the-loop workflows or polling external dependencies. Whether you're processing orders, coordinating microservices, or orchestrating agentic AI applications, durable functions maintain state automatically and recover from failures while you write code in familiar programming languages.

Key benefits

Write resilient code naturally: With familiar programming constructs, you write code that handles failures automatically. Built-in checkpointing, transparent retries, and automatic recovery mean your business logic stays clean and focused.

Pay only for what you use: During wait operations, your function suspends without incurring compute charges. For long-running workflows that wait hours or days, you pay only for actual processing time, not idle waiting.

Operational simplicity: With Lambda's serverless model, you get automatic scaling, including scale-to-zero, without managing infrastructure. Durable functions handle state management, retry logic, and failure recovery automatically, reducing operational overhead.

When to use durable functions

Short-lived coordination: Coordinate payments, inventory, and shipping across multiple services with automatic rollback on failures. Process orders through validation, payment authorization, inventory allocation, and fulfillment with guaranteed completion.

Process payments with confidence: Build resilient payment flows that maintain transaction state through failures and handle retries automatically. Coordinate multi-step authorization, fraud checks, and settlement across payment providers with full auditability across steps.

Build reliable AI workflows: Create multi-step AI workflows that chain model calls, incorporate human feedback, and handle long-running tasks deterministically during failures. Automatically resume after suspension, and only pay for active execution time.

Orchestrate complex order fulfillment: Coordinate order processing across inventory, payment, shipping, and notification systems with built-in resilience. Automatically handle partial failures, preserve order state despite interruptions, and efficiently wait for external events without consuming compute resources.

Automate multi-step business workflows: Build reliable workflows for employee onboarding, loan approvals, and compliance processes that span days or weeks. Maintain workflow state across human approvals, system integrations, and scheduled tasks while providing full visibility into process status and history.

How durable functions compare to Step Functions

Both, durable functions and Step Functions, provide workflow orchestration with automatic state management. The key differences are where they run and how you define workflows:

- **Durable functions:** Run within Lambda, use standard programming languages, managed within Lambda environment
- **Step Functions:** Standalone service, graph-based DSL or visual designer, fully managed with zero maintenance

Durable functions are ideal for application development in Lambda where workflows are tightly coupled with business logic. Step Functions excels at workflow orchestration across AWS services where you need visual design, native integrations to 220+ services, and zero-maintenance infrastructure.

For a detailed comparison, see Durable functions or Step Functions.

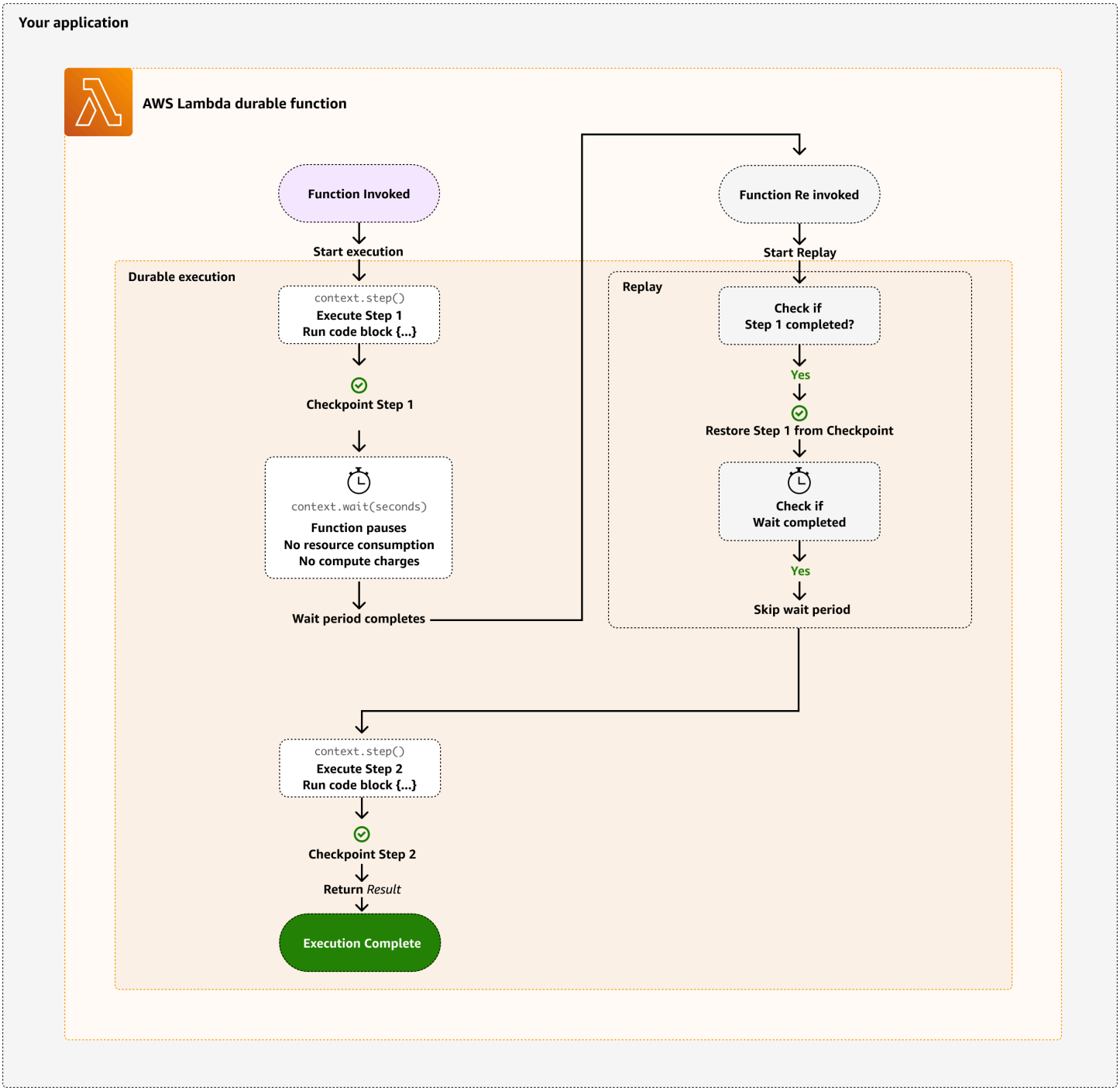
How it works

Under the hood, durable functions are regular Lambda functions using a checkpoint/replay mechanism to track progress and support long-running operations through user-defined suspension points, commonly referred to as durable execution. After your function resumes from a pause or interruption, the system performs replay. During replay, your code runs from

the beginning but skips over completed checkpoints, using stored results instead of re-executing completed operations. This replay mechanism ensures consistency while enabling long-running executions.

To harness this checkpoint-and-replay mechanism in your applications, Lambda provides a durable execution SDK. The SDK abstracts away the complexity of managing checkpoints and replay, exposing simple primitives called durable operations that you use in your code. The SDK is available for JavaScript, TypeScript, Python and Java, integrating seamlessly with your existing Lambda development workflow.

With the SDK, you wrap your Lambda event handler, which then provides a DurableContext alongside your event. This context gives you access to durable operations like steps and waits. You write your function logic as normal sequential code, but instead of calling services directly, you wrap those calls in steps for automatic checkpointing and retries. When you need to pause execution, you add waits that suspend your function without incurring charges. The SDK handles all the complexity of state management and replay behind the scenes, so your code remains clean and readable.



Next steps

- Get started with durable functions
- Explore the durable execution SDK
- Durable functions or Step Functions

- Monitor and debug durable functions
- Review security and permissions
- Follow best practices

Basic concepts

Lambda provides durable execution SDKs for JavaScript, TypeScript, and Python. These SDKs are the foundation for building durable functions, providing the primitives you need to checkpoint progress, handle retries, and manage execution flow. For complete SDK documentation and examples, see the JavaScript/TypeScript SDK and Python SDK on GitHub.

Durable execution

A **durable execution** represents the complete lifecycle of a Lambda durable function, using a checkpoint and replay mechanism to track business logic progress, suspend execution, and recover from failures. When functions resume after suspension or interruptions, previously completed checkpoints are replayed and the function continues execution.

The lifecycle may include multiple invocations of a Lambda function to complete the execution, particularly after suspensions or failure recovery. This approach enables your function to run for extended periods (up to one year) while maintaining reliable progress despite interruptions.

How replay works

Lambda keeps a running log of all durable operations (steps, waits, and other operations) as your function executes. When your function needs to pause or encounters an interruption, Lambda saves this checkpoint log and stops the execution. When it's time to resume, Lambda invokes your function again from the beginning and replays the checkpoint log, substituting stored values for completed operations. This means your code runs again, but previously completed steps don't re-execute. Their stored results are used instead.

This replay mechanism is fundamental to understanding durable functions. Your code must be deterministic during replay, meaning it produces the same results given the same inputs. Avoid operations with side effects (like generating random numbers or getting the current time) outside of steps, as these can produce different values during replay and cause non-deterministic behavior.

DurableContext

DurableContext is the context object your durable function receives. It provides methods for durable operations like steps and waits that create checkpoints and manage execution flow.

Your durable function receives a `DurableContext` instead of the default Lambda context:

TypeScript

```
import {
  DurableContext,
  withDurableExecution,
} from "@aws/durable-execution-sdk-js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const result = await context.step(async () => {
      return "step completed";
    });
    return result;
  },
);
```

Python

```
from aws_durable_execution_sdk_python import (
    DurableContext,
    durable_execution,
    durable_step,
)

@durable_step
def my_step(step_context, data):
    # Your business logic
    return result

@durable_execution
def handler(event, context: DurableContext):
    result = context.step(my_step(event["data"]))
    return result
```

The Python SDK for durable functions uses synchronous methods and doesn't support `await`. The TypeScript SDK uses `async/await`.

Steps

Steps runs business logic with built-in retries and automatic checkpointing. Each step saves its result, ensuring your function can resume from any completed step after interruptions.

TypeScript

```
// Each step is automatically checkpointed
const order = await context.step(async () => processOrder(event));
const payment = await context.step(async () => processPayment(order));
const result = await context.step(async () => completeOrder(payment));
```

Python

```
# Each step is automatically checkpointed
order = context.step(lambda: process_order(event))
payment = context.step(lambda: process_payment(order))
result = context.step(lambda: complete_order(payment))
```

Wait States

Wait states are planned pauses where your function stops running (and stops charging) until it's time to continue. Use them to wait for time periods, external callbacks, or specific conditions.

TypeScript

```
// Wait for 1 hour without consuming resources
await context.wait({ seconds:3600 });

// Wait for external callback
const approval = await context.waitForCallback(
  async (callbackId) => sendApprovalRequest(callbackId)
);
```

Python

```
# Wait for 1 hour without consuming resources
```

```
context.wait(Duration.from_seconds(3600))

# Wait for external callback
approval = context.wait_for_callback(
    lambda callback_id: send_approval_request(callback_id)
)
```

When your function encounters a wait or needs to pause, Lambda saves the checkpoint log and stops the execution. When it's time to resume, Lambda invokes your function again and replays the checkpoint log, substituting stored values for completed operations.

For more complex workflows, durable Lambda functions also come with advanced operations like `parallel()` for concurrent execution, `map()` for processing arrays, `runInChildContext()` for nested operations, and `waitForCondition()` for polling. See Examples for detailed examples and guidance on when to use each operation.

Invoking other functions

Invoke allows a durable function to call other Lambda functions and wait for their results. The calling function suspends while the invoked function executes, creating a checkpoint that preserves the result. This enables you to build modular workflows where specialized functions handle specific tasks.

Use `context.invoke()` to call other functions from within your durable function. The invocation is checkpointed, so if your function is interrupted after the invoked function completes, it resumes with the stored result without re-invoking the function.

TypeScript

```
// Invoke another function and wait for result
const customerData = await context.invoke(
  'validate-customer',
  'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
  { customerId: event.customerId }
);

// Use the result in subsequent steps
const order = await context.step(async () => {
  return processOrder(customerData);
});
```

Python

```
# Invoke another function and wait for result
customer_data = context.invoke(
    'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
    {'customerId': event['customerId']},
    name='validate-customer'
)

# Use the result in subsequent steps
order = context.step(
    lambda: process_order(customer_data),
    name='process-order'
)
```

The invoked function can be either a durable or standard Lambda function. If you invoke a durable function, the calling function waits for the complete durable execution to finish. This pattern is common in microservices architectures where each function handles a specific domain, allowing you to compose complex workflows from specialized, reusable functions.

> **Note**
>
> Cross-account invocations are not supported. The invoked function must be in the same AWS account as the calling function.

Durable function configuration

Durable functions have specific configuration settings that control execution behavior and data retention. These settings are separate from standard Lambda function configuration and apply to the entire durable execution lifecycle.

The **DurableConfig** object defines the configuration for durable functions:

```
{
  "ExecutionTimeout": Integer,
  "RetentionPeriodInDays": Integer
}
```

Execution timeout

The **execution timeout** controls how long a durable execution can run from start to completion. This is different from the Lambda function timeout, which controls how long a single function invocation can run.

A durable execution can span multiple Lambda function invocations as it progresses through checkpoints, waits, and replays. The execution timeout applies to the total elapsed time of the durable execution, not to individual function invocations.

Understanding the difference

The Lambda function timeout (maximum 15 minutes) limits each individual invocation of your function. The durable execution timeout (maximum 1 year) limits the total time from when the execution starts until it completes, fails, or times out. During this period, your function may be invoked multiple times as it processes steps, waits, and recovers from failures.

For example, if you set a durable execution timeout of 24 hours and a Lambda function timeout of 5 minutes:

- Each function invocation must complete within 5 minutes
- The entire durable execution can run for up to 24 hours
- Your function can be invoked many times during those 24 hours
- Wait operations don't count against the Lambda function timeout but do count against the execution timeout

You can configure the execution timeout when creating a durable function using the Lambda console, AWS CLI, or AWS SAM. In the Lambda console, choose your function, then Configuration, Durable execution. Set the Execution timeout value in seconds (default: 86400 seconds / 24 hours, minimum: 60 seconds, maximum: 31536000 seconds / 1 year).

> **Note**
>
> The execution timeout and Lambda function timeout are different settings. The Lambda function timeout controls how long each individual invocation can run (maximum 15 minutes). The execution timeout controls the total elapsed time for the entire durable execution (maximum 1 year).

Retention period

The **retention period** controls how long Lambda retains execution history and checkpoint data after a durable execution completes. This data includes step results, execution state, and the complete checkpoint log.

After the retention period expires, Lambda deletes the execution history and checkpoint data. You can no longer retrieve execution details or replay the execution. The retention period starts when the execution reaches a terminal state (SUCCEEDED, FAILED, STOPPED, or TIMED_OUT).

You can configure the retention period when creating a durable function using the Lambda console, AWS CLI, or AWS SAM. In the Lambda console, choose your function, then Configuration, Durable execution. Set the Retention period value in days (default: 14 days, minimum: 1 day, maximum: 90 days).

Choose a retention period based on your compliance requirements, debugging needs, and cost considerations. Longer retention periods provide more time for debugging and auditing but increase storage costs.

See also

- Durable functions or Step Functions – Compare durable functions with Step Functions to understand when each approach is most effective.

Creating Lambda durable functions

To get started with Lambda durable functions, use the Lambda console to create a durable function. In a few minutes, you can create and deploy a durable function that uses steps and waits to demonstrate checkpoint-based execution.

As you carry out the tutorial, you'll learn fundamental durable function concepts, like how to use the `DurableContext` object, create checkpoints with steps, and pause execution with waits. You'll also learn how replay works when your function resumes after a wait.

To keep things simple, this tutorial shows you how to create your function using either the Python or Node.js runtime. With these interpreted languages, you can edit function code directly in the console's built-in code editor.

Durable functions in Java currently can only be deployed through container images. For more information on creating durable functions from container images, see Supported runtimes for durable functions or Deploy Lambda durable functions with Infrastructure as Code.

> **Note**
>
> Durable functions currently support Python and Node.js (JavaScript/TypeScript) runtimes and container images (OCI), such as Java. For a complete list of supported runtime versions and container image options, see Supported runtimes for durable functions. For more information about using container images with Lambda, see Creating Lambda container images in the Lambda Developer Guide.

> **Tip**
>
> To learn how to build **serverless solutions**, check out the Serverless Developer Guide.

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Create a Lambda durable function with the console

In this example, your durable function processes an order through multiple steps with automatic checkpointing. The function takes a JSON object containing an order ID, validates the order, processes payment, and confirms the order. Each step is automatically checkpointed, so if the function is interrupted, it resumes from the last completed step.

Your function also demonstrates a wait operation, pausing execution for a short period to simulate waiting for external confirmation.

To create a durable function with the console

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Select **Author from scratch**.
4. In the **Basic information** pane, for **Function name**, enter **`myDurableFunction`**.
5. For **Runtime**, choose either **Node.js 24** or **Python 3.14**.
6. Select **Enable durable execution**.

Lambda creates your durable function with an execution role that includes permissions for checkpoint operations (`lambda:CheckpointDurableExecution` and `lambda:GetDurableExecutionState`).

> **Note**
>
> Lambda runtimes include the Durable Execution SDK, so you can test durable functions without packaging dependencies. However, we recommend including the SDK in your deployment package for production. This ensures version consistency and avoids potential runtime updates that might affect your function.

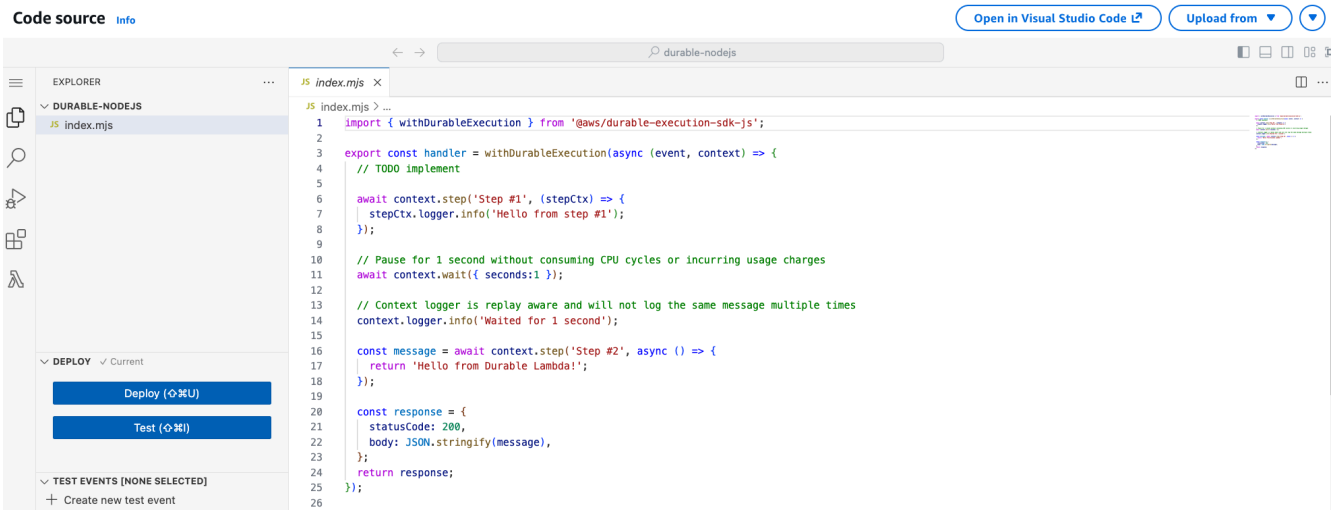
Use the console's built-in code editor to add your durable function code.

Node.js

To modify the code in the console

1. Choose the **Code** tab.

   In the console's built-in code editor, you should see the function code that Lambda created. If you don't see the **index.mjs** tab in the code editor, select **index.mjs** in the file explorer as shown on the following diagram.



2. Paste the following code into the **index.mjs** tab, replacing the code that Lambda created.

```
import {
  withDurableExecution,
} from "@aws/durable-execution-sdk-js";

export const handler = withDurableExecution(
  async (event, context) => {
    const orderId = event.orderId;
```

```
    // Step 1: Validate order
    const validationResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Validating order ${orderId}`);
      return { orderId, status: "validated" };
    });

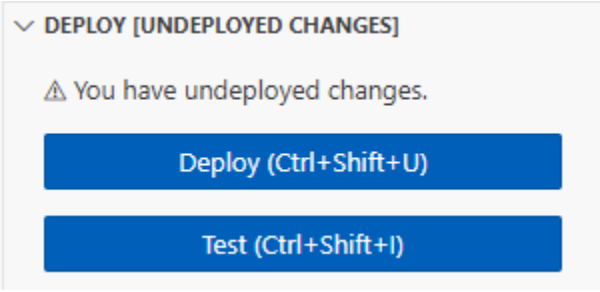
    // Step 2: Process payment
    const paymentResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Processing payment for order ${orderId}`);
      return { orderId, status: "paid", amount: 99.99 };
    });

    // Wait for 10 seconds to simulate external confirmation
    await context.wait({ seconds: 10 });

    // Step 3: Confirm order
    const confirmationResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Confirming order ${orderId}`);
      return { orderId, status: "confirmed" };
    });

    return {
      orderId: orderId,
      status: "completed",
      steps: [validationResult, paymentResult, confirmationResult]
    };
  }
);
```

3. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Understanding your durable function code

Before you move to the next step, let's look at the function code and understand key durable function concepts.

- The `withDurableExecution` wrapper:

  Your durable function is wrapped with `withDurableExecution`. This wrapper enables durable execution by providing the `DurableContext` object and managing checkpoint operations.

- The `DurableContext` object:

  Instead of the standard Lambda context, your function receives a `DurableContext`. This object provides methods for durable operations like `step()` and `wait()` that create checkpoints.

- Steps and checkpoints:

  Each `context.step()` call creates a checkpoint before and after execution. If your function is interrupted, it resumes from the last completed checkpoint. The function doesn't re-execute completed steps. It uses their stored results instead.

- Wait operations:

  The `context.wait()` call pauses execution without consuming compute resources. When the wait completes, Lambda invokes your function again and replays the checkpoint log, substituting stored values for completed steps.

- Replay mechanism:

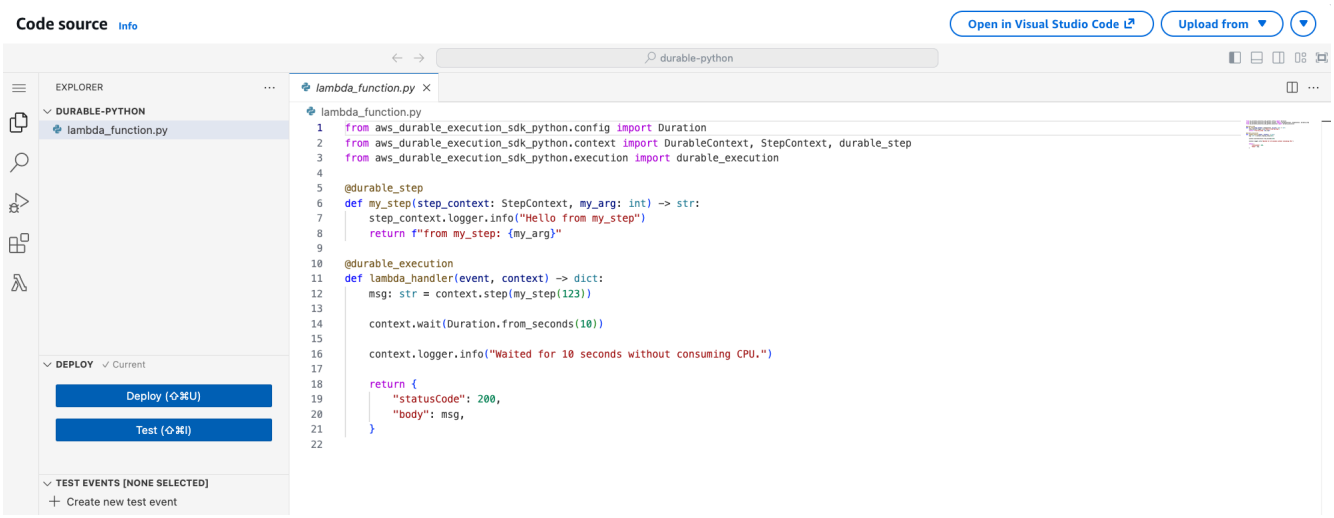
  When your function resumes after a wait or interruption, Lambda runs your code from the beginning. However, completed steps don't re-execute. Lambda replays their results from the checkpoint log. This is why your code must be deterministic.

Python

To modify the code in the console

1. Choose the **Code** tab.

   In the console's built-in code editor, you should see the function code that Lambda created. If you don't see the **lambda_function.py** tab in the code editor, select **lambda_function.py** in the file explorer as shown on the following diagram.



2. Paste the following code into the **lambda_function.py** tab, replacing the code that Lambda created.

```
from aws_durable_execution_sdk_python import (
    DurableContext,
    durable_execution,
    durable_step,
)
from aws_durable_execution_sdk_python.config import Duration

@durable_step
def validate_order(step_context, order_id):
    step_context.logger.info(f"Validating order {order_id}")
    return {"orderId": order_id, "status": "validated"}

@durable_step
def process_payment(step_context, order_id):
    step_context.logger.info(f"Processing payment for order {order_id}")
    return {"orderId": order_id, "status": "paid", "amount": 99.99}

@durable_step
def confirm_order(step_context, order_id):
    step_context.logger.info(f"Confirming order {order_id}")
    return {"orderId": order_id, "status": "confirmed"}

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']

    # Step 1: Validate order
```

```
    validation_result = context.step(validate_order(order_id))

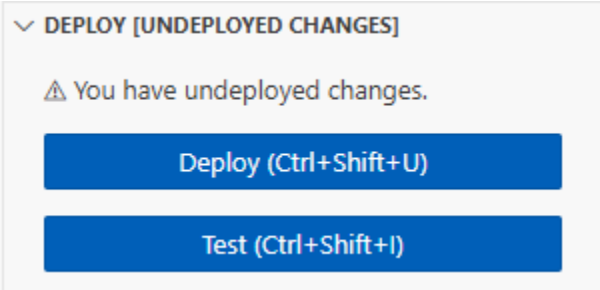
    # Step 2: Process payment
    payment_result = context.step(process_payment(order_id))

    # Wait for 10 seconds to simulate external confirmation
    context.wait(Duration.from_seconds(10))

    # Step 3: Confirm order
    confirmation_result = context.step(confirm_order(order_id))

    return {
        "orderId": order_id,
        "status": "completed",
        "steps": [validation_result, payment_result, confirmation_result]
    }
```

3. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Understanding your durable function code

Before you move to the next step, let's look at the function code and understand key durable function concepts.

- The `@durable_execution` decorator:

  Your handler function is decorated with `@durable_execution`. This decorator enables durable execution by providing the `DurableContext` object and managing checkpoint operations.
- The `@durable_step` decorator:

  Each step function is decorated with `@durable_step`. This decorator marks the function as a durable step that creates checkpoints.
- The `DurableContext` object:

Instead of the standard Lambda context, your function receives a `DurableContext`. This object provides methods for durable operations like `step()` and `wait()` that create checkpoints.

- Steps and checkpoints:

  Each `context.step()` call creates a checkpoint before and after execution. If your function is interrupted, it resumes from the last completed checkpoint. The function doesn't re-execute completed steps. It uses their stored results instead.

- Wait operations:

  The `context.wait()` call pauses execution without consuming compute resources. When the wait completes, Lambda invokes your function again and replays the checkpoint log, substituting stored values for completed steps.

- Python SDK is synchronous:

  Note that the Python SDK doesn't use `await`. All durable operations are synchronous method calls.

Invoke the durable function using the console code editor

When no explicit version is specified (or published), the console invokes the durable function using the $LATEST version qualifier. However, for deterministic execution of your code, you must always use a qualified ARN pointing to a stable version.

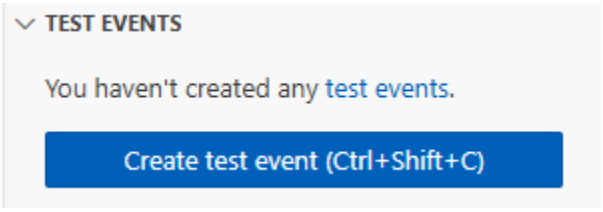
To publish a version of your function

1. Choose the **Versions** tab.
2. Choose **Publish new version**.
3. For **Version description**, enter **`Initial version`** (optional).
4. Choose **Publish**.
5. Lambda creates version 1 of your function. Note that the function ARN now includes `:1` at the end, indicating this is version 1.

Now create a test event to send to your function. The event is a JSON formatted document containing an order ID.

To create the test event

1. In the **TEST EVENTS** section of the console code editor, choose **Create test event**.



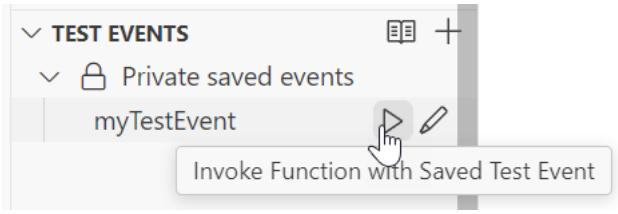
2. For **Event Name**, enter **`myTestEvent`**.
3. In the **Event JSON** section, replace the default JSON with the following:

```
{
  "orderId": "order-12345"
}
```

4. Choose **Save**.

To test your durable function and view execution

In the **TEST EVENTS** section of the console code editor, choose the run icon next to your test event:



Your durable function starts executing. Because it includes a 10-second wait, the initial invocation completes quickly, and the function resumes after the wait period. You can view the execution progress in the **Durable executions** tab.

To view your durable function execution

1. Choose the **Durable executions** tab.
2. Find your execution in the list. The execution shows the current status (Running, Succeeded, or Failed).
3. Choose the execution ID to view details, including:

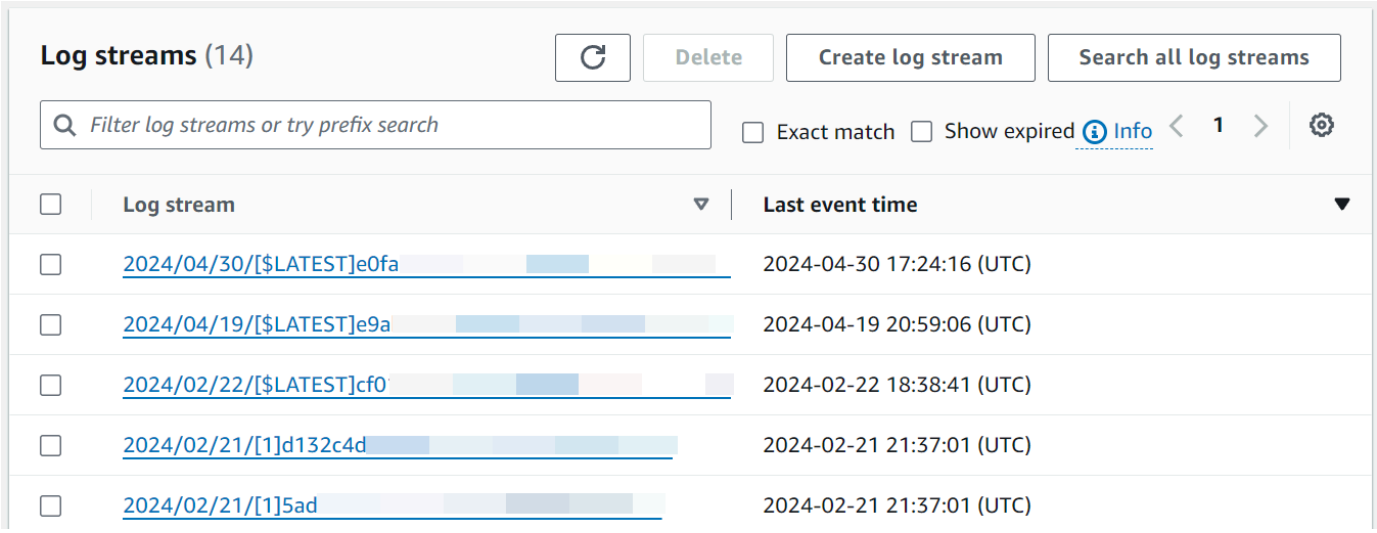
   - Execution timeline showing when each step completed
   - Checkpoint history

- Wait periods
- Step results

You can also view your function's logs in CloudWatch Logs to see the console output from each step.

To view your function's invocation records in CloudWatch Logs

1. Open the Log groups page of the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/myDurableFunction`).
3. Scroll down and choose the **Log stream** for the function invocations you want to look at.



You should see log entries for each invocation of your function, including the initial execution and the replay after the wait.

> **Note**
>
> When you use the logger from the `DurableContext` (such as `context.logger` or `stepContext.logger`), logs also appear in the durable execution and step views in the Lambda console. These logs may take a moment to load.

Clean up

When you're finished working with the example durable function, delete it. You can also delete the log group that stores the function's logs, and the execution role that the console created.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the log group

1. Open the Log groups page of the CloudWatch console.
2. Select the function's log group (`/aws/lambda/myDurableFunction`).
3. Choose **Actions**, **Delete log group(s)**.
4. In the **Delete log group(s)** dialog box, choose **Delete**.

To delete the execution role

1. Open the Roles page of the AWS Identity and Access Management (IAM) console.
2. Select the function's execution role (for example, `myDurableFunction-role-31exxmpl`).
3. Choose **Delete**.
4. In the **Delete role** dialog box, enter the role name, and then choose **Delete**.

Additional resources and next steps

Now that you've created and tested a simple durable function using the console, take these next steps:

- Learn about common use cases for durable functions, including distributed transactions, order processing, and human review workflows. See Examples.
- Understand how to monitor durable function executions with CloudWatch metrics and execution history. See Monitoring and debugging.

- Learn about invoking durable functions synchronously and asynchronously, and managing long-running executions. See Invoking durable functions.
- Follow best practices for writing deterministic code, managing checkpoint sizes, and optimizing costs. See Best practices.
- Learn how to test durable functions locally and in the cloud. See Testing durable functions.
- Compare durable functions with Step Functions to understand when each approach is most effective. See Durable functions or Step Functions.

Deploy and invoke Lambda durable functions with the AWS CLI

Use the AWS CLI to create and deploy Lambda durable functions with imperative commands. This approach gives you direct control over each step of the deployment process.

Prerequisites

- Install and configure the AWS CLI. For instructions, see Installing the AWS CLI.
- Create a deployment package with your function code and the durable execution SDK.
- Create an IAM execution role with checkpoint permissions.

Create the execution role

Create an IAM role with permissions for basic Lambda execution and checkpoint operations.

To create the execution role

1. Create a trust policy document that allows Lambda to assume the role. Save this as `trust-policy.json`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
```

```
  ]
}
```

2. Create the role:

```
aws iam create-role \
  --role-name durable-function-role \
  --assume-role-policy-document file://trust-policy.json
```

3. Attach the durable execution policy for checkpoint operations and basic execution:

```
aws iam attach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy
```

The `AWSLambdaBasicDurableExecutionRolePolicy` managed policy includes the required permissions for checkpoint operations (`lambda:CheckpointDurableExecution` and `lambda:GetDurableExecutionState`) and basic Lambda execution.

Create the durable function

Create your durable function with the `--durable-config` parameter.

To create a durable function

1. Package your function code with dependencies into a .zip file:

```
zip -r function.zip index.mjs node_modules/
```

2. Create the function with durable execution enabled:

```
aws lambda create-function \
  --function-name myDurableFunction \
  --runtime nodejs22.x \
  --role arn:aws:iam::123456789012:role/durable-function-role \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'
```

Note

You can only enable durable execution when creating the function. You cannot enable it on existing functions.

Note

Currently Durable functions in Java can only be created through container images. For more information on creating durable functions from container images, see Supported runtimes for durable functions.

Publish a version

While durable functions can be invoked using the $LATEST version qualifier, you must always use a qualified ARN pointing to a stable version to ensure deterministic execution of your code.

```
aws lambda publish-version \
  --function-name myDurableFunction \
  --description "Initial version"
```

The command returns the version ARN. Note the version number (for example, :1) at the end of the ARN.

Optionally, create an alias that points to the version:

```
aws lambda create-alias \
  --function-name myDurableFunction \
  --name prod \
  --function-version 1
```

Invoke the durable function

Invoke your durable function using the qualified ARN (version or alias).

> **Note**
>
> **Idempotent invocations:** To prevent duplicate executions when retrying failed invocations, you can provide an execution name that ensures at-most-once execution semantics. See Idempotency for details.

Synchronous invocation

For executions that complete within 15 minutes, use synchronous invocation:

```
aws lambda invoke \
  --function-name myDurableFunction:1 \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Or using an alias:

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

Asynchronous invocation

For long-running executions, use asynchronous invocation:

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --invocation-type Event \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

With asynchronous invocation, Lambda returns immediately. The function continues executing in the background.

> **Note**
>
> You can use $LATEST for prototyping and testing in the console. For production workloads, use a published version or alias.

Manage durable executions

Use the following commands to manage and monitor durable function executions.

List executions

List all executions for a durable function:

```
aws lambda list-durable-executions-by-function \
  --function-name myDurableFunction
```

Get execution details

Get details about a specific execution:

```
aws lambda get-durable-execution \
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

Get execution history

View the checkpoint history for an execution:

```
aws lambda get-durable-execution-history \
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

Stop an execution

Stop a running durable execution:

```
aws lambda stop-durable-execution \
```

```
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

Update function code

Update your durable function code and publish a new version:

To update and publish a new version

1. Update the function code:

```
aws lambda update-function-code \
  --function-name myDurableFunction \
  --zip-file fileb://function.zip
```

2. Wait for the update to complete:

```
aws lambda wait function-updated \
  --function-name myDurableFunction
```

3. Publish a new version:

```
aws lambda publish-version \
  --function-name myDurableFunction \
  --description "Updated order processing logic"
```

4. Update the alias to point to the new version:

```
aws lambda update-alias \
  --function-name myDurableFunction \
  --name prod \
  --function-version 2
```

Important

Running executions continue using the version they started with. New invocations use the updated alias version.

View function logs

View your durable function's logs in CloudWatch Logs:

```
aws logs tail /aws/lambda/myDurableFunction --follow
```

Filter logs for a specific execution:

```
aws logs filter-log-events \
  --log-group-name /aws/lambda/myDurableFunction \
  --filter-pattern "exec-abc123"
```

Clean up resources

Delete your durable function and associated resources:

```
# Delete the function
aws lambda delete-function --function-name myDurableFunction

# Delete the IAM role policies
aws iam detach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole

aws iam detach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy

# Delete the role
aws iam delete-role --role-name durable-function-role
```

Next steps

After deploying your durable function with the AWS CLI:

- Monitor executions using the `list-durable-executions-by-function` and `get-durable-execution` commands
- View checkpoint operations in AWS CloudTrail data events
- Set up CloudWatch alarms for execution failures or long-running executions
- Automate deployments using shell scripts or CI/CD pipelines

For more information about AWS CLI commands for Lambda, see the AWS CLI Command Reference.

Deploy Lambda durable functions with Infrastructure as Code

You can deploy Lambda durable functions using Infrastructure as Code (IaC) tools like AWS CloudFormation, AWS CDK, AWS Serverless Application Model, or Terraform. These tools let you define your function, execution role, and permissions in code, making deployments repeatable and version-controlled.

All three tools require you to:

- Enable durable execution on the function
- Grant checkpoint permissions to the execution role
- Publish a version or create an alias (durable functions require qualified ARNs)

Durable functions from a ZIP

AWS CloudFormation

Use CloudFormation to define your durable function in a template. The following example creates a durable function with the required permissions.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda durable function example

Resources:
  DurableFunctionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy

  DurableFunction:
```

```
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs22.x
      Handler: index.handler
      Role: !GetAtt DurableFunctionRole.Arn
      Code:
        ZipFile: |
          // Your durable function code here
          export const handler = async (event, context) => {
            return { statusCode: 200 };
          };
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7

  DurableFunctionVersion:
    Type: AWS::Lambda::Version
    Properties:
      FunctionName: !Ref DurableFunction
      Description: Initial version

  DurableFunctionAlias:
    Type: AWS::Lambda::Alias
    Properties:
      FunctionName: !Ref DurableFunction
      FunctionVersion: !GetAtt DurableFunctionVersion.Version
      Name: prod

Outputs:
  FunctionArn:
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunctionAlias
```

To deploy the template

```
aws cloudformation deploy \
  --template-file template.yaml \
  --stack-name my-durable-function-stack \
  --capabilities CAPABILITY_IAM
```

AWS CDK

AWS CDK lets you define infrastructure using programming languages. The following examples show how to create a durable function using TypeScript and Python.

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as iam from 'aws-cdk-lib/aws-iam';
import { Construct } from 'constructs';

export class DurableFunctionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the durable function
    const durableFunction = new lambda.Function(this, 'DurableFunction', {
      runtime: lambda.Runtime.NODEJS_22_X,
      handler: 'index.handler',
      code: lambda.Code.fromAsset('lambda'),
      functionName: 'myDurableFunction',
      durableConfig: { executionTimeout: Duration.hours(1), retentionPeriod:
 Duration.days(30) },
    });

    // Create version and alias
    const version = durableFunction.currentVersion;
    const alias = new lambda.Alias(this, 'ProdAlias', {
      aliasName: 'prod',
      version: version,
    });

    // Output the alias ARN
    new cdk.CfnOutput(this, 'FunctionAliasArn', {
      value: alias.functionArn,
      description: 'Use this ARN to invoke the durable function',
    });
  }
}
```

Python

```
from aws_cdk import (
    Stack,
    aws_lambda as lambda_,
    aws_iam as iam,
    CfnOutput,
)
from constructs import Construct

class DurableFunctionStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        # Create the durable function
        durable_function = lambda_.Function(
            self, 'DurableFunction',
            runtime=lambda_.Runtime.NODEJS_22_X,
            handler='index.handler',
            code=lambda_.Code.from_asset('lambda'),
            function_name='myDurableFunction',
            durable_execution={execution_timeout: Duration.hours(1),
 retention_period: Duration.days(30)}
        )

        # Add durable execution managed policy for checkpoint permissions
        durable_function.role.add_managed_policy(
            iam.ManagedPolicy.from_aws_managed_policy_name('service-role/
AWSLambdaBasicDurableExecutionRolePolicy')
        )

        # Create version and alias
        version = durable_function.current_version
        alias = lambda_.Alias(
            self, 'ProdAlias',
            alias_name='prod',
            version=version
        )

        # Output the alias ARN
        CfnOutput(
            self, 'FunctionAliasArn',
            value=alias.function_arn,
            description='Use this ARN to invoke the durable function'
```

```
        )
```

To deploy the CDK stack

```
cdk deploy
```

AWS Serverless Application Model

AWS SAM simplifies CloudFormation templates for serverless applications. The following template creates a durable function with AWS SAM.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda durable function with SAM

Resources:
  DurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs22.x
      Handler: index.handler
      CodeUri: ./src
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7
      Policies:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy
      AutoPublishAlias: prod

Outputs:
  FunctionArn:
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunction.Alias
```

To deploy the SAM template

```
sam build
```

```
sam deploy --guided
```

Terraform

Terraform is a popular open-source IaC tool that supports AWS resources. The following example creates a durable function with Terraform using the AWS provider version 6.25.0 or later.

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 6.25.0"
    }
  }
}

provider "aws" {
  region = "us-east-2"
}

# IAM Role for Lambda Function
resource "aws_iam_role" "lambda_role" {
  name = "durable-function-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = {
        Service = "lambda.amazonaws.com"
      }
    }]
  })
}

# Attach durable execution policy for checkpoint operations
resource "aws_iam_role_policy_attachment" "lambda_durable" {
  policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy"
  role       = aws_iam_role.lambda_role.name
}
```

```
# Lambda Function with Durable Execution enabled
resource "aws_lambda_function" "durable_function" {
  filename      = "function.zip"
  function_name = "myDurableFunction"
  role          = aws_iam_role.lambda_role.arn
  handler       = "index.handler"
  runtime       = "nodejs22.x"
  timeout       = 30
  memory_size   = 512

  durable_config {
    execution_timeout = 900
    retention_period  = 7
  }
}

# Publish a version
resource "aws_lambda_alias" "prod" {
  name             = "prod"
  function_name    = aws_lambda_function.durable_function.function_name
  function_version = aws_lambda_function.durable_function.version
}

output "function_arn" {
  description = "ARN of the Lambda function"
  value       = aws_lambda_function.durable_function.arn
}

output "alias_arn" {
  description = "ARN of the function alias (use this for invocations)"
  value       = aws_lambda_alias.prod.arn
}
```

To deploy with Terraform

```
terraform init
terraform plan
terraform apply
```

> **Note**
>
> Terraform support for Lambda durable functions requires AWS provider version 6.25.0 or later. Update your provider version if you're using an older version.

Durable functions from an OCI container image

You can also create Durable functions based off of container images. For instructions on how to build a container image, see Supported runtimes for durable functions.

AWS CDK

AWS CDK lets you define infrastructure using programming languages. The following examples show how to create a durable function using TypeScript from a container image.

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as iam from 'aws-cdk-lib/aws-iam';
import { Construct } from 'constructs';

export class DurableFunctionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the durable function
    const durableFunction = new lambda.DockerImageFunction(this, 'DurableFunction', {
      code: lambda.DockerImageCode.fromImageAsset('./lambda', {
        platform: cdk.aws_ecr_assets.Platform.LINUX_AMD64,
      }),
      functionName: 'myDurableFunction',
      memorySize: 512,
      timeout: cdk.Duration.seconds(30),
      durableConfig: { executionTimeout: cdk.Duration.hours(1), retentionPeriod:
 cdk.Duration.days(30) },
    });

    // Create version and alias
    const version = durableFunction.currentVersion;
    const alias = new lambda.Alias(this, 'ProdAlias', {
      aliasName: 'prod',
      version: version,
```

```
    });

    // Output the alias ARN
    new cdk.CfnOutput(this, 'FunctionAliasArn', {
      value: alias.functionArn,
      description: 'Use this ARN to invoke the durable function',
    });
  }
}
```

To deploy the CDK stack

```
cdk deploy
```

AWS Serverless Application Model

AWS SAM simplifies CloudFormation templates for serverless applications. The following template creates a durable function with AWS SAM.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda durable function with SAM

Resources:
  DurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: myDurableFunction
      PackageType: Image
      ImageUri: ./src
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7
      Policies:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy
      AutoPublishAlias: prod
    Metadata:
      DockerTag: latest
      DockerContext: ./src
      Dockerfile: Dockerfile

Outputs:
  FunctionArn:
```

```
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunction.Alias
```

To deploy the SAM template

```
sam build
sam deploy --guided
```

Common configuration patterns

Regardless of which IaC tool you use, follow these patterns for durable functions:

Enable durable execution

Set the `DurableConfig` property on your function to enable durable execution. This property is only available when creating the function. You cannot enable durable execution on existing functions.

Grant checkpoint permissions

Attach the `AWSLambdaBasicDurableExecutionRolePolicy` managed policy to the execution role. This policy includes the required `lambda:CheckpointDurableExecution` and `lambda:GetDurableExecutionState` permissions.

Use qualified ARNs

Create a version or alias for your function. Durable functions require qualified ARNs (with version or alias) for invocation. Use `AutoPublishAlias` in AWS SAM or create explicit versions in CloudFormation, AWS CDK, and Terraform.

Package dependencies

Include the durable execution SDK in your deployment package. For Node.js, install `@aws/durable-execution-sdk-js`. For Python, install `aws-durable-execution-sdk-python`.

Next steps

After deploying your durable function:

- Test your function using the qualified ARN (version or alias)

- Monitor execution progress in the Lambda console under the Durable executions tab
- View checkpoint operations in AWS CloudTrail data events
- Review CloudWatch Logs for function output and replay behavior

For more information about deploying Lambda functions with IaC tools, see:

- CloudFormation AWS::Lambda::Function reference
- AWS CDK Lambda module documentation
- AWS SAM Developer Guide

Configure Lambda durable functions

Durable execution settings control how long your Lambda function can run and how long the service retains execution history. Configure these settings to enable durable execution for your function.

Enable durable execution

Configure the `DurableConfig` object when creating your function to set execution timeout and history retention. You can only enable durable execution when creating a function. You cannot enable it on existing functions.

AWS CLI

```
aws lambda create-function \
  --function-name my-durable-function \
  --runtime nodejs24.x \
  --role arn:aws:iam::123456789012:role/my-durable-role \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 30}'
```

CloudFormation

```
Resources:
```

```
MyDurableFunction:
  Type: AWS::Lambda::Function
  Properties:
    FunctionName: my-durable-function
    Runtime: nodejs24.x
    Handler: index.handler
    Code:
      ZipFile: |
        // Your durable function code
    DurableConfig:
      ExecutionTimeout: 3600
      RetentionPeriodInDays: 30
```

Configuration parameters:

- `ExecutionTimeout` – The maximum time in seconds that a durable execution can run before Lambda stops the execution. This timeout applies to the entire durable execution, not individual function invocations. Valid range: 1–31622400.
- `RetentionPeriodInDays` – The number of days to retain execution history after a durable execution completes. After this period, execution history is no longer available through the `GetDurableExecutionHistory` API. Valid range: 1–90.

For the full API reference, see DurableConfig in the Lambda API Reference.

Configuration best practices

Follow these best practices when configuring durable functions for production use:

- **Set appropriate execution timeouts** – Configure `ExecutionTimeout` based on your workflow's maximum expected duration. Do not set unnecessarily long timeouts as they affect cost and resource allocation.
- **Balance retention with storage costs** – Set `RetentionPeriodInDays` based on your debugging and audit requirements. Longer retention periods increase storage costs.
- **Monitor state size** – Large state objects increase storage costs and can impact performance. Keep state minimal and use external storage for large data.
- **Configure appropriate logging** – Enable detailed logging for troubleshooting long-running workflows, but consider the impact on log volume and costs.

Production configuration example:

```
{
  "ExecutionTimeout": 86400,
  "RetentionPeriodInDays": 7
}
```

This example sets a 24-hour (86,400 seconds) execution timeout with a 7-day retention period, which balances debugging visibility with storage costs for most production workloads.

Durable functions or Step Functions

Both Lambda durable functions and AWS Step Functions enable reliable workflow orchestration with automatic state management and failure recovery. They serve different developer preferences and architectural patterns. Durable functions are optimized for application development within Lambda, while Step Functions is built for workflow orchestration across AWS services.

When to use durable functions

Use durable functions when:

- Your team prefers standard programming languages and familiar development tools
- Your application logic is primarily within Lambda functions
- You want fine-grained control over execution state in code
- You're building Lambda-centric applications with tight coupling between workflow and business logic
- You want to iterate quickly without switching between code and visual/JSON designers

When to use Step Functions

Use Step Functions when:

- You need visual workflow representation for cross-team visibility
- You're orchestrating multiple AWS services and want native integrations without custom SDK code
- You require zero-maintenance infrastructure (no patching, runtime updates)

- Non-technical stakeholders need to understand and validate workflow logic

Decision framework

Use the following questions to determine which service fits your use case:

- **What's your primary focus?** Application development in Lambda → durable functions. Workflow orchestration across AWS → Step Functions.
- **What's your preferred programming model?** Standard programming languages → durable functions. Graph-based DSL or visual designer → Step Functions.
- **How many AWS services are involved?** Primarily Lambda → durable functions. Multiple AWS services → Step Functions.
- **What development tools do you use?** Lambda developer experience, IDE with LLM agent, programming language-specific unit test frameworks, AWS SAM, AWS CDK, AWS Toolkit → durable functions. Visual workflow builder, AWS CDK to model workflows → Step Functions.
- **Who manages the infrastructure?** Want flexibility within Lambda → durable functions. Want fully managed, zero-maintenance → Step Functions.

Feature comparison

The following table compares key features between Step Functions and Lambda durable functions:

Feature	AWS Step Functions	Lambda durable functions
Primary focus	Workflow orchestration across AWS	Application development in Lambda
Service type	Standalone, dedicated workflow service	Runs within Lambda
Programming model	Graph-based, Amazon States Language DSL or AWS CDK	Standard programming languages (JavaScript/TypeScript, Python)
Development tools	Visual builder in Console / AWS Toolkit IDE extension, AWS CDK	Lambda DX within IDE and LLM agents, unit test frameworks, AWS SAM, AWS Toolkit IDE extension

Feature	AWS Step Functions	Lambda durable functions
Integrations	220+ AWS services, 16k APIs	Lambda event-driven programming model extension (event sources)
Management	Fully managed, runtime agnostic, zero maintenance (no patching, runtime updates)	Managed within Lambda environment
Best for	Business process and IT automation, data processing, AI workflows	Distributed transactions, stateful application logic, function orchestration, data processing, AI workflows

Hybrid architectures

Many applications benefit from using both services. A common pattern is using durable functions for application-level logic within Lambda, while Step Functions coordinates high-level workflows across multiple AWS services beyond Lambda functions.

Migration considerations

Starting simple, evolving complex: Begin with durable functions for Lambda-centric workflows. Add Step Functions when you need multi-service orchestration or visual workflow design.

Existing Step Functions users: Keep Step Functions for established cross-service workflows. Consider durable functions for new Lambda application logic that needs reliability.

Related resources

- Lambda durable functions
- Orchestrating Lambda functions with Step Functions
- Getting started with durable functions

Examples and use cases

Lambda durable functions enable you to build fault-tolerant, multi-step applications using durable operations like steps and waits. With automatic checkpointing and a checkpoint-replay model. where execution restarts from the beginning after failure but skips completed checkpoints, your functions can recover from failures and resume execution without losing progress.

Short-lived fault-tolerant processes

Use durable functions to build reliable operations that typically complete within minutes. While these processes are shorter than long-running workflows, they still benefit from automatic checkpointing and fault tolerance across distributed systems. Durable functions help ensure your multi-step processes complete successfully even when individual service calls fail, without requiring complex error handling or state management code.

Common scenarios include hotel booking systems, restaurant reservation platforms, ride-sharing trip requests, event ticket purchases, and SaaS subscription upgrades. These scenarios share common characteristics: multiple service calls that must complete together, the need for automatic retry on transient failures, and the requirement to maintain consistent state across distributed systems.

Distributed transactions across microservices

Coordinate payments, inventory, and shipping across multiple services with automatic rollback on failures. Each service operation is wrapped in a step, ensuring the transaction can recover from any point if a service fails.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, amount, items } = event;

    // Reserve inventory across multiple warehouses
    const inventory = await context.step("reserve-inventory", async () => {
      return await inventoryService.reserve(items);
```

```
    });

    // Process payment
    const payment = await context.step("process-payment", async () => {
      return await paymentService.charge(amount);
    });

    // Create shipment
    const shipment = await context.step("create-shipment", async () => {
      return await shippingService.createShipment(orderId, inventory);
    });

    return { orderId, status: 'completed', shipment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    amount = event['amount']
    items = event['items']

    # Reserve inventory across multiple warehouses
    inventory = context.step(
        lambda _: inventory_service.reserve(items),
        name='reserve-inventory'
    )

    # Process payment
    payment = context.step(
        lambda _: payment_service.charge(amount),
        name='process-payment'
    )

    # Create shipment
    shipment = context.step(
        lambda _: shipping_service.create_shipment(order_id, inventory),
```

```
        name='create-shipment'
    )

    return {'orderId': order_id, 'status': 'completed', 'shipment': shipment}
```

If any step fails, the function automatically retries from the last successful checkpoint. The inventory reservation persists even if payment processing fails temporarily. When the function retries, it skips the completed inventory step and proceeds directly to payment processing. This eliminates duplicate reservations and ensures consistent state across your distributed system.

Order processing with multiple steps

Process orders through validation, payment authorization, inventory allocation, and fulfillment with automatic retry and recovery. Each step is checkpointed, ensuring the order progresses even if individual steps fail and retry.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, customerId, items } = event;

    // Validate order details
    const validation = await context.step("validate-order", async () => {
      const customer = await customerService.validate(customerId);
      const itemsValid = await inventoryService.validateItems(items);
      return { customer, itemsValid };
    });

    if (!validation.itemsValid) {
      return { orderId, status: 'rejected', reason: 'invalid_items' };
    }

    // Authorize payment
    const authorization = await context.step("authorize-payment", async () => {
      return await paymentService.authorize(
        validation.customer.paymentMethod,
```

```
        calculateTotal(items)
      );
    });

    // Allocate inventory
    const allocation = await context.step("allocate-inventory", async () => {
      return await inventoryService.allocate(items);
    });

    // Fulfill order
    const fulfillment = await context.step("fulfill-order", async () => {
      return await fulfillmentService.createShipment({
        orderId,
        items: allocation.allocatedItems,
        address: validation.customer.shippingAddress
      });
    });

    return {
      orderId,
      status: 'completed',
      trackingNumber: fulfillment.trackingNumber
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    customer_id = event['customerId']
    items = event['items']

    # Validate order details
    def validate_order(_):
        customer = customer_service.validate(customer_id)
        items_valid = inventory_service.validate_items(items)
        return {'customer': customer, 'itemsValid': items_valid}
```

```
    validation = context.step(validate_order, name='validate-order')

    if not validation['itemsValid']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
'invalid_items'}

    # Authorize payment
    authorization = context.step(
        lambda _: payment_service.authorize(
            validation['customer']['paymentMethod'],
            calculate_total(items)
        ),
        name='authorize-payment'
    )

    # Allocate inventory
    allocation = context.step(
        lambda _: inventory_service.allocate(items),
        name='allocate-inventory'
    )

    # Fulfill order
    fulfillment = context.step(
        lambda _: fulfillment_service.create_shipment({
            'orderId': order_id,
            'items': allocation['allocatedItems'],
            'address': validation['customer']['shippingAddress']
        }),
        name='fulfill-order'
    )

    return {
        'orderId': order_id,
        'status': 'completed',
        'trackingNumber': fulfillment['trackingNumber']
    }
```

This pattern ensures orders never get stuck in intermediate states. If validation fails, the order is rejected before payment authorization. If payment authorization fails, inventory isn't allocated. Each step builds on the previous one with automatic retry and recovery.

Note

The conditional check `if (!validation.itemsValid)` is outside a step and will re-execute during replay. This is safe because it's deterministic—it always produces the same result given the same validation object.

Long-running processes

Use durable functions for processes that span hours, days, or weeks. Wait operations suspend execution without incurring compute charges, making long-running processes cost-effective. During wait periods, your function stops running and Lambda recycles the execution environment. When it's time to resume, Lambda invokes your function again and replays from the last checkpoint.

This execution model makes durable functions ideal for processes that need to pause for extended periods, whether waiting for human decisions, external system responses, scheduled processing windows, or time-based delays. You pay only for active compute time, not for waiting.

Common scenarios include document approval processes, scheduled batch processing, multi-day onboarding processes, subscription trial processes, and delayed notification systems. These scenarios share common characteristics: extended wait periods measured in hours or days, the need to maintain execution state across those waits, and cost-sensitive requirements where paying for idle compute time is prohibitive.

Human-in-the-loop approvals

Pause execution for document reviews, approvals, or decisions while maintaining execution state. The function waits for external callbacks without consuming resources, resuming automatically when approval is received.

This pattern is essential for processes that require human judgment or external validation. The function suspends at the callback point, incurring no compute charges while waiting. When someone submits their decision via API, Lambda invokes your function again and replays from the checkpoint, continuing with the approval result.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";
```

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { documentId, reviewers } = event;

    // Step 1: Prepare document for review
    const prepared = await context.step("prepare-document", async () => {
      return await documentService.prepare(documentId);
    });

    // Step 2: Request approval with callback
    const approval = await context.waitForCallback(
      "approval-callback",
      async (callbackId) => {
        await notificationService.sendApprovalRequest({
          documentId,
          reviewers,
          callbackId,
          expiresIn: 86400
        });
      },
      {
        timeout: { seconds: 86400 }
      }
    );

    // Function resumes here when approval is received
    if (approval?.approved) {
      const finalized = await context.step("finalize-document", async () => {
        return await documentService.finalize(documentId, approval.comments);
      });

      return {
        status: 'approved',
        documentId,
        finalizedAt: finalized.timestamp
      };
    }

    // Handle rejection
    await context.step("archive-rejected", async () => {
      await documentService.archive(documentId, approval?.reason);
    });

    return {
```

```
      status: 'rejected',
      documentId,
      reason: approval?.reason
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 WaitConfig

@durable_execution
def lambda_handler(event, context: DurableContext):
    document_id = event['documentId']
    reviewers = event['reviewers']

    # Step 1: Prepare document for review
    prepared = context.step(
        lambda _: document_service.prepare(document_id),
        name='prepare-document'
    )

    # Step 2: Request approval with callback
    def send_approval_request(callback_id):
        notification_service.send_approval_request({
            'documentId': document_id,
            'reviewers': reviewers,
            'callbackId': callback_id,
            'expiresIn': 86400
        })

    approval = context.wait_for_callback(
        send_approval_request,
        name='approval-callback',
        config=WaitConfig(timeout=86400)
    )

    # Function resumes here when approval is received
    if approval and approval.get('approved'):
        finalized = context.step(
```

```
            lambda _: document_service.finalize(document_id,
approval.get('comments')),
            name='finalize-document'
        )

        return {
            'status': 'approved',
            'documentId': document_id,
            'finalizedAt': finalized['timestamp']
        }

    # Handle rejection
    context.step(
        lambda _: document_service.archive(document_id, approval.get('reason') if
approval else None),
        name='archive-rejected'
    )

    return {
        'status': 'rejected',
        'documentId': document_id,
        'reason': approval.get('reason') if approval else None
    }
```

When the callback is received and your function resumes, it replays from the beginning. The prepare-document step returns its checkpointed result instantly. The waitForCallback operation also returns instantly with the stored approval result instead of waiting again. Execution then continues to the finalization or archival steps.

Multi-stage data pipelines

Process large datasets through extraction, transformation, and loading phases with checkpoints between stages. Each stage can take hours to complete, and checkpoints enable the pipeline to resume from any stage if interrupted.

This pattern is ideal for ETL workflows, data migrations, or batch processing jobs where you need to process data in stages with recovery points between them. If a stage fails, the pipeline resumes from the last completed stage rather than restarting from the beginning. You can also use wait operations to pause between stages; respecting rate limits, waiting for downstream systems to be ready, or scheduling processing during off-peak hours.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { datasetId, batchSize } = event;

    // Stage 1: Extract data from source
    const extracted = await context.step("extract-data", async () => {
      const records = await sourceDatabase.extractRecords(datasetId);
      return { recordCount: records.length, records };
    });

    // Wait 5 minutes to respect source system rate limits
    await context.wait({ seconds: 300 });

    // Stage 2: Transform data in batches
    const transformed = await context.step("transform-data", async () => {
      const batches = chunkArray(extracted.records, batchSize);
      const results = [];

      for (const batch of batches) {
        const transformed = await transformService.processBatch(batch);
        results.push(transformed);
      }

      return { batchCount: batches.length, results };
    });

    // Wait until off-peak hours (e.g., 2 AM)
    const now = new Date();
    const targetHour = 2;
    const msUntilTarget = calculateMsUntilHour(now, targetHour);
    await context.wait({ seconds: Math.floor(msUntilTarget / 1000) });

    // Stage 3: Load data to destination
    const loaded = await context.step("load-data", async () => {
      let loadedCount = 0;

      for (const result of transformed.results) {
        await destinationDatabase.loadBatch(result);
```

```
        loadedCount += result.length;
      }

      return { loadedCount };
    });

    // Stage 4: Verify and finalize
    const verified = await context.step("verify-pipeline", async () => {
      const verification = await destinationDatabase.verifyRecords(datasetId);
      await pipelineService.markComplete(datasetId, verification);
      return verification;
    });

    return {
      datasetId,
      recordsProcessed: extracted.recordCount,
      batchesProcessed: transformed.batchCount,
      recordsLoaded: loaded.loadedCount,
      verified: verified.success
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution
from datetime import datetime

@durable_execution
def lambda_handler(event, context: DurableContext):
    dataset_id = event['datasetId']
    batch_size = event['batchSize']

    # Stage 1: Extract data from source
    def extract_data(_):
        records = source_database.extract_records(dataset_id)
        return {'recordCount': len(records), 'records': records}

    extracted = context.step(extract_data, name='extract-data')

    # Wait 5 minutes to respect source system rate limits
```

```
    context.wait(Duration.from_seconds(300))

    # Stage 2: Transform data in batches
    def transform_data(_):
        batches = chunk_array(extracted['records'], batch_size)
        results = []

        for batch in batches:
            transformed = transform_service.process_batch(batch)
            results.append(transformed)

        return {'batchCount': len(batches), 'results': results}

    transformed = context.step(transform_data, name='transform-data')

    # Wait until off-peak hours (e.g., 2 AM)
    now = datetime.now()
    target_hour = 2
    ms_until_target = calculate_ms_until_hour(now, target_hour)
    context.wait(ms_until_target // 1000)

    # Stage 3: Load data to destination
    def load_data(_):
        loaded_count = 0

        for result in transformed['results']:
            destination_database.load_batch(result)
            loaded_count += len(result)

        return {'loadedCount': loaded_count}

    loaded = context.step(load_data, name='load-data')

    # Stage 4: Verify and finalize
    def verify_pipeline(_):
        verification = destination_database.verify_records(dataset_id)
        pipeline_service.mark_complete(dataset_id, verification)
        return verification

    verified = context.step(verify_pipeline, name='verify-pipeline')

    return {
        'datasetId': dataset_id,
        'recordsProcessed': extracted['recordCount'],
```

```
        'batchesProcessed': transformed['batchCount'],
        'recordsLoaded': loaded['loadedCount'],
        'verified': verified['success']
    }
```

Each stage is wrapped in a step, creating a checkpoint that allows the pipeline to resume from any stage if interrupted. The 5-minute wait between extract and transform respects source system rate limits without consuming compute resources, while the wait until 2 AM schedules the expensive load operation during off-peak hours.

Note

The `new Date()` call and `calculateMsUntilHour()` function are outside steps and will re-execute during replay. For time-based operations that must be consistent across replays, calculate the timestamp inside a step or use it only for wait durations (which are checkpointed).

Advanced patterns

Use durable functions to build complex multi-step applications that combine multiple durable operations, parallel execution, array processing, conditional logic, and polling. These patterns let you build sophisticated applications that coordinate many tasks while maintaining fault tolerance and automatic recovery.

Advanced patterns go beyond simple sequential steps. You can run operations concurrently with `parallel()`, process arrays with `map()`, wait for external conditions with `waitForCondition()`, and combine these primitives to build reliable applications. Each durable operation creates its own checkpoints, so your application can recover from any point if interrupted.

User onboarding processes

Guide users through registration, email verification, profile setup, and initial configuration with retry handling. This example combines sequential steps, callbacks, and conditional logic to create a complete onboarding process.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";
```

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { userId, email } = event;

    // Step 1: Create user account
    const user = await context.step("create-account", async () => {
      return await userService.createAccount(userId, email);
    });

    // Step 2: Send verification email
    await context.step("send-verification", async () => {
      return await emailService.sendVerification(email);
    });

    // Step 3: Wait for email verification (up to 48 hours)
    const verified = await context.waitForCallback(
      "email-verification",
      async (callbackId) => {
        await notificationService.sendVerificationLink({
          email,
          callbackId,
          expiresIn: 172800
        });
      },
      {
        timeout: { seconds: 172800 }
      }
    );

    if (!verified) {
      await context.step("send-reminder", async () => {
        await emailService.sendReminder(email);
      });

      return {
        status: "verification_timeout",
        userId,
        message: "Email verification not completed within 48 hours"
      };
    }

    // Step 4: Initialize user profile in parallel
    const setupResults = await context.parallel("profile-setup", [
```

```
      async (ctx: DurableContext) => {
        return await ctx.step("create-preferences", async () => {
          return await preferencesService.createDefaults(userId);
        });
      },

      async (ctx: DurableContext) => {
        return await ctx.step("setup-notifications", async () => {
          return await notificationService.setupDefaults(userId);
        });
      },

      async (ctx: DurableContext) => {
        return await ctx.step("create-welcome-content", async () => {
          return await contentService.createWelcome(userId);
        });
      }
    ]);

    // Step 5: Send welcome email
    await context.step("send-welcome", async () => {
      const [preferences, notifications, content] = setupResults.getResults();
      return await emailService.sendWelcome({
        email,
        preferences,
        notifications,
        content
      });
    });

    return {
      status: "onboarding_complete",
      userId,
      completedAt: new Date().toISOString()
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 WaitConfig
from datetime import datetime

@durable_execution
def lambda_handler(event, context: DurableContext):
    user_id = event['userId']
    email = event['email']

    # Step 1: Create user account
    user = context.step(
        lambda _: user_service.create_account(user_id, email),
        name='create-account'
    )

    # Step 2: Send verification email
    context.step(
        lambda _: email_service.send_verification(email),
        name='send-verification'
    )

    # Step 3: Wait for email verification (up to 48 hours)
    def send_verification_link(callback_id):
        notification_service.send_verification_link({
            'email': email,
            'callbackId': callback_id,
            'expiresIn': 172800
        })

    verified = context.wait_for_callback(
        send_verification_link,
        name='email-verification',
        config=WaitConfig(timeout=172800)
    )

    if not verified:
        context.step(
            lambda _: email_service.send_reminder(email),
            name='send-reminder'
        )

        return {
            'status': 'verification_timeout',
            'userId': user_id,
```

```
            'message': 'Email verification not completed within 48 hours'
        }

    # Step 4: Initialize user profile in parallel
    def create_preferences(ctx: DurableContext):
        return ctx.step(
            lambda _: preferences_service.create_defaults(user_id),
            name='create-preferences'
        )

    def setup_notifications(ctx: DurableContext):
        return ctx.step(
            lambda _: notification_service.setup_defaults(user_id),
            name='setup-notifications'
        )

    def create_welcome_content(ctx: DurableContext):
        return ctx.step(
            lambda _: content_service.create_welcome(user_id),
            name='create-welcome-content'
        )

    setup_results = context.parallel(
        [create_preferences, setup_notifications, create_welcome_content],
        name='profile-setup'
    )

    # Step 5: Send welcome email
    def send_welcome(_):
        results = setup_results.get_results()
        preferences, notifications, content = results[0], results[1], results[2]
        return email_service.send_welcome({
            'email': email,
            'preferences': preferences,
            'notifications': notifications,
            'content': content
        })

    context.step(send_welcome, name='send-welcome')

    return {
        'status': 'onboarding_complete',
        'userId': user_id,
        'completedAt': datetime.now().isoformat()
```

```
    }
```

The process combines sequential steps with checkpoints for account creation and email sending, then pauses for up to 48 hours waiting for email verification without consuming resources. Conditional logic handles different paths based on whether verification completes or times out. Profile setup tasks run concurrently using parallel operations to reduce total execution time, and each step retries automatically on transient failures to help ensure the onboarding completes reliably.

Chained invocations across functions

Invoke other Lambda functions from within a durable function using `context.invoke()`. The calling function suspends while waiting for the invoked function to complete, creating a checkpoint that preserves the result. If the calling function is interrupted after the invoked function completes, it resumes with the stored result without re-invoking the function.

Use this pattern when you have specialized functions that handle specific domains (customer validation, payment processing, inventory management) and need to coordinate them in a workflow. Each function maintains its own logic and can be invoked by multiple orchestrator functions, avoiding code duplication.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

// Main orchestrator function
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, customerId } = event;

    // Step 1: Validate customer by invoking customer service function
    const customer = await context.invoke(
      "validate-customer",
      "arn:aws:lambda:us-east-1:123456789012:function:customer-service:1",
      { customerId }
    );
```

```
    if (!customer.isValid) {
      return { orderId, status: "rejected", reason: "invalid_customer" };
    }

    // Step 2: Check inventory by invoking inventory service function
    const inventory = await context.invoke(
      "check-inventory",
      "arn:aws:lambda:us-east-1:123456789012:function:inventory-service:1",
      { orderId, items: event.items }
    );

    if (!inventory.available) {
      return { orderId, status: "rejected", reason: "insufficient_inventory" };
    }

    // Step 3: Process payment by invoking payment service function
    const payment = await context.invoke(
      "process-payment",
      "arn:aws:lambda:us-east-1:123456789012:function:payment-service:1",
      {
        customerId,
        amount: inventory.totalAmount,
        paymentMethod: customer.paymentMethod
      }
    );

    // Step 4: Create shipment by invoking fulfillment service function
    const shipment = await context.invoke(
      "create-shipment",
      "arn:aws:lambda:us-east-1:123456789012:function:fulfillment-service:1",
      {
        orderId,
        items: inventory.allocatedItems,
        address: customer.shippingAddress
      }
    );

    return {
      orderId,
      status: "completed",
      trackingNumber: shipment.trackingNumber,
      estimatedDelivery: shipment.estimatedDelivery
    };
  }
```

```
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

# Main orchestrator function
@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    customer_id = event['customerId']

    # Step 1: Validate customer by invoking customer service function
    customer = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
        {'customerId': customer_id},
        name='validate-customer'
    )

    if not customer['isValid']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'invalid_customer'}

    # Step 2: Check inventory by invoking inventory service function
    inventory = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:inventory-service:1',
        {'orderId': order_id, 'items': event['items']},
        name='check-inventory'
    )

    if not inventory['available']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'insufficient_inventory'}

    # Step 3: Process payment by invoking payment service function
    payment = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:payment-service:1',
        {
            'customerId': customer_id,
            'amount': inventory['totalAmount'],
            'paymentMethod': customer['paymentMethod']
```

```
        },
        name='process-payment'
    )

    # Step 4: Create shipment by invoking fulfillment service function
    shipment = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:fulfillment-service:1',
        {
            'orderId': order_id,
            'items': inventory['allocatedItems'],
            'address': customer['shippingAddress']
        },
        name='create-shipment'
    )

    return {
        'orderId': order_id,
        'status': 'completed',
        'trackingNumber': shipment['trackingNumber'],
        'estimatedDelivery': shipment['estimatedDelivery']
    }
```

Each invocation creates a checkpoint in the orchestrator function. If the orchestrator is interrupted after the customer validation completes, it resumes from that checkpoint with the stored customer data, skipping the validation invocation. This prevents duplicate calls to downstream services and ensures consistent execution across interruptions.

The invoked functions can be either durable or standard Lambda functions. If you invoke a durable function, it can have its own multi-step workflow with waits and checkpoints. The orchestrator simply waits for the complete durable execution to finish, receiving the final result.

> **Note**
>
> Cross-account invocations are not supported. All invoked functions must be in the same AWS account as the calling function.

Batch processing with checkpoints

Process millions of records with automatic recovery from the last successful checkpoint after failures. This example demonstrates how durable functions combine `map()` operations with chunking and rate limiting to handle large-scale data processing.

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

interface Batch {
  batchIndex: number;
  recordIds: string[];
}

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { datasetId, batchSize = 1000 } = event;

    // Step 1: Get all record IDs to process
    const recordIds = await context.step("fetch-record-ids", async () => {
      return await dataService.getRecordIds(datasetId);
    });

    // Step 2: Split into batches
    const batches: Batch[] = [];
    for (let i = 0; i < recordIds.length; i += batchSize) {
      batches.push({
        batchIndex: Math.floor(i / batchSize),
        recordIds: recordIds.slice(i, i + batchSize)
      });
    }

    // Step 3: Process batches with controlled concurrency
    const batchResults = await context.map(
      "process-batches",
      batches,
      async (ctx: DurableContext, batch: Batch, index: number) => {
        const processed = await ctx.step(`batch-${batch.batchIndex}`, async () => {
          const results = [];
          for (const recordId of batch.recordIds) {
```

```
            const result = await recordService.process(recordId);
            results.push(result);
          }
          return results;
        });

        const validated = await ctx.step(`validate-${batch.batchIndex}`, async () =>
 {
          return await validationService.validateBatch(processed);
        });

        return {
          batchIndex: batch.batchIndex,
          recordCount: batch.recordIds.length,
          successCount: validated.successCount,
          failureCount: validated.failureCount
        };
      },
      {
        maxConcurrency: 5
      }
    );

    // Step 4: Aggregate results
    const summary = await context.step("aggregate-results", async () => {
      const results = batchResults.getResults();
      const totalSuccess = results.reduce((sum, r) => sum + r.successCount, 0);
      const totalFailure = results.reduce((sum, r) => sum + r.failureCount, 0);

      return {
        datasetId,
        totalRecords: recordIds.length,
        batchesProcessed: batches.length,
        successCount: totalSuccess,
        failureCount: totalFailure,
        completedAt: new Date().toISOString()
      };
    });

    return summary;
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 MapConfig
from datetime import datetime
from typing import List, Dict

@durable_execution
def lambda_handler(event, context: DurableContext):
    dataset_id = event['datasetId']
    batch_size = event.get('batchSize', 1000)

    # Step 1: Get all record IDs to process
    record_ids = context.step(
        lambda _: data_service.get_record_ids(dataset_id),
        name='fetch-record-ids'
    )

    # Step 2: Split into batches
    batches = []
    for i in range(0, len(record_ids), batch_size):
        batches.append({
            'batchIndex': i // batch_size,
            'recordIds': record_ids[i:i + batch_size]
        })

    # Step 3: Process batches with controlled concurrency
    def process_batch(ctx: DurableContext, batch: Dict, index: int):
        batch_index = batch['batchIndex']

        def process_records(_):
            results = []
            for record_id in batch['recordIds']:
                result = record_service.process(record_id)
                results.append(result)
            return results

        processed = ctx.step(process_records, name=f'batch-{batch_index}')

        validated = ctx.step(
            lambda _: validation_service.validate_batch(processed),
            name=f'validate-{batch_index}'
        )
```

```
        return {
            'batchIndex': batch_index,
            'recordCount': len(batch['recordIds']),
            'successCount': validated['successCount'],
            'failureCount': validated['failureCount']
        }

    batch_results = context.map(
        process_batch,
        batches,
        name='process-batches',
        config=MapConfig(max_concurrency=5)
    )

    # Step 4: Aggregate results
    def aggregate_results(_):
        results = batch_results.get_results()
        total_success = sum(r['successCount'] for r in results)
        total_failure = sum(r['failureCount'] for r in results)

        return {
            'datasetId': dataset_id,
            'totalRecords': len(record_ids),
            'batchesProcessed': len(batches),
            'successCount': total_success,
            'failureCount': total_failure,
            'completedAt': datetime.now().isoformat()
        }

    summary = context.step(aggregate_results, name='aggregate-results')

    return summary
```

Records are split into manageable batches to avoid overwhelming memory or downstream services, then multiple batches process concurrently with `maxConcurrency` controlling the parallelism. Each batch has its own checkpoint, so failures only retry the failed batch rather than reprocessing all records. This pattern is ideal for ETL jobs, data migrations, or bulk operations where processing can take hours.

Next steps

- Explore basic concepts to understand DurableContext, steps, and waits
- Review best practices for writing deterministic code and optimizing performance
- Learn about testing durable functions locally and in the cloud
- Compare durable functions with Step Functions to understand when each approach is most effective. See Durable functions or Step Functions.

Security and permissions for Lambda durable functions

Lambda durable functions require specific IAM permissions to manage checkpoint operations. Follow the principle of least privilege by granting only the permissions your function needs.

Execution role permissions

Your durable function's execution role needs permissions to create checkpoints and retrieve execution state. The following policy shows the minimum required permissions:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:function-name:*"
        }
    ]
}
```

When you create a durable function using the console, Lambda automatically adds these permissions to the execution role. If you create the function using the AWS CLI or AWS CloudFormation, add these permissions to your execution role.

For Lambda to assume your execution role, the role's trust policy must specify the Lambda service principal (`lambda.amazonaws.com`) as a trusted service. The following example shows a trust policy that grants Lambda permission to assume the role.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Least privilege principle

Scope the `Resource` element to specific function ARNs instead of using wildcards. This limits the execution role to checkpoint operations for only the functions that need them.

Example: Scoped permissions for multiple functions

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": [
                "arn:aws:lambda:us-east-1:123456789012:function:orderProcessor:*",
                "arn:aws:lambda:us-east-1:123456789012:function:paymentHandler:*"
            ]
        }
    ]
}
```

Alternatively, you can use the AWS managed policy `AWSLambdaBasicDurableExecutionRolePolicy` which includes the required durable execution permissions along with basic Lambda execution permissions for Amazon CloudWatch Logs.

State encryption

Lambda durable functions automatically enable encryption at rest using AWS owned keys at no charge. Each function execution maintains isolated state that other executions cannot access. Customer managed keys (CMK) are not supported.

Checkpoint data includes:

- Step results and return values
- Execution progress and timeline
- Wait state information

All data is encrypted in transit using TLS when Lambda reads or writes checkpoint data.

Custom encryption with custom serializers and deserializers

For critical security requirements, you can implement your own encryption and decryption mechanism using custom serializers and deserializers (SerDer) using durable SDK. This approach gives you full control over the encryption keys and algorithms used to protect checkpoint data.

> **Important**
>
> When you use custom encryption, you lose visibility of operation results in the Lambda console and API responses. Checkpoint data appears encrypted in execution history and cannot be inspected without decryption.

Your function's execution role needs `kms:Encrypt` and `kms:Decrypt` permissions for the AWS KMS key used in the custom SerDer implementation.

CloudTrail logging

Lambda logs checkpoint operations as data events in AWS CloudTrail. You can use CloudTrail to audit when checkpoints are created, track execution state changes, and monitor access to durable execution data.

Checkpoint operations appear in CloudTrail logs with the following event names:

- `CheckpointDurableExecution` - Logged when a step completes and creates a checkpoint

- `GetDurableExecutionState` - Logged when Lambda retrieves execution state during replay

To enable data event logging for durable functions, configure a CloudTrail trail to log Lambda data events. For more information, see Logging data events in the CloudTrail User Guide.

Example: CloudTrail log entry for checkpoint operation

```
{
    "eventVersion": "1.08",
    "eventTime": "2024-11-16T10:30:45Z",
    "eventName": "CheckpointDurableExecution",
    "eventSource": "lambda.amazonaws.com",
    "requestParameters": {
        "functionName": "myDurableFunction",
        "executionId": "exec-abc123",
        "stepId": "step-1"
    },
    "responseElements": null,
    "eventType": "AwsApiCall"
}
```

Cross-account considerations

If you invoke durable functions across AWS accounts, the calling account needs `lambda:InvokeFunction` permission, but checkpoint operations always use the execution role in the function's account. The calling account cannot access checkpoint data or execution state directly.

This isolation ensures that checkpoint data remains secure within the function's account, even when invoked from external accounts.

Inherited Lambda security features

Durable functions inherit all security, governance, and compliance features from Lambda, including VPC connectivity, environment variable encryption, dead letter queues, reserved concurrency, function URLs, code signing, and compliance certifications (SOC, PCI DSS, HIPAA, etc.).

For detailed information about Lambda security features, see Security in AWS Lambda in the Lambda Developer Guide. The only additional security considerations for durable functions are the checkpoint permissions documented in this guide.

Durable execution SDK

The durable execution SDK is the foundation for building durable functions. It provides the primitives you need to checkpoint progress, handle retries, and manage execution flow. The SDK abstracts the complexity of checkpoint management and replay, letting you write sequential code that automatically becomes fault-tolerant.

The SDK is available for JavaScript, TypeScript, Python and Java. For complete API documentation and examples, see the JavaScript/TypeScript SDK, Python SDK and Java SDK on GitHub.

DurableContext

The SDK provides your function with a `DurableContext` object that exposes all durable operations. This context replaces the standard Lambda context and provides methods for creating checkpoints, managing execution flow, and coordinating with external systems.

To use the SDK, wrap your Lambda handler with the durable execution wrapper:

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Your function receives DurableContext instead of Lambda context
    // Use context.step(), context.wait(), etc.
    return result;
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext

@durable_execution
def handler(event: dict, context: DurableContext):
    # Your function receives DurableContext
```

```
    # Use context.step(), context.wait(), etc.
    return result
```

Java

```
import software.amazon.lambda.durable.DurableContext;
import software.amazon.lambda.durable.DurableHandler;

public class Handler extends DurableHandler<Object, String> {
    @Override
    public String handleRequest(Object input, DurableContext context) {
        // Your function receives DurableContext
        // Use context.step(), context.wait(), etc.
        return result;
    }
}
```

The wrapper intercepts your function invocation, loads any existing checkpoint log, and provides the `DurableContext` that manages replay and checkpointing.

What the SDK does

The SDK handles three critical responsibilities that enable durable execution:

Checkpoint management: The SDK automatically creates checkpoints as your function executes durable operations. Each checkpoint records the operation type, inputs, and results. When your function completes a step, the SDK persists the checkpoint before continuing. This ensures your function can resume from any completed operation if interrupted.

Replay coordination: When your function resumes after a pause or interruption, the SDK performs replay. It runs your code from the beginning but skips completed operations, using stored checkpoint results instead of re-executing them. The SDK ensures replay is deterministic—given the same inputs and checkpoint log, your function produces the same results.

State isolation: The SDK maintains execution state separately from your business logic. Each durable execution has its own checkpoint log that other executions cannot access. The SDK encrypts checkpoint data at rest and ensures state remains consistent across replays.

How checkpointing works

When you call a durable operation, the SDK follows this sequence:

1. **Check for existing checkpoint:** The SDK checks if this operation already completed in a previous invocation. If a checkpoint exists, the SDK returns the stored result without re-executing the operation.
2. **Execute the operation:** If no checkpoint exists, the SDK executes your operation code. For steps, this means calling your function. For waits, this means scheduling resumption.
3. **Create checkpoint:** After the operation completes, the SDK serializes the result and creates a checkpoint. The checkpoint includes the operation type, name, inputs, result, and timestamp.
4. **Persist checkpoint:** The SDK calls the Lambda checkpoint API to persist the checkpoint. This ensures the checkpoint is durable before continuing execution.
5. **Return result:** The SDK returns the operation result to your code, which continues to the next operation.

This sequence ensures that once an operation completes, its result is safely stored. If your function is interrupted at any point, the SDK can replay up to the last completed checkpoint.

Replay behavior

When your function resumes after a pause or interruption, the SDK performs replay:

1. **Load checkpoint log:** The SDK retrieves the checkpoint log for this execution from Lambda.
2. **Run from beginning:** The SDK invokes your handler function from the start, not from where it paused.
3. **Skip completed durable operations:** As your code calls durable operations, the SDK checks each against the checkpoint log. For completed durable operations, the SDK returns the stored result without executing the operation code.

> **Note**
>
> If a child context's result was larger than the maximum checkpoint size (256 KB), the context's code is executed again during replay. This allows you to construct large results from the durable operations that ran inside the context, which will be looked up from the checkpoint log. Therefore it is imperative to only run deterministic code in

the context itself. When using child contexts with large results, it is a best practice to perform long-running or non-deterministic work inside of steps and only perform short-running tasks which combine the results in the context itself.

4. **Resume at interruption point:** When the SDK reaches an operation without a checkpoint, it executes normally and creates new checkpoints as durable operations complete.

This replay mechanism requires your code to be deterministic. Given the same inputs and checkpoint log, your function must make the same sequence of durable operation calls. The SDK enforces this by validating that operation names and types match the checkpoint log during replay.

Available durable operations

The `DurableContext` provides operations for different coordination patterns. Each durable operation creates checkpoints automatically, ensuring your function can resume from any point.

Steps

Executes business logic with automatic checkpointing and retry. Use steps for operations that call external services, perform calculations, or execute any logic that should be checkpointed. The SDK creates a checkpoint before and after the step, storing the result for replay.

TypeScript

```
const result = await context.step('process-payment', async () => {
  return await paymentService.charge(amount);
});
```

Python

```
result = context.step(
    lambda _: payment_service.charge(amount),
    name='process-payment'
)
```

Java

```
var result = context.step("process-payment", Payment.class,
    () -> paymentService.charge(amount)
);
```

Steps support configurable retry strategies, execution semantics (at-most-once or at-least-once), and custom serialization.

Waits

Pauses execution for a specified duration without consuming compute resources. The SDK creates a checkpoint, terminates the function invocation, and schedules resumption. When the wait completes, Lambda invokes your function again and the SDK replays to the wait point before continuing.

TypeScript

```
// Wait 1 hour without charges
await context.wait({ seconds: 3600 });
```

Python

```
# Wait 1 hour without charges
context.wait(Duration.from_seconds(3600))
```

Java

```
// Wait 1 hour without charges
context.wait(Duration.ofHours(1));
```

Callbacks

Callbacks enable your function to pause and wait for external systems to provide input. When you create a callback, the SDK generates a unique callback ID and creates a checkpoint. Your function then suspends (terminates the invocation) without incurring compute charges. External systems submit callback results using the `SendDurableExecutionCallbackSuccess` or `SendDurableExecutionCallbackFailure` Lambda APIs. When a callback is submitted, Lambda invokes your function again, the SDK replays to the callback point, and your function continues with the callback result.

The SDK provides two methods for working with callbacks:

createCallback: Creates a callback and returns both a promise and a callback ID. You send the callback ID to an external system, which submits the result using the Lambda API.

TypeScript

```
const [promise, callbackId] = await context.createCallback('approval', {
  timeout: { hours: 24 }
});

await sendApprovalRequest(callbackId, requestData);
const approval = await promise;
```

Python

```
callback = context.create_callback(
    name='approval',
    config=CallbackConfig(timeout_seconds=86400)
)

context.step(
    lambda _: send_approval_request(callback.callback_id),
    name='send_request'
)

approval = callback.result()
```

Java

```
var config = CallbackConfig.builder(Duration.ofHours(24)).timeout()

var callback = context.createCallback("approval", String.class, config);

context.step("send-request", String.class, () -> {
    notificationService.sendApprovalRequest(callback.callbackId(), requestData);
    return "request-sent";
});

// Blocks until the callback finishes or times out
String approval = callback.get();
```

waitForCallback: Simplifies callback handling by combining callback creation and submission in one operation. The SDK creates the callback, executes your submitter function with the callback ID, and waits for the result.

TypeScript

```
const result = await context.waitForCallback(
  'external-api',
  async (callbackId, ctx) => {
    await submitToExternalAPI(callbackId, requestData);
  },
  { timeout: { minutes: 30 } }
);
```

Python

```
result = context.wait_for_callback(
    lambda callback_id: submit_to_external_api(callback_id, request_data),
    name='external-api',
    config=WaitForCallbackConfig(timeout_seconds=1800)
)
```

Java

```
var result = context.waitForCallback(
    "external-api",
    String.class,
    (callbackId, ctx) -> {
        submitToExternalAPI(callbackId, requestData);
    },
    WaitForCallbackConfig.builder()
        .callbackConfig(CallbackConfig.builder()
            .timeout(Duration.ofMinutes(30))
            .build())
        .build());
```

Configure timeouts to prevent functions from waiting indefinitely. If a callback times out, the SDK throws a `CallbackError` and your function can handle the timeout case. Use heartbeat timeouts for long-running callbacks to detect when external systems stop responding.

Use callbacks for human-in-the-loop workflows, external system integration, webhook responses, or any scenario where execution must pause for external input.

Parallel execution

Executes multiple operations concurrently with optional concurrency control. The SDK manages parallel execution, creates checkpoints for each operation, and handles failures according to your completion policy.

TypeScript

```
const results = await context.parallel([
  async (ctx) => ctx.step('task1', async () => processTask1()),
  async (ctx) => ctx.step('task2', async () => processTask2()),
  async (ctx) => ctx.step('task3', async () => processTask3())
]);
```

Python

```
results = context.parallel([
    lambda ctx: ctx.step(lambda _: process_task1(), name='task1'),
    lambda ctx: ctx.step(lambda _: process_task2(), name='task2'),
    lambda ctx: ctx.step(lambda _: process_task3(), name='task3')
])
```

Java

```
DurableFuture<String> f1;
DurableFuture<Integer> f2;
DurableFuture<Boolean> f3;
try (var parallel = context.parallel("tasks")) {
    f1 = parallel.branch("string-task",  String.class,  ctx -> ctx.step("string-
task",  String.class,  s -> processString()));
    f2 = parallel.branch("integer-task", Integer.class, ctx -> ctx.step("integer-
task", Integer.class, s -> processInteger()));
    f3 = parallel.branch("boolean-task", Boolean.class, ctx -> ctx.step("boolean-
task", Boolean.class, s -> processBoolean()));
}
String stringResult = f1.get();
int integerResult = f2.get();
boolean booleanResult = f3.get();
```

Use `parallel` to execute independent operations concurrently.

Map

Concurrently execute an operation on each item in an array with optional concurrency control. The SDK manages concurrent execution, creates checkpoints for each operation, and handles failures according to your completion policy.

TypeScript

```
const results = await context.map(itemArray, async (ctx, item, index) =>
```

```
  ctx.step('task', async () => processItem(item, index))
);
```

Python

```
results = context.map(
    item_array,
    lambda ctx, item, index: ctx.step(
        lambda _: process_item(item, index),
        name='task'
    )
)
```

Java

```
var results = context.map(
    "process-items",
    itemArray,
    String.class,
    (item, index, ctx) -> ctx.step("task", String.class, s -> processItem(item,
 index)));
```

Use `map` to process arrays with concurrency control.

Child contexts

Creates an isolated execution context for grouping operations. Child contexts have their own checkpoint log and can contain multiple steps, waits, and other operations. The SDK treats the entire child context as a single unit for retry and recovery.

Use child contexts to organize complex workflows, implement sub-workflows, or isolate operations that should retry together.

TypeScript

```
const result = await context.runInChildContext(
  'batch-processing',
  async (childCtx) => {
    return await processBatch(childCtx, items);
  }
);
```

Python

```
result = context.run_in_child_context(
    lambda child_ctx: process_batch(child_ctx, items),
    name='batch-processing'
)
```

Java

```
var result = context.runInChildContext(
    "batch-processing",
    String.class,
    childCtx -> process_batch(childCtx, items)
);
```

The replay mechanism demands that durable operations happen in a deterministic order. Using multiple child contexts you can have multiple streams of work execute concurrently, and the determinism applies separately within each context. This allows you to build high-performance functions which efficiently utilize multiple CPU cores.

For example, imagine we start two child contexts, A and B. On the initial invocation, the steps within the contexts were run in this order, with the 'A' steps running concurrently with the 'B' steps: A1, B1, B2, A2, A3. Upon replay, the timing is much faster as results are retrieved from checkpoint log, and the steps happen to be encountered in a different order: B1, A1, A2, B2, A3. Because the 'A' steps were encountered in the correct order (A1, A2, A3) and the 'B' steps were encountered in the correct order (B1, B2), the need for determinism was satisfied correctly.

Conditional waits

Polls for a condition with automatic checkpointing between attempts. The SDK executes your check function, creates a checkpoint with the result, waits according to your strategy, and repeats until the condition is met.

TypeScript

```
const result = await context.waitForCondition(
  async (state, ctx) => {
    const status = await checkJobStatus(state.jobId);
    return { ...state, status };
  },
  {
    initialState: { jobId: 'job-123', status: 'pending' },
    waitStrategy: (state) =>
      state.status === 'completed'
        ? { shouldContinue: false }
        : { shouldContinue: true, delay: { seconds: 30 } }
  }
);
```

Python

```
result = context.wait_for_condition(
    lambda state, ctx: check_job_status(state['jobId']),
    config=WaitForConditionConfig(
        initial_state={'jobId': 'job-123', 'status': 'pending'},
        wait_strategy=lambda state, attempt:
            {'should_continue': False} if state['status'] == 'completed'
            else {'should_continue': True, 'delay': 30}
    )
)
```

Java

```
record JobState(String jobId, String status) {}
```

```
var result = context.waitForCondition(
    "check-job",
    JobState.class,
    (state, ctx) -> {
        var status = checkJobStatus(state.jobId());
        var updatedState = new JobState(state.jobId(), status);
        if ("completed".equals(status)) {
            return WaitForConditionResult.stopPolling(updatedState);
        }
        return WaitForConditionResult.continuePolling(updatedState);
    },
    WaitForConditionConfig.<JobState>builder()
        .initialState(new JobState("job-123", "pending"))
        .waitStrategy((state, attempt) -> Duration.ofSeconds(30))
        .build());
```

Use `waitForCondition` for polling external systems, waiting for resources to be ready, or implementing retry with backoff.

Function invocation

Invokes another Lambda function and waits for its result. The SDK creates a checkpoint, invokes the target function, and resumes your function when the invocation completes. This enables function composition and workflow decomposition.

TypeScript

```
const result = await context.invoke(
  'invoke-processor',
  'arn:aws:lambda:us-east-1:123456789012:function:processor:1',
  { data: inputData }
);
```

Python

```
result = context.invoke(
```

```
    'arn:aws:lambda:us-east-1:123456789012:function:processor:1',
    {'data': input_data},
    name='invoke-processor'
)
```

Java

```
var result = context.invoke(
    "invoke-processor",
    "arn:aws:lambda:us-east-1:123456789012:function:processor:1",
    inputData,
    Result.class,
    InvokeConfig.builder().build()
);
```

How durable operations are metered

Each durable operation you call through `DurableContext` creates checkpoints to track execution progress and store state data. These operations incur charges based on their usage, and the checkpoints may contain data that contributes to your data write and retention costs. Stored data includes invocation event data, payloads returned from steps, and data passed when completing callbacks. Understanding how durable operations are metered helps you estimate execution costs and optimize your workflows. For details on pricing, see the Lambda pricing page.

Payload size refers to the size of the serialized data that a durable operation persists. The data is measured in bytes and the size can vary depending on the serializer used by the operation. The payload of an operation could be the result itself for successful completions, or the serialized error object if the operation failed.

Basic operations

Basic operations are the fundamental building blocks for durable functions:

Operation	Checkpoint timing	Number of operations	Data persisted
Execution	Started	1	Input payload size
Execution	Completed (Succeeded/Failed/Stopped)	0	Output payload size
Step	Retry/Succeeded/Failed	1 + N retries	Returned payload size from each attempt
Wait	Started	1	N/A
WaitForCondition	Each poll attempt	1 + N polls	Returned payload size from each poll attempt
Invocation-level Retry	Started	1	Payload for error object

Callback operations

Callback operations enable your function to pause and wait for external systems to provide input. These operations create checkpoints when the callback is created and when it's completed:

Operation	Checkpoint timing	Number of operations	Data persisted
CreateCallback	Started	1	N/A
Callback completion via API call	Completed	0	Callback payload

Operation	Checkpoint timing	Number of operations	Data persisted
WaitForCallback	Started	3 + N retries (context + callback + step)	Payloads returned by submitter step attempts, plus two copies of the callback payload

Compound operations

Compound operations combine multiple durable operations to handle complex coordination patterns like parallel execution, array processing, and nested contexts:

Operation	Checkpoint timing	Number of operations	Data persisted
Parallel	Started	1 + N branches (1 parent context + N child contexts)	Up to two copies of the returned payload size from each branch, plus the statuses of each branch
Map	Started	1 + N branches (1 parent context + N child contexts)	Up to two copies of the returned payload size from each iteration, plus the statuses of each iteration

Operation	Checkpoint timing	Number of operations	Data persisted
Promise helpers	Completed	1	Returned payload size from the promise
RunInChildContext	Succeeded/Failed	1	Returned payload size from the child context

For contexts, such as from `runInChildContext` or used internally by compound operations, results smaller than 256 KB are checkpointed directly. Larger results aren't stored—instead, they're reconstructed during replay by re-processing the context's operations.

Supported runtimes for durable functions

Durable functions are available for selected managed runtimes and OCI container images for additional runtime version flexibility. You can create durable functions for Node.js and Python using managed runtimes directly in the console or programmatically through infrastructure-as-code. Durable functions in Java currently can only be deployed through container images.

Lambda managed runtimes

The following managed runtimes support durable functions when you create functions in the Lambda console or using the AWS CLI with the `--durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'` parameter. For complete information about Lambda runtimes, see Lambda runtimes.

Language	Runtime
Node.js	nodejs22.x
Node.js	nodejs24.x
Python	python3.13
Python	python3.14

> **Note**
>
> Lambda runtimes include the durable execution SDK for testing and development. However, we recommend including the SDK in your deployment package for production. This ensures version consistency and avoids potential runtime updates that might affect your function behavior.

Node.js

Install the SDK in your Node.js project:

```
npm install @aws/durable-execution-sdk-js
```

The SDK supports JavaScript and TypeScript. For TypeScript projects, the SDK includes type definitions.

Python

Install the SDK in your Python project:

```
pip install aws-durable-execution-sdk-python
```

The Python SDK uses synchronous methods and doesn't require `async/await`.

Java

Add a dependency to `pom.xml`:

```
<dependency>
    <groupId>software.amazon.lambda.durable</groupId>
    <artifactId>aws-durable-execution-sdk-java</artifactId>
    <version>VERSION</version>
</dependency>
```

Install the SDK in your Java project:

```
mvn install
```

The Java SDK provides both synchronous and asynchronous versions of each method.

Container images

You can use durable functions with container images to support additional runtime versions or custom runtime configurations. Container images let you use runtime versions not available as managed runtimes or customize your runtime environment.

To create a durable function using a container image:

1. Create a Dockerfile based on an Lambda base image
2. Install the durable execution SDK in your container
3. Build and push the container image to Amazon Elastic Container Registry
4. Create the Lambda function from the container image with durable execution enabled

Container example

Create a Dockerfile:

Python

Create a Dockerfile for Python 3.11:

```
FROM public.ecr.aws/lambda/python:3.11

# Copy requirements file
COPY requirements.txt ${LAMBDA_TASK_ROOT}/

# Install dependencies including durable SDK
RUN pip install -r requirements.txt
```

```
# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}/

# Set the handler
CMD [ "lambda_function.handler" ]
```

Create a `requirements.txt` file:

```
aws-durable-execution-sdk-python
```

Java

Create a Dockerfile for Java 25:

```
FROM --platform=linux/amd64 public.ecr.aws/lambda/java:25

# Install Maven
RUN dnf install -y maven

WORKDIR /var/task

# Copy Maven configuration and source code
COPY pom.xml .
COPY src ./src

# Build
RUN mvn clean package -DskipTests

# Move JAR to lib directory
RUN mv target/*.jar lib/

# Set the handler
CMD ["src.path.to.lambdaFunction::handler"]
```

Build and push the image:

```
# Build the image
docker build -t my-durable-function .

# Tag for ECR
docker tag my-durable-function:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-
durable-function:latest

# Push to ECR
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-durable-function:latest
```

Create the function with durable execution enabled:

```
aws lambda create-function \
  --function-name myDurableFunction \
  --package-type Image \
  --code ImageUri=123456789012.dkr.ecr.us-east-1.amazonaws.com/my-durable-
function:latest \
  --role arn:aws:iam::123456789012:role/lambda-execution-role \
  --durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'
```

For more information about using container images with Lambda, see Creating Lambda container images in the Lambda Developer Guide.

Runtime considerations

SDK version management: Include the durable execution SDK in your deployment package or container image. This ensures your function uses a specific SDK version and isn't affected by runtime updates. Pin SDK versions in your `package.json` or `requirements.txt` to control when you upgrade.

Runtime updates: AWS updates managed runtimes to include security patches and bug fixes. These updates may include new SDK versions. To avoid unexpected behavior, include the SDK in your deployment package and test thoroughly before deploying to production.

Container image size: Container images have a maximum uncompressed size of 10 GB. The durable execution SDK adds minimal size to your image. Optimize your container by using multi-stage builds and removing unnecessary dependencies.

Cold start performance: Container images may have longer cold start times than managed runtimes. The durable execution SDK has minimal impact on cold start performance. Use provisioned concurrency if cold start latency is critical for your application.

Invoking durable Lambda functions

Durable Lambda functions support the same invocation methods as standard Lambda functions. You can invoke durable functions synchronously, asynchronously, or through event source mappings. The invocation process is identical to standard functions, but durable functions provide additional capabilities for long-running executions and automatic state management.

Invocation methods

Synchronous invocation: Invoke a durable function and wait for the response. Synchronous invocations are limited by the Lambda to 15 minutes (or less, depending on the configured function and execution timeout). Use synchronous invocation when you need immediate results or when integrating with APIs and services that expect a response. You can use wait operations for efficient computation without disrupting the caller—the invocation waits for the entire durable execution to complete. For idempotent execution starts, use the execution name parameter as described in Idempotency.

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

Asynchronous invocation: Queue an event for processing without waiting for a response. Lambda places the event in a queue and returns immediately. Asynchronous invocations support execution durations up to 1 year. Use asynchronous invocation for fire-and-forget scenarios or when processing can happen in the background. For idempotent execution starts, use the execution name parameter as described in Idempotency.

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --invocation-type Event \
```

```
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

Event source mappings: Configure Lambda to automatically invoke your durable function when records are available from stream or queue-based services like Amazon SQS, Kinesis, or DynamoDB. Event source mappings poll the event source and invoke your function with batches of records. For details about using event source mappings with durable functions, including execution duration limits, see Event source mappings with durable functions.

For complete details about each invocation method, see synchronous invocation and asynchronous invocation.

> **Note**
>
> Durable functions support dead-letter queues (DLQs) for error handling, but don't support Lambda destinations. Configure a DLQ to capture records from failed invocations.

Qualified ARNs requirement

Durable functions require qualified identifiers for invocation. You must invoke durable functions using a version number, alias, or $LATEST. You can use either a full qualified ARN or a function name with version/alias suffix. You cannot use an unqualified identifier (without a version or alias suffix).

Valid invocations:

```
# Using full ARN with version number
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:1

# Using full ARN with alias
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:prod

# Using full ARN with $LATEST
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:$LATEST

# Using function name with version number
```

```
my-durable-function:1

# Using function name with alias
my-durable-function:prod
```

Invalid invocations:

```
# Unqualified ARN (not allowed)
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function

# Unqualified function name (not allowed)
my-durable-function
```

This requirement ensures that durable executions remain consistent throughout their lifecycle. When a durable execution starts, it's pinned to the specific function version. If your function pauses and resumes hours or days later, Lambda invokes the same version that started the execution, ensuring code consistency across the entire workflow.

Best practice

Use numbered versions or aliases for production durable functions rather than $LATEST. Numbered versions are immutable and support deterministic replay. Optionally, aliases provide a stable reference that you can update to point to new versions without changing invocation code. When you update an alias, new executions use the new version, while in-progress executions continue with their original version. You may use $LATEST for prototyping or to shorten deployment times during development, understanding that executions might not replay correctly (or even fail) if the underlying code changes during running executions.

Understanding execution lifecycle

When you invoke a durable function, Lambda creates a durable execution that can span multiple function invocations:

1. **Initial invocation:** Your invocation request creates a new durable execution. Lambda assigns a unique execution ID and starts processing.
2. **Execution and checkpointing:** As your function executes durable operations, the SDK creates checkpoints that track progress.

3. **Suspension (if needed):** If your function uses durable waits, such as `wait` or `waitForCallback`, or automatic step retries, Lambda suspends the execution and stops charging for compute time.
4. **Resumption:** When it's time to resume (including after retries), Lambda invokes your function again. The SDK replays the checkpoint log and continues from where execution paused.
5. **Completion:** When your function returns a final result or throws an unhandled error, the durable execution completes.

For synchronous invocations, the caller waits for the entire durable execution to complete, including any wait operations. If the execution exceeds the invocation timeout (15 minutes or less), the invocation times out. For asynchronous invocations, Lambda returns immediately and the execution continues independently. Use the durable execution APIs to track execution status and retrieve final results.

Invoking from application code

Use the AWS SDKs to invoke durable functions from your application code. The invocation process is identical to standard functions:

TypeScript

```
import { LambdaClient, InvokeCommand } from '@aws-sdk/client-lambda';

const client = new LambdaClient({});

// Synchronous invocation
const response = await client.send(new InvokeCommand({
  FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
  Payload: JSON.stringify({ orderId: '12345' })
}));

const result = JSON.parse(Buffer.from(response.Payload!).toString());

// Asynchronous invocation
await client.send(new InvokeCommand({
  FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
  InvocationType: 'Event',
```

```
  Payload: JSON.stringify({ orderId: '12345' })
}));
```

Python

```
import boto3
import json

client = boto3.client('lambda')

# Synchronous invocation
response = client.invoke(
    FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    Payload=json.dumps({'orderId': '12345'})
)

result = json.loads(response['Payload'].read())

# Asynchronous invocation
client.invoke(
    FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    InvocationType='Event',
    Payload=json.dumps({'orderId': '12345'})
)
```

Chained invocations

Durable functions can invoke other durable and non-durable functions using the `invoke` operation from `DurableContext`. This creates a chained invocation where the calling function waits (suspends) for the invoked function to complete:

TypeScript

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
```

```
    // Invoke another durable function and wait for result
    const result = await context.invoke(
      'process-order',
      'arn:aws:lambda:us-east-1:123456789012:function:order-processor:1',
      { orderId: event.orderId }
    );

    return { statusCode: 200, body: JSON.stringify(result) };
  }
);
```

Python

```
@durable_execution
def handler(event, context: DurableContext):
    # Invoke another durable function and wait for result
    result = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:order-processor:1',
        {'orderId': event['orderId']},
        name='process-order'
    )

    return {'statusCode': 200, 'body': json.dumps(result)}
```

Chained invocations create a checkpoint in the calling function. If the calling function is interrupted, it resumes from the checkpoint with the invoked function's result, without re-invoking the function.

Note

Cross-account chained invocations are not supported. The invoked function must be in the same AWS account as the calling function.

Event source mappings with durable functions

Durable functions work with all Lambda event source mappings. Configure event source mappings for durable functions the same way you configure them for standard functions. Event source mappings automatically poll event sources like Amazon SQS, Kinesis, and DynamoDB Streams, and invoke your function with batches of records.

Event source mappings are useful for durable functions that process streams or queues with complex, multi-step workflows. For example, you can create a durable function that processes Amazon SQS messages with retries, external API calls, and human approvals.

How event source mappings invoke durable functions

Event source mappings invoke durable functions synchronously, waiting for the complete durable execution to finish before processing the next batch or marking records as processed. If the total durable execution time exceeds 15 minutes, the execution times out and fails. The event source mapping receives a timeout exception and handles it according to its retry configuration.

15-minute execution limit

When durable functions are invoked by event source mappings, the total durable execution duration cannot exceed 15 minutes. This limit applies to the entire durable execution from start to completion, not just individual function invocations.

This 15-minute limit is separate from the Lambda function timeout (also 15 minutes maximum). The function timeout controls how long each individual invocation can run, while the durable execution timeout controls the total elapsed time from execution start to completion.

Example scenarios:

- **Valid:** A durable function processes an Amazon SQS message with three steps, each taking 2 minutes, then waits 5 minutes before completing a final step. Total execution time: 11 minutes. This works because the total is under 15 minutes.
- **Invalid:** A durable function processes an Amazon SQS message, completes initial processing in 2 minutes, then waits 20 minutes for an external callback before completing. Total execution time: 22 minutes. This exceeds the 15-minute limit and will fail.
- **Invalid:** A durable function processes a Kinesis record with multiple wait operations totaling 30 minutes between steps. Even though each individual invocation completes quickly, the total execution time exceeds 15 minutes.

> **Important**
>
> Configure your durable execution timeout to 15 minutes or less when using event source mappings, otherwise creation of the event source mapping will fail. If your workflow requires longer execution times, use the intermediary function pattern described below.

Configuring event source mappings

Configure event source mappings for durable functions using the Lambda console, AWS CLI, or AWS SDKs. All standard event source mapping properties apply to durable functions:

```
aws lambda create-event-source-mapping \
  --function-name arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:1
 \
  --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
  --batch-size 10 \
  --maximum-batching-window-in-seconds 5
```

Remember to use a qualified ARN (with version number or alias) when configuring event source mappings for durable functions.

Error handling with event source mappings

Event source mappings provide built-in error handling that works with durable functions:

- **Retry behavior:** If the initial invocation fails, the event source mapping retries according to its retry configuration. Configure maximum retry attempts and retry intervals based on your requirements.
- **Dead-letter queues:** Configure a dead-letter queue to capture records that fail after all retries. This prevents message loss and enables manual inspection of failed records.
- **Partial batch failures:** For Amazon SQS and Kinesis, use partial batch failure reporting to process records individually and only retry failed records.
- **Bisect on error:** For Kinesis and DynamoDB Streams, enable bisect on error to split failed batches and isolate problematic records.

> **Note**
>
> Durable functions support dead-letter queues (DLQs) for error handling, but don't support Lambda destinations. Configure a DLQ to capture records from failed invocations.

For complete information about event source mapping error handling, see event source mappings.

Using an intermediary function for long-running workflows

If your workflow requires more than 15 minutes to complete, use an intermediary standard Lambda function between the event source mapping and your durable function. The intermediary function receives events from the event source mapping and invokes the durable function asynchronously, removing the 15-minute execution limit.

This pattern decouples the event source mapping's synchronous invocation model from the durable function's long-running execution model. The event source mapping invokes the intermediary function, which quickly returns after starting the durable execution. The durable function then runs independently for as long as needed (up to 1 year).

Architecture

The intermediary function pattern uses three components:

1. **Event source mapping:** Polls the event source (Amazon SQS, Kinesis, DynamoDB Streams) and invokes the intermediary function synchronously with batches of records.
2. **Intermediary function:** A standard Lambda function that receives events from the event source mapping, validates and transforms the data if needed, and invokes the durable function asynchronously. This function completes quickly (typically under 1 second) and returns control to the event source mapping.
3. **Durable function:** Processes the event with complex, multi-step logic that can run for extended periods. Invoked asynchronously, so it's not constrained by the 15-minute limit.

Implementation

The intermediary function receives the entire event from the event source mapping and invokes the durable function asynchronously. Use the execution name parameter to ensure idempotent execution starts, preventing duplicate processing if the event source mapping retries:

TypeScript

```
import { LambdaClient, InvokeCommand } from '@aws-sdk/client-lambda';
import { SQSEvent } from 'aws-lambda';
import { createHash } from 'crypto';

const lambda = new LambdaClient({});

export const handler = async (event: SQSEvent) => {
  // Invoke durable function asynchronously with execution name
  await lambda.send(new InvokeCommand({
    FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    InvocationType: 'Event',
    Payload: JSON.stringify({
      executionName: event.Name,
      event: event
    })
  }));

  return { statusCode: 200 };
};
```

Python

```
import boto3
import json
import hashlib

lambda_client = boto3.client('lambda')

def handler(event, context):
    # Invoke durable function asynchronously with execution name
    lambda_client.invoke(
        FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
        InvocationType='Event',
        Payload=json.dumps({
            'executionName': execution_name,
            'event': event["name"]
```

```
        })
    )

    return {'statusCode': 200}
```

For idempotency in the intermediary function itself, use Powertools for AWS Lambda to prevent duplicate invocations of the durable function if the event source mapping retries the intermediary function.

The durable function receives the payload with the execution name and processes all records with long-running logic:

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (payload: any, context: DurableContext) => {
    const sqsEvent = payload.event;

    // Process each record with complex, multi-step logic
    const results = await context.map(
      sqsEvent.Records,
      async (ctx, record) => {
        const validated = await ctx.step('validate', async () => {
          return validateOrder(JSON.parse(record.body));
        });

        // Wait for external approval (could take hours or days)
        const approval = await ctx.waitForCallback(
          'approval',
          async (callbackId) => {
            await requestApproval(callbackId, validated);
          },
          { timeout: { hours: 48 } }
        );

        // Complete processing
        return await ctx.step('complete', async () => {
```

```
          return completeOrder(validated, approval);
        });
      }
    );

    return { statusCode: 200, processed: results.getResults().length };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
from aws_durable_execution_sdk_python.config import Duration, WaitForCallbackConfig
from collections.abc import Sequence
import json

def validate_order(order_data: dict) -> dict:
    """Validate order data - always passes."""
    return order_data

def request_approval(callback_id: str, validated_order: dict) -> None:
    """Request approval for the order - always passes."""
    pass

def complete_order(validated_order: dict, approval_result: str) -> dict:
    """Complete the order processing - always passes."""
    return validated_order

@durable_execution
def lambda_handler(payload, context: DurableContext):
    sqs_event = payload['event']

    def process_record(
        ctx: DurableContext,
        record: dict,
        index: int,
        items: Sequence[dict]
    ) -> dict:
        validated = ctx.step(
            lambda _: validate_order(json.loads(record['body'])),
            name=f'validate-{index}'
```

```
        )

        approval = ctx.wait_for_callback(
            submitter=lambda callback_id, wait_ctx: request_approval(callback_id,
validated),
            name=f'approval-{index}',
            config=WaitForCallbackConfig(timeout=Duration.from_seconds(172800))
        )

        return ctx.step(
            lambda _: complete_order(validated, approval),
            name=f'complete-{index}'
        )

    results = context.map(
        inputs=sqs_event['Records'],
        func=process_record,
        name='process-records'
    )

    return {
        'statusCode': 200,
        'started': results.started_count,
        'completed': results.success_count,
        'failed': results.failure_count,
        'total': results.total_count
    }
```

Key considerations

This pattern removes the 15-minute execution limit by decoupling the event source mapping from the durable execution. The intermediary function returns immediately after starting the durable execution, allowing the event source mapping to continue processing. The durable function then runs independently for as long as needed.

The intermediary function succeeds when it invokes the durable function, not when the durable execution completes. If the durable execution fails later, the event source mapping won't retry because it already processed the batch successfully. Implement error handling in the durable function and configure dead-letter queues for failed executions.

Use the execution name parameter to ensure idempotent execution starts. If the event source mapping retries the intermediary function, the durable function won't start a duplicate execution because the execution name already exists.

Supported event sources

Durable functions support all Lambda event sources that use event source mappings:

- Amazon SQS queues (standard and FIFO)
- Kinesis streams
- DynamoDB Streams
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon MQ (ActiveMQ and RabbitMQ)
- Amazon DocumentDB change streams

All event source types are subject to the 15-minute durable execution limit when invoking durable functions.

Retries for Lambda durable functions

Durable functions provide automatic retry capabilities that make your applications resilient to transient failures. The SDK handles retries at two levels: step retries for business logic failures and backend retries for infrastructure failures.

Step retries

When an uncaught exception occurs within a step, the SDK automatically retries the step based on the configured retry strategy. Step retries are checkpointed operations that allow the SDK to suspend execution and resume later without losing progress.

Step retry behavior

The following table describes how the SDK handles exceptions within steps:

Scenario	What happens	Metering impact
Exception in step with remaining retry attempts	The SDK creates a checkpoin t for the retry and suspends the function. On the next invocation, the step retries with the configured backoff delay.	1 operation + error payload size
Exception in step with no remaining retry attempts	The step fails and throws an exception. If your handler code doesn't catch this exception, the entire execution fails.	1 operation + error payload size

When a step needs to retry, the SDK checkpoints the retry state and exits the Lambda invocation if no other work is running. This allows the SDK to implement backoff delays without consuming compute resources. The function resumes automatically after the backoff period.

Configuring step retry strategies

Configure retry strategies to control how steps handle failures. You can specify maximum attempts, backoff intervals, and conditions for retrying.

Exponential backoff with max attempts:

TypeScript

```
const result = await context.step('call-api', async () => {
  const response = await fetch('https://api.example.com/data');
  if (!response.ok) throw new Error(`API error: ${response.status}`);
  return await response.json();
}, {
  retryStrategy: (error, attemptCount) => {
    if (attemptCount >= 5) {
      return { shouldRetry: false };
    }
    // Exponential backoff: 2s, 4s, 8s, 16s, 32s (capped at 300s)
    const delay = Math.min(2 * Math.pow(2, attemptCount - 1), 300);
```

```
    return { shouldRetry: true, delay: { seconds: delay } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    if attempt_count >= 5:
        return RetryDecision(should_retry=False)
    # Exponential backoff: 2s, 4s, 8s, 16s, 32s (capped at 300s)
    delay = min(2 * (2 ** (attempt_count - 1)), 300)
    return RetryDecision(should_retry=True, delay=delay)

result = context.step(
    lambda _: call_external_api(),
    name='call-api',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

Fixed interval backoff:

TypeScript

```
const orders = await context.step('query-orders', async () => {
  return await queryDatabase(event.userId);
}, {
  retryStrategy: (error, attemptCount) => {
    if (attemptCount >= 3) {
      return { shouldRetry: false };
    }
    return { shouldRetry: true, delay: { seconds: 5 } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    if attempt_count >= 3:
        return RetryDecision(should_retry=False)
    return RetryDecision(should_retry=True, delay=5)

orders = context.step(
    lambda _: query_database(event['userId']),
    name='query-orders',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

Conditional retry (retry only specific errors):

TypeScript

```
const result = await context.step('call-rate-limited-api', async () => {
  const response = await fetch('https://api.example.com/data');

  if (response.status === 429) throw new Error('RATE_LIMIT');
  if (response.status === 504) throw new Error('TIMEOUT');
  if (!response.ok) throw new Error(`API_ERROR_${response.status}`);

  return await response.json();
}, {
  retryStrategy: (error, attemptCount) => {
    // Only retry rate limits and timeouts
    const isRetryable = error.message === 'RATE_LIMIT' || error.message ===
 'TIMEOUT';

    if (!isRetryable || attemptCount >= 3) {
      return { shouldRetry: false };
    }

    // Exponential backoff: 1s, 2s, 4s (capped at 30s)
    const delay = Math.min(Math.pow(2, attemptCount - 1), 30);
    return { shouldRetry: true, delay: { seconds: delay } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    # Only retry rate limits and timeouts
    is_retryable = str(error) in ['RATE_LIMIT', 'TIMEOUT']

    if not is_retryable or attempt_count >= 3:
        return RetryDecision(should_retry=False)

    # Exponential backoff: 1s, 2s, 4s (capped at 30s)
    delay = min(2 ** (attempt_count - 1), 30)
    return RetryDecision(should_retry=True, delay=delay)

result = context.step(
    lambda _: call_rate_limited_api(),
    name='call-rate-limited-api',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

Disable retries:

TypeScript

```
const isDuplicate = await context.step('check-duplicate', async () => {
  return await checkIfOrderExists(event.orderId);
}, {
  retryStrategy: () => ({ shouldRetry: false })
});
```

Python

```
is_duplicate = context.step(
    lambda _: check_if_order_exists(event['orderId']),
    name='check-duplicate',
    config=StepConfig(
        retry_strategy=lambda error, attempt: {'should_retry': False}
    )
```

```
)
```

When the retry strategy returns `shouldRetry: false`, the step fails immediately without retries. Use this for operations that should not be retried, such as idempotency checks or operations with side effects that cannot be safely repeated.

Exceptions outside steps

When an uncaught exception occurs in your handler code but outside any step, the SDK marks the execution as failed. This ensures errors in your application logic are properly captured and reported.

Scenario	What happens	Metering impact
Exception in handler code outside any step	The SDK marks the execution as FAILED and returns the error. The exception is not automatically retried.	Error payload size

To enable automatic retry for error-prone code, wrap it in a step with a retry strategy. Steps provide automatic retry with configurable backoff, while code outside steps fails immediately.

Backend retries

Backend retries occur when Lambda encounters infrastructure failures, runtime errors, or when the SDK cannot communicate with the durable execution service. Lambda automatically retries these failures to help your durable functions can recover from transient infrastructure issues.

Backend retry scenarios

Lambda automatically retries your function when it encounters the following scenarios:

- **Internal service errors** - When Lambda or the durable execution service returns a 5xx error, indicating a temporary service issue.
- **Throttling** - When your function is throttled due to concurrency limits or service quotas.
- **Timeouts** - When the SDK cannot reach the durable execution service within the timeout period.
- **Sandbox initialization failures** - When Lambda cannot initialize the execution environment.

- **Runtime errors** - When the Lambda runtime encounters errors outside your function code, such as out-of-memory errors or process crashes.
- **Invalid checkpoint token errors** - When the checkpoint token is no longer valid, typically due to service-side state changes.

The following table describes how the SDK handles these scenarios:

Scenario	What happens	Metering impact
Runtime error outside durable handler (OOM, timeout, crash)	Lambda automatically retries the invocation. The SDK replays from the last checkpoint, skipping completed steps.	Error payload size + 1 operation per retry
Service error (5xx) or timeout when calling `CheckpointDurableExecution` / `GetDurableExecutionState` APIs	Lambda automatically retries the invocation. The SDK replays from the last checkpoint.	Error payload size + 1 operation per retry
Throttling (429) or invalid checkpoint token when calling `CheckpointDurableExecution` / `GetDurableExecutionState` APIs	Lambda automatically retries the invocation with exponential backoff. The SDK replays from the last checkpoint.	Error payload size + 1 operation per retry
Client error (4xx, except 429 and invalid token) when `CheckpointDurableExecution` / `GetDurableExecutionState` APIs	The SDK marks the execution as FAILED. No automatic retry occurs because the error indicates a permanent issue.	Error payload size

Backend retries use exponential backoff and continue until the function succeeds or the execution timeout is reached. During replay, the SDK skips completed checkpoints and continues execution from the last successful operation, ensuring your function doesn't re-execute completed work.

Retry best practices

Follow these best practices when configuring retry strategies:

- **Configure explicit retry strategies** - Don't rely on default retry behavior in production. Configure explicit retry strategies with appropriate max attempts and backoff intervals for your use case.
- **Use conditional retries** - Implement `shouldRetry` logic to retry only transient errors (rate limits, timeouts) and fail fast on permanent errors (validation failures, not found).
- **Set appropriate max attempts** - Balance between resilience and execution time. Too many retries can delay failure detection, while too few can cause unnecessary failures.
- **Use exponential backoff** - Exponential backoff reduces load on downstream services and increases the likelihood of recovery from transient failures.
- **Wrap error-prone code in steps** - Code outside steps cannot be automatically retried. Wrap external API calls, database queries, and other error-prone operations in steps with retry strategies.
- **Monitor retry metrics** - Track step retry operations and execution failures in Amazon CloudWatch to identify patterns and optimize retry strategies.

Idempotency

Durable functions provide built-in idempotency for execution starts through execution names. When you provide an execution name, Lambda uses it to prevent duplicate executions and enable safe retries of invocation requests. Steps have at-least-once execution semantics by default—during replay, the SDK returns checkpointed results without re-executing completed steps, but your business logic must be idempotent to handle potential retries before completion.

> **Note**
>
> Lambda event source mappings (ESM) don't support idempotency at launch. Therefore, each invocation (including retries) starts a new durable execution. To ensure idempotent execution with event source mappings, either implement idempotency logic in your function code such as with Powertools for AWS Lambda or use a regular Lambda function

as proxy (dispatcher) to invoke a durable function with an idempotency key (execution name parameter).

Execution names

You can provide an execution name when invoking a durable function. The execution name acts as an idempotency key, allowing you to safely retry invocation requests without creating duplicate executions. If you don't provide a name, Lambda generates a unique execution ID automatically.

Execution names must be unique within your account and region. When you invoke a function with an execution name that already exists, Lambda behavior depends on the existing execution's state and whether the payload matches.

Idempotency behavior

The following table describes how Lambda handles invocation requests based on whether you provide an execution name, the existing execution state, and whether the payload matches:

Scenario	Name provided?	Existing execution status	Payload identical?	Behavior
1	No	N/A	N/A	**New execution started:** Lambda generates a unique execution ID and starts a new execution
2	Yes	Never existed or retention expired	N/A	**New execution started:** Lambda starts a new execution with the provided name
3	Yes	Running	Yes	**Idempotent start:** Lambda returns the existing execution

Scenario	Name provided?	Existing execution status	Payload identical?	Behavior
				information without starting a duplicate . For synchronous invocations, this acts as a reattach to the running execution
4	Yes	Running	No	**Error:** Lambda returns `DurableExecutionAlreadyExists` error because an execution with this name is already running with different payload
5	Yes	Closed (succeeded, failed, stopped, or timed out)	Yes	**Idempotent start:** Lambda returns the existing execution information without starting a new execution. The closed execution result is returned

Scenario	Name provided?	Existing execution status	Payload identical?	Behavior
6	Yes	Closed (succeeded, failed, stopped, or timed out)	No	**Error:** Lambda returns `DurableExecutionAlreadyExists` error because an execution with this name already completed with different payload

Note

Scenarios 3 and 5 demonstrate idempotent behavior where Lambda safely handles duplicate invocation requests by returning existing execution information instead of creating duplicates.

Step idempotency

Steps have at-least-once execution semantics by default. When your function replays after a wait, callback, or failure, the SDK checks each step against the checkpoint log. For steps that already completed, the SDK returns the checkpointed result without re-executing the step logic. However, if a step fails or the function is interrupted before the step completes, the step may execute multiple times.

Your business logic wrapped in steps must be idempotent to handle potential retries. Use idempotency keys to ensure operations like payments or database writes execute only once, even if the step retries.

Example: Using idempotency keys in steps

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';
import { randomUUID } from 'crypto';
```

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate idempotency key once
    const idempotencyKey = await context.step('generate-key', async () => {
      return randomUUID();
    });

    // Use idempotency key in payment API to prevent duplicate charges
    const payment = await context.step('process-payment', async () => {
      return paymentAPI.charge({
        amount: event.amount,
        idempotencyKey: idempotencyKey
      });
    });

    return { statusCode: 200, payment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate idempotency key once
    idempotency_key = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-key'
    )

    # Use idempotency key in payment API to prevent duplicate charges
    payment = context.step(
        lambda _: payment_api.charge(
            amount=event['amount'],
            idempotency_key=idempotency_key
        ),
        name='process-payment'
    )
```

```
    return {'statusCode': 200, 'payment': payment}
```

You can configure steps to use at-most-once execution semantics by setting the execution mode to `AT_MOST_ONCE_PER_RETRY`. This ensures the step executes at most once per retry attempt, but may not execute at all if the function is interrupted before the step completes.

The SDK enforces deterministic replay by validating that step names and order match the checkpoint log during replay. If your code attempts to execute steps in a different order or with different names, the SDK throws a `NonDeterministicExecutionError`.

How replay works with completed steps:

1. First invocation: Function executes step A, creates checkpoint, then waits
2. Second invocation (after wait): Function replays from beginning, step A returns checkpointed result instantly without re-executing, then continues to step B
3. Third invocation (after another wait): Function replays from beginning, steps A and B return checkpointed results instantly, then continues to step C

This replay mechanism ensures that completed steps don't re-execute, but your business logic must still be idempotent to handle retries before completion.

Testing Lambda durable functions

AWS provides dedicated testing SDKs for durable functions that let you run and inspect executions both locally and in the cloud. Install the testing SDK for your language:

TypeScript

```
npm install --save-dev @aws/durable-execution-sdk-js-testing
```

For complete documentation and examples, see the TypeScript testing SDK on GitHub.

Python

```
pip install aws-durable-execution-sdk-python-testing
```

For complete documentation and examples, see the Python testing SDK on GitHub.

The testing SDK provides two testing modes: local testing for fast unit tests, and cloud testing for integration tests against deployed functions.

Local testing

Local testing runs your durable functions in your development environment without requiring deployed resources. The test runner runs your function code directly and captures all operations for inspection.

Use local testing for unit tests, test-driven development, and CI/CD pipelines. Tests run locally without network latency or additional costs.

Example test:

TypeScript

```
import { withDurableExecution } from '@aws/durable-execution-sdk-js';
import { DurableFunctionTestRunner } from '@aws/durable-execution-sdk-js-testing';

const handler = withDurableExecution(async (event, context) => {
  const result = await context.step('calculate', async () => {
    return event.a + event.b;
  });
  return result;
});

test('addition works correctly', async () => {
  const runner = new DurableFunctionTestRunner({ handler });
  const result = await runner.run({ a: 5, b: 3 });

  expect(result.status).toBe('SUCCEEDED');
  expect(result.result).toBe(8);

  const step = result.getStep('calculate');
  expect(step.result).toBe(8);
});
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
from aws_durable_execution_sdk_python_testing import DurableFunctionTestRunner
from aws_durable_execution_sdk_python.execution import InvocationStatus

@durable_execution
def handler(event: dict, context: DurableContext) -> int:
    result = context.step(lambda _: event["a"] + event["b"], name="calculate")
    return result

def test_addition():
    runner = DurableFunctionTestRunner(handler=handler)
    with runner:
        result = runner.run(input={"a": 5, "b": 3}, timeout=10)

    assert result.status is InvocationStatus.SUCCEEDED
    assert result.result == 8

    step = result.get_step("calculate")
    assert step.result == 8
```

The test runner captures execution state including the final result, individual step results, wait operations, callbacks, and any errors. You can inspect operations by name or iterate through all operations to verify execution behavior.

Execution stores

The testing SDK uses execution stores to persist test execution data. By default, tests use an in-memory store that's fast and requires no cleanup. For debugging or analyzing execution history, you can use a filesystem store that saves executions as JSON files.

In-memory store (default):

The in-memory store keeps execution data in memory during test runs. Data is lost when tests complete, making it ideal for standard unit tests and CI/CD pipelines where you don't need to inspect executions after tests finish.

Filesystem store:

The filesystem store persists execution data to disk as JSON files. Each execution is saved in a separate file, making it easy to inspect execution history after tests complete. Use the filesystem store when debugging complex test failures or analyzing execution patterns over time.

Configure the store using environment variables:

```
# Use filesystem store
export AWS_DEX_STORE_TYPE=filesystem
export AWS_DEX_STORE_PATH=./test-executions

# Run tests
pytest tests/
```

Execution files are stored with sanitized names and contain the complete execution state including operations, checkpoints, and results. The filesystem store automatically creates the storage directory if it doesn't exist.

Cloud testing

Cloud testing invokes deployed durable functions in AWS and retrieves their execution history using the Lambda API. Use cloud testing to verify behavior in production-like environments with real AWS services and configurations.

Cloud testing requires a deployed function and AWS credentials with permissions to invoke functions and read execution history:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction",
                "lambda:GetDurableExecution",
                "lambda:GetDurableExecutionHistory"
            ],
```

```
            "Resource": "arn:aws:lambda:region:account-id:function:function-name"
        }
    ]
}
```

Example cloud test:

TypeScript

```
import { DurableFunctionCloudTestRunner } from '@aws/durable-execution-sdk-js-
testing';

test('deployed function processes orders', async () => {
  const runner = new DurableFunctionCloudTestRunner({
    functionName: 'order-processor',
    region: 'us-east-1'
  });

  const result = await runner.run({ orderId: 'order-123' });

  expect(result.status).toBe('SUCCEEDED');
  expect(result.result.status).toBe('completed');
});
```

Python

```
from aws_durable_execution_sdk_python_testing import (
    DurableFunctionCloudTestRunner,
    DurableFunctionCloudTestRunnerConfig
)

def test_deployed_function():
    config = DurableFunctionCloudTestRunnerConfig(
        function_name="order-processor",
        region="us-east-1"
    )
    runner = DurableFunctionCloudTestRunner(config=config)

    result = runner.run(input={"orderId": "order-123"})
```

```
    assert result.status is InvocationStatus.SUCCEEDED
    assert result.result["status"] == "completed"
```

Cloud tests invoke the actual deployed function and retrieve execution history from AWS. This lets you verify integration with other AWS services, validate performance characteristics, and test with production-like data and configurations.

What to test

Test durable functions by verifying execution outcomes, operation behavior, and error handling. Focus on business logic correctness rather than implementation details.

Verify execution results: Check that functions return the expected values for given inputs. Test both successful executions and error cases to ensure functions handle invalid input appropriately.

Inspect operation execution: Verify that steps, waits, and callbacks execute as expected. Check step results to ensure intermediate operations produce correct values. Validate that wait operations are configured with appropriate timeouts and that callbacks are created with correct settings.

Test error handling: Verify functions fail correctly with descriptive error messages when given invalid input. Test retry behavior by simulating transient failures and confirming operations retry appropriately. Check that permanent failures don't trigger unnecessary retries.

Validate workflows: For multi-step workflows, verify operations execute in the correct order. Test conditional branching to ensure different execution paths work correctly. Validate parallel operations execute concurrently and produce expected results.

The SDK documentation repositories contain extensive examples of testing patterns including multi-step workflows, error scenarios, timeout handling, and polling patterns.

Testing strategy

Use local testing for unit tests during development and in CI/CD pipelines. Local tests run fast, don't require AWS credentials, and provide immediate feedback on code changes. Write local tests to verify business logic, error handling, and operation behavior.

Use cloud testing for integration tests before deploying to production. Cloud tests verify behavior with real AWS services and configurations, validate performance characteristics, and test end-to-end workflows. Run cloud tests in staging environments to catch integration issues before they reach production.

Mock external dependencies in local tests to isolate function logic and keep tests fast. Use cloud tests to verify actual integration with external services like databases, APIs, and other AWS services.

Write focused tests that verify one specific behavior. Use descriptive test names that explain what's being tested. Group related tests together and use test fixtures for common setup code. Keep tests simple and avoid complex test logic that's hard to understand.

Debugging failures

When tests fail, inspect the execution result to understand what went wrong. Check the execution status to see if the function succeeded, failed, or timed out. Read error messages to understand the failure cause.

Inspect individual operation results to find where behavior diverged from expectations. Check step results to see what values were produced. Verify operation ordering to confirm operations executed in the expected sequence. Count operations to ensure the right number of steps, waits, and callbacks were created.

Common issues include non-deterministic code that produces different results on replay, shared state through global variables that breaks during replay, and missing operations due to conditional logic errors. Use standard debuggers and logging to step through function code and track execution flow.

For cloud tests, inspect execution history in CloudWatch Logs to see detailed operation logs. Use tracing to track execution flow across services and identify bottlenecks.

Monitoring durable functions

You can monitor your durable functions using CloudWatch metrics, CloudWatch Logs, and tracing. Because durable functions can run for extended periods and span multiple function invocations, monitoring them requires understanding their unique execution patterns, including checkpoints, state transitions, and replay behavior.

CloudWatch metrics

Lambda automatically publishes metrics to CloudWatch at no additional charge. Durable functions provide additional metrics beyond standard Lambda metrics to help you monitor long-running workflows, state management, and resource utilization.

Durable execution metrics

Lambda emits the following metrics for durable executions:

Metric	Description
`ApproximateRunningDurableExecutions`	Number of durable executions in the RUNNING state
`ApproximateRunningDurableExecutionsUtilization`	Percentage of your account's maximum running durable executions quota currently in use
`DurableExecutionDuration`	Elapsed wall-clock time in milliseconds that a durable execution remained in the RUNNING state
`DurableExecutionStarted`	Number of durable executions that started
`DurableExecutionStopped`	Number of durable executions stopped using the StopDurableExecution API
`DurableExecutionSucceeded`	Number of durable executions that completed successfully
`DurableExecutionFailed`	Number of durable executions that completed with a failure
`DurableExecutionTimedOut`	Number of durable executions that exceeded their configured execution timeout
`DurableExecutionOperations`	Cumulative number of operations performed within a durable execution (max: 3,000)

Metric	Description
DurableExecutionStorageWrittenBytes	Cumulative amount of data in bytes persisted by a durable execution (max: 100 MB)

CloudWatch metrics

Lambda emits standard invocation, performance, and concurrency metrics for durable functions. Because a durable execution can span multiple function invocations as it progresses through checkpoints and replays, these metrics behave differently than for standard functions:

- **Invocations:** Counts each function invocation, including replays. A single durable execution can generate multiple invocation data points.
- **Duration:** Measures each function invocation separately. Use `DurableExecutionDuration` for total time taken by a single durable execution.
- **Errors:** Tracks function invocation failures. Use `DurableExecutionFailed` for execution-level failures.

For a complete list of standard Lambda metrics, see Types of metrics for Lambda functions.

Creating CloudWatch alarms

Create CloudWatch alarms to notify you when metrics exceed thresholds. Common alarms include:

- `ApproximateRunningDurableExecutionsUtilization` exceeds 80% of your quota
- `DurableExecutionFailed` increases above a threshold
- `DurableExecutionTimedOut` indicates executions are timing out
- `DurableExecutionStorageWrittenBytes` approaches storage limits

For more information, see Using CloudWatch alarms..

EventBridge events

Lambda publishes durable execution status change events to EventBridge. You can use these events to trigger workflows, send notifications, or track execution lifecycle changes across your durable functions.

Durable execution status change events

Lambda emits an event to EventBridge whenever a durable execution changes status. These events have the following characteristics:

- **Source:** `aws.lambda`
- **Detail type:** `Durable Execution Status Change`

Status change events are published for the following execution states:

- `RUNNING` - Execution started
- `SUCCEEDED` - Execution completed successfully
- `STOPPED` - Execution stopped using the StopDurableExecution API
- `FAILED` - Execution failed with an error
- `TIMED_OUT` - Execution exceeded the configured timeout

The following example shows a durable execution status change event:

```
{
  "version": "0",
  "id": "d019b03c-a8a3-9d58-85de-241e96206538",
  "detail-type": "Durable Execution Status Change",
  "source": "aws.lambda",
  "account": "123456789012",
  "time": "2025-11-20T13:08:22Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "durableExecutionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST/durable-execution/090c4189-b18b-4296-9d0c-cfd01dc3a122/9f7d84c9-ea3d-3ffc-
b3e5-5ec51c34ffc9",
    "durableExecutionName": "order-123",
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:2",
    "status": "RUNNING",
    "startTimestamp": "2025-11-20T13:08:22.345Z"
  }
}
```

For terminal states (`SUCCEEDED`, `STOPPED`, `FAILED`, `TIMED_OUT`), the event includes an `endTimestamp` field indicating when the execution completed.

Creating EventBridge rules

Create rules to route durable execution status change events to targets like Amazon Simple Notification Service, Amazon Simple Queue Service, or other Lambda functions.

The following example creates a rule that matches all durable execution status changes:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"]
}
```

The following example creates a rule that matches only failed executions:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "status": ["FAILED"]
  }
}
```

The following example creates a rule that matches status changes for a specific function:

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "functionArn": [{
      "prefix": "arn:aws:lambda:us-east-1:123456789012:function:my-function"
    }]
  }
}
```

For more information about creating rules, see Amazon EventBridge tutorials in the EventBridge User Guide.

AWS X-Ray tracing

You can enable X-Ray tracing on your durable functions. Lambda passes the X-Ray trace header to the durable execution, allowing you to trace requests across your workflow.

To enable X-Ray; tracing using the Lambda console, choose your function, then choose Configuration, Monitoring and operations tools, and turn on Active tracing under X-Ray.

To enable X-Ray tracing using the AWS CLI:

```
aws lambda update-function-configuration \
    --function-name my-durable-function \
    --tracing-config Mode=Active
```

To enable AWS X-Ray tracing using AWS SAM:

```
Resources:
  MyDurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      DurableConfig:
        ExecutionTimeout: 3600
```

For more information about X-Ray, see the AWS X-Ray Developer Guide.

Best practices for Lambda durable functions

Durable functions use a replay-based execution model that requires different patterns than traditional Lambda functions. Follow these best practices to build reliable, cost-effective workflows.

Write deterministic code

During replay, your function runs from the beginning and must follow the same execution path as the original run. Code outside durable operations must be deterministic, producing the same results given the same inputs.

Wrap non-deterministic operations in steps:

- Random number generation and UUIDs
- Current time or timestamps
- External API calls and database queries
- File system operations

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';
import { randomUUID } from 'crypto';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate transaction ID inside a step
    const transactionId = await context.step('generate-transaction-id', async () =>
 {
      return randomUUID();
    });

    // Use the same ID throughout execution, even during replay
    const payment = await context.step('process-payment', async () => {
      return processPayment(event.amount, transactionId);
    });

    return { statusCode: 200, transactionId, payment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate transaction ID inside a step
    transaction_id = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-transaction-id'
    )

    # Use the same ID throughout execution, even during replay
    payment = context.step(
        lambda _: process_payment(event['amount'], transaction_id),
        name='process-payment'
    )

    return {'statusCode': 200, 'transactionId': transaction_id, 'payment': payment}
```

Important

Don't use global variables or closures to share state between steps. Pass data through return values. Global state breaks during replay because steps return cached results but global variables reset.

Avoid closure mutations: Variables captured in closures can lose mutations during replay. Steps return cached results, but variable updates outside the step aren't replayed.

TypeScript

```
// # WRONG: Mutations lost on replay
export const handler = withDurableExecution(async (event, context) => {
  let total = 0;

  for (const item of items) {
    await context.step(async () => {
      total += item.price; // ## Mutation lost on replay!
      return saveItem(item);
    });
  }
```

```
  return { total }; // Inconsistent value!
});

// # CORRECT: Accumulate with return values
export const handler = withDurableExecution(async (event, context) => {
  let total = 0;

  for (const item of items) {
    total = await context.step(async () => {
      const newTotal = total + item.price;
      await saveItem(item);
      return newTotal; // Return updated value
    });
  }

  return { total }; // Consistent!
});

// # EVEN BETTER: Use map for parallel processing
export const handler = withDurableExecution(async (event, context) => {
  const results = await context.map(
    items,
    async (ctx, item) => {
      await ctx.step(async () => saveItem(item));
      return item.price;
    }
  );

  const total = results.getResults().reduce((sum, price) => sum + price, 0);
  return { total };
});
```

Python

```
# # WRONG: Mutations lost on replay
@durable_execution
def handler(event, context: DurableContext):
    total = 0

    for item in items:
```

```
        context.step(
            lambda _: save_item_and_mutate(item, total),  # ## Mutation lost on
 replay!
            name=f'save-item-{item["id"]}'
        )

    return {'total': total}  # Inconsistent value!

# # CORRECT: Accumulate with return values
@durable_execution
def handler(event, context: DurableContext):
    total = 0

    for item in items:
        total = context.step(
            lambda _: save_item_and_return_total(item, total),
            name=f'save-item-{item["id"]}'
        )

    return {'total': total}  # Consistent!

# # EVEN BETTER: Use map for parallel processing
@durable_execution
def handler(event, context: DurableContext):
    def process_item(ctx, item):
        ctx.step(lambda _: save_item(item))
        return item['price']

    results = context.map(items, process_item)
    total = sum(results.get_results())

    return {'total': total}
```

Design for idempotency

Operations may execute multiple times due to retries or replay. Non-idempotent operations cause duplicate side effects like charging customers twice or sending multiple emails.

Use idempotency tokens: Generate tokens inside steps and include them with external API calls to prevent duplicate operations.

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate idempotency token once
    const idempotencyToken = await context.step('generate-idempotency-token', async
 () => {
      return crypto.randomUUID();
    });

    // Use token to prevent duplicate charges
    const charge = await context.step('charge-payment', async () => {
      return paymentService.charge({
        amount: event.amount,
        cardToken: event.cardToken,
        idempotencyKey: idempotencyToken
      });
    });

    return { statusCode: 200, charge };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate idempotency token once
    idempotency_token = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-idempotency-token'
    )

    # Use token to prevent duplicate charges
```

```
    def charge_payment(_):
        return payment_service.charge(
            amount=event['amount'],
            card_token=event['cardToken'],
            idempotency_key=idempotency_token
        )

    charge = context.step(charge_payment, name='charge-payment')

    return {'statusCode': 200, 'charge': charge}
```

Use at-most-once semantics: For critical operations that must never duplicate (financial transactions, inventory deductions), configure at-most-once execution mode.

TypeScript

```
// Critical operation that must not duplicate
await context.step('deduct-inventory', async () => {
  return inventoryService.deduct(event.productId, event.quantity);
}, {
  executionMode: 'AT_MOST_ONCE_PER_RETRY'
});
```

Python

```
# Critical operation that must not duplicate
context.step(
    lambda _: inventory_service.deduct(event['productId'], event['quantity']),
    name='deduct-inventory',
    config=StepConfig(execution_mode='AT_MOST_ONCE_PER_RETRY')
)
```

Database idempotency: Use check-before-write patterns, conditional updates, or upsert operations to prevent duplicate records.

Manage state efficiently

Every checkpoint saves state to persistent storage. Large state objects increase costs, slow checkpointing, and impact performance. Store only essential workflow coordination data.

Keep state minimal:

- Store IDs and references, not full objects
- Fetch detailed data within steps as needed
- Use Amazon S3 or DynamoDB for large data, pass references in state
- Avoid passing large payloads between steps

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Store only the order ID, not the full order object
    const orderId = event.orderId;

    // Fetch data within each step as needed
    await context.step('validate-order', async () => {
      const order = await orderService.getOrder(orderId);
      return validateOrder(order);
    });

    await context.step('process-payment', async () => {
      const order = await orderService.getOrder(orderId);
      return processPayment(order);
    });

    return { statusCode: 200, orderId };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext

@durable_execution
def handler(event, context: DurableContext):
    # Store only the order ID, not the full order object
    order_id = event['orderId']

    # Fetch data within each step as needed
    context.step(
        lambda _: validate_order(order_service.get_order(order_id)),
        name='validate-order'
    )

    context.step(
        lambda _: process_payment(order_service.get_order(order_id)),
        name='process-payment'
    )

    return {'statusCode': 200, 'orderId': order_id}
```

Design effective steps

Steps are the fundamental unit of work in durable functions. Well-designed steps make workflows easier to understand, debug, and maintain.

Step design principles:

- **Use descriptive names** - Names like `validate-order` instead of `step1` make logs and errors easier to understand
- **Keep names static** - Don't use dynamic names with timestamps or random values. Step names must be deterministic for replay
- **Balance granularity** - Break complex operations into focused steps, but avoid excessive tiny steps that increase checkpoint overhead
- **Group related operations** - Operations that should succeed or fail together belong in the same step

Use wait operations efficiently

Wait operations suspend execution without consuming resources or incurring costs. Use them instead of keeping Lambda running.

Time-based waits: Use `context.wait()` for delays instead of `setTimeout` or `sleep`.

External callbacks: Use `context.waitForCallback()` when waiting for external systems. Always set timeouts to prevent indefinite waits.

Polling: Use `context.waitForCondition()` with exponential backoff to poll external services without overwhelming them.

TypeScript

```
// Wait 24 hours without cost
await context.wait({ seconds: 86400 });

// Wait for external callback with timeout
const result = await context.waitForCallback(
  'external-job',
  async (callbackId) => {
    await externalService.submitJob({
      data: event.data,
      webhookUrl: `https://api.example.com/callbacks/${callbackId}`
    });
  },
  { timeout: { seconds: 3600 } }
);
```

Python

```
# Wait 24 hours without cost
context.wait(86400)

# Wait for external callback with timeout
result = context.wait_for_callback(
    lambda callback_id: external_service.submit_job(
        data=event['data'],
```

```
        webhook_url=f'https://api.example.com/callbacks/{callback_id}'
    ),
    name='external-job',
    config=WaitForCallbackConfig(timeout_seconds=3600)
)
```

Additional considerations

Error handling: Retry transient failures like network timeouts and rate limits. Don't retry permanent failures like invalid input or authentication errors. Configure retry strategies with appropriate max attempts and backoff rates. For detailed examples, see Error handling and retries.

Performance: Minimize checkpoint size by storing references instead of full payloads. Use `context.parallel()` and `context.map()` to execute independent operations concurrently. Batch related operations to reduce checkpoint overhead.

Versioning: Invoke functions with version numbers or aliases to pin executions to specific code versions. Ensure new code versions can handle state from older versions. Don't rename steps or change their behavior in ways that break replay.

Serialization: Use JSON-compatible types for operation inputs and results. Convert dates to ISO strings and custom objects to plain objects before passing them to durable operations.

Monitoring: Enable structured logging with execution IDs and step names. Set up CloudWatch alarms for error rates and execution duration. Use tracing to identify bottlenecks. For detailed guidance, see Monitoring and debugging.

Testing: Test happy path, error handling, and replay behavior. Test timeout scenarios for callbacks and waits. Use local testing to reduce iteration time. For detailed guidance, see Testing durable functions.

Common mistakes to avoid: Don't nest `context.step()` calls, use child contexts instead. Wrap non-deterministic operations in steps. Always set timeouts for callbacks. Balance step granularity with checkpoint overhead. Store references instead of large objects in state.

Additional resources

- Python SDK documentation - Complete API reference, testing patterns, and advanced examples

- TypeScript SDK documentation - Complete API reference, testing patterns, and advanced examples

Lambda Managed Instances

Lambda Managed Instances enables you to run Lambda functions on your current-generation Amazon EC2 instances, including Graviton4, network-optimized instances, and other specialized compute options, without managing instance lifecycles, operating system and language runtime patching, routing, load balancing, or scaling policies. With Lambda Managed Instances, you benefit from EC2 pricing advantages, including EC2 Savings Plans and Reserved Instances.

For a list of supported instance types, go to the AWS Lambda Pricing page and select your AWS Region.

Key capabilities

Lambda Managed Instances provides the following capabilities:

- **Choose suitable instances** - Select appropriate instances based on performance and cost requirements, including access to the latest CPUs like Graviton4, configurable memory-CPU ratios, and high-bandwidth networking.
- **Automatic provisioning** - AWS automatically provisions suitable instances and spins up function execution environments.
- **Dynamic scaling** - Instances scale dynamically based on your function traffic patterns.
- **Fully managed experience** - AWS handles infrastructure management, scaling, patching, and routing, with the same extensive event-source integrations you're familiar with.

When to use Lambda Managed Instances

Consider Lambda Managed Instances for the following use cases:

- **High volume-predictable workloads** - Ideal for steady-state workloads without unexpected traffic spikes. Lambda Managed Instances scale to handle traffic doubling within five minutes by default.
- **Performance-critical applications** - Access to latest CPUs, varying memory-CPU ratios, and high network throughput
- **Regulatory requirements** - Granular governance needs with control over VPC and instance placement

- **Variety of applications** - Event-driven applications, media/data processing, web applications, and legacy workloads migrating to serverless

How it works

Lambda Managed Instances uses capacity providers as the foundation for running your functions:

1. **Create a capacity provider** - Define where your functions run by specifying VPC configuration and optionally, instance requirements, and scaling configuration
2. **Create your function** - Create Lambda functions as usual and attach them to a capacity provider
3. **Publish a function version** - Function versions become active on capacity provider instances once published

When you publish a function version with a capacity provider, Lambda launches Managed Instances in your account. It launches three instances by default for AZ resiliency and starts three execution environments before marking your function version ACTIVE. If you attach a function to an existing capacity provider that is already running other functions, Lambda may not spin up new instances if the available instances already have capacity to accommodate the new function's execution environments.

Concurrency model

Lambda Managed Instances support multi-concurrent invocations, where one execution environment can handle multiple invocations at the same time. This differs from the Lambda (default) compute type, which provides a single concurrency model where one execution environment can run a maximum of one invoke at a time. Multi-concurrency yields better utilization of your underlying EC2 instances and is especially beneficial for IO-heavy applications like web services or batch jobs. This change in execution model means that thread safety, state management, and context isolation must be handled differently depending on the runtime.

Tenancy and isolation

Lambda (default) compute type is multi-tenant, making use of Firecracker microVM technology to provide isolation between execution environments running on shared Lambda fleets. Lambda Managed Instances run in your account, providing the latest EC2 hardware and pricing options.

Managed Instances use containers running on EC2 Nitro instances to provide isolation rather than Firecracker. Capacity providers serve as the security boundary for Lambda functions. Functions execute in containers within instances.

Understanding managed instances

Lambda Managed Instances functions run on EC2 managed instances in your account. These instances are fully managed by Lambda, which means you have restricted permissions on them compared to standard EC2 instances. You can identify Lambda Managed Instances in your account by:

- The presence of the `Operator` field in EC2 `DescribeInstances` output
- The `aws:lambda:capacity-provider` tag on the instance

You cannot perform standard EC2 operations directly on these instances, such as terminating them manually. To destroy managed instances, delete the associated capacity provider. Lambda will then terminate the instances as part of the capacity provider deletion process.

Pricing

Lambda Managed Instances uses EC2-based pricing with a 15% management fee on top of the EC2 instance cost. This pricing model supports EC2 Savings Plans, Reserved Instances and any other pricing discounts applied to your EC2 usage. Refer to pricing page for additional details: https://aws.amazon.com/lambda/pricing/

Important: EC2 pricing discounts only apply to the underlying EC2 compute, not to the management fee.

How Lambda Managed Instances differs from the Lambda (default) compute type

Lambda Managed Instances changes how Lambda processes requests compared to Lambda (default).

Key differences:

	Lambda (default)	Lambda Managed Instances
Concurrency model	Single concurrency model where one execution environment can support a maximum of one invocation at a time	Multi-concurrent invocations where one execution environme nt can handle multiple invocatio ns simultaneously, increasing throughput especially for IO-heavy applications
Tenancy and isolation	Multi-tenant, using Firecracker microVM technology to provide isolation between execution environments running on shared Lambda fleets	Run in your account, using EC2 Nitro to provide isolation. Capacity providers serve as the security boundary, with functions executing in containers within instances
Pricing model	Per-request duration pricing	Instance-based pricing with EC2 pricing models, including On-Demand and Reserved Instances , and savings options such as Compute Savings Plans
Scaling behavior	Scales when there is no free execution environment to handle an incoming invocation (cold start). Scales to zero without traffic	Scales asynchronously based on CPU resource utilization only, without cold starts. Scales to minimum execution environments configured without traffic
Best suited for	Functions with bursty traffic that can handle some cold-start time, or applications without sustained load that benefit from scale to zero	High volume predictable traffic functions when you want the flexibility, pricing plans, and hardware options of EC2

Next steps

- Learn about capacity providers for Lambda Managed Instances

- Understand scaling for Lambda Managed Instances
- Review runtime-specific guides for Java, Node.js, and Python
- Configure VPC connectivity for your capacity providers
- Understand security and permissions for Lambda Managed Instances

Getting started with Lambda Managed Instances

Creating a Lambda Managed Instance function (console)

You can use the Lambda console to create a Managed Instance function that runs on Amazon EC2 instances managed by a capacity provider.

Important: Before creating a Managed Instance function, you must first create a capacity provider. These functions require a capacity provider to define the Amazon EC2 infrastructure that will run your functions.

To create a Lambda Managed Instance function (console)

1. Open the Lambda console.
2. Choose **Capacity providers** from the left navigation pane.
3. Choose **Create capacity provider**.
4. In the **Capacity provider settings** section, enter a name for your capacity provider.
5. Select VPC and permissions for your capacity provider. You can either use an existing or create a new one. For information about creating the required operator role, see Lambda Operator role for Lambda Managed Instances.
6. Expand **Advanced settings**.
7. Define your **Instance requirements** by choosing the processor architecture and instance types.
8. Under **Auto scaling**, specify the maximum number of EC2 vCPUs for your capacity provider. You can also choose **Manual instance scaling mode** to set your own scaling value for precise control.
9. Choose **Create capacity provider** to create a new one.
10. Next, choose **Create function**.
11. Select **Author from scratch**.
12. In the **Basic information** pane, provide a **Function name**.
13. For **Runtime**, choose any of the supported Runtimes.

14. Choose the **Architecture** for your function (same as the one you selected for capacity provider). By default, **x86_64**.
15. Under **Permissions**, ensure you have permission for the chosen **Execution role**. Otherwise, you can create a new role.
16. Under **Additional configurations**, pick the **Compute type** as **Lambda Managed Instances**.
17. Capacity provider ARN of the capacity provider you created in the previous steps should be pre-selected.
18. Choose **Memory size** and **Execution environment memory (GiB) per vCPU ratio**.
19. Choose **Create function**.

Your Lambda Managed Instance function is created and will provision capacity on your specified capacity provider. Function creation typically takes several minutes. Once complete, you can edit your function code and run your first test.

Creating a Lambda Managed Instance function (AWS CLI)

Prerequisites

Before you begin, make sure you have the following:

- **AWS CLI** – Install and configure the AWS CLI. For more information, see Installing or updating the latest version of the AWS CLI.
- **IAM permissions** – Your IAM user or role must have permissions to create Lambda functions, capacity providers, and pass IAM roles. Note that you'll also need `iam:CreateServiceLinkedRole` if it's the first time creating a capacity provider in the account or if the Service Linked Role (SLR) was deleted.

Step 1: Create the required IAM roles

Lambda Managed Instances require two IAM roles: an execution role for your function and an operator role for the capacity provider. The operator role allows Lambda to launch, terminate, and monitor Amazon EC2 instances on your behalf. The function execution role grants the function permissions to access other AWS services and resources.

To create the Lambda execution role

1. Create a trust policy document that allows Lambda to assume the role:

```
cat > lambda-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
```

2. Create the execution role:

```
aws iam create-role \
  --role-name MyLambdaExecutionRole \
  --assume-role-policy-document file://lambda-trust-policy.json
```

3. Attach the basic execution policy:

```
aws iam attach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

To create the capacity provider operator role

1. Create a trust policy document that allows Lambda to assume the operator role:

```
cat > operator-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
```

```
    }
  ]
}
EOF
```

2. Create the operator role:

```
aws iam create-role \
  --role-name MyCapacityProviderOperatorRole \
  --assume-role-policy-document file://operator-trust-policy.json
```

3. Attach the required EC2 permissions policy:

```
aws iam attach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator
```

Step 2: Set up VPC resources

Lambda Managed Instances run in your VPC and require a subnet and security group.

To create VPC resources

1. Create a VPC:

```
VPC_ID=$(aws ec2 create-vpc \
  --cidr-block 10.0.0.0/16 \
  --query 'Vpc.VpcId' \
  --output text)
```

2. Create a subnet:

```
SUBNET_ID=$(aws ec2 create-subnet \
  --vpc-id $VPC_ID \
  --cidr-block 10.0.1.0/24 \
  --query 'Subnet.SubnetId' \
  --output text)
```

3. Create a security group:

```
SECURITY_GROUP_ID=$(aws ec2 create-security-group \
  --group-name my-capacity-provider-sg \
```

```
  --description "Security group for Lambda Managed Instances" \
  --vpc-id $VPC_ID \
  --query 'GroupId' \
  --output text)
```

Note: Your Lambda Managed Instances functions require VPC configuration to access resources outside the VPC and to transmit telemetry data to CloudWatch Logs and X-Ray. For configuration details, see Networking for Lambda Managed Instances.

Step 3: Create a capacity provider

A capacity provider manages the EC2 instances that run your Lambda functions.

To create a capacity provider

```
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)

aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config SubnetIds=[$SUBNET_ID],SecurityGroupIds=[$SECURITY_GROUP_ID] \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::${ACCOUNT_ID}:role/
MyCapacityProviderOperatorRole \
  --instance-requirements Architectures=[x86_64] \
  --capacity-provider-scaling-config MaxVCpuCount=30
```

This command creates a capacity provider with the following configuration:

- **VPC configuration** – Specifies the subnet and security group for the EC2 instances
- **Permissions** – Defines the IAM role that Lambda uses to manage EC2 instances
- **Instance requirements** – Specifies the x86_64 architecture
- **Scaling configuration** – Sets a maximum of 30 vCPUs for the capacity provider

Step 4: Create a Lambda function with inline code

To create a function with inline code

1. First, create a simple Python function and package it inline:

```
# Create a temporary directory for the function code
mkdir -p /tmp/my-lambda-function
```

```
cd /tmp/my-lambda-function

# Create a simple Python handler
cat > lambda_function.py << 'EOF'
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': json.dumps({
            'message': 'Hello from Lambda Managed Instances!',
            'event': event
        })
    }
EOF

# Create a ZIP file
zip function.zip lambda_function.py
```

2. Create the Lambda function using the inline ZIP file:

```
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
REGION=$(aws configure get region)

aws lambda create-function \
  --function-name my-managed-instance-function \
  --package-type Zip \
  --runtime python3.13 \
  --handler lambda_function.lambda_handler \
  --zip-file fileb:///tmp/my-lambda-function/function.zip \
  --role arn:aws:iam::${ACCOUNT_ID}:role/MyLambdaExecutionRole \
  --architectures x86_64 \
  --memory-size 2048 \
  --ephemeral-storage Size=512 \
  --capacity-provider-config
 LambdaManagedInstancesCapacityProviderConfig={CapacityProviderArn=arn:aws:lambda:
${REGION}:${ACCOUNT_ID}:capacity-provider:my-capacity-provider}
```

The function is created with:

- **Runtime** – Python 3.13
- **Handler** – The `lambda_handler` function in `lambda_function.py`
- **Memory** – 2048 MB

- **Ephemeral storage** – 512 MB
- **Capacity provider** – Links to the capacity provider you created

Step 5: Publish a function version

To run your function on Lambda Managed Instances, you must publish a version.

To publish a function version

```
aws lambda publish-version \
  --function-name my-managed-instance-function
```

This command publishes version 1 of your function and deploys it to the capacity provider.

Step 6: Invoke your function

After publishing, you can invoke your function.

To invoke your function

```
aws lambda invoke \
  --function-name my-managed-instance-function:1 \
  --payload '{"name": "World"}' \
  response.json

# View the response
cat response.json
```

The function runs on the EC2 instances managed by your capacity provider and returns a response.

Clean up

To avoid incurring charges, delete the resources you created:

1. Delete the function:

   ```
   aws lambda delete-function --function-name my-managed-instance-function
   ```

2. Delete the capacity provider:

   ```
   aws lambda delete-capacity-provider --capacity-provider-name my-capacity-provider
   ```

3. Delete the VPC resources:

```
aws ec2 delete-security-group --group-id $SECURITY_GROUP_ID
aws ec2 delete-subnet --subnet-id $SUBNET_ID
aws ec2 delete-vpc --vpc-id $VPC_ID
```

4. Delete the IAM roles:

```
aws iam detach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
aws iam detach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator

aws iam delete-role --role-name MyLambdaExecutionRole
aws iam delete-role --role-name MyCapacityProviderOperatorRole
```

Core concepts

Lambda Managed Instances introduces several core concepts that differ from traditional Lambda functions. Understanding these concepts is essential for effectively deploying and managing your functions on EC2 infrastructure.

Capacity providers form the foundation of Lambda Managed Instances. A capacity provider defines the compute infrastructure where your functions execute, including VPC configuration, instance requirements, and scaling policies. Capacity providers also serve as the security boundary for your functions, meaning all functions assigned to the same capacity provider must be mutually trusted.

Scaling behavior differs significantly from traditional Lambda functions. Instead of scaling on-demand when invocations arrive, Managed Instances scale asynchronously based on CPU resource utilization. This approach eliminates cold starts but requires planning for traffic growth. If your traffic more than doubles within 5 minutes, you may experience throttles as Lambda scales up capacity to meet demand.

Security and permissions require careful consideration. You need operator role permissions to allow Lambda to manage EC2 resources in your capacity providers. Additionally, users need the

`lambda:PassCapacityProvider` permission to assign functions to capacity providers, acting as a security gate to control which functions can run on specific infrastructure.

Multi-concurrent execution is a key characteristic of Managed Instances. Each execution environment can handle multiple invocations simultaneously, maximizing resource utilization for IO-heavy applications. This differs from traditional Lambda where each environment processes one request at a time. This execution model requires attention to thread safety, state management, and context isolation depending on your runtime.

The following sections provide detailed information about each core concept.

Capacity providers

A capacity provider is the foundation for running Lambda Managed Instances. It acts as the security boundary for your functions and defines the compute resources that Lambda will provision and manage on your behalf.

When you create a capacity provider, you specify:

- **VPC configuration** - The subnets and security groups where instances will run
- **Permissions** - IAM roles for Lambda to manage EC2 resources
- **Instance requirements** (optional) - Architecture and instance type preferences
- **Scaling configuration** (optional) - How Lambda scales your instances

Understanding capacity providers as security boundary

Capacity providers serve as the security boundary for Lambda functions within your VPC, replacing Firecracker-based isolation. Functions execute in containers within instances, but containers do not provide strong security isolation between functions, unlike Firecracker microVMs.

Key security concepts:

- **Capacity Provider:** The security boundary that defines trust levels for Lambda functions
- **Container Isolation:** Containers are NOT a security provider - do not rely on them for security between untrusted workloads
- **Trust Separation:** Separate workloads that are not mutually trusted by using different capacity providers

Creating a capacity provider

You can create a capacity provider using the AWS CLI, AWS Management Console, or AWS SDKs.

Using AWS CLI:

```
aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config
 SubnetIds=subnet-12345,subnet-67890,subnet-11111,SecurityGroupIds=sg-12345 \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::123456789012:role/
MyOperatorRole \
  --instance-requirements Architectures=x86_64 \
  --capacity-provider-scaling-config ScalingMode=Auto
```

Required parameters

CapacityProviderName

- A unique name for your capacity provider
- Must be unique within your AWS account

VpcConfig

- **SubnetIds** (required): At least one subnet, maximum of 16. Use subnets across multiple Availability Zones for resiliency
- **SecurityGroupIds** (optional): Security groups for your instances. Defaults to the VPC default security group if not specified

PermissionsConfig

- **CapacityProviderOperatorRoleArn** (required): IAM role that allows Lambda to manage EC2 resources in your capacity provider

Optional parameters

InstanceRequirements

Specify the architecture and instance types for your capacity provider:

- **Architectures**: Choose `x86_64` or `arm64`. Default is `x86_64`
- **AllowedInstanceTypes**: Specify allowed instance types. Example: `m5.8xlarge`
- **ExcludedInstanceTypes**: Specify excluded instance types using wildcards. You can specify only one of AllowedInstanceTypes or ExcludedInstanceTypes

By default, Lambda chooses optimal instance types for your workload. We recommend letting Lambda Managed Instances choose instance types for you, as restricting the number of possible instance types may result in lower availability.

CapacityProviderScalingConfig

Configure how Lambda scales your instances:

- **ScalingMode**: Set to `Auto` for automatic scaling or `Manual` for manual control. Default is `Auto`
- **MaxVCpuCount**: Maximum number of vCPUs for the capacity provider. Default is 400.
- **ScalingPolicies**: Define target tracking scaling policies for CPU and memory utilization

KmsKeyArn

Specify a AWS KMS key for EBS encryption. Defaults to AWS managed key if not specified.

Tags

Add tags to organize and manage your capacity providers.

Managing capacity providers

Updating a capacity provider

You can update certain properties of a capacity provider using the `UpdateCapacityProvider` API.

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --capacity-provider-scaling-config ScalingMode=Auto
```

Deleting a capacity provider

You can delete a capacity provider when it's no longer needed using the `DeleteCapacityProvider` API.

```
aws lambda delete-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

Note: You cannot delete a capacity provider that has function versions attached to it.

Viewing capacity provider details

Retrieve information about a capacity provider using the `GetCapacityProvider` API.

```
aws lambda get-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

Capacity provider states

A capacity provider can be in one of the following states:

- **Pending**: The capacity provider is being created
- **Active**: The capacity provider is ready to use
- **Failed**: The capacity provider creation failed
- **Deleting**: The capacity provider is being deleted

Quotas

- **Maximum capacity providers per account**: 1,000
- **Maximum function versions per capacity provider**: 100 (cannot be increased)

Best practices

1. **Separate by trust level**: Create different capacity providers for workloads with different security requirements
2. **Use descriptive names**: Name capacity providers to clearly indicate their intended use and trust level (e.g., `production-trusted`, `dev-sandbox`)
3. **Use multiple Availability Zones**: Specify subnets across multiple AZs for high availability
4. **Let Lambda choose instance types**: Unless you have specific hardware requirements, allow Lambda to select optimal instance types for availability

5. **Monitor usage**: Use AWS CloudTrail to monitor capacity provider assignments and access patterns

Next steps

- Learn about scaling Lambda Managed Instances
- Understand security and permissions for Lambda Managed Instances
- Configure VPC connectivity for your capacity providers
- Review runtime-specific guides for Java, Node.js, and Python

Scaling Lambda Managed Instances

Lambda Managed Instances does not scale when invocations arrive and does not support cold starts. Instead, it scales asynchronously using resource consumption signals. Managed Instances currently scales based on CPU resource utilization and multi-concurrency saturation.

Key differences:

- **Lambda (default):** Scales when there is no free execution environment to handle an incoming invocation (cold start)
- **Lambda Managed Instances:** Scales asynchronously based on CPU resource utilization and multi-concurrency saturation of execution environments

If your traffic more than doubles within 5 minutes, you may see throttles as Lambda scales up instances and execution environments to meet demand.

The scaling lifecycle

Lambda Managed Instances uses a distributed architecture to manage scaling:

Components:

- **Managed Instances** - Run in your account in the subnets you provide
- **Router and Scaler** - Shared Lambda components that route invocations and manage scaling
- **Lambda Agent** - Runs on each Managed Instance to manage execution environment lifecycle and monitor resource consumption

How it works:

1. When you publish a function version with a capacity provider, Lambda launches Managed Instances in your account. It launches three by default for AZ resiliency and starts three execution environments before marking your function version ACTIVE.
2. Each Managed Instance can run execution environments for multiple functions mapped to the same capacity provider.
3. As traffic flows into your application, execution environments consume resources. The Lambda Agent notifies the Scaler, which decides whether to scale new execution environments or Managed Instances.
4. If Router attempts to send an invocation to an execution environment with high resource consumption, the Lambda Agent on that instance notifies it to retry on another.
5. As traffic decreases, the Lambda Agent notifies Scaler, which makes a decision to scale down execution environments and scale in Managed Instances.

Adjusting scaling behavior

You can customize the scaling behavior of Managed Instances through four controls:

Function level controls

1. Function memory and vCPUs

Choose the memory size and vCPU allocation for your function. The smallest supported function size is 2GB and 1vCPU.

Considerations:

- Pick a memory and vCPU setting that will support multi-concurrent executions of your function
- You cannot configure a function with less than 1 vCPU because functions running on Managed Instances should support multi-concurrent workloads
- You cannot choose less than 2GB because this matches the 2 to 1 memory to vCPU ratio of c instances, which have the lowest ratio
- For Python applications, you may need to choose a higher ratio of memory to vCPUs, such as 4 to 1 or 8 to 1, because of the way Python handles multi-concurrency
- If you are running CPU-intensive operations or perform little IO, you should choose more than one vCPU

2. Maximum concurrency

Set the maximum concurrency per execution environment.

Default behavior: Lambda chooses sensible defaults that balance resource consumption and throughput that work for a wide variety of applications.

Adjustment guidelines:

- **Increase concurrency:** If your function invocations use very little CPU, you can increase maximum concurrency up to a maximum of 64 per vCPU
- **Decrease concurrency:** If your application consumes a large amount of memory and very little CPU, you can reduce your maximum concurrency

Important: Since Lambda Managed Instances are meant for multi-concurrent applications, execution environments with very low concurrency may experience throttles when scaling.

Capacity provider level controls

3. Target resource utilization

Choose your own target for CPU utilization consumption.

Default behavior: Lambda maintains enough headroom for your traffic to double within 5 minutes without throttles.

Optimization options:

- If your workload is very steady or if your application is not sensitive to throttles, you may set the target to a high level to achieve higher utilization and lower costs
- If you want to maintain headroom for bursts of traffic, you can set resource targets to a low level, which will require more capacity

4. Instance type selection

Set allowed or excluded instance types.

Default behavior: Lambda chooses the best instance types for your workload. We recommend letting Lambda Managed Instances choose instance types for you, as restricting the number of possible instance types may result in lower availability.

Custom configuration:

- **Specific hardware requirements:** Set allowed instance types to a list of compatible instances. For example, if you have an application that requires high network bandwidth, you can select several n instance types
- **Cost optimization:** For testing or development environments, you might choose smaller instance types, like m7a.large instance types

Next steps

- Learn about capacity providers for Lambda Managed Instances
- Review runtime-specific guides for handling multi-concurrency
- Configure VPC connectivity for your capacity providers
- Monitor scaling metrics to optimize scaling behavior

Security and permissions

Lambda Managed Instances use **capacity providers as trust boundaries**. Functions execute in containers within these instances, but containers do not provide security isolation between workloads. All functions assigned to the same capacity provider must be mutually trusted.

Key Security Concepts

- **Capacity Provider**: The security boundary that defines trust levels for Lambda functions
- **Container Isolation**: Containers are not a security boundary - do not rely on them for security between untrusted workloads
- **Trust Separation**: Separate workloads that are not mutually trusted by using different capacity providers

Required Permissions

PassCapacityProvider Action

Users need the `lambda:PassCapacityProvider` permission to assign functions to capacity providers. This permission acts as a security gate, ensuring only authorized users can place functions in specific capacity providers.

Account administrators control which functions can use specific capacity providers through the `lambda:PassCapacityProvider` IAM action. This action is required when:

- Creating functions that use Lambda Managed Instances
- Updating function configurations to use a capacity provider
- Deploying functions via infrastructure as code

Example IAM Policy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "lambda:PassCapacityProvider",
      "Resource": "arn:aws:lambda:*:*:capacity-provider:trusted-workloads-*"
    }
  ]
}
```

Service-Linked Role

AWS Lambda uses the `AWSServiceRoleForLambda` service-linked role to manage Lambda Managed Instances ec2 resources in your capacity providers.

Best Practices

1. **Separate by Trust Level**: Create different capacity providers for workloads with different security requirements
2. **Use Descriptive Names**: Name capacity providers to clearly indicate their intended use and trust level (e.g., `production-trusted`, `dev-sandbox`)
3. **Apply Least Privilege**: Grant `PassCapacityProvider` permissions only for necessary capacity providers
4. **Monitor Usage**: Use AWS CloudTrail to monitor capacity provider assignments and access patterns

Next steps

- Learn about capacity providers for Lambda Managed Instances
- Understand scaling for Lambda Managed Instances
- Configure VPC connectivity for your capacity providers
- Review runtime-specific guides for Java, Node.js, and Python

Lambda operator role for Lambda Managed Instances

When you use Lambda Managed Instances, Lambda needs permissions to manage compute capacity in your account. The operator role provides these permissions through IAM policies that allow Lambda to manage EC2 instances in the capacity provider.

Lambda assumes the operator role when performing these management operations, similar to how Lambda assumes an execution role when your function runs.

Creating an operator role

You can create an operator role in the IAM console or with the AWS CLI. The role must include:

- **Permissions policy** – Grants permissions to manage capacity providers and associated resources
- **Trust policy** – Allows the Lambda service (`lambda.amazonaws.com`) to assume the role

Permissions policy

The operator role needs permissions to manage capacity providers and the underlying compute resources. At minimum, the role requires the permissions in the AWSLambdaManagedEC2ResourceOperator managed policy, currently:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:CreateTags",
        "ec2:AttachNetworkInterface"
      ],
```

```
      "Resource": [
        "arn:aws:ec2:*:*:instance/*",
        "arn:aws:ec2:*:*:network-interface/*",
        "arn:aws:ec2:*:*:volume/*"
      ],
      "Condition": {
        "StringEquals": {
          "ec2:ManagedResourceOperator": "scaler.lambda.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeAvailabilityZones",
        "ec2:DescribeCapacityReservations",
        "ec2:DescribeInstances",
        "ec2:DescribeInstanceStatus",
        "ec2:DescribeInstanceTypeOfferings",
        "ec2:DescribeInstanceTypes",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:subnet/*",
        "arn:aws:ec2:*:*:security-group/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:image/*"
      ],
```

```
      "Condition": {
        "StringEquals": {
          "ec2:Owner": "amazon"
        }
      }
    }
  ]
}
```

Trust policy

The trust policy allows Lambda to assume the operator role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Service-Linked Role for Lambda Managed Instances

To responsibly manage the lifecycle of Lambda Managed Instances, Lambda requires persistent access to terminate managed instances in your account. Lambda uses an AWS Identity and Access Management (IAM) service-linked role (SLR) to perform these operations.

Automatic creation: The service-linked role is automatically created the first time you create a capacity provider. The user creating the first capacity provider must have the `iam:CreateServiceLinkedRole` permission for the `lambda.amazonaws.com` principal.

Permissions: The service-linked role grants Lambda the following permissions on managed instances:

- `ec2:TerminateInstances` – To terminate instances at the end of their lifecycle
- `ec2:DescribeInstances` – To enumerate managed instances

Deletion: You can only delete this service-linked role after you have deleted all Lambda Managed Instances capacity providers in your account.

For more information about service-linked roles, see the section called “Using service-linked roles”.

Understanding the Lambda Managed Instances execution environment

Lambda Managed Instances provide an alternative deployment model that runs your function code on customer-owned Amazon EC2 instances while Lambda manages the operational aspects. The execution environment for Managed Instances has several important differences from Lambda (default) functions, particularly in how it handles concurrent invocations and manages container lifecycles.

Note: For information about the Lambda (default) execution environment, see Understanding the Lambda execution environment lifecycle.

Execution environment lifecycle

The lifecycle of a Lambda Managed Instances function execution environment differs from Lambda (default) in several key ways:

Init phase

During the Init phase, Lambda performs the following steps:

- Initialize and register all extensions
- Bootstrap the runtime entrypoint. Runtime spawns the configured number of runtime workers (implementation depends on runtime)
- Run function initialization code (code outside the handler)
- Wait for at least one runtime worker to signal readiness by calling `/runtime/invocation/next`

The Init phase is considered complete when extensions have initialized and at least one runtime worker has called `/runtime/invocation/next`. The function is then ready to process invocations.

> **Note**
>
> For Lambda Managed Instances functions, initialization can take up to 15 minutes. The timeout is the maximum of 130 seconds or the configured function timeout (up to 900 seconds).

Invoke phase

The Invoke phase for Lambda Managed Instances functions has several unique characteristics:

Continuous operation. Unlike Lambda (default), the execution environment remains continuously active, processing invocations as they arrive without freezing between invocations.

Parallel processing. Multiple invocations can execute simultaneously within the same execution environment, each handled by a different runtime worker.

Independent timeouts. The function's configured timeout applies to each individual invocation. When an invocation times out, Lambda marks that specific invocation as failed but does not interrupt other running invocations or terminate the execution environment.

Backpressure handling. If all runtime workers are busy processing invocations, new invocation requests are rejected until a worker becomes available.

Error handling and recovery

Error handling in Lambda Managed Instances function execution environments differs from Lambda (default):

Invoke timeouts. When an individual invocation times out, Lambda returns a timeout error for that invocation. However, Lambda Managed Instances does not enforce the timeout—your code will keep running. As a function developer, you are responsible for detecting and handling the timeout. The context object exposes the remaining time for the invocation, with a zero or negative value indicating a timeout. Other concurrent invocations in the execution environment continue processing normally.

Runtime worker failures. If a runtime worker process crashes, the execution environment continues operating with the remaining healthy workers.

Extension crashes. If an extension process crashes during initialization or operation, the entire execution environment is marked as unhealthy and is terminated. Lambda creates a new execution environment to replace it.

No reset/repair. Unlike Lambda (default), Managed Instances do not attempt to reset and reinitialize the execution environment after errors. Instead, unhealthy containers are terminated and replaced with new ones.

$LATEST.PUBLISHED version in Lambda Managed Instances

Lambda Managed Instances functions support the same numbered versioning workflow as Lambda (default). If you prefer not to maintain numbered versions, Lambda Managed Instances introduces a new version type: `$LATEST.PUBLISHED`. This version allows you to create or republish a latest published version as needed with updated code or configuration, without managing numbered versions.

Key difference from $LATEST: When you invoke a Lambda Managed Instances function using an unqualified ARN, Lambda implicitly invokes the `$LATEST.PUBLISHED` version rather than the unpublished $LATEST version.

The following AWS CLI command creates or republishes the `$LATEST.PUBLISHED` version.

```
aws lambda publish-version --function-name my-function --publish-to LATEST_PUBLISHED
```

You should see the following output:

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:
$LATEST.PUBLISHED",
  "Version": "$LATEST.PUBLISHED",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs24.x",
  ...
}
```

> **Note**
>
> If you use AWS CloudFormation or the Lambda console to create a Lambda Managed Instances function, Lambda automatically creates the `$LATEST.PUBLISHED` version.

Lambda Managed Instances runtimes

Lambda processes requests differently when using Lambda Managed Instances. Instead of handling requests sequentially in each execution environment, Lambda Managed Instances process multiple requests concurrently within each execution environment. This change in execution model means that functions using Lambda Managed Instances need to consider thread safety, state management, and context isolation, concerns which do not arise in the Lambda (default) single-concurrency model. In addition, the multi-concurrency implementation varies between runtimes.

Supported languages

Lambda Managed Instances can be used with the following programming languages and runtimes:

- **Java:** Java 21 and later.
- **Python:** Python 3.13 and later.
- **Node.js:** Node.js 22 and later.
- **.NET:** .NET 8 and later.
- **Rust:** Supported using the OS-only runtime `provided.al2023` and later.

Language-specific considerations

Each programming language implements multi-concurrency differently. You need to understand how multi-concurrency is implemented in your chosen programming language to apply the appropriate concurrency best practices.

Java

Uses a single process with OS threads for concurrency. Multiple threads execute the handler method simultaneously, requiring thread-safe handling of state and shared resources.

Python

Uses multiple Python processes where each concurrent request runs in a separate process. This protects against most concurrency issues, though care is required for shared resources such as the `/tmp` directory.

Node.js

Uses worker threads with asynchronous execution. Concurrent requests are distributed across worker threads, and each worker thread can also handle concurrent requests asynchronously, requiring safe handling of state and shared resources.

.NET

Uses .NET Tasks with asynchronous processing of multiple concurrent requests. Requires safe handling of state and shared resources.

Rust

Uses a single process with async tasks powered by Tokio. The handler must be `Clone` + `Send`.

Next steps

For detailed information about each runtime, see the following topics:

- Java runtime for Lambda Managed Instances
- Node.js runtime for Lambda Managed Instances
- Python runtime for Lambda Managed Instances
- .NET runtime for Lambda Managed Instances
- Rust support for Lambda Managed Instances

Java runtime for Lambda Managed Instances

For Java runtimes, Lambda Managed Instances use OS threads for concurrency. Lambda loads your handler object once per execution environment during initialization and then creates multiple threads. These threads execute in parallel and require thread-safe handling of state and shared resources. Each thread shares the same handler object and any static fields.

Concurrency configuration

The maximum number of concurrent requests which Lambda sends to each execution environment is controlled by the `PerExecutionEnvironmentMaxConcurrency` setting in the function

configuration. This is an optional setting, and the default value varies depending on the runtime. For Java runtimes, the default is 32 concurrent requests per vCPU, or you can configure your own value. This value also determines the number of threads used by the Java runtime. Lambda automatically adjusts the number of concurrent requests up to the configured maximum based on the capacity of each execution environment to absorb those requests.

Building functions for multi-concurrency

You should apply the same thread safety practices when using Lambda Managed Instances as you would in any other multi-threaded environment. Since the handler object is shared across all runtime worker threads, any mutable state must be thread-safe. This includes collections, database connections, and any static objects that are modified during request processing.

AWS SDK clients are thread safe and do not require special handling.

Example: Database connection pools

The following code uses a static database connection object which is shared between threads. Depending on the connection library used, this may not be thread safe.

```
public class DBQueryHandler implements RequestHandler<Object, String> {
    // Single connection shared across all threads - NOT SAFE
    private static Connection connection;

    public DBQueryHandler() {
        this.connection = DriverManager.getConnection(jdbcUrl, username, password);
    }

    @Override
    public String handleRequest(Object input, Context context) {
        PreparedStatement stmt = connection.prepareStatement(query);
        ResultSet rs = stmt.executeQuery();
        // Multiple threads using same connection causes issues
        return result.toString();
    }
}
```

A thread-safe approach is to use a connection pool. In the following example, the function handler retrieves a connection from the pool. The connection is only used in the context of a single request.

```
public class DBQueryHandler implements RequestHandler<Object, String> {
```

```
    private static HikariDataSource dataSource;

    public DBQueryHandler() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/your_database");
        dataSource = new HikariDataSource(config); // Create pool once per Lambda
 container
    }

    @Override
    public String handleRequest(Object input, Context context) {
        String query = "SELECT column_name FROM your_table LIMIT 10";
        StringBuilder result = new StringBuilder("Data:\n");

        // try-with-resources automatically calls close() on the connection,
        // which returns it to the HikariCP pool (does NOT close the physical DB
 connection)
        try (Connection connection = dataSource.getConnection();
             PreparedStatement stmt = connection.prepareStatement(query);
             ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) {
                result.append(rs.getString("column_name")).append("\n");
            }

        } catch (Exception e) {
            context.getLogger().log("Error: " + e.getMessage());
            return "Error";
        }

        return result.toString();
    }
}
```

Example: Collections

Standard Java collections are not thread safe:

```
public class Handler implements RequestHandler<Object, String> {
    private static List<String> items = new ArrayList<>();
    private static Map<String, Object> cache = new HashMap<>();

    @Override
```

```
    public String handleRequest(Object input, Context context) {
        items.add("list item");  // Not thread-safe
        cache.put("key", input); // Not thread-safe
        return "Success";
    }
}
```

Instead, use thread-safe collections:

```
public class Handler implements RequestHandler<Object, String> {
    private static final List<String> items =
        Collections.synchronizedList(new ArrayList<>());
    private static final ConcurrentHashMap<String, Object> cache =
        new ConcurrentHashMap<>();

    @Override
    public String handleRequest(Object input, Context context) {
        items.add("list item");  // Thread-safe
        cache.put("key", input); // Thread-safe
        return "Success";
    }
}
```

Shared /tmp directory

The `/tmp` directory is shared across all concurrent requests in the execution environment. Concurrent writes to the same file can cause data corruption, for example if another process overwrites the file. To address this, either implement file locking for shared files or use unique file names per thread or per request to avoid conflicts. Remember to clean up unneeded files to avoid exhausting the available space.

Logging

Log interleaving (log entries from different requests being interleaved in logs) is normal in multi-concurrent systems.

Functions using Lambda Managed Instances always use the structured JSON log format introduced with advanced logging controls. This format includes the `requestId`, allowing log entries to be correlated to a single request. When you use the `LambdaLogger` object from `context.getLogger()` the `requestId` is automatically included in each log entry. For further information, see the section called “Using Lambda advanced logging controls with Java”.

Request context

The `context` object is bound to the request thread. Using `context.getAwsRequestId()` provides thread-safe access to the request ID for the current request.

Use `context.getXrayTraceId()` to access the X-Ray trace ID. This provides thread-safe access to the trace ID for the current request. Lambda does not support the `_X_AMZN_TRACE_ID` environment variable with Lambda Managed Instances. The X-Ray trace ID is propagated automatically when using the AWS SDK.

Use `com.amazonaws.services.lambda.runtime.Context.getRemainingTimeInMillis()` to detect timeouts. See the section called “Error handling and recovery” for more information.

If you use virtual threads in your program or create threads during initialization, you will need to pass any required request context to these threads.

Initialization and shutdown

Function initialization occurs once per execution environment. Objects created during initialization are shared across threads.

For Lambda functions with extensions, the execution environment emits a SIGTERM signal during shut down. This signal is used by extensions to trigger clean up tasks, such as flushing buffers. You can subscribe to SIGTERM events to trigger function clean-up tasks, such as closing database connections. To learn more about the execution environment lifecycle, see the section called “Execution environment”.

Dependency versions

Lambda Managed Instances requires the following minimum package versions:

- AWS SDK for Java 2.0: version 2.34.0 or later
- AWS X-Ray SDK for Java: version 2.20.0 or later
- AWS Distro for OpenTelemetry - Instrumentation for Java: version 2.20.0 or later
- Powertools for AWS Lambda (Java): version 2.8.0 or later

Powertools for AWS Lambda (Java)

Powertools for AWS Lambda (Java) is compatible with Lambda Managed Instances and provides utilities for logging, tracing, metrics, and more. For more information, see Powertools for AWS Lambda (Java).

Next steps

- Review Node.js runtime for Lambda Managed Instances
- Review Python runtime for Lambda Managed Instances
- Review .NET runtime for Lambda Managed Instances
- Learn about scaling Lambda Managed Instances

Node.js runtime for Lambda Managed Instances

For Node.js runtimes, Lambda Managed Instances uses worker threads with `async/await`-based execution to handle concurrent requests. Function initialization occurs once per worker thread. Concurrent invocations are handled across two dimensions: worker threads provide parallelism across vCPUs, and asynchronous execution provides concurrency within each thread. Each concurrent request handled by the same worker thread shares the same handler object and global state, requiring safe handling under multiple concurrent requests.

Maximum concurrency

The maximum number of concurrent requests which Lambda sends to each execution environment is controlled by the `PerExecutionEnvironmentMaxConcurrency` setting in the function configuration. This is an optional setting, and the default value varies depending on the runtime. For Node.js runtimes, the default is 64 concurrent requests per vCPU, or you can configure your own value. Lambda automatically adjusts the number of concurrent requests up to the configured maximum based on the capacity of each execution environment to absorb those requests.

For Node.js, the number of concurrent requests that each execution environment can process is determined by the number of worker threads and the capacity of each worker thread to process concurrent requests asynchronously. The default number of worker threads is determined by the number of vCPUs available, or you can configure the number of worker threads by setting the `AWS_LAMBDA_NODEJS_WORKER_COUNT` environment variable. We recommend using async function handlers since this allows processing multiple requests per worker thread. If your function handler is synchronous, each worker thread can only process a single request at a time.

Building functions for multi-concurrency

With an async function handler, each runtime worker processes multiple requests concurrently. Global objects will be shared across multiple concurrent requests. For mutable objects, avoid using global state or use `AsyncLocalStorage`.

AWS SDK clients are async safe and do not require special handling.

Example: Global state

The following code uses a global object which is mutated inside the function handler. This is not async-safe.

```
let state = {
    currentUser: null,
    requestData: null
};

export const handler = async (event, context) => {
    state.currentUser = event.userId;
    state.requestData = event.data;

    await processData(state.requestData);

    // state.currentUser might now belong to a different request
    return { user: state.currentUser };
};
```

Initialising the `state` object inside the function handler avoids shared global state.

```
export const handler = async (event, context) => {
    let state = {
        currentUser: event.userId,
        requestData: event.data
    };

    await processData(state.requestData);

    return { user: state.currentUser };
};
```

Example: Database connections

The following code uses a shared client object which is shared between multiple invocations. Depending on the connection library used, this may not be concurrency safe.

```
const { Client } = require('pg');

// Single connection created at init time
const client = new Client({
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD
});

// Connect once during cold start
client.connect();

exports.handler = async (event) => {
  // Multiple parallel invocations share this single connection = BAD
  // With multi-concurrent Lambda, queries will collide
  const result = await client.query('SELECT * FROM users WHERE id = $1',
 [event.userId]);

  return {
    statusCode: 200,
    body: JSON.stringify(result.rows[0])
  };
};
```

A concurrency-safe approach is to use a connection pool. The pool uses a separate connection for each concurrent database query.

```
const { Pool } = require('pg');

// Connection pool created at init time
const pool = new Pool({
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  max: 20,  // Max connections in pool
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000
});
```

```
exports.handler = async (event) => {
  // Pool gives each parallel invocation its own connection
  const result = await pool.query('SELECT * FROM users WHERE id = $1', [event.userId]);

  return {
    statusCode: 200,
    body: JSON.stringify(result.rows[0])
  };
};
```

Node.js 22 callback-based handlers

When using Node.js 22, you cannot use a callback-based function handler with Lambda Managed Instances. Callback-based handlers are only supported for Lambda (default) functions. For Node.js 24 and later runtimes, callback-based function handlers are deprecated for both Lambda (default) and Lambda Managed Instances.

Instead, use an `async` function handler when using Lambda Managed Instances. For more information, see Define Lambda function handler in Node.js.

Shared /tmp directory

The `/tmp` directory is shared across all concurrent requests in the execution environment. Concurrent writes to the same file can cause data corruption, for example if another process overwrites the file. To address this, either implement file locking for shared files or use unique file names per request to avoid conflicts. Remember to clean up unneeded files to avoid exhausting the available space.

Logging

Log interleaving (log entries from different requests being interleaved in logs) is normal in multi-concurrent systems. Functions using Lambda Managed Instances always use the structured JSON log format introduced with advanced logging controls. This format includes the `requestId`, allowing log entries to be correlated to a single request. When you use the `console` logger, the `requestId` is automatically included in each log entry. For further information, see the section called “Using Lambda advanced logging controls with Node.js”.

Popular third-party logging libraries, such as Winston, typically include support for using console for log output.

Request context

Using `context.awsRequestId` provides async-safe access to the request ID for the current request.

Use `context.xRayTraceId` to access the X-Ray trace ID. This provides concurrency-safe access to the trace ID for the current request. Lambda does not support the `_X_AMZN_TRACE_ID` environment variable with Lambda Managed Instances. The X-Ray trace ID is propagated automatically when using the AWS SDK.

Use `context.getRemainingTimeInMillis()` to detect timeouts. See the section called "Error handling and recovery" for more information.

Initialization and shutdown

Function initialization occurs once per worker thread. You may see repeat log entries if your function emits logs during initialization.

For Lambda functions with extensions, the execution environment emits a SIGTERM signal during shut down. This signal is used by extensions to trigger clean up tasks, such as flushing buffers. Lambda (default) functions with extensions can also subscribe to the SIGTERM signal using `process.on()`. This is not supported for functions using Lambda Managed Instances since `process.on()` cannot be used with worker threads. To learn more about the execution environment lifecycle, see the section called "Execution environment".

Dependency versions

Lambda Managed Instances requires the following minimum package versions:

- AWS SDK for JavaScript v3: version 3.933.0 or later
- AWS X-Ray SDK for Node.js: version 3.12.0 or later
- AWS Distro for OpenTelemetry - Instrumentation for JavaScript: version 0.8.0 or later
- Powertools for AWS Lambda (TypeScript): version 2.29.0 or later

Powertools for AWS Lambda (TypeScript)

Powertools for AWS Lambda (TypeScript) is compatible with Lambda Managed Instances and provides utilities for logging, tracing, metrics, and more. For more information, see Powertools for AWS Lambda (TypeScript).

Next steps

- Review Java runtime for Lambda Managed Instances
- Review Python runtime for Lambda Managed Instances
- Review .NET runtime for Lambda Managed Instances
- Learn about scaling Lambda Managed Instances

Python runtime for Lambda Managed Instances

The Lambda runtime uses multiple Python processes to handle concurrent requests. Each concurrent request runs in a separate process with its own memory space and initialization. Each process handles one request at a time, synchronously. Processes don't share memory directly, so global variables, module-level caches, and singleton objects are isolated between concurrent requests.

Concurrency configuration

The maximum number of concurrent requests which Lambda sends to each execution environment is controlled by the `PerExecutionEnvironmentMaxConcurrency` setting in the function configuration. This is an optional setting, and the default value varies depending on the runtime. For Python runtimes, the default is 16 concurrent requests per vCPU, or you can configure your own value. This value also determines the number of processes used by the Python runtime. Lambda automatically adjusts the number of concurrent requests up to the configured maximum based on the capacity of each execution environment to absorb those requests.

> **Important**
>
> Using process-based concurrency means each runtime worker process performs its own initialization. Total memory usage equals the per-process memory multiplied by the number of concurrent processes. If you are loading large libraries or data sets and have high concurrency, you will have a large memory footprint. According to your workload, you may need to tune your CPU-to-memory ratio or use a lower concurrency setting to avoid exceeding the available memory. You can use the `MemoryUtilization` metric in CloudWatch to track memory consumption.

Building functions for multi-concurrency

Due to the process-based multi-concurrency model, Lambda Managed Instances functions using Python runtimes do not access in-memory resources concurrently from multiple invokes. You do not need to apply coding practices for in-memory concurrency safety.

Shared /tmp directory

The `/tmp` directory is shared across all concurrent requests in the execution environment. Concurrent writes to the same file can cause data corruption, for example if another process overwrites the file. To address this, either implement file locking for shared files or use unique file names per process or per request to avoid conflicts. Remember to clean up unneeded files to avoid exhausting the available space.

Logging

Log interleaving (log entries from different requests being interleaved in logs) is normal in multi-concurrent systems.

Functions using Lambda Managed Instances always use the structured JSON log format introduced with advanced logging controls. This format includes the `requestId`, allowing log entries to be correlated to a single request. When you use the `logging` module from the Python standard library in Lambda, the `requestId` is automatically included in each log entry. For further information, see Using Lambda advanced logging controls with Python.

Request context

Use `context.aws_request_id` to access to the request ID for the current request.

With Python runtimes, you can use the `_X_AMZN_TRACE_ID` environment variable to access the X-Ray trace ID with Lambda Managed Instances. The X-Ray trace ID is propagated automatically when using the AWS SDK.

Use `context.get_remaining_time_in_millis()` to detect timeouts. See the section called "Error handling and recovery" for more information.

Initialization and shutdown

Function initialization occurs once per process. You may see repeat log entries if your function emits logs during initialization.

For Lambda functions with extensions, the execution environment emits a SIGTERM signal during shut down. This signal is used by extensions to trigger clean up tasks, such as flushing buffers. You can subscribe to SIGTERM events to trigger function clean-up tasks, such as closing database connections. To learn more about the execution environment lifecycle, see the section called "Execution environment".

Dependency versions

Lambda Managed Instances requires the following minimum package versions:

- Powertools for AWS Lambda (Python): version 3.23.0 or later

Powertools for AWS Lambda (Python)

Powertools for AWS Lambda (Python) is compatible with Lambda Managed Instances and provides utilities for logging, tracing, metrics, and more. For more information, see Powertools for AWS Lambda (Python).

Next steps

- Review Java runtime for Lambda Managed Instances
- Review Node.js runtime for Lambda Managed Instances
- Review .NET runtime for Lambda Managed Instances
- Learn about scaling Lambda Managed Instances

.NET runtime for Lambda Managed Instances

For .NET runtimes, Lambda Managed Instances use a single .NET process per execution environment. Multiple concurrent requests are processed using .NET Tasks.

Concurrency configuration

The maximum number of concurrent requests which Lambda sends to each execution environment is controlled by the `PerExecutionEnvironmentMaxConcurrency` setting in the function configuration. This is an optional setting, and the default value varies depending on the runtime. For .NET runtimes, the default is 32 concurrent requests per vCPU, or you can configure your own value. Lambda automatically adjusts the number of concurrent requests up to the configured maximum based on the capacity of each execution environment to absorb those requests.

Building functions for multi-concurrency

You should apply the same concurrency safety practices when using Lambda Managed Instances as you would in any other multi-concurrent environment. Since the handler object is shared across all Tasks any mutable state must be thread-safe. This includes collections, database connections and any static objects that are modified during request processing.

AWS SDK clients are thread safe and do not require special handling.

Example: Database connection pools

The following code uses a static database connection object which is shared between concurrent requests. The `SqlConnection` object is not thread safe.

```
public class DBQueryHandler
{
    // Single connection shared across threads - NOT SAFE
    private SqlConnection connection;

    public DBQueryHandler()
    {
        connection = new SqlConnection("your-connection-string-here");
        connection.Open();
    }

    public string Handle(object input, ILambdaContext context)
    {
        using var cmd = connection.CreateCommand();
        cmd.CommandText = "SELECT ..."; // your query

        using var reader = cmd.ExecuteReader();

        ...
    }
}
```

To address this, use a separate connection for each request, drawn from a connection pool. ADO.NET providers like `Microsoft.Data.SqlClient` automatically support connection pooling when the connection object is opened.

```
public class DBQueryHandler
```

```
{
    public DBQueryHandler()
    {
    }

    public string Handle(object input, ILambdaContext context)
    {
        using var connection = new SqlConnection("your-connection-string-here");
        connection.Open();
        using var cmd = connection.CreateCommand();
        cmd.CommandText = "SELECT ..."; // your query

        using var reader = cmd.ExecuteReader();

        ...
    }
}
```

Example: Collections

Standard .NET collections are not thread safe:

```
public class Handler
{
    private static List<string> items = new List<string>();
    private static Dictionary<string, object> cache = new Dictionary<string, object>();

    public string FunctionHandler(object input, ILambdaContext context)
    {
        items.Add(context.AwsRequestId);
        cache["key"] = input;

        return "Success";
    }
}
```

Use collections from the `System.Collections.Concurrent` namespace for concurrency safety:

```
public class Handler
{
    private static ConcurrentBag<string> items = new ConcurrentBag<string>();
    private static ConcurrentDictionary<string, object> cache = new
 ConcurrentDictionary<string, object>();
```

```
    public string FunctionHandler(object input, ILambdaContext context)
    {
        items.Add(context.AwsRequestId);
        cache["key"] = input;

        return "Success";
    }
}
```

Shared /tmp directory

The `/tmp` directory is shared across all concurrent requests in the execution environment. Concurrent writes to the same file can cause data corruption, for example if another request overwrites the file. To address this, either implement file locking for shared files or use unique file names per request to avoid conflicts. Remember to clean up unneeded files to avoid exhausting the available space.

Logging

Log interleaving (log entries from different requests being interleaved in logs) is normal in multi-concurrent systems. Functions using Lambda Managed Instances always use the structured JSON log format introduced with advanced logging controls. This format includes the `requestId`, allowing log entries to be correlated to a single request. When you use the `context.Logger` object to generate logs, the `requestId` is automatically included in each log entry. For further information, see the section called “Using Lambda advanced logging controls with .NET”.

Request context

Use the `context.AwsRequestId` property to access to the request ID for the current request.

Use the `context.TraceId` property to access the X-Ray trace ID. This provides concurrency-safe access to the trace ID for the current request. Lambda does not support the `_X_AMZN_TRACE_ID` environment variable with Lambda Managed Instances. The X-Ray trace ID is propagated automatically when using the AWS SDK.

Use `ILambdaContext.RemainingTime` to detect timeouts. See the section called “Error handling and recovery” for more information.

Initialization and shutdown

Function initialization occurs once per execution environment. Objects created during initialization are shared across requests.

For Lambda functions with extensions, the execution environment emits a SIGTERM signal during shut down. This signal is used by extensions to trigger clean up tasks, such as flushing buffers. You can subscribe to SIGTERM events to trigger function clean-up tasks, such as closing database connections. To learn more about the execution environment lifecycle, see the section called "Execution environment".

Dependency versions

Lambda Managed Instances requires the following minimum package versions:

- Amazon.Lambda.Core: version 2.7.1 or later
- Amazon.Lambda.RuntimeSupport: version 1.14.1 or later
- OpenTelemetry.Instrumentation.AWSLambda: version 1.14.0 or later
- AWSXRayRecorder.Core: version 2.16.0 or later
- AWSSDK.Core: version 4.0.0.32 or later

Powertools for AWS Lambda (.NET)

Powertools for AWS Lambda (.NET) and AWS Distro for OpenTelemetry - Instrumentation for DotNet currently do not support Lambda Managed Instances.

Next steps

- Review Java runtime for Lambda Managed Instances
- Review Node.js runtime for Lambda Managed Instances
- Review Python runtime for Lambda Managed Instances
- Learn about scaling Lambda Managed Instances

Rust support for Lambda Managed Instances

Concurrency configuration

The maximum number of concurrent requests which Lambda sends to each execution environment is controlled by the `PerExecutionEnvironmentMaxConcurrency` setting in the function configuration. This is an optional setting, and the default value for Rust is 8 concurrent requests per vCPU, or you can configure your own value. This value determines the number of Tokio tasks spawned by the runtime and is static for the lifetime of the execution environment. Each worker handles exactly one in-flight request at a time, with no multiplexing per worker. Lambda automatically adjusts the number of concurrent requests up to the configured maximum based on the capacity of each execution environment to absorb those requests.

Building functions for multi-concurrency

You should apply the same thread safety practices when using Lambda Managed Instances as you would in any other multi-threaded environment. Since the handler object is shared across all worker threads, any mutable state must be thread-safe. This includes collections, database connections, and any static objects that are modified during request processing.

To enable concurrent request handling, add the `concurrency-tokio` feature flag to your `Cargo.toml` file.

```
[dependencies]
lambda_runtime = { version = "1", features = ["concurrency-tokio"] }
```

The `lambda_runtime::run_concurrent(…)` entry point must be called from within a Tokio runtime, typically provided by the `#[tokio::main]` attribute on your main function. Your handler closure must implement `Clone` + `Send`. This allows the framework to share your handler across multiple async tasks safely. If those bounds are not met, your code will not compile.

When you need shared state across invocations (a database pool, a config struct), wrap it in `Arc` and clone the `Arc` into each invocation.

All AWS SDK for Rust clients are concurrency-safe and require no special handling.

Example: AWS SDK client

The following example uses an S3 client to upload an object on each invocation. The client is cloned directly into the closure without `Arc`:

```
let config = aws_config::load_defaults(BehaviorVersion::latest()).await;
let s3_client = aws_sdk_s3::Client::new(&config);

run_concurrent(service_fn(move |event: LambdaEvent<Request>| {
    let s3_client = s3_client.clone(); // cheap clone, no Arc needed
    async move {
        s3_client.put_object()
            .bucket(&event.payload.bucket)
            .key(&event.payload.key)
            .body(event.payload.body.into_bytes().into())
            .send()
            .await?;
        Ok(Response { message: "uploaded".into() })
    }
}))
.await
```

Example: Database connection pools

When your handler needs access to shared state such as a client and configuration, wrap it in `Arc` and clone the `Arc` into each invocation:

```
#[derive(Debug)]
struct AppState {
    dynamodb_client: DynamoDbClient,
    table_name: String,
    cache_ttl: Duration,
}

let config = aws_config::load_defaults(BehaviorVersion::latest()).await;
let state = Arc::new(AppState {
    dynamodb_client: DynamoDbClient::new(&config),
    table_name: std::env::var("TABLE_NAME").expect("TABLE_NAME must be set"),
    cache_ttl: Duration::from_secs(300),
});

run_concurrent(service_fn(move |event: LambdaEvent<Request>| {
    let state = state.clone();
    async move { handle(event, state).await }
}))
.await
```

Shared /tmp directory

The `/tmp` directory is shared across all concurrent invocations in the same execution environment. Use unique file names per invocation (e.g. include the request ID) or implement explicit file locking to avoid data corruption.

Logging

Log interleaving (log entries from different requests being interleaved in logs) is normal in multi-concurrent systems. Functions using Lambda Managed Instances support structured JSON log format via Lambda's advanced logging controls. This format includes the `requestId`, allowing log entries to be correlated to a single request. For further information, see the section called "Implementing advanced logging with the Tracing crate".

Request Context

The `Context` object is passed directly to each handler invocation. Use `event.context.request_id` to access the request ID for the current request.

Use `event.context.xray_trace_id` to access the X-Ray trace ID. Lambda does not support the `_X_AMZN_TRACE_ID` environment variable with Lambda Managed Instances. The X-Ray trace ID is propagated automatically when using the AWS SDK for Rust.

Use `event.context.deadline` to detect timeouts — it contains the invocation deadline in milliseconds.

Initialization and shutdown

Function initialization occurs once per execution environment. Objects created during initialization are shared across requests.

For Lambda functions with extensions, the execution environment emits a SIGTERM signal during shut down. This signal is used by extensions to trigger clean up tasks, such as flushing buffers. `lambda_runtime` offers a helper to simplify configuring graceful shutdown signal handling, `spawn_graceful_shutdown_handler()`. To learn more about the execution environment lifecycle, see the section called "Execution environment".

Dependency versions

Lambda Managed Instances requires the following minimum package version:

- `lambda_runtime`: version 1.1.1 or later, with the `concurrency-tokio` feature enabled
- The minimum supported Rust version (MSRV) is 1.84.0.

Networking for Lambda Managed Instances

When running Lambda Managed Instances functions, you need to configure network connectivity to enable your functions to access resources outside the VPC. This includes AWS services such as Amazon S3 and DynamoDB. The connectivity is also needed for transmitting telemetry data to CloudWatch Logs and X-Ray.

Connectivity options

There are three primary approaches for configuring VPC connectivity, each with different trade-offs for cost, security, and complexity.

Public subnet with an internet gateway

This option uses a public subnet with direct internet access through an internet gateway. You can choose between IPv4 and IPv6 configurations.

IPv4 with internet gateway

To configure IPv4 connectivity with an internet gateway

1. Create or use an existing public subnet with an IPv4 CIDR block.
2. Attach an internet gateway to your VPC.
3. Update the route table to route `0.0.0.0/0` traffic to the internet gateway.
4. Ensure resources have public IPv4 addresses or Elastic IP addresses assigned.
5. Configure security groups to allow outbound traffic on the required ports.

This configuration provides bidirectional connectivity, allowing both outbound connections from your functions and inbound connections from the internet.

IPv6 with internet gateway

To configure IPv6 connectivity with an internet gateway

1. Enable IPv6 on your VPC.

2. Create or use an existing public subnet with an IPv6 CIDR block assigned.
3. Attach an internet gateway to your VPC (the same internet gateway can handle both IPv4 and IPv6).
4. Update the route table to route `::/0` traffic to the internet gateway.
5. Verify that the AWS services you need to access support IPv6 in your Region.
6. Configure security groups to allow outbound traffic on the required ports.

This configuration provides bidirectional connectivity using IPv6 addressing.

IPv6 with egress-only internet gateway

To configure IPv6 connectivity with an egress-only internet gateway

1. Enable IPv6 on your VPC.
2. Create or use an existing public subnet with an IPv6 CIDR block assigned.
3. Attach an egress-only internet gateway to your VPC.
4. Update the route table to route `::/0` traffic to the egress-only internet gateway.
5. Verify that the AWS services you need to access support IPv6 in your Region.
6. Configure security groups to allow outbound traffic on the required ports.

This configuration provides outbound-only connectivity, preventing inbound connections from the internet while allowing your functions to initiate outbound connections.

VPC endpoints

VPC endpoints enable you to privately connect your VPC to supported AWS services without requiring an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Traffic between your VPC and the AWS service does not leave the Amazon network.

To configure VPC endpoints

1. Open the Amazon VPC console at console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Endpoints**.
3. Choose **Create endpoint**.
4. For **Service category**, choose **AWS services**.

5. For **Service name**, select the service endpoint you need (for example, `com.amazonaws.region.s3` for Amazon S3).
6. For **VPC**, select your VPC.
7. For **Subnets**, select the subnets where you want to create endpoint network interfaces. For high availability, select subnets in multiple Availability Zones.
8. For **Security groups**, select the security groups to associate with the endpoint network interfaces. The security groups must allow inbound traffic from your function's security group on the required ports.
9. Choose **Create endpoint**.

Repeat these steps for each AWS service that your functions need to access.

Private subnet with NAT gateway

This option uses a NAT gateway to provide internet access for resources in private subnets while keeping the resources private.

To configure a private subnet with NAT gateway

1. Create a public subnet (if one doesn't already exist) with a CIDR block.
2. Attach an internet gateway to your VPC.
3. Create a NAT gateway in the public subnet and assign an Elastic IP address.
4. Update the public subnet route table to add a route: `0.0.0.0/0` → internet gateway.
5. Create or use an existing private subnet with a CIDR block.
6. Update the private subnet route table to add a route: `0.0.0.0/0` → NAT gateway.
7. Configure security groups to allow outbound traffic on the required ports.

For high availability, deploy one NAT gateway in each Availability Zone and configure route tables per Availability Zone to use the local NAT gateway. This prevents cross-AZ data transfer charges and improves resilience.

Choosing a connectivity option

Consider the following factors when choosing a connectivity option:

Public subnet with internet gateway

- Simplest configuration with lowest cost
- Suitable for development and testing environments
- Resources can receive inbound connections from the internet (security consideration)
- Supports both IPv4 and IPv6

VPC endpoints

- Highest security, traffic stays within the AWS network
- Lower latency compared to internet routing
- Recommended for production environments with strict security requirements
- Higher cost per endpoint, per Availability Zone, and per GB processed
- Requires an endpoint in each Availability Zone for high availability

Private subnet with NAT gateway

- Resources remain private with no inbound internet access
- Standard enterprise architecture pattern
- Supports all IPv4 internet traffic
- Moderate cost with NAT gateway hourly and data processing charges
- Supports IPv4 only

Next steps

- Learn about capacity providers for Lambda Managed Instances
- Understand scaling for Lambda Managed Instances
- Review runtime-specific guides for Java, Node.js, and Python
- Understand security and permissions for Lambda Managed Instances

Monitoring Lambda Managed Instances

You can monitor Lambda Managed Instances using CloudWatch metrics. Lambda automatically publishes metrics to CloudWatch to help you monitor resource utilization, track costs, and optimize performance.

Available metrics

Lambda Managed Instances provides metrics at two levels: capacity provider level and execution environment level.

Capacity provider level metrics

Capacity provider level metrics provide visibility into overall resource utilization across your instances. These metrics use the following dimensions:

- **CapacityProviderName** - The name of your capacity provider
- **InstanceType** - The EC2 instance type

Resource utilization metrics:

- **CPUUtilization** - The percentage of CPU utilization across instances in the capacity provider
- **MemoryUtilization** - The percentage of memory utilization across instances in the capacity provider

Capacity metrics:

- **vCPUAvailable** - The amount of vCPU available on instances for allocation (in count)
- **MemoryAvailable** - The amount of memory available on instances for allocation (in bytes)
- **vCPUAllocated** - The amount of vCPU allocated on instances for execution environments (in count)
- **MemoryAllocated** - The amount of memory allocated on instances for execution environments (in bytes)

Execution environment level metrics

Execution environment level metrics provide visibility into resource utilization and concurrency for individual functions. These metrics use the following dimensions:

- **CapacityProviderName** - The name of your capacity provider
- **FunctionName** - The name of your Lambda function
- **Resource** - By resource, view metrics for a specific version of a function.

> **Note**
>
> For Lambda Managed Instances (LMI), the `Resource` dimension supports function versions only. The format is `<FunctionName>:<FunctionVersion>`.

Available execution environment metrics:

- **ExecutionEnvironmentConcurrency** - The maximum concurrency over a 5-minute sample period
- **ExecutionEnvironmentConcurrencyLimit** - The maximum concurrency limit per execution environment
- **ExecutionEnvironmentCPUUtilization** - The percentage of CPU utilization for the function's execution environments
- **ExecutionEnvironmentMemoryUtilization** - The percentage of memory utilization for the function's execution environments

Metric frequency and retention

Lambda Managed Instances metrics are published at 5-minute intervals and retained for 15 months.

Viewing metrics in CloudWatch

To view Lambda Managed Instances metrics in the CloudWatch console

1. Open the CloudWatch console at console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Metrics**.
3. In the **All metrics** tab, choose **AWS/Lambda**.
4. Choose the metric dimension you want to view:
   - For capacity provider level metrics, filter by **CapacityProviderName** and **InstanceType**
   - For execution environment level metrics, filter by **CapacityProviderName**, **FunctionName**, and **Resource**
5. Select the metrics you want to monitor.

Using metrics to optimize performance

Monitor CPU and memory utilization to understand if your functions are properly sized. High utilization may indicate the need for larger instance types or increased function memory allocation. Track concurrency metrics to understand scaling behavior and identify potential throttling.

Monitor capacity metrics to verify sufficient resources are available for your workloads. The **vCPUAvailable** and **MemoryAvailable** metrics help you understand remaining capacity on your instances.

Next steps

- Learn about scaling Lambda Managed Instances
- Review runtime-specific guides for Java, Node.js, and Python
- Configure VPC connectivity for your capacity providers
- Understand security and permissions for Lambda Managed Instances

Lambda Managed Instances quotas

This page describes the service quotas for AWS Lambda Managed Instances. These quotas are separate from AWS Lambda (default) quotas. Some quotas can be increased upon request.

Lambda API request quotas

These quotas control the rate at which you can make API calls to manage Lambda Managed Instances capacity providers. The read and write API rate limits apply to all capacity provider operations combined, including creating, updating, describing, and deleting capacity providers.

Resource	Quota
The maximum combined rate (requests per second) for all capacity provider read APIs	15 requests per second. Cannot be increased.
The maximum combined rate (requests per second) for all capacity provider write APIs	1 request per second. Cannot be increased.

Lambda Managed Instances resource quotas

These quotas define the limits for core Lambda Managed Instances resources within your AWS account. They govern the number of capacity providers you can create and the number of function versions that can be associated with each capacity provider.

Resource	Quota
Capacity providers	1,000. The maximum number of capacity providers created in an account.
Function versions per capacity provider	100. The maximum number of function versions per capacity provider. Cannot be increased.

Event source mapping quotas

These quotas control the throughput and configuration limits for processing events from various AWS services on Lambda Managed Instances. The throughput limits ensure predictable performance while the mapping count limits help maintain service stability. Event source mappings on Lambda Managed Instances support Amazon SQS, DynamoDB Streams, Amazon Kinesis Data Streams, Amazon MSK, and self-managed Apache Kafka as event sources.

Resource	Quota
Standard SQS event source mapping throughput on Lambda Managed Instances	5 MB per second. Cannot be increased.
Standard Kafka event source mapping throughput on Lambda Managed Instances	1 MB per second. Cannot be increased.
Standard Kafka event source mappings on Lambda Managed Instances	100 event source mappings. Cannot be increased.

Resource	Quota
Kinesis event source mapping throughput on Lambda Managed Instances	25 MB per second. Can be increased.
DynamoDB event source mapping throughput on Lambda Managed Instances	10 MB per second. Can be increased.
Invoke request throughput for asynchronous invocations on Lambda Managed Instances	5 MB per second. Can be increased.

Requesting a quota increase

For quotas that can be increased, you can request an increase through the Service Quotas console.

To request a quota increase

1. Open the Service Quotas console at console.aws.amazon.com/servicequotas/.
2. In the navigation pane, choose **AWS services**.
3. Choose **AWS Lambda**.
4. Select the quota you want to increase.
5. Choose **Request quota increase**.
6. Enter the new quota value and provide a justification for the increase.
7. Choose **Request**.

Next steps

- Learn about capacity providers for Lambda Managed Instances
- Understand scaling for Lambda Managed Instances
- Review runtime-specific guides for Java, Node.js, and Python
- Configure VPC connectivity for your capacity providers

Best practices for Lambda Managed Instances

Capacity provider configuration

Separate capacity providers by trust level. Create different capacity providers for workloads with different security requirements. All functions assigned to the same capacity provider must be mutually trusted, as capacity providers serve as the security boundary.

Use descriptive names. Name capacity providers to clearly indicate their intended use and trust level (for example, `production-trusted`, `dev-sandbox`). This helps teams understand the purpose and security posture of each capacity provider.

Use multiple Availability Zones. Specify subnets across multiple Availability Zones when creating capacity providers. Lambda launches three instances by default for AZ resiliency, ensuring high availability for your functions.

Instance type selection

Let Lambda choose instance types. By default, Lambda chooses the best instance types for your workload. We recommend letting Lambda Managed Instances choose instance types for you, as restricting the number of possible instance types may result in lower availability.

Specify instance types for specific requirements. If you have specific hardware requirements, set allowed instance types to a list of compatible instances. For example:

- For applications requiring high network bandwidth, select several n instance types
- For testing or development environments with cost constraints, choose smaller instance types like m7a.large

Function configuration

Choose appropriate memory and vCPU settings. Select memory and vCPU configurations that support multi-concurrent executions of your function. The minimum supported function size is 2GB and 1 vCPU.

- For Python applications, choose a higher ratio of memory to vCPUs (such as 4 to 1 or 8 to 1) because of the way Python handles multi-concurrency
- For CPU-intensive operations or functions that perform little IO, choose more than one vCPU

- For IO-heavy applications like web services or batch jobs, multi-concurrency provides the most benefit

Configure maximum concurrency appropriately. Lambda chooses sensible defaults for maximum concurrency that balance resource consumption and throughput. Adjust this setting based on your function's resource usage:

- Increase maximum concurrency (up to 64 per vCPU) if your function invocations use very little CPU
- Decrease maximum concurrency if your application consumes a large amount of memory and very little CPU

Note that execution environments with very low concurrency may experience throttles and difficulty scaling.

Scaling configuration

Set appropriate target resource utilization. By default, Lambda maintains enough headroom for your traffic to double within 5 minutes without throttles. Adjust this based on your workload characteristics:

- For very steady workloads or applications not sensitive to throttles, set the target to a high level to achieve higher utilization and lower costs
- For workloads with potential traffic bursts, set resource targets to a low level to maintain additional headroom

Plan for traffic growth. If your traffic more than doubles within 5 minutes, you may see throttles as Lambda scales up instances and execution environments. Design your application to handle potential throttling during rapid scale-up periods.

Security

Apply least privilege for PassCapacityProvider permissions. Grant `lambda:PassCapacityProvider` permissions only for necessary capacity providers. Use resource-level permissions to restrict which capacity providers users can assign to functions.

Monitor capacity provider usage. Use AWS CloudTrail to monitor capacity provider assignments and access patterns. This helps identify unauthorized access attempts and ensures compliance with security policies.

Separate untrusted workloads. Do not rely on containers for security isolation between untrusted workloads. Use different capacity providers to separate workloads that are not mutually trusted.

Cost optimization

Leverage EC2 pricing options. Take advantage of EC2 Savings Plans and Reserved Instances to reduce costs. These pricing options apply to the underlying EC2 compute (the 15% management fee is not discounted).

Optimize for steady-state workloads. Lambda Managed Instances are best suited for steady-state functions with predictable high-volume traffic. For bursty traffic patterns, Lambda (default) may be more cost-effective.

Monitor resource utilization. Track CloudWatch metrics to understand CPU and memory utilization. Adjust function memory allocation and instance type selection based on actual usage patterns to optimize costs.

Monitoring and observability

Monitor capacity provider metrics. Track capacity provider level metrics including CPUUtilization, MemoryUtilization, vCPUAvailable, and MemoryAvailable to verify sufficient resources are available for your workloads.

Monitor execution environment metrics. Track execution environment level metrics including ExecutionEnvironmentConcurrency and ExecutionEnvironmentConcurrencyLimit to understand scaling behavior and identify potential throttling.

Set up CloudWatch alarms. Create CloudWatch alarms for key metrics to proactively identify issues:

- High CPU or memory utilization
- Low available capacity
- Approaching concurrency limits

Language-specific considerations

Follow language-specific best practices. Each programming language handles multi-concurrency differently. Review the language-specific guides for detailed recommendations:

- **Java:** Use thread-safe collections, `AtomicInteger`, and `ThreadLocal` for request-specific state
- **Node.js:** Use InvokeStore for all request-specific state and avoid global variables
- **Python:** Use unique file names in `/tmp` with request IDs and consider process-based memory isolation
- **Rust:** Use `run_concurrent` instead of `run`, with the `concurrency-tokio` feature enabled. The handler must be `Clone` + `Send`.

Test for thread safety and concurrency issues. Before deploying to production, thoroughly test your functions for thread safety issues, race conditions, and proper state isolation under concurrent load.

Next steps

- Learn about capacity providers for Lambda Managed Instances
- Understand scaling for Lambda Managed Instances
- Review runtime-specific guides for Java, Node.js, and Python
- Configure VPC connectivity for your capacity providers
- Monitor Lambda Managed Instances with CloudWatch metrics

Troubleshooting Lambda Managed Instances

Throttling and scaling issues

High error rates during scale-up

Problem: You experience throttling errors (HTTP 429) when traffic increases rapidly.

Cause: Lambda Managed Instances scale asynchronously based on CPU resource utilization and multi-concurrency saturation. If your traffic more than doubles within 5 minutes, you may see throttles as Lambda scales up instances and execution environments to meet demand.

Solution:

- **Adjust target resource utilization:** If your workload has predictable traffic patterns, set a lower target resource utilization to maintain additional headroom for traffic bursts.
- **Pre-warm capacity:** For planned traffic increases, gradually ramp up traffic over a longer period to allow scaling to keep pace.
- **Monitor scaling metrics:** Track throttle error metrics to understand the reason for throttles and capacity scaling issues.
- **Review function configuration:** Ensure your function memory and vCPU settings support multi-concurrent executions. Increase function memory or vCPU allocation if needed.

Slow scale-down

Problem: Instances take a long time to scale down after traffic decreases.

Cause: Lambda Managed Instances scale down gradually to maintain availability and avoid rapid capacity changes that could impact performance.

Solution:

This is expected behavior. Lambda scales down instances conservatively to ensure stability. Monitor your CloudWatch metrics to track the number of running instances.

Concurrency issues

Execution environments with low concurrency experience throttles

Problem: Your functions experience throttling despite having available capacity.

Cause: Execution environments with very low maximum concurrency may have difficulty scaling effectively. Lambda Managed Instances are designed for multi-concurrent applications.

Solution:

- **Increase maximum concurrency:** If your function invocations use very little CPU, increase the maximum concurrency setting up to 64 per vCPU.
- **Optimize function code:** Review your function code to reduce CPU consumption per invocation, allowing higher concurrency.

- **Adjust function memory and vCPU:** Ensure your function has sufficient resources to handle multiple concurrent invocations.

Thread safety issues (Java runtime)

Problem: Your Java function produces incorrect results or experiences race conditions under load.

Cause: Multiple threads execute the handler method simultaneously, and shared state is not thread-safe.

Solution:

- Use `AtomicInteger` or `AtomicLong` for counters instead of primitive types
- Replace `HashMap` with `ConcurrentHashMap`
- Use `Collections.synchronizedList()` to wrap `ArrayList`
- Use `ThreadLocal` for request-specific state
- Access trace IDs from the Lambda Context object, not environment variables

For detailed guidance, see the Java runtime for Lambda Managed Instances documentation.

State isolation issues (Node.js runtime)

Problem: Your Node.js function returns data from different requests or experiences data corruption.

Cause: Global variables are shared across concurrent invocations on the same worker thread. When async operations yield control, other invocations can modify shared state.

Solution:

- Install and use `@aws/lambda-invoke-store` for all request-specific state
- Replace global variables with `InvokeStore.set()` and `InvokeStore.get()`
- Use unique file names in `/tmp` with request IDs
- Access trace IDs using `InvokeStore.getXRayTraceId()` instead of environment variables

For detailed guidance, see the Node.js runtime for Lambda Managed Instances documentation.

File conflicts (Python runtime)

Problem: Your Python function reads incorrect data from files in `/tmp`.

Cause: Multiple processes share the `/tmp` directory. Concurrent writes to the same file can cause data corruption.

Solution:

- Use unique file names with request IDs: `/tmp/request_{context.request_id}.txt`
- Use file locking with `fcntl.flock()` for shared files
- Clean up temporary files with `os.remove()` after use

For detailed guidance, see the Python runtime for Lambda Managed Instances documentation.

Performance issues

High memory utilization

Problem: Your functions experience high memory utilization or out-of-memory errors.

Cause: Each concurrent request in Python runs in a separate process with its own memory space. Total memory usage equals per-process memory multiplied by concurrent processes.

Solution:

- Monitor the `MemoryUtilization` metric in CloudWatch
- Reduce the `MaxConcurrency` setting if memory usage approaches the function's memory limit
- Increase function memory allocation to support higher concurrency
- Optimize memory usage by loading data on-demand instead of during initialization

Inconsistent performance

Problem: Function performance varies significantly between invocations.

Cause: Lambda may select different instance types based on availability, or functions may be running on instances with varying resource availability.

Solution:

- **Specify allowed instance types:** If you have specific performance requirements, configure allowed instance types in your capacity provider to limit the instance types Lambda can select.
- **Monitor instance-level metrics:** Track `CPUUtilization` and `MemoryUtilization` at the capacity provider level to identify resource constraints.
- **Review capacity metrics:** Check `vCPUAvailable` and `MemoryAvailable` to ensure sufficient resources are available on your instances.

Capacity provider issues

Function version fails to become ACTIVE

Problem: Your function version remains in a pending state after publishing.

Cause: Lambda is launching Managed Instances and starting execution environments. This process takes time, especially for the first function version on a new capacity provider.

Solution:

Wait for Lambda to complete the initialization process. Lambda launches three instances by default for AZ resiliency and starts three execution environments before marking your function version ACTIVE. This typically takes several minutes.

Cannot delete capacity provider

Problem: You receive an error when attempting to delete a capacity provider.

Cause: You cannot delete a capacity provider that has function versions attached to it.

Solution:

1. Identify all function versions using the capacity provider with the `ListFunctionVersionsByCapacityProvider` API.
2. Delete or update those function versions to remove the capacity provider association.
3. Retry deleting the capacity provider.

Generic error messages during function publishing

Problem: You encounter generic error messages such as "Internal error occurred during publishing" when publishing functions.

Solution:

- **Check IAM permissions:** Ensure you have the `lambda:PassCapacityProvider` permission for the capacity provider you're trying to use.
- **Verify capacity provider configuration:** Confirm that your capacity provider is in the ACTIVE state using the `GetCapacityProvider` API.
- **Review VPC configuration:** Ensure the subnets and security groups specified in your capacity provider are correctly configured and accessible.
- **Check AWS CloudTrail logs:** Review CloudTrail logs for detailed error information about the failed operation.

Monitoring and observability issues

Missing CloudWatch metrics

Problem: You don't see expected metrics in CloudWatch for your capacity provider or functions.

Cause: Metrics are published at 5-minute intervals. New capacity providers or functions may not have metrics available immediately.

Solution:

Wait at least 5-10 minutes after publishing a function version before expecting metrics to appear in CloudWatch. Verify you're looking at the correct namespace (`AWS/Lambda`) and dimensions (`CapacityProviderName`, `FunctionName`, or `InstanceType`).

Cannot find CloudWatch logs

Problem: Your function executes successfully, but you cannot find logs in CloudWatch Logs.

Cause: Lambda Managed Instances run in your VPC and require network connectivity to send logs to CloudWatch Logs. Without proper VPC connectivity configuration, your functions cannot reach the CloudWatch Logs service endpoint.

Solution:

Configure VPC connectivity to enable your functions to send logs to CloudWatch Logs. You have three options:

Option 1: VPC endpoint for CloudWatch Logs (recommended for production)

1. Open the Amazon VPC console at console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Endpoints**.
3. Choose **Create endpoint**.
4. For **Service category**, choose **AWS services**.
5. For **Service name**, select `com.amazonaws.region.logs` (replace `region` with your AWS Region).
6. For **VPC**, select the VPC used by your capacity provider.
7. For **Subnets**, select the subnets where you want to create endpoint network interfaces. For high availability, select subnets in multiple Availability Zones.
8. For **Security groups**, select security groups that allow inbound HTTPS traffic (port 443) from your function's security group.
9. Enable **Private DNS** for the endpoint.
10. Choose **Create endpoint**.

Option 2: Public subnet with internet gateway

If your capacity provider uses public subnets, ensure:

1. An internet gateway is attached to your VPC
2. The route table routes `0.0.0.0/0` traffic to the internet gateway
3. Security groups allow outbound HTTPS traffic on port 443

Option 3: Private subnet with NAT gateway

If your capacity provider uses private subnets, ensure:

1. A NAT gateway exists in a public subnet
2. The private subnet route table routes `0.0.0.0/0` traffic to the NAT gateway
3. The public subnet route table routes `0.0.0.0/0` traffic to an internet gateway
4. Security groups allow outbound HTTPS traffic on port 443

For detailed guidance on VPC connectivity options, see VPC connectivity for Lambda Managed Instances.

Difficulty correlating logs from concurrent requests

Problem: Logs from different requests are interleaved, making it difficult to trace individual requests.

Cause: Log interleaving is expected and standard behavior in multi-concurrent systems.

Solution:

- **Use structured logging with JSON format:** Include request ID in all log statements
- **Java:** Use Log4j with `ThreadContext` to automatically include request ID
- **Node.js:** Use `console.log()` with JSON formatting and include `InvokeStore.getRequestId()`
- **Python:** Use the standard logging module with JSON formatting and include `context.request_id`

For detailed guidance, see the runtime-specific documentation pages.

Getting additional help

If you continue to experience issues after trying these solutions:

1. **Review CloudWatch metrics:** Check capacity provider and execution environment metrics to identify resource constraints or scaling issues.
2. **Check AWS CloudTrail logs:** Review CloudTrail logs for detailed information about API calls and errors.
3. **Contact AWS Support:** If you cannot resolve the issue, contact AWS Support with details about your capacity provider configuration, function configuration, and the specific error messages you're encountering.

Next steps

- Learn about capacity providers for Lambda Managed Instances
- Understand scaling for Lambda Managed Instances
- Review runtime-specific guides for Java, Node.js, and Python
- Monitor Lambda Managed Instances with CloudWatch metrics

- Review best practices for Lambda Managed Instances

Lambda runtimes

Lambda supports multiple languages through the use of *runtimes*. A runtime provides a language-specific environment that relays invocation events, context information, and responses between Lambda and the function. You can use runtimes that Lambda provides, or build your own.

Lambda is agnostic to your choice of runtime. For simple functions, interpreted languages like Python and Node.js offer the fastest performance. For functions with more complex computation, compiled languages like Java are often slower to initialize but run quickly in the Lambda handler. Choice of runtime is also influenced by developer preference and language familiarity.

Each major programming language release has a separate runtime, with a unique *runtime identifier*, such as `nodejs24.x` or `python3.14`. To configure a function to use a new major language version, you need to change the runtime identifier. Since AWS Lambda cannot guarantee backward compatibility between major versions, this is a customer-driven operation.

For a function defined as a container image, you choose a runtime and the Linux distribution when you create the container image. To change the runtime, you create a new container image.

When you use a .zip file archive for the deployment package, you choose a runtime when you create the function. To change the runtime, you can update your function's configuration. The runtime is paired with one of the Amazon Linux distributions. The underlying execution environment provides additional libraries and environment variables that you can access from your function code.

Lambda invokes your function in an execution environment. The execution environment provides a secure and isolated runtime environment that manages the resources required to run your function. Lambda re-uses the execution environment from a previous invocation if one is available, or it can create a new execution environment.

To use other languages in Lambda, such as Go or Rust, use an OS-only runtime. The Lambda execution environment provides a runtime interface for getting invocation events and sending responses. You can deploy other languages by implementing a custom runtime alongside your function code, or in a layer.

Supported runtimes

The following table lists the supported Lambda runtimes and projected deprecation dates. After a runtime is deprecated, you're still able to create and update functions for a limited period. For

more information, see the section called "Runtime use after deprecation". The table provides the currently forecasted dates for runtime deprecation, based on our the section called "Runtime deprecation policy". These dates are provided for planning purposes and are subject to change.

Important

Amazon Linux 2 is scheduled for end of life on June 30, 2026. Lambda runtimes and container base images for Java 8 (AL2), Java 11, Java 17, Python 3.10, Python 3.11, and provided.al2 will continue to receive patches for critical and selected important Amazon Linux 2 security issues, in addition to language runtime patches, until the deprecation dates shown in the table below.

We recommend customers upgrade to an Amazon Linux 2023-based runtime as soon as possible. For customers upgrading to Java 21 or Java 25, you can use AWS Transform custom to assist with these upgrades. For customers unable to upgrade their Java version, we plan to release Amazon Linux 2023-based runtimes for Java 8, Java 11, and Java 17 before the end of Q2 2026.

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Node.js 24	`nodejs24.x`	Amazon Linux 2023	Apr 30, 2028	Jun 1, 2028	Jul 1, 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	Apr 30, 2027	Jun 1, 2027	Jul 1, 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2023	Apr 30, 2026	Aug 31, 2026	Sep 30, 2026
Python 3.14	`python3.14`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
Python 3.13	`python3.13`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Python 3.12	`python3.12`	Amazon Linux 2023	Oct 31, 2028	Nov 30, 2028	Jan 10, 2029
Python 3.11	`python3.11`	Amazon Linux 2	Jun 30, 2027	Jul 31, 2027	Aug 31, 2027
Python 3.10	`python3.10`	Amazon Linux 2	Oct 31, 2026	Nov 30, 2026	Jan 15, 2027
Java 25	`java25`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
Java 21	`java21`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
Java 17	`java17`	Amazon Linux 2	Jun 30, 2027	Jul 31, 2027	Aug 31, 2027
Java 11	`java11`	Amazon Linux 2	Jun 30, 2027	Jul 31, 2027	Aug 31, 2027
Java 8	`java8.al2`	Amazon Linux 2	Jun 30, 2027	Jul 31, 2027	Aug 31, 2027
.NET 10	`dotnet10`	Amazon Linux 2023	Nov 14, 2028	Dec 14, 2028	Jan 15, 2029
.NET 9 (container only)	`dotnet9`	Amazon Linux 2023	Nov 10, 2026	Not scheduled	Not scheduled
.NET 8	`dotnet8`	Amazon Linux 2023	Nov 10, 2026	Dec 10, 2026	Jan 11, 2027

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Ruby 3.4	`ruby3.4`	Amazon Linux 2023	Mar 31, 2028	Apr 30, 2028	May 31, 2028
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	Mar 31, 2027	Apr 30, 2027	May 31, 2027
Ruby 3.2	`ruby3.2`	Amazon Linux 2	Mar 31, 2026	Aug 31, 2026	Sep 30, 2026
OS-only Runtime	`provided.al2023`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
OS-only Runtime	`provided.al2`	Amazon Linux 2	Jul 31, 2026	Aug 31, 2026	Sep 30, 2026

Note

For new regions, Lambda will not support runtimes that are set to be deprecated within the next 6 months.

Lambda keeps managed runtimes and their corresponding container base images up to date with patches and support for minor version releases. For more information see Lambda runtime updates.

To programmatically interact with other AWS services and resources from your Lambda function, you can use one of AWS SDKs. The Node.js, Python, and Ruby runtimes include a version of the AWS SDK. However, to maintain full control of your dependencies, and to maximize backward compatibility during automatic runtime updates, we recommend that you always include the SDK modules your code uses (along with any dependencies) in your function's deployment package or in a Lambda layer.

We recommend that you use the runtime-included SDK version only when you can't include additional packages in your deployment. For example, when you create your function using the Lambda console code editor or using inline function code in an CloudFormation template.

Lambda periodically updates the versions of the AWS SDKs included in the Node.js, Python, and Ruby runtimes. To determine the version of the AWS SDK included in the runtime you're using, see the following sections:

- Runtime-included SDK versions (Node.js)
- Runtime-included SDK versions (Python)
- Runtime-included SDK versions (Ruby)

Lambda continues to support the Go programming language after deprecation of the Go 1.x runtime. For more information, see Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2 on the *AWS Compute Blog*.

All supported Lambda runtimes support both x86_64 and arm64 architectures.

New runtime releases

Lambda provides managed runtimes for new language versions only when the release reaches the long-term support (LTS) phase of the language's release cycle. For example, for the Node.js release cycle, when the release reaches the Active LTS phase.

Before the release reaches the long-term support phase, it remains in development and can still be subject to breaking changes. Lambda applies runtime updates automatically by default, so breaking changes to a runtime version could stop your functions from working as expected.

Lambda doesn't provide managed runtimes for language versions which aren't scheduled for LTS release.

The following list shows the target launch month for upcoming Lambda runtimes. These dates are indicative only and subject to change.

- **Ruby 3.5** - March 2026
- **Java 8, 11, and 17 on AL2023** - Q2 2026
- **Node.js 26** - November 2026
- **Python 3.15** - November 2026

Runtime deprecation policy

Lambda runtimes for .zip file archives are built around a combination of operating system, programming language, and software libraries that are subject to maintenance and security updates. Lambda's standard deprecation policy is to deprecate a runtime when any major component of the runtime reaches the end of community long-term support (LTS) and security updates are no longer available. Most usually, this is the language runtime, though in some cases, a runtime can be deprecated because the operating system (OS) reaches end of LTS.

After a runtime is deprecated, AWS may no longer apply security patches or updates to that runtime, and functions using that runtime are no longer eligible for technical support. Such deprecated runtimes are provided 'as-is', without any warranties, and may contain bugs, errors, defects, or other vulnerabilities.

To learn more about managing runtime upgrades and deprecation, see the following sections and Managing AWS Lambda runtime upgrades on the *AWS Compute Blog*.

> **Important**
>
> Lambda occasionally delays deprecation of a Lambda runtime for a limited period beyond the end of support date of the language version that the runtime supports. During this period, Lambda only applies security patches to the runtime OS. Lambda doesn't apply security patches to programming language runtimes after they reach their end of support date.

Shared responsibility model

Lambda is responsible for curating and publishing security updates for all supported managed runtimes and container base images. By default, Lambda will apply these updates automatically to functions using managed runtimes. Where the default automatic runtime update setting has been changed, see the runtime management controls shared responsibility model. For functions deployed using container images, you're responsible for rebuilding your function's container image from the latest base image and redeploying the container image.

When a runtime is deprecated, Lambda's responsibility for updating the managed runtime and container base images ceases. You are responsible for upgrading your functions to use a supported runtime or base image.

In all cases, you are responsible for applying updates to your function code, including its dependencies. Your responsibilities under the shared responsibility model are summarized in the following table.

Runtime lifecycle phase	Lambda's responsibilities	Your responsibilities
Supported managed runtime	Provide regular runtime updates with security patches and other updates. Apply runtime updates automatically by default (see the section called “Runtime update modes” for non-default behaviors).	Update your function code, including dependencies, to address any security vulnerabilities.
Supported container image	Provide regular updates to container base image with security patches and other updates.	Update your function code, including dependencies, to address any security vulnerabilities. Regularly re-build and re-deploy your container image using the latest base image.
Managed runtime approaching deprecation	Notify customers prior to runtime deprecation via documentation, Health Dashboard, email, and Trusted Advisor. Responsibility for runtime updates ends at deprecation.	Monitor Lambda documentation, Health Dashboard, email, or Trusted Advisor for runtime deprecation information. Upgrade functions to a supported runtime before the previous runtime is deprecated.
Container image approaching deprecation	Deprecation notifications are not available for functions using container images.	Be aware of deprecation schedules and upgrade functions to a supported base

Runtime lifecycle phase	Lambda's responsibilities	Your responsibilities
	Responsibility for container base image updates ends at deprecation.	image before the previous image is deprecated.

Runtime use after deprecation

After a runtime is deprecated, AWS may no longer apply security patches or updates to that runtime, and functions using that runtime are no longer eligible for technical support. While you can continue to invoke your functions indefinitely, AWS strongly recommends migrating to a supported runtime. Deprecated runtimes are provided 'as-is', without any warranties, and may contain bugs, errors, defects, or other vulnerabilities. Functions that use a deprecated runtime may also experience degraded performance or other issues, such as a certificate expiry, that can cause them to stop working properly.

You can update a function to use a newer supported runtime at any time after a runtime is deprecated. We recommend testing your function with the new runtime before applying changes in production environments. You will not be able to revert to the deprecated runtime after function updates are blocked. We recommend using function versions and aliases to enable safe deployment with rollback.

The following timeline describes what happens when a runtime is deprecated:

Runtime lifecycle phase	When	What
Deprecation notice period	At least 180 days before deprecation	• AWS sends notifications through email and the Health Dashboard to accounts that have functions using this runtime in their `$LATEST` version. • Affected functions are also listed in the Health Dashboard Scheduled changes tab and the AWS Trusted Advisor deprecated runtimes check.

Runtime lifecycle phase	When	What
Deprecation	Deprecation date	• AWS may no longer apply security updates or other updates. • Functions are no longer eligible for technical support. • You can no longer create or update functions using the deprecated runtime in the Lambda console. You can continue to create and update functions through the AWS CLI, AWS SAM, or CloudFormation.
Block function create	At least 30 days after deprecation	• Lambda begins blocking creation of new functions. • You can continue to update code and configuration for existing functions through the AWS CLI, AWS SAM, or CloudFormation..
Block function update	At least 60 days after deprecation	• Lambda begins blocking the update of code and configuration for existing functions. • You can still upgrade the function configuration to a supported runtime. However, rolling back to the deprecated runtime may be blocked.

Note

For some runtimes, AWS is delaying the block-function-create and block-function-update dates beyond the usual 30 and 60 days after deprecation. AWS has made this change in response to customer feedback to give you more time to upgrade your functions. Refer to the tables in the section called "Supported runtimes" and the section called "Deprecated runtimes" to see the dates for your runtime. Lambda will not start blocking function creates or updates before the dates given in these tables.

Receiving runtime deprecation notifications

When a runtime approaches its deprecation date, Lambda sends you an email alert if any functions in your AWS account use that runtime. Notifications are also displayed in the Health Dashboard and in AWS Trusted Advisor.

- Receiving email notifications:

  Lambda sends you an email alert at least **180 days** before a runtime is deprecated. This email lists the $LATEST versions of all functions using the runtime. To see a full list of affected function versions, use Trusted Advisor or see the section called "Get data about functions by runtime".

  Lambda sends email notification to your AWS account's primary account contact. For information about viewing or updating the email addresses in your account, see Updating contact information in the *AWS General Reference*.

- Receiving notifications through the Health Dashboard:

  The Health Dashboard displays a notification at least **180 days** before a runtime is deprecated. Notifications appear on the **Your account health** page under Other notifications. The **Affected resources** tab of the notification lists the $LATEST versions of all functions using the runtime.

  > **Note**
  >
  > To see a full and up-to-date list of affected function versions, use Trusted Advisor or see the section called "Get data about functions by runtime".

  Health Dashboard notifications expire 90 days after the affected runtime is deprecated.

- Using AWS Trusted Advisor

  Trusted Advisor displays a notification at least **180 days** before a runtime is deprecated. Notifications appear on the Security page. A list of your affected functions is displayed under **AWS Lambda Functions Using Deprecated Runtimes**. This list of functions shows both $LATEST and published versions and updates automatically to reflect your functions' current status.

  You can turn on weekly email notifications from Trusted Advisor in the Preferences page of the Trusted Advisor console.

Deprecated runtimes

The following runtimes have reached end of support:

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Python 3.9	`python3.9`	Amazon Linux 2	Dec 15, 2025	Aug 31, 2026	Sep 30, 2026
Node.js 18	`nodejs18.x`	Amazon Linux 2	Sep 1, 2025	Aug 31, 2026	Sep 30, 2026
.NET 6	`dotnet6`	Amazon Linux 2	Dec 20, 2024	Aug 31, 2026	Sep 30, 2026
Python 3.8	`python3.8`	Amazon Linux 2	Oct 14, 2024	Aug 31, 2026	Sep 30, 2026
Node.js 16	`nodejs16.x`	Amazon Linux 2	Jun 12, 2024	Aug 31, 2026	Sep 30, 2026
.NET 7 (container only)	`dotnet7`	Amazon Linux 2	May 14, 2024	N/A	N/A
Java 8	`java8`	Amazon Linux	Jan 8, 2024	Feb 8, 2024	Sep 30, 2026
Go 1.x	`go1.x`	Amazon Linux	Jan 8, 2024	Feb 8, 2024	Sep 30, 2026
OS-only Runtime	`provided`	Amazon Linux	Jan 8, 2024	Feb 8, 2024	Sep 30, 2026
Ruby 2.7	`ruby2.7`	Amazon Linux 2	Dec 7, 2023	Jan 9, 2024	Sep 30, 2026

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Node.js 14	`nodejs14.x`	Amazon Linux 2	Dec 4, 2023	Jan 9, 2024	Sep 30, 2026
Python 3.7	`python3.7`	Amazon Linux	Dec 4, 2023	Jan 9, 2024	Sep 30, 2026
.NET Core 3.1	`dotnetcore3.1`	Amazon Linux 2	Apr 3, 2023	Apr 3, 2023	May 3, 2023
Node.js 12	`nodejs12.x`	Amazon Linux 2	Mar 31, 2023	Mar 31, 2023	Apr 30, 2023
Python 3.6	`python3.6`	Amazon Linux	Jul 18, 2022	Jul 18, 2022	Aug 29, 2022
.NET 5 (container only)	`dotnet5.0`	Amazon Linux 2	May 10, 2022	N/A	N/A
.NET Core 2.1	`dotnetcore2.1`	Amazon Linux	Jan 5, 2022	Jan 5, 2022	Apr 13, 2022
Node.js 10	`nodejs10.x`	Amazon Linux 2	Jul 30, 2021	Jul 30, 2021	Feb 14, 2022
Ruby 2.5	`ruby2.5`	Amazon Linux	Jul 30, 2021	Jul 30, 2021	Mar 31, 2022
Python 2.7	`python2.7`	Amazon Linux	Jul 15, 2021	Jul 15, 2021	May 30, 2022
Node.js 8.10	`nodejs8.10`	Amazon Linux	Mar 6, 2020	Feb 4, 2020	Mar 6, 2020

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Node.js 4.3	`nodejs4.3`	Amazon Linux	Mar 5, 2020	Feb 3, 2020	Mar 5, 2020
Node.js 4.3 edge	`nodejs4.3-edge`	Amazon Linux	Mar 5, 2020	Mar 31, 2019	Apr 30, 2019
Node.js 6.10	`nodejs6.10`	Amazon Linux	Aug 12, 2019	Jul 12, 2019	Aug 12, 2019
.NET Core 1.0	`dotnetcore1.0`	Amazon Linux	Jun 27, 2019	Jun 30, 2019	Jul 30, 2019
.NET Core 2.0	`dotnetcore2.0`	Amazon Linux	May 30, 2019	Apr 30, 2019	May 30, 2019
Node.js 0.10	`nodejs`	Amazon Linux	Aug 30, 2016	Sep 30, 2016	Oct 31, 2016

In almost all cases, the end-of-life date of a language version or operating system is known well in advance. The following links give end-of-life schedules for each language that Lambda supports as a managed runtime.

Language and framework support policies

- **Node.js** – github.com
- **Python** – devguide.python.org
- **Ruby** – www.ruby-lang.org
- **Java** – www.oracle.com and Corretto FAQs
- **Go** – golang.org
- **.NET** – dotnet.microsoft.com

Understanding how Lambda manages runtime version updates

Lambda keeps each managed runtime up to date with security updates, bug fixes, new features, performance enhancements, and support for minor version releases. These runtime updates are published as *runtime versions*. Lambda applies runtime updates to functions by migrating the function from an earlier runtime version to a new runtime version.

By default, for functions using managed runtimes, Lambda applies runtime updates automatically. With automatic runtime updates, Lambda takes on the operational burden of patching the runtime versions. For most customers, automatic updates are the right choice. You can change this default behavior by configuring runtime management settings.

Lambda also publishes each new runtime version as a container image. To update runtime versions for container-based functions, you must create a new container image from the updated base image and redeploy your function.

Each runtime version is associated with a version number and an ARN (Amazon Resource Name). Runtime version numbers use a numbering scheme that Lambda defines, independent of the version numbers that the programming language uses. Runtime version numbers are not always sequential. For example, version 42 might be followed by version 45. The runtime version ARN is a unique identifier for each runtime version. You can view the ARN of your function's current runtime version in the Lambda console, or the `INIT_START` line of your function logs.

Runtime versions should not be confused with runtime identifiers. Each runtime has a unique **runtime identifier**, such as `python3.14` or `nodejs24.x`. These correspond to each major programming language release. Runtime versions describe the patch version of an individual runtime.

> **Note**
>
> The ARN for the same runtime version number can vary between AWS Regions and CPU architectures.

Topics

- Backward compatibility
- Runtime update modes
- Two-phase runtime version rollout

- Configuring Lambda runtime management settings
- Rolling back a Lambda runtime version
- Identifying Lambda runtime version changes
- Understanding the shared responsibility model for Lambda runtime management
- Controlling Lambda runtime update permissions for high-compliance applications

Backward compatibility

Lambda strives to provide runtime updates that are backward compatible with existing functions. However, as with software patching, there are rare cases in which a runtime update can negatively impact an existing function. For example, security patches can expose an underlying issue with an existing function that depends on the previous, insecure behavior.

When building and deploying your function, it is important to understand how to manage your dependencies to avoid potential incompatibilities with a future runtime update. For example, suppose your function has a dependency on package A, which in turn depends on package B. Both packages are included in the Lambda runtime (for example, they could be parts of the SDK or its dependencies, or parts of the runtime system libraries).

Consider the following scenarios:

Deployment	Patching compatible	Reason
• **Package A:** Use from runtime • **Package B:** Use from runtime	Yes	Future runtime updates to packages A and B are backward compatible.
• **Package A:** In deployment package • **Package B:** In deployment package	Yes	Your deployment takes precedence, so future runtime updates to packages A and B have no effect.
• **Package A:** In deployment package	Yes*	Future runtime updates to package B are backward compatible.

Deployment	Patching compatible	Reason
• **Package B:** Use from runtime		*If A and B are tightly coupled, compatibility issues can occur. For example, the `boto3` and `botocore` packages in the AWS SDK for Python should be deployed together.
• **Package A:** Use from runtime • **Package B:** In deployment package	No	Future runtime updates to package A might require an updated version of package B. However, the deployed version of package B takes precedence, and might not be forward compatible with the updated version of package A.

To maintain compatibility with future runtime updates, follow these best practices:

- **When possible, package all dependencies:** Include all required libraries, including the AWS SDK and its dependencies, in your deployment package. This ensures a stable, compatible set of components.
- **Use runtime-provided SDKs sparingly:** Only rely on the runtime-provided SDK when you can't include additional packages (for example, when using the Lambda console code editor or inline code in an AWS CloudFormation template).
- **Avoid overriding system libraries:** Don't deploy custom operating system libraries that may conflict with future runtime updates.

Runtime update modes

Lambda strives to provide runtime updates that are backward compatible with existing functions. However, as with software patching, there are rare cases in which a runtime update can negatively impact an existing function. For example, security patches can expose an underlying issue with an

existing function that depends on the previous, insecure behavior. Lambda runtime management controls help reduce the risk of impact to your workloads in the rare event of a runtime version incompatibility. For each function version ($LATEST or published version), you can choose one of the following runtime update modes:

- **Auto (default)** – Automatically update to the most recent and secure runtime version using Two-phase runtime version rollout. We recommend this mode for most customers so that you always benefit from runtime updates.
- **Function update** – Update to the most recent and secure runtime version when you update your function. When you update your function, Lambda updates the runtime of your function to the most recent and secure runtime version. This approach synchronizes runtime updates with function deployments, giving you control over when Lambda applies runtime updates. With this mode, you can detect and mitigate rare runtime update incompatibilities early. When using this mode, you must regularly update your functions to keep their runtime up to date.
- **Manual** – Manually update your runtime version. You specify a runtime version in your function configuration. The function uses this runtime version indefinitely. In the rare case in which a new runtime version is incompatible with an existing function, you can use this mode to roll back your function to an earlier runtime version. We recommend against using **Manual** mode to try to achieve runtime consistency across deployments. For more information, see Rolling back a Lambda runtime version.

Responsibility for applying runtime updates to your functions varies according to which runtime update mode you choose. For more information, see Understanding the shared responsibility model for Lambda runtime management.

Two-phase runtime version rollout

Lambda introduces new runtime versions in the following order:

1. In the first phase, Lambda applies the new runtime version whenever you create or update a function. A function gets updated when you call the UpdateFunctionCode or UpdateFunctionConfiguration API operations.
2. In the second phase, Lambda updates any function that uses the **Auto** runtime update mode and that hasn't already been updated to the new runtime version.

The overall duration of the rollout process varies according to multiple factors, including the severity of any security patches included in the runtime update.

If you're actively developing and deploying your functions, you will most likely pick up new runtime versions during the first phase. This synchronizes runtime updates with function updates. In the rare event that the latest runtime version negatively impacts your application, this approach lets you take prompt corrective action. Functions that aren't in active development still receive the operational benefit of automatic runtime updates during the second phase.

This approach doesn't affect functions set to **Function update** or **Manual** mode. Functions using **Function update** mode receive the latest runtime updates only when you create or update them. Functions using **Manual** mode don't receive runtime updates.

Lambda publishes new runtime versions in a gradual, rolling fashion across AWS Regions. If your functions are set to **Auto** or **Function update** modes, it's possible that functions deployed at the same time to different Regions, or at different times in the same Region, will pick up different runtime versions. Customers who require guaranteed runtime version consistency across their environments should use container images to deploy their Lambda functions. The **Manual** mode is designed as a temporary mitigation to enable runtime version rollback in the rare event that a runtime version is incompatible with your function.

Configuring Lambda runtime management settings

You can configure runtime management settings using the Lambda console or the AWS Command Line Interface (AWS CLI).

> **Note**
>
> You can configure runtime management settings separately for each function version.

To configure how Lambda updates your runtime version (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. On the **Code** tab, under **Runtime settings**, choose **Edit runtime management configuration**.
4. Under **Runtime management configuration**, choose one of the following:

   - To have your function update to the latest runtime version automatically, choose **Auto**.

- To have your function update to the latest runtime version when you change the function, choose **Function update**.
- To have your function update to the latest runtime version only when you change the runtime version ARN, choose **Manual**. You can find the runtime version ARN under **Runtime management configuration**. You can also find the ARN in the `INIT_START` line of your function logs.

For more information about these options, see Runtime update modes.

5. Choose **Save**.

To configure how Lambda updates your runtime version (AWS CLI)

To configure runtime management for a function, run the put-runtime-management-config AWS CLI command. When using `Manual` mode, you must also provide the runtime version ARN.

```
aws lambda put-runtime-management-config \
  --function-name my-function \
  --update-runtime-on Manual \
  --runtime-version-arn arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1
```

You should see output similar to the following:

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
  "RuntimeVersionArn": "arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

Rolling back a Lambda runtime version

In the rare event that a new runtime version is incompatible with your existing function, you can roll back its runtime version to an earlier one. This keeps your application working and minimizes disruption, providing time to remedy the incompatibility before returning to the latest runtime version.

Lambda doesn't impose a time limit on how long you can use any particular runtime version. However, we strongly recommend updating to the latest runtime version as soon as possible to benefit from the latest security patches, performance improvements, and features. Lambda provides the option to roll back to an earlier runtime version only as a temporary mitigation in the rare event of a runtime update compatibility issue. Functions using an earlier runtime version for an extended period may eventually experience degraded performance or issues, such as a certificate expiry, which can cause them to stop working properly.

You can roll back a runtime version in the following ways:

- Using the Manual runtime update mode
- Using published function versions

For more information, see Introducing AWS Lambda runtime management controls on the AWS Compute Blog.

Roll back a runtime version using Manual runtime update mode

If you're using the **Auto** runtime version update mode, or you're using the $LATEST runtime version, you can roll back your runtime version using the **Manual** mode. For the function version you want to roll back, change the runtime version update mode to **Manual** and specify the ARN of the previous runtime version. For more information about finding the ARN of the previous runtime version, see Identifying Lambda runtime version changes.

> **Note**
>
> If the $LATEST version of your function is configured to use **Manual** mode, then you can't change the CPU architecture or runtime version that your function uses. To make these changes, you must change to **Auto** or **Function update** mode.

Roll back a runtime version using published function versions

Published function versions are an immutable snapshot of the $LATEST function code and configuration at the time that you created them. In **Auto** mode, Lambda automatically updates the runtime version of published function versions during phase two of the runtime version rollout. In **Function update** mode, Lambda doesn't update the runtime version of published function versions.

Published function versions using **Function update** mode therefore create a static snapshot of the function code, configuration, and runtime version. By using **Function update** mode with function versions, you can synchronize runtime updates with your deployments. You can also coordinate rollback of code, configuration, and runtime versions by redirecting traffic to an earlier published function version. You can integrate this approach into your continuous integration and continuous delivery (CI/CD) for fully automatic rollback in the rare event of runtime update incompatibility. When using this approach, you must update your function regularly and publish new function versions to pick up the latest runtime updates. For more information, see Understanding the shared responsibility model for Lambda runtime management.

Identifying Lambda runtime version changes

The runtime version number and ARN are logged in the `INIT_START` log line, which Lambda emits to CloudWatch Logs each time that it creates a new execution environment. Because the execution environment uses the same runtime version for all function invocations, Lambda emits the `INIT_START` log line only when Lambda executes the init phase. Lambda doesn't emit this log line for each function invocation. Lambda emits the log line to CloudWatch Logs, but it is not visible in the console.

> **Note**
>
> Runtime version numbers are not always sequential. For example, version 42 might be followed by version 45.

Example INIT_START log line

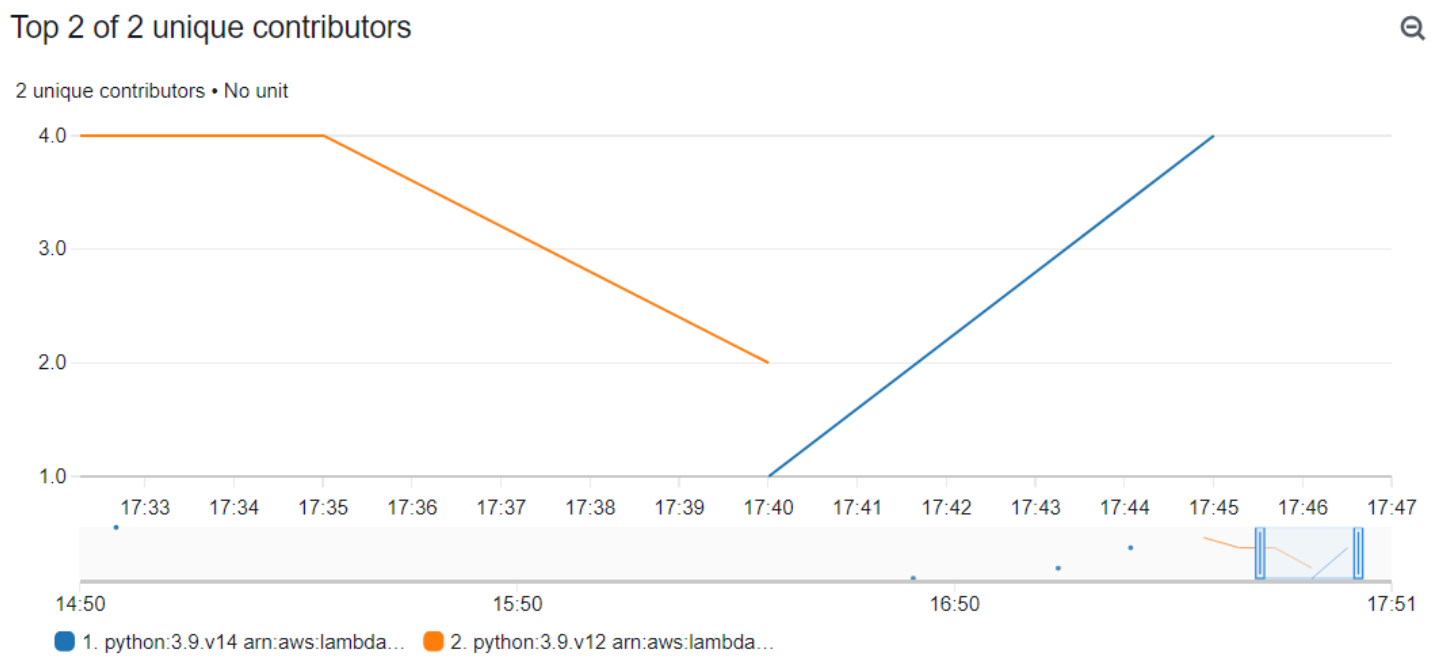
```
INIT_START Runtime Version: python:3.13.v14    Runtime Version ARN: arn:aws:lambda:eu-south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

Rather than working directly with the logs, you can use Amazon CloudWatch Contributor Insights to identify transitions between runtime versions. The following rule counts the distinct runtime versions from each `INIT_START` log line. To use the rule, replace the example log group name `/aws/lambda/*` with the appropriate prefix for your function or group of functions.

```
{
  "Schema": {
    "Name": "CloudWatchLogRule",
```

```
    "Version": 1
  },
  "AggregateOn": "Count",
  "Contribution": {
    "Filters": [
      {
        "Match": "eventType",
        "In": [
          "INIT_START"
        ]
      }
    ],
    "Keys": [
      "runtimeVersion",
      "runtimeVersionArn"
    ]
  },
  "LogFormat": "CLF",
  "LogGroupNames": [
    "/aws/lambda/*"
  ],
  "Fields": {
    "1": "eventType",
    "4": "runtimeVersion",
    "8": "runtimeVersionArn"
  }
}
```

The following CloudWatch Contributor Insights report shows an example of a runtime version transition as captured by the preceding rule. The orange line shows execution environment initialization for the earlier runtime version (**python:3.13.v12**), and the blue line shows execution environment initialization for the new runtime version (**python:3.13.v14**).



Understanding the shared responsibility model for Lambda runtime management

Lambda is responsible for curating and publishing security updates for all supported managed runtimes and container images. Responsibility for updating existing functions to use the latest runtime version varies depending on which runtime update mode you use.

Lambda is responsible for applying runtime updates to all functions configured to use the **Auto** runtime update mode.

For functions configured with the **Function update** runtime update mode, you're responsible for regularly updating your function. Lambda is responsible for applying runtime updates when you make those updates. If you don't update your function, then Lambda doesn't update the runtime. If you don't regularly update your function, then we strongly recommend configuring it for automatic runtime updates so that it continues to receive security updates.

For functions configured to use the **Manual** runtime update mode, you're responsible for updating your function to use the latest runtime version. We strongly recommend that you use this mode only to roll back the runtime version as a temporary mitigation in the rare event of runtime update incompatibility. We also recommend that you change to **Auto** mode as quickly as possible to minimize the time in which your functions aren't patched.

If you're using container images to deploy your functions, then Lambda is responsible for publishing updated base images. In this case, you're responsible for rebuilding your function's container image from the latest base image and redeploying the container image.

This is summarized in the following table:

Deployment mode	Lambda's responsibility	Customer's responsibility
Managed runtime, **Auto** mode	Publish new runtime versions containing the latest patches. Apply runtime patches to existing functions.	Roll back to a previous runtime version in the rare event of a runtime update compatibility issue. Follow best practices for backward compatibility.
Managed runtime, **Function update** mode	Publish new runtime versions containing the latest patches.	Update functions regularly to pick up the latest runtime version. Switch a function to **Auto** mode when you're not regularly updating the function. Roll back to a previous runtime version in the rare event of a runtime update compatibility issue. Follow best practices for backward compatibility.
Managed runtime, **Manual** mode	Publish new runtime versions containing the latest patches.	Use this mode only for temporary runtime rollback in the rare event of a runtime update compatibility issue. Switch functions to **Auto** or **Function update** mode and the latest runtime version as soon as possible.
Container image	Publish new container images containing the latest patches.	Redeploy functions regularly using the latest container base image to pick up the latest patches.

For more information about shared responsibility with AWS, see Shared Responsibility Model.

Controlling Lambda runtime update permissions for high-compliance applications

To meet patching requirements, Lambda customers typically rely on automatic runtime updates. If your application is subject to strict patching freshness requirements, you may want to limit use of earlier runtime versions. You can restrict Lambda's runtime management controls by using AWS Identity and Access Management (IAM) to deny users in your AWS account access to the PutRuntimeManagementConfig API operation. This operation is used to choose the runtime update mode for a function. Denying access to this operation causes all functions to default to the **Auto** mode. You can apply this restriction across your organization by using a service control policies (SCP). If you must roll back a function to an earlier runtime version, you can grant a policy exception on a case-by-case basis.

Retrieve data about Lambda functions that use a deprecated runtime

When a Lambda runtime is approaching deprecation, Lambda alerts you through email and provides notifications in the Health Dashboard and Trusted Advisor. These emails and notifications list the $LATEST versions of functions using the runtime. To list all of your function versions that use a particular runtime, you can use the AWS Command Line Interface (AWS CLI) or one of the AWS SDKs.

If you have a large number of functions which use a runtime that is due to be deprecated, you can also use the AWS CLI or AWS SDKs to help you prioritize updates to your most commonly invoked functions.

Refer to the following sections to learn how to use the AWS CLI and AWS SDKs to gather data about functions that use a particular runtime.

Listing function versions that use a particular runtime

To use the AWS CLI to list all of your function versions that use a particular runtime, run the following command. Replace `RUNTIME_IDENTIFIER` with the name of the runtime that's being deprecated and choose your own AWS Region. To list only $LATEST function versions, omit `--function-version ALL` from the command.

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --
query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

> **Tip**
>
> The example command lists functions in the `us-east-1` region for a particular AWS account You'll need to repeat this command for each region in which your account has functions and for each of your AWS accounts.

You can also list functions that use a particular runtime using one of the AWS SDKs. The following example code uses the V3 AWS SDK for JavaScript and the AWS SDK for Python (Boto3) to return a list of the function ARNs for functions using a particular runtime. The example code also returns the CloudWatch log group for each of the listed functions. You can use this log group to find the

last invocation date for the function. See the following section the section called “Identifying most commonly and most recently invoked functions” for more information.

Node.js

Example JavaScript code to list functions using a particular runtime

```
import { LambdaClient, ListFunctionsCommand } from "@aws-sdk/client-lambda";
const lambdaClient = new LambdaClient();

const command = new ListFunctionsCommand({
    FunctionVersion: "ALL",
    MaxItems: 50
});
const response = await lambdaClient.send(command);

for (const f of response.Functions){
    if (f.Runtime == '<your_runtime>'){ // Use the runtime id, e.g. 'nodejs24.x' or
 'python3.14'
        console.log(f.FunctionArn);
        // get the CloudWatch log group of the function to
        // use later for finding the last invocation date
        console.log(f.LoggingConfig.LogGroup);
    }
}
// If your account has more functions than the specified
// MaxItems, use the returned pagination token in the
// next request with the 'Marker' parameter
if ('NextMarker' in response){
    let paginationToken = response.NextMarker;
  }
```

Python

Example Python code to list functions using a particular runtime

```
import boto3
from botocore.exceptions import ClientError

def list_lambda_functions(target_runtime):

    lambda_client = boto3.client('lambda')
```

```
    response = lambda_client.list_functions(
        FunctionVersion='ALL',
        MaxItems=50
    )
    if not response['Functions']:
            print("No Lambda functions found")
    else:
        for function in response['Functions']:
            if function['PackageType']=='Zip' and function['Runtime'] ==
 target_runtime:
                print(function['FunctionArn'])
                # Print the CloudWatch log group of the function
                # to use later for finding last invocation date
                print(function['LoggingConfig']['LogGroup'])

    if 'NextMarker' in response:
       pagination_token = response['NextMarker']

if __name__ == "__main__":
    # Replace python3.12 with the appropriate runtime ID for your Lambda functions
    list_lambda_functions('python3.12')
```

To learn more about using an AWS SDK to list your functions using the ListFunctions action, see the SDK documentation for your preferred programming language.

You can also use the AWS Config Advanced queries feature to list all your functions that use an affected runtime. This query only returns function $LATEST versions, but you can aggregate queries to list function across all regions and multiple AWS accounts with a single command. To learn more, see Querying the Current Configuration State of AWS Auto Scaling Resources in the *AWS Config Developer Guide*.

Identifying most commonly and most recently invoked functions

If your AWS account contains functions that use a runtime that's due to be deprecated, you might want to prioritize updating functions that are frequently invoked or functions that have been invoked recently.

If you have only a few functions, you can use the CloudWatch Logs console to gather this information by looking at your functions' log streams. See View log data sent to CloudWatch Logs for more information.

To see the number of recent function invocations, you can also use the CloudWatch metrics information shown in the Lambda console. To view this information, do the following:

1. Open the Functions page of the Lambda console.
2. Select the function you want to see invocation statistics for.
3. Choose the **Monitor** tab.
4. Set the time period you wish to view statistics for using the date range picker. Recent invocations are displayed in the **Invocations** pane.

For accounts with larger numbers of functions, it can be more efficient to gather this data programmatically using the AWS CLI or one of the AWS SDKs using the DescribeLogStreams and GetMetricStatistics API actions.

The following examples provide code snippets using the V3 AWS SDK for JavaScript and the AWS SDK for Python (Boto3) to identify the last invoke date for a particular function and to determine the number of invocations for a particular function in the last 14 days.

Node.js

Example JavaScript code to find last invocation time for a function

```
import { CloudWatchLogsClient, DescribeLogStreamsCommand } from "@aws-sdk/client-
cloudwatch-logs";
const cloudWatchLogsClient = new CloudWatchLogsClient();
const command = new DescribeLogStreamsCommand({
    logGroupName: '<your_log_group_name>',
    orderBy: 'LastEventTime',
    descending: true,
    limit: 1
});
try {
    const response = await cloudWatchLogsClient.send(command);
    const lastEventTimestamp = response.logStreams.length > 0 ?
        response.logStreams[0].lastEventTimestamp : null;
    // Convert the UNIX timestamp to a human-readable format for display
    const date = new Date(lastEventTimestamp).toLocaleDateString();
    const time = new Date(lastEventTimestamp).toLocaleTimeString();
    console.log(`${date} ${time}`);

} catch (e){
    console.error('Log group not found.')
```

```
}
```

Python

Example Python code to find last invocation time for a function

```
import boto3
from datetime import datetime

cloudwatch_logs_client  = boto3.client('logs')

response = cloudwatch_logs_client.describe_log_streams(
    logGroupName='<your_log_group_name>',
    orderBy='LastEventTime',
    descending=True,
    limit=1
)

try:
    if len(response['logStreams']) > 0:
        last_event_timestamp = response['logStreams'][0]['lastEventTimestamp']
        print(datetime.fromtimestamp(last_event_timestamp/1000)) # Convert timestamp
 from ms to seconds
    else:
        last_event_timestamp = None
except:
    print('Log group not found')
```

Tip

You can find your function's log group name using the ListFunctions API operation. See the code in the section called “Listing function versions that use a particular runtime” for an example of how to do this.

Node.js

Example JavaScript code to find number of invocations in last 14 days

```
import { CloudWatchClient, GetMetricStatisticsCommand } from "@aws-sdk/client-
cloudwatch";
```

```
const cloudWatchClient = new CloudWatchClient();
const command = new GetMetricStatisticsCommand({
    Namespace: 'AWS/Lambda',
    MetricName: 'Invocations',
    StartTime: new Date(Date.now()-86400*1000*14), // 14 days ago
    EndTime: new Date(Date.now()),
    Period: 86400 * 14, // 14 days.
    Statistics: ['Sum'],
    Dimensions: [{
        Name: 'FunctionName',
        Value: '<your_function_name>'
    }]
});
const response = await cloudWatchClient.send(command);
const invokesInLast14Days = response.Datapoints.length > 0 ?
    response.Datapoints[0].Sum : 0;

console.log('Number of invocations: ' + invokesInLast14Days);
```

Python

Example Python code to find number of invocations in last 14 days

```
import boto3
from datetime import datetime, timedelta

cloudwatch_client = boto3.client('cloudwatch')

response = cloudwatch_client.get_metric_statistics(
    Namespace='AWS/Lambda',
    MetricName='Invocations',
    Dimensions=[
        {
            'Name': 'FunctionName',
            'Value': '<your_function_name>'
        },
    ],
    StartTime=datetime.now() - timedelta(days=14),
    EndTime=datetime.now(),
    Period=86400 * 14, # 14 days
    Statistics=[
        'Sum'
    ]
```

```
)

if len(response['Datapoints']) > 0:
    invokes_in_last_14_days = int(response['Datapoints'][0]['Sum'])
else:
    invokes_in_last_14_days = 0

print(f'Number of invocations: {invokes_in_last_14_days}')
```

Modifying the runtime environment

You can use internal extensions to modify the runtime process. Internal extensions are not separate processes—they run as part of the runtime process.

Lambda provides language-specific environment variables that you can set to add options and tools to the runtime. Lambda also provides wrapper scripts, which allow Lambda to delegate the runtime startup to your script. You can create a wrapper script to customize the runtime startup behavior.

Language-specific environment variables

Lambda supports configuration-only ways to enable code to be pre-loaded during function initialization through the following language-specific environment variables:

- `JAVA_TOOL_OPTIONS` – On Java, Lambda supports this environment variable to set additional command-line variables in Lambda. This environment variable allows you to specify the initialization of tools, specifically the launching of native or Java programming language agents using the `agentlib` or `javaagent` options. For more information, see `JAVA_TOOL_OPTIONS` environment variable.
- `NODE_OPTIONS` – Available in Node.js runtimes.
- `DOTNET_STARTUP_HOOKS` – On .NET Core 3.1 and above, this environment variable specifies a path to an assembly (dll) that Lambda can use.

Using language-specific environment variables is the preferred way to set startup properties.

Wrapper scripts

You can create a *wrapper script* to customize the runtime startup behavior of your Lambda function. A wrapper script enables you to set configuration parameters that cannot be set through language-specific environment variables.

> **Note**
>
> Invocations may fail if the wrapper script does not successfully start the runtime process.

Wrapper scripts are supported on all native Lambda runtimes. Wrapper scripts are not supported on OS-only runtimes (the `provided` runtime family).

When you use a wrapper script for your function, Lambda starts the runtime using your script. Lambda sends to your script the path to the interpreter and all of the original arguments for the standard runtime startup. Your script can extend or transform the startup behavior of the program. For example, the script can inject and alter arguments, set environment variables, or capture metrics, errors, and other diagnostic information.

You specify the script by setting the value of the `AWS_LAMBDA_EXEC_WRAPPER` environment variable as the file system path of an executable binary or script.

Example: Create and use a wrapper script as a Lambda layer

In the following example, you create a wrapper script to start the Python interpreter with the `-X importtime` option. When you run the function, Lambda generates a log entry to show the duration of the import time for each import.

To create and use a wrapper script as a layer

1. Create a directory for the layer:

```
mkdir -p python-wrapper-layer/bin
cd python-wrapper-layer/bin
```

2. In the `bin` directory, paste the following code into a new file named `importtime_wrapper`. This is the wrapper script.

```
#!/bin/bash

# the path to the interpreter and all of the originally intended arguments
args=("$@")

# the extra options to pass to the interpreter
extra_args=("-X" "importtime")

# insert the extra options
args=("${args[@]:0:$#-1}" "${extra_args[@]}" "${args[@]: -1}")

# start the runtime with the extra options
exec "${args[@]}"
```

3. Give the script executable permissions:

```
chmod +x importtime_wrapper
```

4. Create a .zip file for the layer:

```
cd ..
zip -r ../python-wrapper-layer.zip .
```

5. Confirm that your .zip file has the following directory structure:

```
python-wrapper-layer.zip
# bin
    # importtime_wrapper
```

6. Create a layer using the .zip package.
7. Create a function using the Lambda console.
   a. Open the Lambda console.
   b. Choose **Create function**.
   c. Enter a **Function name**.
   d. For **Runtime**, choose the **Latest supported** Python runtime.
   e. Choose **Create function**.
8. Add the layer to your function.
   a. Choose your function, and then choose the **Code** tab if it's not already selected.
   b. Scroll down to the **Layers** section, and then choose **Add a layer**.
   c. For **Layer source**, select **Custom layers**, and then choose your layer from the **Custom layers** dropdown list.
   d. For **Version**, choose **1**.
   e. Choose **Add**.
9. Add the wrapper environment variable.
   a. Choose the **Configuration** tab, then choose **Environment variables**.
   b. Under **Environment variables**, choose **Edit**.
   c. Choose **Add environment variable**.
   d. For **Key**, enter `AWS_LAMBDA_EXEC_WRAPPER`.

e. For **Value**, enter `/opt/bin/importtime_wrapper` (`/opt/` + your .zip layer's folder structure).

f. Choose **Save**.

10. Test the wrapper script.

a. Choose the **Test** tab.

b. Under **Test event**, choose **Test**. You don't need to create a test event—the default event will work.

c. Scroll down to **Log output**. Because your wrapper script started the Python interpreter with the `-X importtime` option, the logs show the time required for each import. For example:

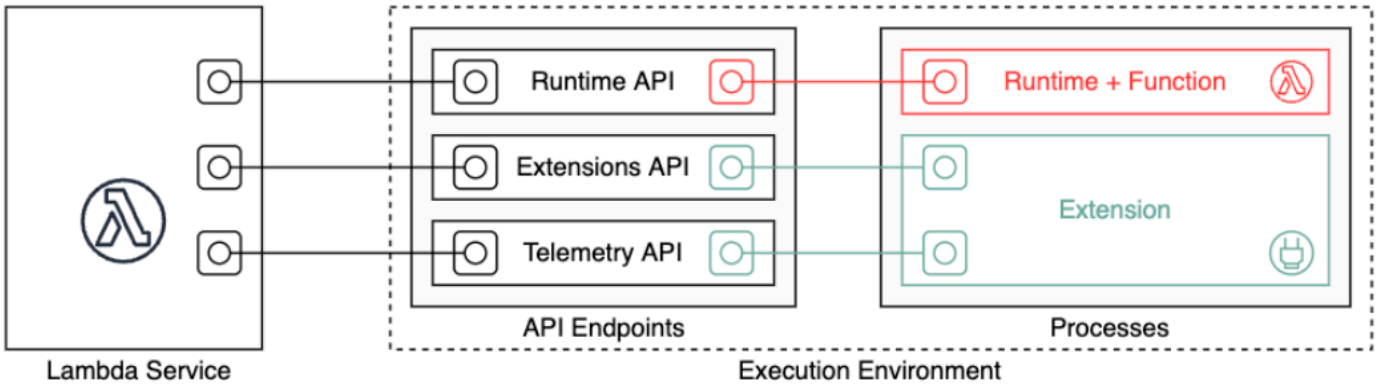
```
532 |           collections
import time:        63 |         63 |           _functools
import time:      1053 |       3646 |         functools
import time:      2163 |       7499 |       enum
import time:       100 |        100 |         _sre
import time:       446 |        446 |           re._constants
import time:       691 |       1136 |         re._parser
import time:       378 |        378 |         re._casefix
import time:       670 |       2283 |       re._compiler
import time:       416 |        416 |       copyreg
```

Using the Lambda runtime API for custom runtimes

AWS Lambda provides an HTTP API for custom runtimes to receive invocation events from Lambda and send response data back within the Lambda execution environment. This section contains the API reference for the Lambda runtime API.

Lambda Managed Instances support concurrent requests

Lambda Managed Instances use the same runtime API as Lambda (default) functions. The key difference is that Managed Instances can accept concurrent `/next` and `/response` requests up to the configured `AWS_LAMBDA_MAX_CONCURRENCY` limit. This enables multiple invocations to be processed simultaneously within a single execution environment. For more information about Managed Instances, see Understanding the Lambda Managed Instances execution environment.



The OpenAPI specification for the runtime API version **2018-06-01** is available in runtime-api.zip

To create an API request URL, runtimes get the API endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable, add the API version, and add the desired resource path.

Example Request

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

API methods

- Next invocation
- Invocation response
- Initialization error
- Invocation error

Next invocation

Path – `/runtime/invocation/next`

Method – **GET**

The runtime sends this message to Lambda to request an invocation event. The response body contains the payload from the invocation, which is a JSON document that contains event data from the function trigger. The response headers contain additional data about the invocation.

Response headers

- `Lambda-Runtime-Aws-Request-Id` – The request ID, which identifies the request that triggered the function invocation.

  For example, `8476a536-e9f4-11e8-9739-2dfe598c3fcd`.
- `Lambda-Runtime-Deadline-Ms` – The date that the function times out in Unix time milliseconds.

  For example, `1542409706888`.
- `Lambda-Runtime-Invoked-Function-Arn` – The ARN of the Lambda function, version, or alias that's specified in the invocation.

  For example, `arn:aws:lambda:us-east-2:123456789012:function:custom-runtime`.
- `Lambda-Runtime-Trace-Id` – The AWS X-Ray tracing header.

  For example, `Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1`.
- `Lambda-Runtime-Client-Context` – For invocations from the AWS Mobile SDK, data about the client application and device.
- `Lambda-Runtime-Cognito-Identity` – For invocations from the AWS Mobile SDK, data about the Amazon Cognito identity provider.

Do not set a timeout on the GET request as the response may be delayed. Between when Lambda bootstraps the runtime and when the runtime has an event to return, the runtime process may be frozen for several seconds.

The request ID tracks the invocation within Lambda. Use it to specify the invocation when you send the response.

The tracing header contains the trace ID, parent ID, and sampling decision. If the request is sampled, the request was sampled by Lambda or an upstream service. The runtime should set the _X_AMZN_TRACE_ID with the value of the header. The X-Ray SDK reads this to get the IDs and determine whether to trace the request.

Invocation response

Path – `/runtime/invocation/AwsRequestId/response`

Method – **POST**

After the function has run to completion, the runtime sends an invocation response to Lambda. For synchronous invocations, Lambda sends the response to the client.

Example success request

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "SUCCESS"
```

Initialization error

If the function returns an error or the runtime encounters an error during initialization, the runtime uses this method to report the error to Lambda.

Path – `/runtime/init/error`

Method – **POST**

Headers

`Lambda-Runtime-Function-Error-Type` – Error type that the runtime encountered. Required: no.

This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

Body parameters

`ErrorRequest` – Information about the error. Required: no.

This field is a JSON object with the following structure:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Note that Lambda accepts any value for `errorType`.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

Example Function error

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Response body parameters

- `StatusResponse` – String. Status information, sent with 202 response codes.
- `ErrorResponse` – Additional error information, sent with the error response codes. ErrorResponse contains an error type and an error message.

Response codes

- 202 – Accepted
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Runtime should exit promptly.

Example initialization error request

```
ERROR="{\"errorMessage\" : \"Failed to load function.\", \"errorType\" :
 \"InvalidFunctionException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR" --
header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

Invocation error

If the function returns an error or the runtime encounters an error, the runtime uses this method to report the error to Lambda.

Path – `/runtime/invocation/`*`AwsRequestId`*`/error`

Method – **POST**

Headers

`Lambda-Runtime-Function-Error-Type` – Error type that the runtime encountered. Required: no.

This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

Body parameters

`ErrorRequest` – Information about the error. Required: no.

This field is a JSON object with the following structure:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Note that Lambda accepts any value for `errorType`.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

Example Function error

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Response body parameters

- `StatusResponse` – String. Status information, sent with 202 response codes.
- `ErrorResponse` – Additional error information, sent with the error response codes. ErrorResponse contains an error type and an error message.

Response codes

- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Runtime should exit promptly.

Example error request

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="{\"errorMessage\" : \"Error parsing event data.\", \"errorType\" :
 \"InvalidEventDataException\"}"
```

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error"
 -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

When to use Lambda's OS-only runtimes

Lambda provides managed runtimes for Java, Python, Node.js, .NET, and Ruby. To create Lambda functions in a programming language that is not available as a managed runtime, use an OS-only runtime (the `provided` runtime family). There are three primary use cases for OS-only runtimes:

- **Native ahead-of-time (AOT) compilation**: Languages such as Go, Rust, Swift, and C++ compile natively to an executable binary, which doesn't require a dedicated language runtime. These languages only need an OS environment in which the compiled binary can run. You can also use Lambda OS-only runtimes to deploy binaries compiled with .NET Native AOT and Java GraalVM Native Image.

  You must include a runtime interface client in your binary. The runtime interface client calls the Using the Lambda runtime API for custom runtimes to retrieve function invocations and then calls your function handler. Lambda provides runtime interface clients for Rust, Go, .NET Native AOT, Swift (experimental), and C++ (experimental).

  You must compile your binary for a Linux environment and for the same instruction set architecture that you plan to use for the function (x86_64 or arm64).
- **Third-party runtimes**: You can run Lambda functions using off-the-shelf runtimes such as Bref for PHP.
- **Custom runtimes**: You can build your own runtime for a language or language version that Lambda doesn't provide a managed runtime for, such as Node.js 19. For more information, see Building a custom runtime for AWS Lambda. This is the least common use case for OS-only runtimes.

Lambda supports the following OS-only runtimes:

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
OS-only Runtime	`provided.al2023`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
OS-only Runtime	`provided.al2`	Amazon Linux 2	Jul 31, 2026	Aug 31, 2026	Sep 30, 2026

The Amazon Linux 2023 (`provided.al2023`) runtime provides several advantages over Amazon Linux 2, including a smaller deployment footprint and updated versions of libraries such as `glibc`.

The `provided.al2023` runtime uses `dnf` as the package manager instead of `yum`, which is the default package manager in Amazon Linux 2. For more information about the differences between `provided.al2023` and `provided.al2`, see Introducing the Amazon Linux 2023 runtime for AWS Lambda on the AWS Compute Blog.

Building a custom runtime for AWS Lambda

You can implement an AWS Lambda runtime in any programming language. A runtime is a program that runs a Lambda function's handler method when the function is invoked. You can include the runtime in your function's deployment package or distribute it in a layer. When you create the Lambda function, choose an OS-only runtime (the `provided` runtime family).

> **Note**
>
> Creating a custom runtime is an advanced use case. If you're looking for information about compiling to a native binary or using a third-party off-the-shelf runtime, see When to use Lambda's OS-only runtimes.

For a walkthrough of the custom runtime deployment process, see Tutorial: Building a custom runtime.

Topics

- Requirements
- Implementing response streaming in a custom runtime
- Building custom runtimes for Lambda Managed Instances

Requirements

Custom runtimes must complete certain initialization and processing tasks. A runtime runs the function's setup code, reads the handler name from an environment variable, and reads invocation events from the Lambda runtime API. The runtime passes the event data to the function handler, and posts the response from the handler back to Lambda.

Initialization tasks

The initialization tasks run once per instance of the function to prepare the environment to handle invocations.

- **Retrieve settings** – Read environment variables to get details about the function and environment.
  - `_HANDLER` – The location to the handler, from the function's configuration. The standard format is *`file`*`.`*`method`*, where `file` is the name of the file without an extension, and `method` is the name of a method or function that's defined in the file.
  - `LAMBDA_TASK_ROOT` – The directory that contains the function code.
  - `AWS_LAMBDA_RUNTIME_API` – The host and port of the runtime API.

  For a full list of available variables, see Defined runtime environment variables.
- **Initialize the function** – Load the handler file and run any global or static code that it contains. Functions should create static resources like SDK clients and database connections once, and reuse them for multiple invocations.
- **Handle errors** – If an error occurs, call the initialization error API and exit immediately.

Initialization counts towards billed execution time and timeout. When an execution triggers the initialization of a new instance of your function, you can see the initialization time in the logs and AWS X-Ray trace.

Example log

```
REPORT RequestId: f8ac1208... Init Duration: 48.26 ms   Duration: 237.17 ms   Billed
 Duration: 300 ms   Memory Size: 128 MB   Max Memory Used: 26 MB
```

Processing tasks

While it runs, a runtime uses the Lambda runtime interface to manage incoming events and report errors. After completing initialization tasks, the runtime processes incoming events in a loop. In your runtime code, perform the following steps in order.

- **Get an event** – Call the next invocation API to get the next event. The response body contains the event data. Response headers contain the request ID and other information.

- **Propagate the tracing header** – Get the X-Ray tracing header from the `Lambda-Runtime-Trace-Id` header in the API response. Set the `_X_AMZN_TRACE_ID` environment variable locally with the same value. The X-Ray SDK uses this value to connect trace data between services.
- **Create a context object** – Create an object with context information from environment variables and headers in the API response.
- **Invoke the function handler** – Pass the event and context object to the handler.
- **Handle the response** – Call the invocation response API to post the response from the handler.
- **Handle errors** – If an error occurs, call the invocation error API.
- **Cleanup** – Release unused resources, send data to other services, or perform additional tasks before getting the next event.

Entrypoint

A custom runtime's entry point is an executable file named `bootstrap`. The bootstrap file can be the runtime, or it can invoke another file that creates the runtime. If the root of your deployment package doesn't contain a file named `bootstrap`, Lambda looks for the file in the function's layers. If the `bootstrap` file doesn't exist or isn't executable, your function returns a `Runtime.InvalidEntrypoint` error upon invocation.

Here's an example `bootstrap` file that uses a bundled version of Node.js to run a JavaScript runtime in a separate file named `runtime.js`.

Example bootstrap

```
#!/bin/sh
    cd $LAMBDA_TASK_ROOT
    ./node-v11.1.0-linux-x64/bin/node runtime.js
```

Implementing response streaming in a custom runtime

For response streaming functions, the `response` and `error` endpoints have slightly modified behavior that lets the runtime stream partial responses to the client and return payloads in chunks. For more information about the specific behavior, see the following:

- `/runtime/invocation/AwsRequestId/response` – Propagates the `Content-Type` header from the runtime to send to the client. Lambda returns the response payload in chunks via HTTP/1.1 chunked transfer encoding. To stream the response to Lambda, the runtime must:

- Set the `Lambda-Runtime-Function-Response-Mode` HTTP header to `streaming`.
- Set the `Transfer-Encoding` header to `chunked`.
- Write the response conforming to the HTTP/1.1 chunked transfer encoding specification.
- Close the underlying connection after it has successfully written the response.

- `/runtime/invocation/AwsRequestId/error` – The runtime can use this endpoint to report function or runtime errors to Lambda, which also accepts the `Transfer-Encoding` header. This endpoint can only be called before the runtime begins sending an invocation response.
- Report midstream errors using error trailers in `/runtime/invocation/AwsRequestId/response` – To report errors that occur after the runtime starts writing the invocation response, the runtime can optionally attach HTTP trailing headers named `Lambda-Runtime-Function-Error-Type` and `Lambda-Runtime-Function-Error-Body`. Lambda treats this as a successful response and forwards the error metadata that the runtime provides to the client.

  > **Note**
  >
  > To attach trailing headers, the runtime must set the `Trailer` header value at the beginning of the HTTP request. This is a requirement of the HTTP/1.1 chunked transfer encoding specification.

  - `Lambda-Runtime-Function-Error-Type` – The error type that the runtime encountered. This header consists of a string value. Lambda accepts any string, but we recommend a format of *`<category.reason>`*. For example, `Runtime.APIKeyNotFound`.
  - `Lambda-Runtime-Function-Error-Body` – Base64-encoded information about the error.

Building custom runtimes for Lambda Managed Instances

Lambda Managed Instances use the same runtime API as Lambda (default) functions. However, there are key differences in how custom runtimes must be implemented to support the concurrent execution model of Managed Instances.

Concurrent request handling

The primary difference when building custom runtimes for Managed Instances is support for concurrent invocations. Unlike Lambda (default) functions where the runtime processes one

invocation at a time, Managed Instances can process multiple invocations simultaneously within a single execution environment.

Your custom runtime must:

- **Support concurrent `/next` requests** – The runtime can make multiple simultaneous calls to the next invocation API, up to the limit specified by the `AWS_LAMBDA_MAX_CONCURRENCY` environment variable.
- **Handle concurrent `/response` requests** – Multiple invocations can call the invocation response API simultaneously.
- **Implement thread-safe request handling** – Ensure that concurrent invocations don't interfere with each other by properly managing shared resources and state.
- **Use unique request IDs** – Track each invocation separately using the `Lambda-Runtime-Aws-Request-Id` header to match responses with their corresponding requests.

Implementation pattern

A typical implementation pattern for Managed Instances runtimes involves creating worker threads or processes to handle concurrent invocations:

1. **Read the concurrency limit** – At initialization, read the `AWS_LAMBDA_MAX_CONCURRENCY` environment variable to determine how many concurrent invocations to support.
2. **Create worker pool** – Initialize a pool of workers (threads, processes, or async tasks) equal to the concurrency limit.
3. **Worker processing loop** – Each worker independently:
   - Calls `/runtime/invocation/next` to get an invocation event
   - Invokes the function handler with the event data
   - Posts the response to `/runtime/invocation/AwsRequestId/response`
   - Repeats the loop

Additional considerations

- **Logging format** – Managed Instances only support JSON log format. Ensure your runtime respects the `AWS_LAMBDA_LOG_FORMAT` environment variable and only uses JSON format. For more information, see Configuring JSON and plain text log formats.

- **Shared resources** – Be cautious when using shared resources like the `/tmp` directory with concurrent invocations. Implement proper locking mechanisms to prevent race conditions.

For more information about Lambda Managed Instances execution environments, see Understanding the Lambda Managed Instances execution environment.

Tutorial: Building a custom runtime

In this tutorial, you create a Lambda function with a custom runtime. You start by including the runtime in the function's deployment package. Then you migrate it to a layer that you manage independently from the function. Finally, you share the runtime layer with the world by updating its resource-based permissions policy.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in Create a Lambda function with the console to create your first Lambda function.

To complete the following steps, you need the AWS CLI version 2. Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

You need an IAM role to create a Lambda function. The role needs permission to send logs to CloudWatch Logs and access the AWS services that your function uses. If you don't have a role for function development, create one now.

To create an execution role

1. Open the roles page in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.

   - **Trusted entity** – **Lambda**.
   - **Permissions** – **AWSLambdaBasicExecutionRole**.
   - **Role name** – **`lambda-role`**.

   The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to CloudWatch Logs.

Create a function

Create a Lambda function with a custom runtime. This example includes two files: a runtime `bootstrap` file and a function handler. Both are implemented in Bash.

1. Create a directory for the project, and then switch to that directory.

   ```
   mkdir runtime-tutorial
   cd runtime-tutorial
   ```

2. Create a new file called `bootstrap`. This is the custom runtime.

   Example bootstrap

   ```
   #!/bin/sh

   set -euo pipefail

   # Initialization - load function handler
   source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

   # Processing
   ```

```
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

The runtime loads a function script from the deployment package. It uses two variables to locate the script. `LAMBDA_TASK_ROOT` tells it where the package was extracted, and `_HANDLER` includes the name of the script.

After the runtime loads the function script, it uses the runtime API to retrieve an invocation event from Lambda, passes the event to the handler, and posts the response back to Lambda. To get the request ID, the runtime saves the headers from the API response to a temporary file, and reads the `Lambda-Runtime-Aws-Request-Id` header from the file.

> **Note**
>
> Runtimes have additional responsibilities, including error handling, and providing context information to the handler. For details, see Requirements.

3. Create a script for the function. The following example script defines a handler function that takes event data, logs it to `stderr`, and returns it.

Example function.sh

```
function handler () {
  EVENT_DATA=$1
  echo "$EVENT_DATA" 1>&2;
```

```
  RESPONSE="Echoing request: '$EVENT_DATA'"

  echo $RESPONSE
}
```

The `runtime-tutorial` directory should now look like this:

```
runtime-tutorial
# bootstrap
# function.sh
```

4. Make the files executable and add them to a .zip file archive. This is the deployment package.

```
chmod 755 function.sh bootstrap
zip function.zip function.sh bootstrap
```

5. Create a function named `bash-runtime`. For `--role`, enter the ARN of your Lambda execution role.

```
aws lambda create-function --function-name bash-runtime \
--zip-file fileb://function.zip --handler function.handler --runtime
 provided.al2023 \
--role arn:aws:iam::123456789012:role/lambda-role
```

6. Invoke the function.

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see a response like this:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

7. Verify the response.

```
cat response.txt
```

You should see a response like this:

```
Echoing request: '{"text":"Hello"}'
```

Create a layer

To separate the runtime code from the function code, create a layer that only contains the runtime. Layers let you develop your function's dependencies independently, and can reduce storage usage when you use the same layer with multiple functions. For more information, see Managing Lambda dependencies with layers.

1. Create a .zip file that contains the `bootstrap` file.

```
zip runtime.zip bootstrap
```

2. Create a layer with the publish-layer-version command.

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

This creates the first version of the layer.

Update the function

To use the runtime layer in the function, configure the function to use the layer, and remove the runtime code from the function.

1. Update the function configuration to pull in the layer.

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
```

This adds the runtime to the function in the `/opt` directory. To ensure that Lambda uses the runtime in the layer, you must remove the `boostrap` from the function's deployment package, as shown in the next two steps.

2. Create a .zip file that contains the function code.

```
zip function-only.zip function.sh
```

3. Update the function code to only include the handler script.

```
aws lambda update-function-code --function-name bash-runtime --zip-file fileb://
function-only.zip
```

4. Invoke the function to confirm that it works with the runtime layer.

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see a response like this:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

5. Verify the response.

```
cat response.txt
```

You should see a response like this:

```
Echoing request: '{"text":"Hello"}'
```

Update the runtime

1. To log information about the execution environment, update the runtime script to output environment variables.

   Example bootstrap

```
#!/bin/sh

set -euo pipefail

# Configure runtime to output environment variables
echo "##  Environment variables:"
env

# Load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

2. Create a .zip file that contains the new version of the `bootstrap` file.

```
zip runtime.zip bootstrap
```

3. Create a new version of the `bash-runtime` layer.

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

4. Configure the function to use the new version of the layer.

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

Share the layer

To share a layer with another AWS account, add a cross-account permissions statement to the layer's resource-based policy. Run the add-layer-version-permission command and specify the account ID as the `principal`. In each statement, you can grant permission to a single account, all accounts, or an organization in AWS Organizations.

The following example grants account 111122223333 access to version 2 of the `bash-runtime` layer.

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

You should see output similar to the following:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

Permissions apply only to a single layer version. Repeat the process each time that you create a new layer version.

Clean up

Delete each version of the layer.

```
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

Because the function holds a reference to version 2 of the layer, it still exists in Lambda. The function continues to work, but functions can no longer be configured to use the deleted version. If you modify the list of layers on the function, you must specify a new version or omit the deleted layer.

Delete the function with the delete-function command.

```
aws lambda delete-function --function-name bash-runtime
```

Open source repositories

AWS Lambda provides a variety of open source tools, libraries, and components to help you build, customize, and optimize your serverless applications. These resources include runtime interface clients, event libraries, container base images, development tools, and sample projects that are maintained by AWS and available on GitHub. By leveraging these open source repositories, you can extend Lambda's capabilities, create custom runtimes, process events from various AWS services, and gain deeper insights into your function's performance. This page provides an overview of the key open source projects that support Lambda development.

Runtime Interface Clients

Lambda Runtime Interface Clients (RICs) are open source libraries that implement the Runtime API and manage the interaction between your function code and the Lambda service. These clients handle receiving invocation events, passing context information, and reporting errors.

The runtime interface clients used by Lambda's managed runtimes and container base images are published as open source. When you build custom runtimes or extend existing ones, you can use these open source libraries to simplify your implementation. The following open source GitHub repositories contain the source code for Lambda's RICs:

- Node.js Runtime Interface Client
- Python Runtime Interface Client
- Java Runtime Interface Client

- Ruby Runtime Interface Client
- .NET Runtime Interface Client
- Rust Runtime Interface Client
- Go Runtime Interface Client
- Swift Runtime Interface Client (experimental)
- C++ Runtime Interface Client (experimental)
- Lambda Base Images

For more information about using these clients to build custom runtimes, see the section called "Building a custom runtime".

Event libraries

Lambda event libraries provide type definitions and helper utilities for processing events from various AWS services. These libraries help you parse and handle event data in a type-safe manner, making it easier to work with events from services like Amazon S3, Amazon DynamoDB, and Amazon API Gateway.

For compiled languages, AWS provides the following event libraries:

- Java Event Library
- .NET Event Libraries
- Go Event Library
- Rust Event Library

For interpreted languages like Node.js, Python, and Ruby, events can be parsed directly as JSON objects without requiring a separate library. However, developers using Node.js and Python can leverage powertools for AWS Lambda, which provides built-in schemas for AWS events that offer type hinting, data validation, and functionality similar to what compiled language libraries provide.

- Powertools for TypeScript
- Powertools for Python

Container base images

AWS provides open source container base images that you can use as a starting point for building container images for your Lambda functions. These base images include the runtime interface client and other components needed to run your functions in the Lambda execution environment.

For more information about the available base images and how to use them, see the AWS Lambda Base Images repository and the section called "Container images".

Development tools

AWS provides additional open source development tools to help you build and optimize your Lambda functions:

Powertools for AWS Lambda

Powertools for AWS Lambda simplifies serverless development with essential utilities to prevent duplicate processing, and batch processing for multi-record handling and Kafka consumer library. These features help you minimize code complexity and operational overhead.

You can also leverage built-in event schema validation, structured logging and tracing, and parameter store integration which are designed to accelerate the creation of production-ready Lambda functions while following AWS well-architected best practices.

GitHub repositories:

- Python
- TypeScript
- Java
- .NET

Java development tools

- Java Profiler (experimental) - A tool for profiling Java Lambda functions.
- Java Libraries - A repository that contains a comprehensive collection of Java libraries and tools for Lambda development, including key projects such as JUnit testing utilities and profiling tools.
- Serverless Java Container - A library that enables you to run existing Java applications on Lambda with minimal changes.

.NET development tools

The AWS Lambda .NET repository provides .NET libraries and tools for Lambda development, including key projects such as for AWS Lambda tools for the .NET CLI and .NET Core server for hosting .NET Core applications.

Sample projects

Explore a comprehensive collection of sample Lambda projects and applications at Serverless Land repositories. These samples demonstrate various Lambda use cases, integration patterns, and best practices to help you get started with your serverless applications.

Configuring AWS Lambda functions

Learn how to configure the core capabilities and options for your Lambda function using the Lambda API or console.

.zip file archives

Create a Lambda function deployment package when you want to include dependencies, custom runtime layers, or any files beyond your function code. The deployment package is a .zip file archive containing your function code and dependencies.

Container images

Use container images to package your function code and dependencies when you need more control over the build process, or if your function requires custom runtime configurations. You can build, test, and deploy Lambda functions as container images using tools like Docker CLI.

Memory

Learn how and when to increase function memory.

Ephemeral storage

Learn how and when to increase your function's temporary storage capacity.

Timeout

Learn how and when to increase your function's timeout value.

Environment variables

You can make your function code portable and keep secrets out of your code by storing them in your function's configuration by using environment variables.

Outbound networking

You can use your Lambda function with AWS resources in an Amazon VPC. Connecting your function to a VPC lets you access resources in a private subnet such as relational databases and caches.

Inbound networking

You can use an interface VPC endpoint to invoke your Lambda functions without crossing the public internet.

File system

You can use your Lambda function to mount a Amazon EFS to a local directory. A file system allows your function code to access and modify shared resources safely and at high concurrency.

Aliases

You can configure your clients to invoke a specific Lambda function version by using an alias, instead of updating the client.

Versions

By publishing a version of your function, you can store your code and configuration as a separate resource that cannot be changed.

Tags

Use tags to enable attribute-based access control (ABAC), to organize your Lambda functions, and to filter and generate reports on your functions using the AWS Cost Explorer or AWS Billing and Cost Management services.

Response streaming

You can configure your Lambda function URLs to stream response payloads back to clients. Response streaming can benefit latency sensitive applications by improving time to first byte (TTFB) performance. This is because you can send partial responses back to the client as they become available. Additionally, you can use response streaming to build functions that return larger payloads.

Metadata endpoint

Use the Lambda metadata endpoint to discover which Availability Zone your function is running in, enabling you to optimize latency by routing to same-AZ resources and to implement AZ-aware resilience patterns.

Deploying Lambda functions as .zip file archives

When you create a Lambda function, you package your function code into a deployment package. Lambda supports two types of deployment packages: container images and .zip file archives. The workflow to create a function depends on the deployment package type. To configure a function defined as a container image, see the section called “Container images”.

You can use the Lambda console and the Lambda API to create a function defined with a .zip file archive. You can also upload an updated .zip file to change the function code.

> **Note**
>
> You cannot change the deployment package type (.zip or container image) for an existing function. For example, you cannot convert a container image function to use a .zip file archive. You must create a new function.

Topics

- Creating the function
- Using the console code editor
- Updating function code
- Changing the runtime
- Changing the architecture
- Using the Lambda API
- Downloading your function code
- CloudFormation
- Encrypting Lambda .zip deployment packages

Creating the function

When you create a function defined with a .zip file archive, you choose a code template, the language version, and the execution role for the function. You add your function code after Lambda creates the function.

To create the function

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch** or **Use a blueprint** to create your function.
4. Under **Basic information**, do the following:
   a. For **Function name**, enter the function name. Function names are limited to 64 characters in length.
   b. For **Runtime**, choose the language version to use for your function.
   c. (Optional) For **Architecture**, choose the instruction set architecture to use for your function. The default architecture is x86_64. When you build the deployment package for your function, make sure that it is compatible with this instruction set architecture.
5. (Optional) Under **Permissions**, expand **Change default execution role**. You can create a new **Execution role** or use an existing role.
6. (Optional) Expand **Advanced settings**. You can choose a **Code signing configuration** for the function. You can also configure an (Amazon VPC) for the function to access.
7. Choose **Create function**.

Lambda creates the new function. You can now use the console to add the function code and configure other function parameters and features. For code deployment instructions, see the handler page for the runtime your function uses.

Node.js

Deploy Node.js Lambda functions with .zip file archives

Python

Working with .zip file archives for Python Lambda functions

Ruby

Deploy Ruby Lambda functions with .zip file archives

Java

Deploy Java Lambda functions with .zip or JAR file archives

Go

Deploy Go Lambda functions with .zip file archives

C#

Build and deploy C# Lambda functions with .zip file archives

PowerShell

Deploy PowerShell Lambda functions with .zip file archives

Using the console code editor

The console creates a Lambda function with a single source file. For scripting languages, you can edit this file and add more files using the built-in code editor. To save your changes, choose **Save**. Then, to run your code, choose **Test**.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to create a deployment package to upload your code to the Lambda function.

Updating function code

For scripting languages (Node.js, Python, and Ruby), you can edit your function code in the embedded code editor. If the code is larger than 3MB, or if you need to add libraries, or for languages that the editor doesn't support (Java, Go, C#), you must upload your function code as a .zip archive. If the .zip file archive is smaller than 50 MB, you can upload the .zip file archive from your local machine. If the file is larger than 50 MB, upload the file to the function from an Amazon S3 bucket.

To upload function code as a .zip archive

1. Open the Functions page of the Lambda console.
2. Choose the function to update and choose the **Code** tab.
3. Under **Code source**, choose **Upload from**.
4. Choose **.zip file**, and then choose **Upload**.
    - In the file chooser, select the new image version, choose **Open**, and then choose **Save**.

5. (Alternative to step 4) Choose **Amazon S3 location**.
    - In the text box, enter the S3 link URL of the .zip file archive, then choose **Save**.

Changing the runtime

If you update the function configuration to use a new runtime, you may need to update the function code to be compatible with the new runtime. If you update the function configuration to use a different runtime, you **must** provide new function code that is compatible with the runtime and architecture. For instructions on how to create a deployment package for the function code, see the handler page for the runtime that the function uses.

The Node.js 20, Python 3.12, Java 21, .NET 8, Ruby 3.3, and later base images are based on the Amazon Linux 2023 minimal container image. Earlier base images use Amazon Linux 2. AL2023 provides several advantages over Amazon Linux 2, including a smaller deployment footprint and updated versions of libraries such as `glibc`. For more information, see Introducing the Amazon Linux 2023 runtime for AWS Lambda on the AWS Compute Blog.

To change the runtime

1. Open the Functions page of the Lambda console.
2. Choose the function to update and choose the **Code** tab.
3. Scroll down to the **Runtime settings** section, which is under the code editor.
4. Choose **Edit**.
    a. For **Runtime**, select the runtime identifier.
    b. For **Handler**, specify file name and handler for your function.
    c. For **Architecture**, choose the instruction set architecture to use for your function.
5. Choose **Save**.

Changing the architecture

Before you can change the instruction set architecture, you need to ensure that your function's code is compatible with the target architecture.

If you use Node.js, Python, or Ruby and you edit your function code in the embedded editor, the existing code may run without modification.

However, if you provide your function code using a .zip file archive deployment package, you must prepare a new .zip file archive that is compiled and built correctly for the target runtime and instruction-set architecture. For instructions, see the handler page for your function runtime.

To change the instruction set architecture

1. Open the Functions page of the Lambda console.
2. Choose the function to update and choose the **Code** tab.
3. Under **Runtime settings**, choose **Edit**.
4. For **Architecture**, choose the instruction set architecture to use for your function.
5. Choose **Save**.

Using the Lambda API

To create and configure a function that uses a .zip file archive, use the following API operations:

- CreateFunction
- UpdateFunctionCode
- UpdateFunctionConfiguration

Downloading your function code

You can download the current unpublished (`$LATEST`) version of your function code .zip via the Lambda console. To do this, first ensure that you have the following IAM permissions:

- `iam:GetPolicy`
- `iam:GetPolicyVersion`
- `iam:GetRole`
- `iam:GetRolePolicy`
- `iam:ListAttachedRolePolicies`
- `iam:ListRolePolicies`
- `iam:ListRoles`

To download the function code .zip

1. Open the Functions page of the Lambda console.
2. Choose the function you want to download the function code .zip for.
3. In the **Function overview**, choose the **Download** button, then choose **Download function code .zip**.

    - Alternatively, choose **Download AWS SAM file** to generate and download a SAM template based on your function's configuration. You can also choose **Download both** to download both the .zip and the SAM template.

CloudFormation

You can use CloudFormation to create a Lambda function that uses a .zip file archive. In your CloudFormation template, the `AWS::Lambda::Function` resource specifies the Lambda function. For descriptions of the properties in the `AWS::Lambda::Function` resource, see AWS::Lambda::Function in the *AWS CloudFormation User Guide*.

In the `AWS::Lambda::Function` resource, set the following properties to create a function defined as a .zip file archive:

- AWS::Lambda::Function
    - PackageType – Set to `Zip`.
    - Code – Enter the Amazon S3 bucket name and .zip file name in the `S3Bucket` and `S3Key`fields. For Node.js or Python, you can provide inline source code of your Lambda function.
    - Runtime – Set the runtime value.
    - Architecture – Set the architecture value to `arm64` to use the AWS Graviton2 processor. By default, the architecture value is `x86_64`.

Encrypting Lambda .zip deployment packages

Lambda always provides server-side encryption at rest for .zip deployment packages and function configuration details with an AWS KMS key. By default, Lambda uses an AWS owned key. If this default behavior suits your workflow, you don't need to set up anything else. AWS doesn't charge you to use this key.

If you prefer, you can provide an AWS KMS customer managed key instead. You might do this to have control over rotation of the KMS key or to meet the requirements of your organization for managing KMS keys. When you use a customer managed key, only users in your account with access to the KMS key can view or manage the function's code or configuration.

Customer managed keys incur standard AWS KMS charges. For more information, see AWS Key Management Service pricing.

Create a customer managed key

You can create a symmetric customer managed key by using the AWS Management Console, or the AWS KMS APIs.

To create a symmetric customer managed key

Follow the steps for Creating symmetric encryption Creating symmetric KMS keys in the *AWS Key Management Service Developer Guide*.

Permissions

Key policy

Key policies control access to your customer managed key. Every customer managed key must have exactly one key policy, which contains statements that determine who can use the key and how they can use it. For more information, see How to change a key policy in the *AWS Key Management Service Developer Guide*.

When you use a customer managed key to encrypt a .zip deployment package, Lambda doesn't add a grant to the key. Instead, your AWS KMS key policy must allow Lambda to call the following AWS KMS API operations on your behalf:

- kms:GenerateDataKey
- kms:Decrypt

The following example key policy allows all Lambda functions in account 111122223333 to call the required AWS KMS operations for the specified customer managed key:

Example AWS KMS key policy

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": [
                "kms:GenerateDataKey",
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-1:111122223333:key/key-id",
            "Condition": {
                "StringLike": {
                "kms:EncryptionContext:aws:lambda:FunctionArn":
 "arn:aws:lambda:us-east-1:111122223333:function:*"
                }
            }
        }
    ]
}
```

For more information about troubleshooting key access, see the *AWS Key Management Service Developer Guide*.

Principal permissions

When you use a customer managed key to encrypt a .zip deployment package, only principals with access to that key can access the .zip deployment package. For example, principals who don't have access to the customer managed key can't download the .zip package using the presigned S3 URL that's included in the GetFunction response. An `AccessDeniedException` is returned in the `Code` section of the response.

Example AWS KMS AccessDeniedException

```
{
```

```
    "Code": {
        "RepositoryType": "S3",
        "Error": {
            "ErrorCode": "AccessDeniedException",
            "Message": "KMS access is denied. Check your KMS permissions. KMS
 Exception: AccessDeniedException KMS Message: User: arn:aws:sts::111122223333:assumed-
role/LambdaTestRole/session is not authorized to perform: kms:Decrypt on resource:
 arn:aws:kms:us-east-1:111122223333:key/key-id with an explicit deny in a resource-
based policy"
        },
        "SourceKMSKeyArn": "arn:aws:kms:us-east-1:111122223333:key/key-id"
    },
 ...
```

For more information about permissions for AWS KMS keys, see Authentication and access control for AWS KMS.

Using a customer managed key for your .zip deployment package

Use the following API parameters to configure customer managed keys for .zip deployment packages:

- SourceKMSKeyArn: Encrypts the source .zip deployment package (the file that you upload).
- KMSKeyArn: Encrypts environment variables and Lambda SnapStart snapshots.

When `SourceKMSKeyArn` and `KMSKeyArn` are both specified, Lambda uses the `KMSKeyArn` key to encrypt the unzipped version of the package that Lambda uses to invoke the function. When `SourceKMSKeyArn` is specified but `KMSKeyArn` is not, Lambda uses an AWS managed key to encrypt the unzipped version of the package.

Lambda console

To add customer managed key encryption when you create a function

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch** or **Container image**.
4. Under **Basic information**, do the following:

    a. For **Function name**, enter the function name.

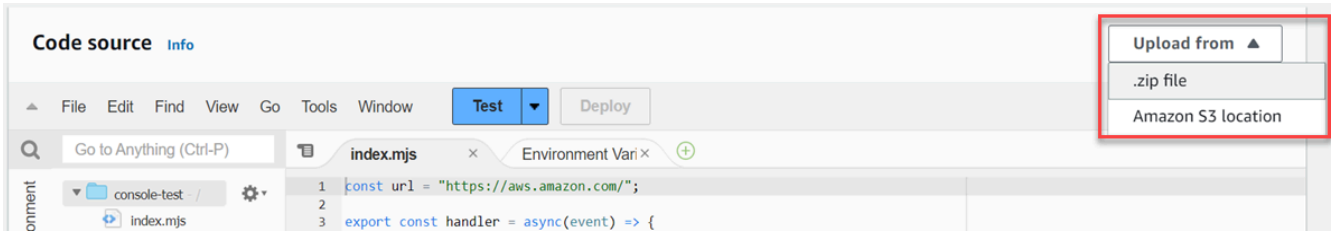
b. For **Runtime**, choose the language version to use for your function.

5. Expand **Advanced settings**, and then select **Enable encryption with an AWS KMS customer managed key**.
6. Choose a customer managed key.
7. Choose **Create function**.

To remove customer managed key encryption, or to use a different key, you must upload the .zip deployment package again.

To add customer managed key encryption to an existing function

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **.zip file** or **Amazon S3 location**.



5. Upload the file or enter the Amazon S3 location.
6. Choose **Enable encryption with an AWS KMS customer managed key**.
7. Choose a customer managed key.
8. Choose **Save**.

AWS CLI

To add customer managed key encryption when you create a function

In the following create-function example:

- `--code`: Specifies the local path to the .zip deployment package (`ZipFile`) and the customer managed key to encrypt it (`SourceKMSKeyArn`).
- `--kms-key-arn`: Specifies the customer managed key to encrypt the environment variables and the unzipped version of the deployment package.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
  --code ZipFile=fileb://myFunction.zip,SourceKMSKeyArn=arn:aws:kms:us-
east-1:111122223333:key/key-id \
  --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key2-id
```

In the following create-function example:

- `--code`: Specifies the location of the .zip file in an Amazon S3 bucket (`S3Bucket`, `S3Key`, `S3ObjectVersion`) and the customer managed key to encrypt it (`SourceKMSKeyArn`).
- `--kms-key-arn`: Specifies the customer managed key to encrypt the environment variables and the unzipped version of the deployment package.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x --handler index.handler \
  --role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
  --code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion,SourceKMSKeyArn=arn:aws:kms:us-
east-1:111122223333:key/key-id \
  --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key2-id
```

To add customer managed key encryption to an existing function

In the following update-function-code example:

- `--zip-file`: Specifies the local path to the .zip deployment package.
- `--source-kms-key-arn`: Specifies the customer managed key to encrypt the zipped version of the deployment package. Lambda uses an AWS owned key to encrypt the unzipped package for function invocations. If you want to use a customer managed key to encrypt the unzipped version of the package, run the update-function-configuration command with the `--kms-key-arn` option.

```
aws lambda update-function-code \
  --function-name myFunction \
```

```
  --zip-file fileb://myFunction.zip \
  --source-kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key-id
```

In the following update-function-code example:

- `--s3-bucket`: Specifies the location of the .zip file in an Amazon S3 bucket.
- `--s3-key`: Specifies the Amazon S3 key of the deployment package.
- `--s3-object-version`: For versioned objects, the version of the deployment package object to use.
- `--source-kms-key-arn`: Specifies the customer managed key to encrypt the zipped version of the deployment package. Lambda uses an AWS owned key to encrypt the unzipped package for function invocations. If you want to use a customer managed key to encrypt the unzipped version of the package, run the update-function-configuration command with the `--kms-key-arn` option.

```
aws lambda update-function-code \
  --function-name myFunction \
  --s3-bucket amzn-s3-demo-bucket \
  --s3-key myFileName.zip \
  --s3-object-version myObject Version
  --source-kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key-id
```

To remove customer managed key encryption from an existing function

In the following update-function-code example, `--zip-file` specifies the local path to the .zip deployment package. When you run this command without the `--source-kms-key-arn` option, Lambda uses an AWS owned key to encrypt the zipped version of the deployment package.

```
aws lambda update-function-code \
  --function-name myFunction \
  --zip-file fileb://myFunction.zip
```

Create a Lambda function using a container image

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

There are three ways to build a container image for a Lambda function:

- Using an AWS base image for Lambda

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.
- Using an AWS OS-only base image

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include a runtime interface client for your language in the image.
- Using a non-AWS base image

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include a runtime interface client for your language in the image.

> **Tip**
>
> To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

To create a Lambda function from a container image, build your image locally and upload it to an Amazon Elastic Container Registry (Amazon ECR) repository. If you're using a container image provided by an AWS Marketplace seller, you need to clone the image to your private Amazon ECR

repository first. Then, specify the repository URI when you create the function. The Amazon ECR repository must be in the same AWS Region as the Lambda function. You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

> **Note**
>
> Lambda does not support Amazon ECR FIPS endpoints for container images. If your repository URI includes `ecr-fips`, you are using a FIPS endpoint. Example: `111122223333.dkr.ecr-fips.us-east-1.amazonaws.com`.

This page explains the base image types and requirements for creating Lambda-compatible container images.

> **Note**
>
> You cannot change the deployment package type (.zip or container image) for an existing function. For example, you cannot convert a container image function to use a .zip file archive. You must create a new function.

Topics

- Requirements
- Using an AWS base image for Lambda
- Using an AWS OS-only base image
- Using a non-AWS base image
- Runtime interface clients
- Amazon ECR permissions
- Function lifecycle

Requirements

Install the AWS CLI version 2 and the Docker CLI. Additionally, note the following requirements:

- The container image must implement the Using the Lambda runtime API for custom runtimes. The AWS open-source runtime interface clients implement the API. You can add a runtime interface client to your preferred base image to make it compatible with Lambda.
- The container image must be able to run on a read-only file system. Your function code can access a writable `/tmp` directory with between 512 MB and 10,240 MB, in 1-MB increments, of storage.
- The default Lambda user must be able to read all the files required to run your function code. Lambda follows security best practices by defining a default Linux user with least-privileged permissions. This means that you don't need to specify a USER in your Dockerfile. Verify that your application code does not rely on files that other Linux users are restricted from running.
- Lambda supports only Linux-based container images.
- Lambda provides multi-architecture base images. However, the image you build for your function must target only one of the architectures. Lambda does not support functions that use multi-architecture container images.

Using an AWS base image for Lambda

You can use one of the AWS base images for Lambda to build the container image for your function code. The base images are preloaded with a language runtime and other components required to run a container image on Lambda. You add your function code and dependencies to the base image and then package it as a container image.

AWS periodically provides updates to the AWS base images for Lambda. If your Dockerfile includes the image name in the FROM property, your Docker client pulls the latest version of the image from the Amazon ECR repository. To use the updated base image, you must rebuild your container image and update the function code.

The Node.js 20, Python 3.12, Java 21, .NET 8, Ruby 3.3, and later base images are based on the Amazon Linux 2023 minimal container image. Earlier base images use Amazon Linux 2. AL2023 provides several advantages over Amazon Linux 2, including a smaller deployment footprint and updated versions of libraries such as `glibc`.

AL2023-based images use `microdnf` (symlinked as `dnf`) as the package manager instead of `yum`, which is the default package manager in Amazon Linux 2. `microdnf` is a standalone implementation of `dnf`. For a list of packages that are included in AL2023-based images, refer to the **Minimal Container** columns in Comparing packages installed on Amazon Linux 2023 Container

Images. For more information about the differences between AL2023 and Amazon Linux 2, see Introducing the Amazon Linux 2023 runtime for AWS Lambda on the AWS Compute Blog.

> **Note**
>
> To run AL2023-based images locally, including with AWS Serverless Application Model (AWS SAM), you must use Docker version 20.10.10 or later.

To build a container image using an AWS base image, choose the instructions for your preferred language:

- Node.js
- TypeScript (uses a Node.js base image)
- Python
- Java
- Go
- .NET
- Ruby

Using an AWS OS-only base image

AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include a runtime interface client for your language in the image.

Tags	Runtime	Operating system	Dockerfile	Deprecation
al2023	OS-only Runtime	Amazon Linux 2023	Dockerfile for OS-only Runtime on GitHub	Jun 30, 2029

Tags	Runtime	Operating system	Dockerfile	Deprecation
al2	OS-only Runtime	Amazon Linux 2	Dockerfile for OS-only Runtime on GitHub	Jul 31, 2026

Amazon Elastic Container Registry Public Gallery: gallery.ecr.aws/lambda/provided

Using a non-AWS base image

Lambda supports any image that conforms to one of the following image manifest formats:

- Docker image manifest V2, schema 2 (used with Docker version 1.10 and newer)
- Open Container Initiative (OCI) Specifications (v1.0.0 and up)

Lambda supports a maximum uncompressed image size of 10 GB, including all layers.

Note

- To make the image compatible with Lambda, you must include a runtime interface client for your language in the image.
- For optimal performance, keep your image manifest size under 25,400 bytes. To reduce image manifest size, minimize the number of layers in your image and reduce annotations.

Runtime interface clients

If you use an OS-only base image or an alternative base image, you must include a runtime interface client in your image. The runtime interface client must extend the Using the Lambda runtime API for custom runtimes, which manages the interaction between Lambda and your function code. AWS provides open-source runtime interface clients for the following languages:

- Node.js
- Python
- Java

- .NET
- Go
- Ruby
- Rust –

If you're using a language that doesn't have an AWS-provided runtime interface client, you must create your own.

Amazon ECR permissions

Before you create a Lambda function from a container image, you must build the image locally and upload it to an Amazon ECR repository. When you create the function, specify the Amazon ECR repository URI.

Make sure that the permissions for the user or role that creates the function includes `GetRepositoryPolicy`, `SetRepositoryPolicy`, `BatchGetImage`, and `GetDownloadUrlForLayer`.

For example, use the IAM console to create a role with the following policy:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "ecr:SetRepositoryPolicy",
        "ecr:GetRepositoryPolicy",
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world"
    }
  ]
}
```

Amazon ECR repository policies

For a function in the same account as the container image in Amazon ECR, you can add `ecr:BatchGetImage` and `ecr:GetDownloadUrlForLayer` permissions to your Amazon ECR repository policy. The following example shows the minimum policy:

```
{
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

For more information about Amazon ECR repository permissions, see Private repository policies in the *Amazon Elastic Container Registry User Guide*.

If the Amazon ECR repository does not include these permissions, Lambda attempts to add them automatically. Lambda can add permissions only if the principal calling Lambda has `ecr:getRepositoryPolicy` and `ecr:setRepositoryPolicy` permissions.

To view or edit your Amazon ECR repository permissions, follow the directions in Setting a private repository policy statement in the *Amazon Elastic Container Registry User Guide*.

Amazon ECR cross-account permissions

A different account in the same region can create a function that uses a container image owned by your account. In the following example, your Amazon ECR repository permissions policy needs the following statements to grant access to account number 123456789012.

- **CrossAccountPermission** – Allows account 123456789012 to create and update Lambda functions that use images from this ECR repository.
- **LambdaECRImageCrossAccountRetrievalPolicy** – Lambda will eventually set a function's state to inactive if it is not invoked for an extended period. This statement is required so that Lambda can retrieve the container image for optimization and caching on behalf of the function owned by 123456789012.

Example— Add cross-account permission to your repository

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "CrossAccountPermission",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      },
      "Resource": "arn:aws:ecr:us-east-1:123456789012:repository/example-lambda-repository"
    },
    {
      "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "ArnLike": {
          "aws:sourceARN": "arn:aws:lambda:us-east-1:123456789012:function:*"
        }
      },
      "Resource": "arn:aws:ecr:us-east-1:123456789012:repository/example-lambda-repository"
    }
  ]
}
```

To give access to multiple accounts, you add the account IDs to the Principal list in the `CrossAccountPermission` policy and to the Condition evaluation list in the `LambdaECRImageCrossAccountRetrievalPolicy`.

If you are working with multiple accounts in an AWS Organization, we recommend that you enumerate each account ID in the ECR permissions policy. This approach aligns with the AWS security best practice of setting narrow permissions in IAM policies.

In addition to Lambda permissions, the user or role that creates the function must also have `BatchGetImage` and `GetDownloadUrlForLayer` permissions.

Function lifecycle

After you upload a new or updated container image, Lambda optimizes the image before the function can process invocations. The optimization process can take a few seconds. The function remains in the `Pending` state until the process completes, when the state transitions to `Active`. You can't invoke the function until it reaches the `Active` state.

If a function is not invoked for multiple weeks, Lambda reclaims its optimized version, and the function transitions to the `Inactive` state. To reactivate the function, you must invoke it. Lambda rejects the first invocation and the function enters the `Pending` state until Lambda re-optimizes the image. The function then returns to the `Active` state.

Lambda periodically fetches the associated container image from the Amazon ECR repository. If the corresponding container image no longer exists on Amazon ECR or permissions are revoked, the function enters the `Failed` state, and Lambda returns a failure for any function invocations.

You can use the Lambda API to get information about a function's state. For more information, see Lambda function states.

Configure Lambda function memory

Lambda allocates CPU power in proportion to the amount of memory configured. *Memory* is the amount of memory available to your Lambda function at runtime. You can increase or decrease the memory and CPU power allocated to your function using the **Memory** setting. You can configure memory between 128 MB and 10,240 MB in 1-MB increments. At 1,769 MB, a function has the equivalent of one vCPU (one vCPU-second of credits per second).

This page describes how and when to update the memory setting for a Lambda function.

Sections

- Determining the appropriate memory setting for a Lambda function
- Configuring function memory (console)
- Configuring function memory (AWS CLI)
- Configuring function memory (AWS SAM)
- Accepting function memory recommendations (console)

Determining the appropriate memory setting for a Lambda function

Memory is the principal lever for controlling the performance of a function. The default setting, 128 MB, is the lowest possible setting. We recommend that you only use 128 MB for simple Lambda functions, such as those that transform and route events to other AWS services. A higher memory allocation can improve performance for functions that use imported libraries, Lambda layers, Amazon Simple Storage Service (Amazon S3) or Amazon Elastic File System (Amazon EFS). Adding more memory proportionally increases the amount of CPU, increasing the overall computational power available. If a function is CPU, network or memory-bound, then increasing the memory setting can dramatically improve its performance.

To find the right memory configuration, monitor your functions with Amazon CloudWatch and set alarms if memory consumption is approaching the configured maximums. This can help identify memory-bound functions. For CPU-bound and IO-bound functions, monitoring the duration can also provide insight. In these cases, increasing the memory can help resolve the compute or network bottlenecks.

You can also consider using the open source AWS Lambda Power Tuning tool. This tool uses AWS Step Functions to run multiple concurrent versions of a Lambda function at different

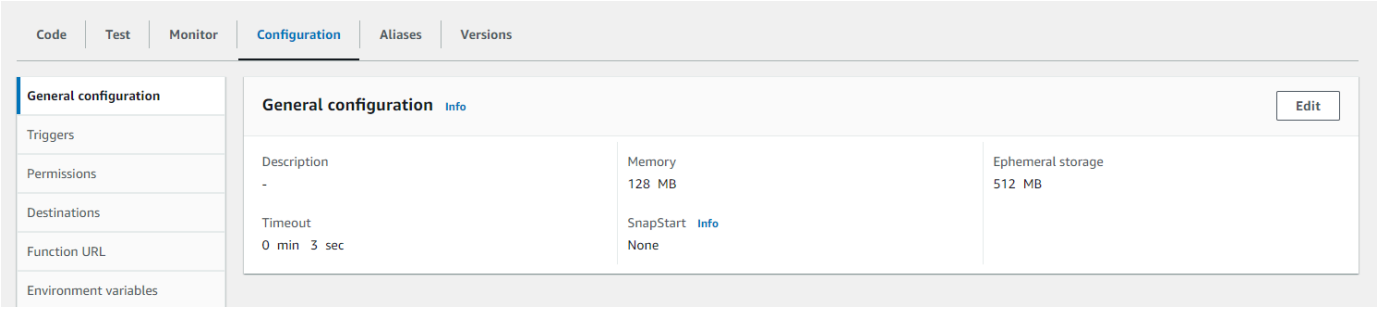
memory allocations and measure the performance. The input function runs in your AWS account, performing live HTTP calls and SDK interaction, to measure likely performance in a live production scenario. You can also implement a CI/CD process to use this tool to automatically measure the performance of new functions that you deploy.

Configuring function memory (console)

You can configure the memory of your function in the Lambda console.

To update the memory of a function

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the **Configuration** tab and then choose **General configuration**.



4. Under **General configuration**, choose **Edit**.
5. For **Memory**, set a value from 128 MB to 10,240 MB.
6. Choose **Save**.

Configuring function memory (AWS CLI)

You can use the update-function-configuration command to configure the memory of your function.

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --memory-size 1024
```

Configuring function memory (AWS SAM)

You can use the AWS Serverless Application Model to configure memory for your function. Update the MemorySize property in your `template.yaml` file and then run sam deploy.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 1024
      # Other function properties...
```

Accepting function memory recommendations (console)

If you have administrator permissions in AWS Identity and Access Management (IAM), you can opt in to receive Lambda function memory setting recommendations from AWS Compute Optimizer. For instructions on opting in to memory recommendations for your account or organization, see Opting in your account in the *AWS Compute Optimizer User Guide*.

> **Note**
>
> Compute Optimizer supports only functions that use x86_64 architecture.

When you've opted in and your Lambda function meets Compute Optimizer requirements, you can view and accept function memory recommendations from Compute Optimizer in the Lambda console in **General configuration**.

Configure ephemeral storage for Lambda functions

Lambda provides ephemeral storage for functions in the `/tmp` directory. This storage is temporary and unique to each execution environment. You can control the amount of ephemeral storage allocated to your function using the **Ephemeral storage** setting. You can configure ephemeral storage between 512 MB and 10,240 MB, in 1-MB increments. All data stored in `/tmp` is encrypted at rest with a key managed by AWS.

This page describes common use cases and how to update the ephemeral storage for a Lambda function.

Sections

- Common use cases for increased ephemeral storage
- Configuring ephemeral storage (console)
- Configuring ephemeral storage (AWS CLI)
- Configuring ephemeral storage (AWS SAM)

Common use cases for increased ephemeral storage

Here are several common use cases that benefit from increased ephemeral storage:

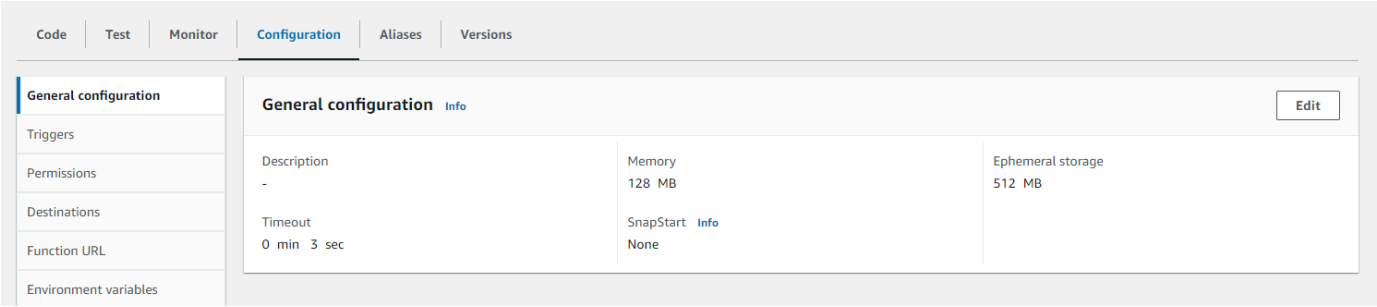
- **Extract-transform-load (ETL) jobs:** Increase ephemeral storage when your code performs intermediate computation or downloads other resources to complete processing. More temporary space enables more complex ETL jobs to run in Lambda functions.
- **Machine learning (ML) inference:** Many inference tasks rely on large reference data files, including libraries and models. With more ephemeral storage, you can download larger models from Amazon Simple Storage Service (Amazon S3) to `/tmp` and use them in your processing.
- **Data processing:** For workloads that download objects from Amazon S3 in response to S3 events, more `/tmp` space makes it possible to handle larger objects without using in-memory processing. Workloads that create PDFs or process media also benefit from more ephemeral storage.
- **Graphics processing:** Image processing is a common use case for Lambda-based applications. For workloads that process large TIFF files or satellite images, more ephemeral storage makes it easier to use libraries and perform the computation in Lambda.

Configuring ephemeral storage (console)

You can configure ephemeral storage in the Lambda console.

To modify ephemeral storage for a function

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the **Configuration** tab and then choose **General configuration**.



4. Under **General configuration**, choose **Edit**.
5. For **Ephemeral storage**, set a value between 512 MB and 10,240 MB, in 1-MB increments.
6. Choose **Save**.

Configuring ephemeral storage (AWS CLI)

You can use the update-function-configuration command to configure ephemeral storage.

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --ephemeral-storage '{"Size": 1024}'
```

Configuring ephemeral storage (AWS SAM)

You can use the AWS Serverless Application Model to configure ephemeral storage for your function. Update the EphemeralStorage property in your `template.yaml` file and then run sam deploy.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs22.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      # Other function properties...
```

Selecting and configuring an instruction set architecture for your Lambda function

The *instruction set architecture* of a Lambda function determines the type of computer processor that Lambda uses to run the function. Lambda provides a choice of instruction set architectures:

- arm64 – 64-bit ARM architecture, for the AWS Graviton2 processor.
- x86_64 – 64-bit x86 architecture, for x86-based processors.

> **Note**
>
> The arm64 architecture is available in most AWS Regions. For more information, see AWS Lambda Pricing. In the memory prices table, choose the **Arm Price** tab, and then open the **Region** dropdown list to see which AWS Regions support arm64 with Lambda.
> For an example of how to create a function with arm64 architecture, see AWS Lambda Functions Powered by AWS Graviton2 Processor.

Topics

- Advantages of using arm64 architecture
- Requirements for migration to arm64 architecture
- Function code compatibility with arm64 architecture
- How to migrate to arm64 architecture
- Configuring the instruction set architecture

Advantages of using arm64 architecture

Lambda functions that use arm64 architecture (AWS Graviton2 processor) can achieve significantly better price and performance than the equivalent function running on x86_64 architecture. Consider using arm64 for compute-intensive applications such as high-performance computing, video encoding, and simulation workloads.

The Graviton2 CPU uses the Neoverse N1 core and supports Armv8.2 (including CRC and crypto extensions) plus several other architectural extensions.

Graviton2 reduces memory read time by providing a larger L2 cache per vCPU, which improves the latency performance of web and mobile backends, microservices, and data processing systems. Graviton2 also provides improved encryption performance and supports instruction sets that improve the latency of CPU-based machine learning inference.

For more information about AWS Graviton2, see AWS Graviton Processor.

Requirements for migration to arm64 architecture

When you select a Lambda function to migrate to arm64 architecture, to ensure a smooth migration, make sure that your function meets the following requirements:

- The deployment package contains only open-source components and source code that you control, so that you can make any necessary updates for the migration.
- If the function code includes third-party dependencies, each library or package provides an arm64 version.

Function code compatibility with arm64 architecture

Your Lambda function code must be compatible with the instruction set architecture of the function. Before you migrate a function to arm64 architecture, note the following points about the current function code:

- If you added your function code using the embedded code editor, your code probably runs on either architecture without modification.
- If you uploaded your function code, you must upload new code that is compatible with your target architecture.
- If your function uses layers, you must check each layer to ensure that it is compatible with the new architecture. If a layer is not compatible, edit the function to replace the current layer version with a compatible layer version.
- If your function uses Lambda extensions, you must check each extension to ensure that it is compatible with the new architecture.
- If your function uses a container image deployment package type, you must create a new container image that is compatible with the architecture of the function.

How to migrate to arm64 architecture

To migrate a Lambda function to the arm64 architecture, we recommend following these steps:

1. Build the list of dependencies for your application or workload. Common dependencies include:
   - All the libraries and packages that the function uses.
   - The tools that you use to build, deploy, and test the function, such as compilers, test suites, continuous integration and continuous delivery (CI/CD) pipelines, provisioning tools, and scripts.
   - The Lambda extensions and third-party tools that you use to monitor the function in production.
2. For each of the dependencies, check the version, and then check whether arm64 versions are available.
3. Build an environment to migrate your application.
4. Bootstrap the application.
5. Test and debug the application.
6. Test the performance of the arm64 function. Compare the performance with the x86_64 version.
7. Update your infrastructure pipeline to support arm64 Lambda functions.
8. Stage your deployment to production.

   For example, use alias routing configuration to split traffic between the x86 and arm64 versions of the function, and compare the performance and latency.

For more information about how to create a code environment for arm64 architecture, including language-specific information for Java, Go, .NET, and Python, see the Getting started with AWS Graviton GitHub repository.

Configuring the instruction set architecture

You can configure the instruction set architecture for new and existing Lambda functions using the Lambda console, AWS SDKs, AWS Command Line Interface (AWS CLI), or CloudFormation. Follow these steps to change the instruction set architecture for an existing Lambda function from the console.

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to configure the instruction set architecture for.

3. On the main **Code** tab, for the **Runtime settings** section, choose **Edit**.
4. Under **Architecture**, choose the instruction set architecture you want your function to use.
5. Choose **Save**.

Configure Lambda function timeout

Lambda runs your code for a set amount of time before timing out. *Timeout* is the maximum amount of time in seconds that a Lambda function can run. The default value for this setting is 3 seconds, but you can adjust this in increments of 1 second up to a maximum value of 900 seconds (15 minutes).

This page describes how and when to update the timeout setting for a Lambda function.

Sections

- Determining the appropriate timeout value for a Lambda function
- Configuring timeout (console)
- Configuring timeout (AWS CLI)
- Configuring timeout (AWS SAM)

Determining the appropriate timeout value for a Lambda function

If the timeout value is close to the average duration of a function, there is a higher risk that the function will time out unexpectedly. The duration of a function can vary based on the amount of data transfer and processing, and the latency of any services the function interacts with. Some common causes of timeout include:

- Downloads from Amazon Simple Storage Service (Amazon S3) are larger or take longer than average.
- A function makes a request to another service, which takes longer to respond.
- The parameters provided to a function require more computational complexity in the function, which causes the invocation to take longer.

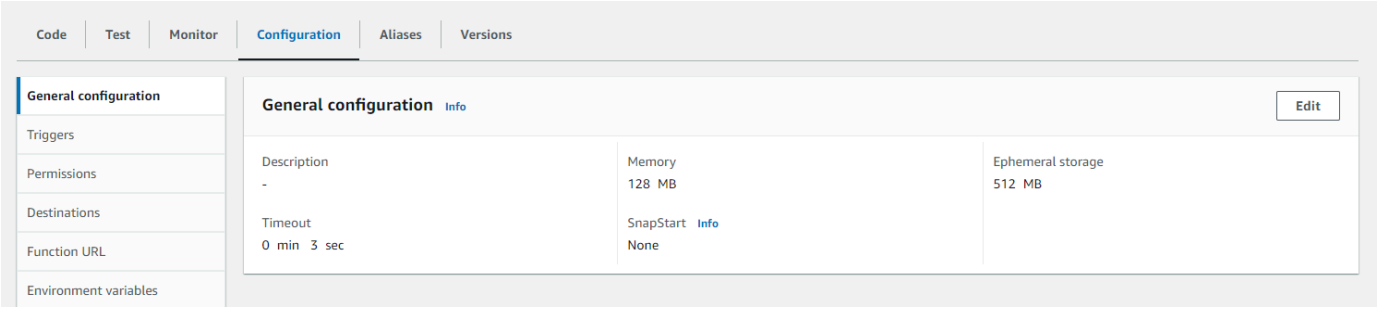
When testing your application, ensure that your tests accurately reflect the size and quantity of data and realistic parameter values. Tests often use small samples for convenience, but you should use datasets at the upper bounds of what is reasonably expected for your workload.

Configuring timeout (console)

You can configure function timeout in the Lambda console.

To modify the timeout for a function

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the **Configuration** tab and then choose **General configuration**.



4. Under **General configuration**, choose **Edit**.
5. For **Timeout**, set a value between 1 and 900 seconds (15 minutes).
6. Choose **Save**.

Configuring timeout (AWS CLI)

You can use the update-function-configuration command to configure the timeout value, in seconds. The following example command increases the function timeout to 120 seconds (2 minutes).

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --timeout 120
```

Configuring timeout (AWS SAM)

You can use the AWS Serverless Application Model to configure the timeout value for your function. Update the Timeout property in your `template.yaml` file and then run sam deploy.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
```

```
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      # Other function properties...
```

Working with Lambda environment variables

You can use environment variables to adjust your function's behavior without updating code. An environment variable is a pair of strings that is stored in a function's version-specific configuration. The Lambda runtime makes environment variables available to your code and sets additional environment variables that contain information about the function and invocation request.

> **Note**
>
> To increase security, we recommend that you use AWS Secrets Manager instead of environment variables to store database credentials and other sensitive information like API keys or authorization tokens. For more information, see Use Secrets Manager secrets in Lambda functions.

Environment variables are not evaluated before the function invocation. Any value you define is considered a literal string and not expanded. Perform the variable evaluation in your function code.

Creating Lambda environment variables

You can configure environment variables in Lambda using the Lambda console, the AWS Command Line Interface (AWS CLI), AWS Serverless Application Model (AWS SAM), or using an AWS SDK.

Console

You define environment variables on the unpublished version of your function. When you publish a version, the environment variables are locked for that version along with other version-specific configuration settings.

You create an environment variable for your function by defining a key and a value. Your function uses the name of the key to retrieve the value of the environment variable.

To set environment variables in the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the **Configuration** tab, then choose **Environment variables**.
4. Under **Environment variables**, choose **Edit**.
5. Choose **Add environment variable**.

6. Enter a key and value.

   Requirements

   - Keys start with a letter and are at least two characters.
   - Keys only contain letters, numbers, and the underscore character (_).
   - Keys aren't reserved by Lambda.
   - The total size of all environment variables doesn't exceed 4 KB.

7. Choose **Save**.

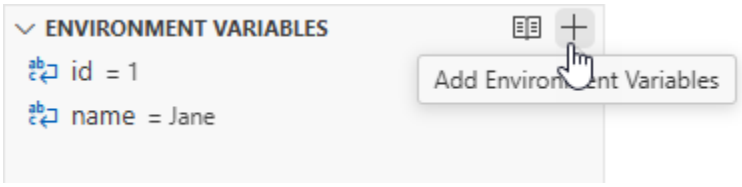
To generate a list of environment variables in the console code editor

You can generate a list of environment variables in the Lambda code editor. This is a quick way to reference your environment variables while you code.

1. Choose the **Code** tab.
2. Scroll down to the **ENVIRONMENT VARIABLES** section of the code editor. Existing environment variables are listed here:



3. To create new environment variables, choose the choose the plus sign (+):



Environment variables remain encrypted when listed in the console code editor. If you enabled encryption helpers for encryption in transit, then those settings remain unchanged. For more information, see Securing Lambda environment variables.

The environment variables list is read-only and is available only on the Lambda console. This file is not included when you download the function's .zip file archive, and you can't add environment variables by uploading this file.

AWS CLI

The following example sets two environment variables on a function named `my-function`.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --environment "Variables={BUCKET=amzn-s3-demo-bucket,KEY=file.txt}"
```

When you apply environment variables with the `update-function-configuration` command, the entire contents of the `Variables` structure is replaced. To retain existing environment variables when you add a new one, include all existing values in your request.

To get the current configuration, use the `get-function-configuration` command.

```
aws lambda get-function-configuration \
  --function-name my-function
```

You should see the following output:

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::111122223333:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    "RevisionId": "0894d3c1-2a3d-4d48-bf7f-abade99f3c15",
    ...
}
```

You can pass the revision ID from the output of `get-function-configuration` as a parameter to `update-function-configuration`. This ensures that the values don't change between when you read the configuration and when you update it.

To configure a function's encryption key, set the `KMSKeyARN` option.

```
aws lambda update-function-configuration \
```

```
  --function-name my-function \
  --kms-key-arn arn:aws:kms:us-east-2:111122223333:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

AWS SAM

You can use the AWS Serverless Application Model to configure environment variables for your function. Update the Environment and Variables properties in your `template.yaml` file and then run sam deploy.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs24.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      Environment:
        Variables:
          BUCKET: amzn-s3-demo-bucket
          KEY: file.txt
      # Other function properties...
```

AWS SDKs

To manage environment variables using an AWS SDK, use the following API operations.

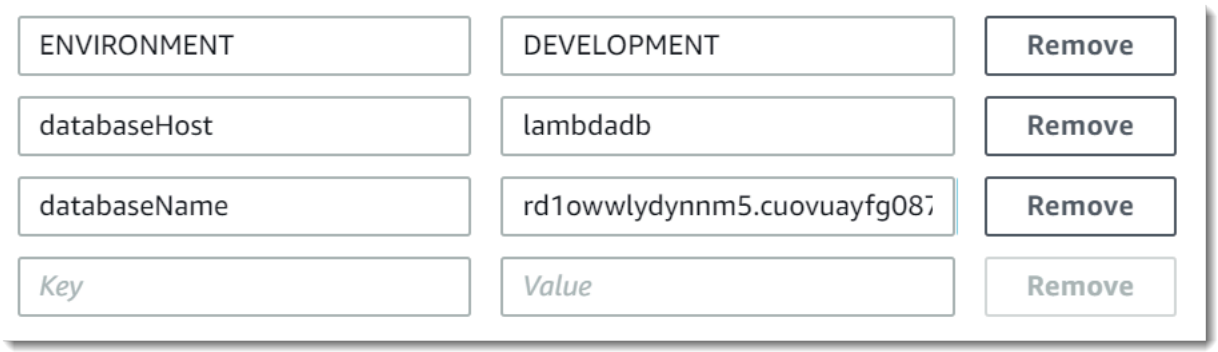
- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

To learn more, refer to the AWS SDK documentation for your preferred programming language.

Example scenario for environment variables

You can use environment variables to customize function behavior in your test environment and production environment. For example, you can create two functions with the same code but different configurations. One function connects to a test database, and the other connects to a production database. In this situation, you use environment variables to pass the hostname and other connection details for the database to the function.

The following example shows how to define the database host and database name as environment variables.



If you want your test environment to generate more debug information than the production environment, you could set an environment variable to configure your test environment to use more verbose logging or more detailed tracing.

For example, in your test environment, you could set an environment variable with the key `LOG_LEVEL` and a value indicating a log level of debug or trace. In your Lambda function's code, you can then use this environment variable to set the log level.

The following code examples in Python and Node.js illustrate how you can achieve this. These examples assume your environment variable has a value of `DEBUG` in Python or `debug` in Node.js.

Python

Example Python code to set log level

```
import os
import logging
```

```
# Initialize the logger
logger = logging.getLogger()

# Get the log level from the environment variable and default to INFO if not set
log_level = os.environ.get('LOG_LEVEL', 'INFO')

# Set the log level
logger.setLevel(log_level)

def lambda_handler(event, context):
    # Produce some example log outputs
    logger.debug('This is a log with detailed debug information - shown only in test
 environment')
    logger.info('This is a log with standard information - shown in production and
 test environments')
```

Node.js (ES module format)

Example Node.js code to set log level

This example uses the `winston` logging library. Use npm to add this library to your function's deployment package. For more information, see the section called “Creating a .zip deployment package with dependencies”.

```
import winston from 'winston';

// Initialize the logger using the log level from environment variables, defaulting
 to INFO if not set
const logger = winston.createLogger({
   level: process.env.LOG_LEVEL || 'info',
   format: winston.format.json(),
   transports: [new winston.transports.Console()]
});

export const handler = async (event) => {
   // Produce some example log outputs
   logger.debug('This is a log with detailed debug information - shown only in test
 environment');
   logger.info('This is a log with standard information - shown in production and
 test environment');

};
```

Retrieving Lambda environment variables

To retrieve environment variables in your function code, use the standard method for your programming language.

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
  region = os.environ['AWS_REGION']
```

> **Note**
>
> In some cases, you may need to use the following format:
>
> ```
> region = os.environ.get('AWS_REGION')
> ```

Ruby

```
region = ENV["AWS_REGION"]
```

Java

```
String region = System.getenv("AWS_REGION");
```

Go

```
var region = os.Getenv("AWS_REGION")
```

C#

```
string region = Environment.GetEnvironmentVariable("AWS_REGION");
```

PowerShell

```
$region = $env:AWS_REGION
```

Lambda stores environment variables securely by encrypting them at rest. You can configure Lambda to use a different encryption key, encrypt environment variable values on the client side, or set environment variables in an CloudFormation template with AWS Secrets Manager.

Defined runtime environment variables

Lambda runtimes set several environment variables during initialization. Most of the environment variables provide information about the function or runtime. The keys for these environment variables are *reserved* and cannot be set in your function configuration.

Reserved environment variables

- `_HANDLER` – The handler location configured on the function.
- `_X_AMZN_TRACE_ID` – The X-Ray tracing header. This environment variable changes with each invocation.
  - This environment variable is not defined for OS-only runtimes (the `provided` runtime family). You can set `_X_AMZN_TRACE_ID` for custom runtimes using the `Lambda-Runtime-Trace-Id` response header from the Next invocation.
  - For Java runtime versions 17 and later, this environment variable is not used. Instead, Lambda stores tracing information in the `com.amazonaws.xray.traceHeader` system property.
- `AWS_DEFAULT_REGION` – The default AWS Region where the Lambda function is executed.
- `AWS_REGION` – The AWS Region where the Lambda function is executed. If defined, this value overrides the `AWS_DEFAULT_REGION`.
  - For more information about using the AWS Region environment variables with AWS SDKs, see AWS Region in the *AWS SDKs and Tools Reference Guide*.
- `AWS_EXECUTION_ENV` – The runtime identifier, prefixed by `AWS_Lambda_` (for example, `AWS_Lambda_java8`). This environment variable is not defined for OS-only runtimes (the `provided` runtime family).
- `AWS_LAMBDA_FUNCTION_NAME` – The name of the function.
- `AWS_LAMBDA_FUNCTION_MEMORY_SIZE` – The amount of memory available to the function in MB.

- `AWS_LAMBDA_FUNCTION_VERSION` – The version of the function being executed.
- `AWS_LAMBDA_INITIALIZATION_TYPE` – The initialization type of the function, which is `on-demand`, `provisioned-concurrency`, `snap-start`, or `lambda-managed-instances`. For information, see Configuring provisioned concurrency, Improving startup performance with Lambda SnapStart, or Lambda Managed Instances.
- `AWS_LAMBDA_LOG_GROUP_NAME`, `AWS_LAMBDA_LOG_STREAM_NAME` – The name of the Amazon CloudWatch Logs group and stream for the function. The `AWS_LAMBDA_LOG_GROUP_NAME` and `AWS_LAMBDA_LOG_STREAM_NAME` environment variables are not available in Lambda SnapStart functions.
- `AWS_ACCESS_KEY`, `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN` – The access keys obtained from the function's execution role.
- `AWS_LAMBDA_RUNTIME_API` – (Custom runtime) The host and port of the runtime API.
- `LAMBDA_TASK_ROOT` – The path to your Lambda function code.
- `LAMBDA_RUNTIME_DIR` – The path to runtime libraries.
- `AWS_LAMBDA_MAX_CONCURRENCY` – (Lambda Managed Instances only) The maximum number of concurrent invocations Lambda will send to one execution environment.
- `AWS_LAMBDA_METADATA_API` – The metadata endpoint server address in the format `{ipv4_address}:{port}` (for example, `169.254.100.1:9001`).
- `AWS_LAMBDA_METADATA_TOKEN` – A unique authentication token for the current execution environment used to authenticate requests to the metadata endpoint. Lambda generates this token automatically at initialization.

The following additional environment variables aren't reserved and can be extended in your function configuration.

Unreserved environment variables

- `LANG` – The locale of the runtime (`en_US.UTF-8`).
- `PATH` – The execution path (`/usr/local/bin:/usr/bin/:/bin:/opt/bin`).
- `LD_LIBRARY_PATH` – The system library path (`/var/lang/lib:/lib64:/usr/lib64:$LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:$LAMBDA_TASK_ROOT/lib:/opt/lib`).
- `NODE_PATH` – (Node.js) The Node.js library path (`/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules`).

- `NODE_OPTIONS` – (Node.js) For Node.js runtimes, you can use `NODE_OPTIONS` to re-enable experimental features that Lambda disables by default.
- `PYTHONPATH` – (Python) The Python library path (`$LAMBDA_RUNTIME_DIR`).
- `GEM_PATH` – (Ruby) The Ruby library path (`$LAMBDA_TASK_ROOT/vendor/bundle/ruby/3.3.0:/opt/ruby/gems/3.3.0`).
- `AWS_XRAY_CONTEXT_MISSING` – For X-Ray tracing, Lambda sets this to `LOG_ERROR` to avoid throwing runtime errors from the X-Ray SDK.
- `AWS_XRAY_DAEMON_ADDRESS` – For X-Ray tracing, the IP address and port of the X-Ray daemon.
- `AWS_LAMBDA_DOTNET_PREJIT` – (.NET) Set this variable to enable or disable .NET specific runtime optimizations. Values include `always`, `never`, and `provisioned-concurrency`. For more information, see Configuring provisioned concurrency for a function.
- `TZ` – The environment's time zone (`:UTC`). The execution environment uses NTP to synchronize the system clock.

The sample values shown reflect the latest runtimes. The presence of specific variables or their values can vary on earlier runtimes.

Securing Lambda environment variables

For securing your environment variables, you can use server-side encryption to protect your data at rest and client-side encryption to protect your data in transit.

> **Note**
>
> To increase database security, we recommend that you use AWS Secrets Manager instead of environment variables to store database credentials. For more information, see Use Secrets Manager secrets in Lambda functions.

Security at rest

Lambda always provides server-side encryption at rest with an AWS KMS key. By default, Lambda uses an AWS managed key. If this default behavior suits your workflow, you don't need to set up anything else. Lambda creates the AWS managed key in your account and manages the permissions for you. AWS doesn't charge you to use this key.

If you prefer, you can provide an AWS KMS customer managed key instead. You might do this to have control over rotation of the KMS key or to meet the requirements of your organization for managing KMS keys. When you use a customer managed key, only users in your account with access to the KMS key can view or manage environment variables on the function.

Customer managed keys incur standard AWS KMS charges. For more information, see AWS Key Management Service pricing.

Security in transit

For additional security, you can enable helpers for encryption in transit, which ensures that your environment variables are encrypted client-side for protection in transit.

To configure encryption for your environment variables

1. Use the AWS Key Management Service (AWS KMS) to create any customer managed keys for Lambda to use for server-side and client-side encryption. For more information, see Creating keys in the *AWS Key Management Service Developer Guide*.
2. Using the Lambda console, navigate to the **Edit environment variables** page.
   a. Open the Functions page of the Lambda console.
   b. Choose a function.
   c. Choose **Configuration**, then choose **Environment variables** from the left navigation bar.
   d. In the **Environment variables** section, choose **Edit**.
   e. Expand **Encryption configuration**.
3. (Optional) Enable console encryption helpers to use client-side encryption to protect your data in transit.
   a. Under **Encryption in transit**, choose **Enable helpers for encryption in transit**.
   b. For each environment variable that you want to enable console encryption helpers for, choose **Encrypt** next to the environment variable.
   c. Under AWS KMS key to encrypt in transit, choose a customer managed key that you created at the beginning of this procedure.
   d. Choose **Execution role policy** and copy the policy. This policy grants permission to your function's execution role to decrypt the environment variables.

      Save this policy to use in the last step of this procedure.

e. Add code to your function that decrypts the environment variables. To see an example, choose **Decrypt secrets snippet**.

4. (Optional) Specify your customer managed key for encryption at rest.

   a. Choose **Use a customer master key**.

   b. Choose a customer managed key that you created at the beginning of this procedure.

5. Choose **Save**.

6. Set up permissions.

   If you're using a customer managed key with server-side encryption, grant permissions to any users or roles that you want to be able to view or manage environment variables on the function. For more information, see Managing permissions to your server-side encryption KMS key.

   If you're enabling client-side encryption for security in transit, your function needs permission to call the `kms:Decrypt` API operation. Add the policy that you saved previously in this procedure to the function's execution role.

Managing permissions to your server-side encryption KMS key

No AWS KMS permissions are required for your user or the function's execution role to use the default encryption key. To use a customer managed key, you need permission to use the key. Lambda uses your permissions to create a grant on the key. This allows Lambda to use it for encryption.

- `kms:ListAliases` – To view keys in the Lambda console.
- `kms:CreateGrant`, `kms:Encrypt` – To configure a customer managed key on a function.
- `kms:Decrypt` – To view and manage environment variables that are encrypted with a customer managed key.

You can get these permissions from your AWS account or from a key's resource-based permissions policy. `ListAliases` is provided by the managed policies for Lambda. Key policies grant the remaining permissions to users in the **Key users** group.

Users without `Decrypt` permissions can still manage functions, but they can't view environment variables or manage them in the Lambda console. To prevent a user from viewing environment

variables, add a statement to the user's permissions that denies access to the default key, a customer managed key, or all keys.

Example IAM policy – Deny access by key ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Deny",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-2:111122223333:key/3be10e2d-
xmpl-4be4-bc9d-0405a71945cc"
        }
    ]
}
```

For details on managing key permissions, see Key policies in AWS KMS in the *AWS Key Management Service Developer Guide*.

Giving Lambda functions access to resources in an Amazon VPC

With Amazon Virtual Private Cloud (Amazon VPC), you can create private networks in your AWS account to host resources such as Amazon Elastic Compute Cloud (Amazon EC2) instances, Amazon Relational Database Service (Amazon RDS) instances, and Amazon ElastiCache instances. You can give your Lambda function access to resources hosted in an Amazon VPC by attaching your function to the VPC through the private subnets that contain the resources. Follow the instructions in the following sections to attach a Lambda function to an Amazon VPC using the Lambda console, the AWS Command Line Interface (AWS CLI), or AWS SAM.

> **Note**
>
> Every Lambda function runs inside a VPC that is owned and managed by the Lambda service. These VPCs are maintained automatically by Lambda and are not visible to customers. Configuring your function to access other AWS resources in an Amazon VPC has no effect on the Lambda-managed VPC your function runs inside.

Sections

- Required IAM permissions
- Attaching Lambda functions to an Amazon VPC in your AWS account
- Internet access when attached to a VPC
- IPv6 support
- Best practices for using Lambda with Amazon VPCs
- Understanding Hyperplane Elastic Network Interfaces (ENIs)
- Using IAM condition keys for VPC settings
- VPC tutorials

Required IAM permissions

To attach a Lambda function to an Amazon VPC in your AWS account, Lambda needs permissions to create and manage the network interfaces it uses to give your function access to the resources in the VPC.

The network interfaces that Lambda creates are known as Hyperplane Elastic Network Interfaces, or Hyperplane ENIs. To learn more about these network interfaces, see the section called "Understanding Hyperplane Elastic Network Interfaces (ENIs)".

You can give your function the permissions it needs by attaching the AWS managed policy AWSLambdaVPCAccessExecutionRole to your function's execution role. When you create a new function in the Lambda console and attach it to a VPC, Lambda automatically adds this permissions policy for you.

If you prefer to create your own IAM permissions policy, make sure to add all of the following permissions and allow them on all resources (`"Resource": "*"`):

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeSubnets
- ec2:DeleteNetworkInterface
- ec2:AssignPrivateIpAddresses
- ec2:UnassignPrivateIpAddresses

Note that your function's role only needs these permissions to create the network interfaces, not to invoke your function. You can still invoke your function successfully when it's attached to an Amazon VPC, even if you remove these permissions from your function's execution role.

To attach your function to a VPC, Lambda also needs to verify network resources using your IAM user role. Ensure that your user role has the following IAM permissions:

- **ec2:DescribeSecurityGroups**
- **ec2:DescribeSubnets**
- **ec2:DescribeVpcs**
- **ec2:GetSecurityGroupsForVpc**

> **Note**
>
> The Amazon EC2 permissions that you grant to your function's execution role are used by the Lambda service to attach your function to a VPC. However, you're also implicitly

granting these permissions to your function's code. This means that your function code is able to make these Amazon EC2 API calls. For advice on following security best practices, see the section called “Security best practices”.

Attaching Lambda functions to an Amazon VPC in your AWS account

Attach your function to an Amazon VPC in your AWS account by using the Lambda console, the AWS CLI or AWS SAM. If you're using the AWS CLI or AWS SAM, or attaching an existing function to a VPC using the Lambda console, make sure that your function's execution role has the necessary permissions listed in the previous section.

Lambda functions can't connect directly to a VPC with dedicated instance tenancy. To connect to resources in a dedicated VPC, peer it to a second VPC with default tenancy.

Lambda console

To attach a function to an Amazon VPC when you create it

1. Open the Functions page of the Lambda console and choose **Create function**.
2. Under **Basic information**, for **Function name**, enter a name for your function.
3. Configure VPC settings for the function by doing the following:
   a. Expand **Advanced settings**.
   b. Select **Enable VPC**, and then select the VPC you want to attach the function to.
   c. (Optional) To allow outbound IPv6 traffic, select **Allow IPv6 traffic for dual-stack subnets**.
   d. Choose the subnets and security groups to create the network interface for. If you selected **Allow IPv6 traffic for dual-stack subnets**, all selected subnets must have an IPv4 CIDR block and an IPv6 CIDR block.

> **Note**
>
> To access private resources, connect your function to private subnets. If your function needs internet access, see the section called “Internet access for VPC functions”. Connecting a function to a public subnet doesn't give it internet access or a public IP address.

4. Choose **Create function**.

To attach an existing function to an Amazon VPC

1. Open the Functions page of the Lambda console and select your function.
2. Choose the **Configuration** tab, then choose **VPC**.
3. Choose **Edit**.
4. Under **VPC**, select the Amazon VPC you want to attach your function to.
5. (Optional) To allow outbound IPv6 traffic, select **Allow IPv6 traffic for dual-stack subnets**.
6. Choose the subnets and security groups to create the network interface for. If you selected **Allow IPv6 traffic for dual-stack subnets**, all selected subnets must have an IPv4 CIDR block and an IPv6 CIDR block.

> **Note**
>
> To access private resources, connect your function to private subnets. If your function needs internet access, see the section called "Internet access for VPC functions". Connecting a function to a public subnet doesn't give it internet access or a public IP address.

7. Choose **Save**.

AWS CLI

To attach a function to an Amazon VPC when you create it

- To create a Lambda function and attach it to a VPC, run the following CLI `create-function` command.

```
aws lambda create-function --function-name my-function \
--runtime nodejs24.x --handler index.js --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/lambda-role \
--vpc-config
 Ipv6AllowedForDualStack=true,SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a03
```

Specify your own subnets and security groups and set `Ipv6AllowedForDualStack` to `true` or `false` according to your use case.

To attach an existing function to an Amazon VPC

- To attach an existing function to a VPC, run the following CLI `update-function-configuration` command.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-0859123
```

To unattach your function from a VPC

- To unattach your function from a VPC, run the following `update-function-configuration`CLI command with an empty list of VPC subnets and security groups.

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

To attach your function to a VPC

- To attach a Lambda function to an Amazon VPC, add the `VpcConfig` property to your function definition as shown in the following example template. For more information about this property, see AWS::Lambda::Function VpcConfig in the *CloudFormation User Guide* (the AWS SAM `VpcConfig` property is passed directly to the `VpcConfig` property of an CloudFormation `AWS::Lambda::Function` resource).

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./lambda_function/
      Handler: lambda_function.handler
      Runtime: python3.12
      VpcConfig:
        SecurityGroupIds:
```

```
          - !Ref MySecurityGroup
        SubnetIds:
          - !Ref MySubnet1
          - !Ref MySubnet2
      Policies:
        - AWSLambdaVPCAccessExecutionRole

MySecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
    GroupDescription: Security group for Lambda function
    VpcId: !Ref MyVPC

MySubnet1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
    CidrBlock: 10.0.1.0/24

MySubnet2:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
    CidrBlock: 10.0.2.0/24

MyVPC:
  Type: AWS::EC2::VPC
  Properties:
    CidrBlock: 10.0.0.0/16
```

For more information about configuring your VPC in AWS SAM, see AWS::EC2::VPC in the *CloudFormation User Guide*.

Internet access when attached to a VPC

By default, Lambda functions have access to the public internet. When you attach your function to a VPC, it can only access resources available within that VPC. To give your function access to the internet, you also need to configure the VPC to have internet access. To learn more, see the section called “Internet access for VPC functions”.

IPv6 support

Your function can connect to resources in dual-stack VPC subnets over IPv6. This option is turned off by default. To allow outbound IPv6 traffic, use the console or the `--vpc-config Ipv6AllowedForDualStack=true` option with the create-function or update-function-configuration command.

> **Note**
>
> To allow outbound IPv6 traffic in a VPC, all of the subnets that are connected to the function must be dual-stack subnets. Lambda doesn't support outbound IPv6 connections for IPv6-only subnets in a VPC or outbound IPv6 connections for functions that are not connected to a VPC.

You can update your function code to explicitly connect to subnet resources over IPv6. The following Python example opens a socket and connects to an IPv6 server.

Example— Connect to IPv6 server

```
def connect_to_server(event, context):
    server_address = event['host']
    server_port = event['port']
    message = event['message']
    run_connect_to_server(server_address, server_port, message)

def run_connect_to_server(server_address, server_port, message):
    sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
    try:
        # Send data
        sock.connect((server_address, int(server_port), 0, 0))
        sock.sendall(message.encode())
        BUFF_SIZE = 4096
        data = b''
        while True:
            segment = sock.recv(BUFF_SIZE)
            data += segment
            # Either 0 or end of data
            if len(segment) < BUFF_SIZE:
                break
        return data
```

```
    finally:
        sock.close()
```

Best practices for using Lambda with Amazon VPCs

To ensure that your Lambda VPC configuration meets best practice guidelines, follow the advice in the following sections.

Security best practices

To attach your Lambda function to a VPC, you need to give your function's execution role a number of Amazon EC2 permissions. These permissions are required to create the network interfaces your function uses to access the resources in the VPC. However, these permissions are also implicitly granted to your function's code. This means that your function code has permission to make these Amazon EC2 API calls.

To follow the principle of least-privilege access, add a deny policy like the following example to your function's execution role. This policy prevents your function code from making calls to the Amazon EC2 APIs, while still allowing the Lambda service to manage VPC resources on your behalf. The policy uses the `lambda:SourceFunctionArn` condition key, which only applies to API calls made by your function code during execution. For more information, see the section called "Source function ARN".

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                 "ec2:CreateNetworkInterface",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeSubnets",
                 "ec2:DetachNetworkInterface",
                 "ec2:AssignPrivateIpAddresses",
                 "ec2:UnassignPrivateIpAddresses"
            ],
            "Resource": [ "*" ],
            "Condition": {
```

```
                "ArnEquals": {
                    "lambda:SourceFunctionArn": [
                        "arn:aws:lambda:us-
west-2:123456789012:function:my_function"
                    ]
                }
            }
        }
    ]
}
```

AWS provides *security groups* and *network Access Control Lists (ACLs)* to increase security in your VPC. Security groups control inbound and outbound traffic for your resources, and network ACLs control inbound and outbound traffic for your subnets. Security groups provide enough access control for most subnets. You can use network ACLs if you want an additional layer of security for your VPC. For general guidelines on security best practices when using Amazon VPCs, see Security best practices for your VPC in the *Amazon Virtual Private Cloud User Guide*.

Performance best practices

When you attach your function to a VPC, Lambda checks to see if there is an available network resource (Hyperplane ENI) it can use to connect to. Hyperplane ENIs are associated with a particular combination of security groups and VPC subnets. If you've already attached one function to a VPC, specifying the same subnets and security groups when you attach another function means that Lambda can share the network resources and avoid the need to create a new Hyperplane ENI. For more information about Hyperplane ENIs and their lifecycle, see the section called "Understanding Hyperplane Elastic Network Interfaces (ENIs)".

Understanding Hyperplane Elastic Network Interfaces (ENIs)

A Hyperplane ENI is a managed resource that acts as a network interface between your Lambda function and the resources you want your function to connect to. The Lambda service creates and manages these ENIs automatically when you attach your function to a VPC.

Hyperplane ENIs are not directly visible to you, and you don't need to configure or manage them. However, knowing how they work can help you to understand your function's behavior when you attach it to a VPC.

The first time you attach a function to a VPC using a particular subnet and security group combination, Lambda creates a Hyperplane ENI. Other functions in your account that use the

same subnet and security group combination can also use this ENI. Wherever possible, Lambda reuses existing ENIs to optimize resource utilization and minimize the creation of new ENIs. Each Hyperplane ENI supports up to 65,000 connections/ports. If the number of connections exceeds this limit, Lambda scales the number of ENIs automatically based on network traffic and concurrency requirements.

For new functions, while Lambda is creating a Hyperplane ENI, your function remains in the Pending state and you can't invoke it. Your function transitions to the Active state only when the Hyperplane ENI is ready, which can take several minutes. For existing functions, you can't perform additional operations that target the function, such as creating versions or updating the function's code, but you can continue to invoke previous versions of the function.

As part of managing the ENI lifecycle, Lambda may delete and recreate ENIs to load balance network traffic across ENIs or to address issues found in ENI health-checks. Additionally, if a Lambda function remains idle for 14 days, Lambda reclaims any unused Hyperplane ENIs and sets the function state to `Inactive`. The next invocation attempt will fail, and the function re-enters the Pending state until Lambda completes the creation or allocation of a Hyperplane ENI. We recommend that your design doesn't rely on the persistence of ENIs.

When you update a function to remove its VPC configuration, Lambda requires up to 20 minutes to delete the attached Hyperplane ENI. Lambda only deletes the ENI if no other function (or published function version) is using that Hyperplane ENI.

Lambda relies on permissions in the function execution role to delete the Hyperplane ENI. If you delete the execution role before Lambda deletes the Hyperplane ENI, Lambda won't be able to delete the Hyperplane ENI. You can manually perform the deletion.

Using IAM condition keys for VPC settings

You can use Lambda-specific condition keys for VPC settings to provide additional permission controls for your Lambda functions. For example, you can require that all functions in your organization are connected to a VPC. You can also specify the subnets and security groups that the function's users can and can't use.

Lambda supports the following condition keys in IAM policies:

- **lambda:VpcIds** – Allow or deny one or more VPCs.
- **lambda:SubnetIds** – Allow or deny one or more subnets.
- **lambda:SecurityGroupIds** – Allow or deny one or more security groups.

The Lambda API operations CreateFunction and UpdateFunctionConfiguration support these condition keys. For more information about using condition keys in IAM policies, see IAM JSON Policy Elements: Condition in the *IAM User Guide*.

> **Tip**
>
> If your function already includes a VPC configuration from a previous API request, you can send an `UpdateFunctionConfiguration` request without the VPC configuration.

Example policies with condition keys for VPC settings

The following examples demonstrate how to use condition keys for VPC settings. After you create a policy statement with the desired restrictions, append the policy statement for the target user or role.

Ensure that users deploy only VPC-connected functions

To ensure that all users deploy only VPC-connected functions, you can deny function create and update operations that don't include a valid VPC ID.

Note that VPC ID is not an input parameter to the `CreateFunction` or `UpdateFunctionConfiguration` request. Lambda retrieves the VPC ID value based on the subnet and security group parameters.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
```

```
        }
      }
    }
  ]
}
```

Deny users access to specific VPCs, subnets, or security groups

To deny users access to specific VPCs, use `StringEquals` to check the value of the `lambda:VpcIds` condition. The following example denies users access to `vpc-1` and `vpc-2`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceOutOfVPC",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Deny",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

To deny users access to specific subnets, use `StringEquals` to check the value of the `lambda:SubnetIds` condition. The following example denies users access to `subnet-1` and `subnet-2`.

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

To deny users access to specific security groups, use `StringEquals` to check the value of the `lambda:SecurityGroupIds` condition. The following example denies users access to `sg-1` and `sg-2`.

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

Allow users to create and update functions with specific VPC settings

To allow users to access specific VPCs, use `StringEquals` to check the value of the `lambda:VpcIds` condition. The following example allows users to access `vpc-1` and `vpc-2`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceStayInSpecificVpc",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Allow",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

To allow users to access specific subnets, use `StringEquals` to check the value of the `lambda:SubnetIds` condition. The following example allows users to access `subnet-1` and `subnet-2`.

```
{
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
```

```
      }
    }
```

To allow users to access specific security groups, use `StringEquals` to check the value of the `lambda:SecurityGroupIds` condition. The following example allows users to access `sg-1` and `sg-2`.

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

VPC tutorials

In the following tutorials, you connect a Lambda function to resources in your VPC.

- Tutorial: Using a Lambda function to access Amazon RDS in an Amazon VPC
- Tutorial: Configuring a Lambda function to access Amazon ElastiCache in an Amazon VPC

Giving Lambda functions access to a resource in an Amazon VPC in another account

You can give your AWS Lambda function access to a resource in a Amazon VPC in Amazon Virtual Private Cloud managed by another account, without exposing either VPC to the internet. This access pattern allows you to share data with other organizations using AWS. Using this access pattern, you can share data between VPCs with a greater level of security and performance than over the internet. Configure your Lambda function to use a Amazon VPC peering connection to access these resources.

> **Warning**
>
> When you allow access between accounts or VPCs, check that your plan meets the security requirements of the respective organizations that manage these accounts. Following the instructions in this document will affect the security posture of your resources.

In this tutorial, you connect two accounts together with a peering connection using IPv4. You configure a Lambda function that is not already connected to a Amazon VPC. You configure DNS resolution to connect your function to resources that do not provide static IPs. To adapt these instructions to other peering scenarios, consult the VPC Peering Guide.

Prerequisites

To give a Lambda function access to a resource in another acccount, you must have:

- A Lambda function, configured to authenticate with and then read from your resource.
- A resource in another account, such as an Amazon RDS cluster, available through Amazon VPC.
- Credentials for your Lambda function's account and your resource's account. If you are not authorized to use your resource's account, contact an authorized user to prepare that account.
- Permission to create and update a VPC (and supporting Amazon VPC resources) to associate with your Lambda function.
- Permission to update the execution role and VPC configuration for your Lambda function.
- Permission to create a VPC peering connection in your Lambda function's account.
- Permission to accept a VPC peering connection in your resource's account.

- Permission to update the configuration of your resource's VPC (and supporting Amazon VPC resources).
- Permission to invoke your Lambda function.

Create an Amazon VPC in your function's account

Create an Amazon VPC, subnets, route tables, and a security group in your Lambda function's account.

To create a VPC, subnets, and other VPC resources using the console

1. Open the Amazon VPC Console at https://console.aws.amazon.com/vpc/.
2. On the dashboard, choose **Create VPC**.
3. For **IPv4 CIDR block**, provide a private CIDR block. Your CIDR block must not overlap with blocks used in your resource's VPC. Don't pick a block your resources' VPC uses to assign IPs to resources or a block already defined in the route tables in your resources VPC. For more information about defining appropriate CIDR blocks, see VPC CIDR blocks.
4. Choose **Customize AZs**.
5. Select the same AZs as your resource.
6. For **Number of public subnets**, choose **0**.
7. For **VPC endpoints**, choose **None**.
8. Choose **Create VPC**.

Grant VPC permissions to your function's execution role

Attach AWSLambdaVPCAccessExecutionRole to your function’s execution role to allow it to connect to VPCs.

To grant VPC permissions to your function's execution role

1. Open the Functions page of the Lambda console.
2. Choose the name of your function.
3. Choose **Configuration**.
4. Choose **Permissions**.
5. Under **Role name**, choose the execution role.

6. In the **Permissions policies** section, choose **Add permissions**.
7. In the dropdown list, choose **Attach policies**.
8. In the search box, enter `AWSLambdaVPCAccessExecutionRole`.
9. To the left of the policy name, choose the checkbox.
10. Choose **Add permissions**.

To attach your function to your Amazon VPC

1. Open the Functions page of the Lambda console.
2. Choose the name of your function.
3. Choose the **Configuration** tab, then choose **VPC**.
4. Choose **Edit**.
5. Under **VPC**, select your VPC
6. Under **Subnets**, choose your subnets.
7. Under **Security groups**, choose the default security group for your VPC.
8. Choose **Save**.

Create a VPC peering connection request

Create a VPC peering connection request from your function's VPC (the requester VPC) to your resource's VPC (the accepter VPC).

To request a VPC peering connection from your function's VPC

1. Open the https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Peering connections**.
3. Choose **Create peering connection**.
4. For **VPC ID (Requester)**, select your function's VPC.
5. For **Account ID**, enter the ID of your resource's account.
6. For **VPC ID (Accepter)**, enter your resource's VPC.

Prepare your resource's account

To create your peering connection and prepare your resource's VPC to use the connection, log in to your resource's account with a role that holds the permissions listed in the prerequisites. The steps to log in may be different based on how the account is secured. For more information about how to sign in to an AWS account, see the AWS Sign-in User Guide. In your resource's account, perform the following procedures.

To accept the VPC peering connection request

1. Open the https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Peering connections**.
3. Select the pending VPC peering connection (the status is pending-acceptance).
4. Choose **Actions**
5. From the dropdown list, choose **Accept request**.
6. When prompted for confirmation, choose **Accept request**.
7. Choose **Modify my route tables now** to add a route to the main route table for your VPC so that you can send and receive traffic across the peering connection.

Inspect the route tables for the resource's VPC. The route generated by Amazon VPC might not establish connectivity, based on how your resource's VPC is set up. Check for conflicts between the new route and existing configuration for the VPC. For more information about troubleshooting, see Troubleshoot a VPC peering connection in the *Amazon Virtual Private Cloud VPC Peering Guide*.

To update the security group for your resource

1. Open the https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Security groups**.
3. Select the security group for your resource.
4. Choose **Actions**.
5. From the dropdown list, choose **Edit inbound rules**.
6. Choose **Add rule**.
7. For **Source** enter your function's account ID and security group ID, separated by a forward slash (for example, 111122223333/sg-1a2b3c4d).
8. Choose **Edit outbound rules**.

9. Check whether outbound traffic is restricted. Default VPC settings allow all outbound traffic. If outbound traffic is restricted, continue to the next step.
10. Choose **Add rule**.
11. For **Destination** enter your function's account ID and security group ID, separated by a forward slash (for example, 111122223333/sg-1a2b3c4d).
12. Choose **Save rules**.

To enable DNS resolution for your peering connection

1. Open the https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Peering connections**.
3. Select your peering connection.
4. Choose **Actions**.
5. Choose **Edit DNS settings**.
6. Below **Accepter DNS resolution**, select **Allow requester VPC to resolve DNS of accepter VPC hosts to private IP**.
7. Choose **Save changes**.

Update VPC configuration in your function's account

Log in to your function's account, then update the VPC configuration.

To add a route for your VPC peering connection

1. Open the https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Route tables**.
3. Select the check box next to the name of the route table for the subnet you associated with your function.
4. Choose **Actions**.
5. Choose **Edit routes**.
6. Choose **Add route**.
7. For **Destination**, enter the CIDR block for your resource's VPC.
8. For **Target**, select your VPC peering connection.
9. Choose **Save changes**.

For more information about considerations you may encounter while updating your route tables, consult Update your route tables for a VPC peering connection.

To update the security group for your Lambda function

1. Open the https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Security groups**.
3. Choose **Actions**.
4. Choose **Edit inbound rules**.
5. Choose **Add rule**.
6. For **Source** enter your resource's account ID and security group ID, separated by a forward slash (for example, 111122223333/sg-1a2b3c4d).
7. Choose **Save rules**.

To enable DNS resolution for your peering connection

1. Open the https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Peering connections**.
3. Select your peering connection.
4. Choose **Actions**.
5. Choose **Edit DNS settings**.
6. Below **Requester DNS resolution**, select **Allow accepter VPC to resolve DNS of requester VPC hosts to private IP**.
7. Choose **Save changes**.

Test your function

To create a test event and inspect your function's output

1. In the **Code source** pane, choose **Test**.
2. Select **Create new event**.
3. In the **Event JSON** panel, replace the default values with an input appropriate for your Lambda function.
4. Choose **Invoke**.

5. In the **Execution results** tab, confirm that **Response** contains your expected output.

Additionally, you can check your function's logs to verify the logs are as you expect.

To view your function's invocation records in CloudWatch Logs

1. Choose the **Monitor** tab.
2. Choose **View CloudWatch logs**.
3. In the **Log streams** tab, choose the log stream for your function's invocation.
4. Confirm your logs are as you expect.

Enable internet access for VPC-connected Lambda functions

By default, Lambda functions run in a Lambda-managed VPC that has internet access. To access resources in a VPC in your account, you can add a VPC configuration to a function. This restricts the function to resources within that VPC, unless the VPC has internet access. This page explains how to provide internet access to VPC-connected Lambda functions.

I don't have a VPC yet

Create the VPC

The **Create VPC workflow** creates all VPC resources required for a Lambda function to access the public internet from a private subnet, including subnets, NAT gateway, internet gateway, and route table entries.

To create the VPC

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. On the dashboard, choose **Create VPC**.
3. For **Resources to create**, choose **VPC and more**.
4. **Configure the VPC**
    a. For **Name tag auto-generation**, enter a name for the VPC.
    b. For **IPv4 CIDR block**, you can keep the default suggestion, or alternatively you can enter the CIDR block required by your application or network.
    c. If your application communicates by using IPv6 addresses, choose **IPv6 CIDR block**, **Amazon-provided IPv6 CIDR block**.
5. **Configure the subnets**
    a. For **Number of Availability Zones**, choose **2**. We recommend at least two AZs for high availability.
    b. For **Number of public subnets**, choose **2**.
    c. For **Number of private subnets**, choose **2**.
    d. You can keep the default CIDR block for the public subnet, or alternatively you can expand **Customize subnet CIDR blocks** and enter a CIDR block. For more information, see Subnet CIDR blocks .
6. For **NAT gateways**, choose **1 per AZ** to improve resiliency.

7. For **Egress only internet gateway**, choose **Yes** if you opted to include an IPv6 CIDR block.
8. For **VPC endpoints**, keep the default (**S3 Gateway**). There is no cost for this option. For more information, see Types of VPC endpoints for Amazon S3.
9. For **DNS options**, keep the default settings.
10. Choose **Create VPC**.

Configure the Lambda function

To configure a VPC when you create a function

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Under **Basic information**, for **Function name**, enter a name for your function.
4. Expand **Advanced settings**.
5. Select **Enable VPC**, and then choose a VPC.
6. (Optional) To allow outbound IPv6 traffic, select **Allow IPv6 traffic for dual-stack subnets**.
7. For **Subnets**, select all private subnets. The private subnets can access the internet through the NAT gateway. Connecting a function to a public subnet doesn't give it internet access.

> **Note**
>
> If you selected **Allow IPv6 traffic for dual-stack subnets**, all selected subnets must have an IPv4 CIDR block and an IPv6 CIDR block.

8. For **Security groups**, select a security group that allows outbound traffic.
9. Choose **Create function**.

Lambda automatically creates an execution role with the AWSLambdaVPCAccessExecutionRole AWS managed policy. The permissions in this policy are required only to create elastic network interfaces for the VPC configuration, not to invoke your function. To apply least-privilege permissions, you can remove the **AWSLambdaVPCAccessExecutionRole** policy from your execution role after you create the function and VPC configuration. For more information, see Required IAM permissions.

To configure a VPC for an existing function

To add a VPC configuration to an existing function, the function's execution role must have permission to create and manage elastic network interfaces. The AWSLambdaVPCAccessExecutionRole AWS managed policy includes the required permissions. To apply least-privilege permissions, you can remove the **AWSLambdaVPCAccessExecutionRole** policy from your execution role after you create the VPC configuration.

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the **Configuration** tab, and then choose **VPC**.
4. Under **VPC**, choose **Edit**.
5. Select the VPC.
6. (Optional) To allow outbound IPv6 traffic, select **Allow IPv6 traffic for dual-stack subnets**.
7. For **Subnets**, select all private subnets. The private subnets can access the internet through the NAT gateway. Connecting a function to a public subnet doesn't give it internet access.

   > **Note**
   >
   > If you selected **Allow IPv6 traffic for dual-stack subnets**, all selected subnets must have an IPv4 CIDR block and an IPv6 CIDR block.

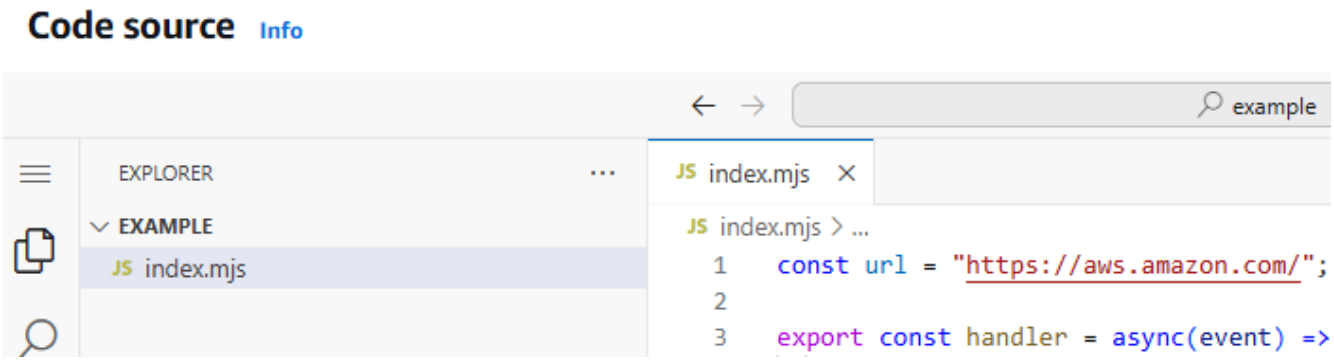
8. For **Security groups**, select a security group that allows outbound traffic.
9. Choose **Save**.

Test the function

Use the following sample code to confirm that your VPC-connected function can reach the public internet. If successful, the code returns a `200` status code. If unsuccessful, the function times out.

Node.js

1. In the **Code source** pane on the Lambda console, paste the following code into the **index.mjs** file. The function makes an HTTP GET request to a public endpoint and returns the HTTP response code to test if the function has access to the public internet.

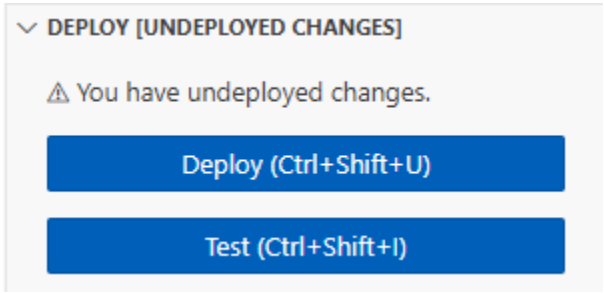


Example— HTTP request with async/await

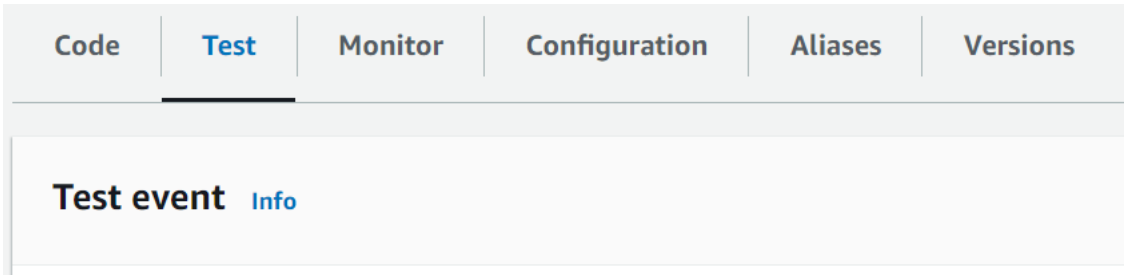
```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

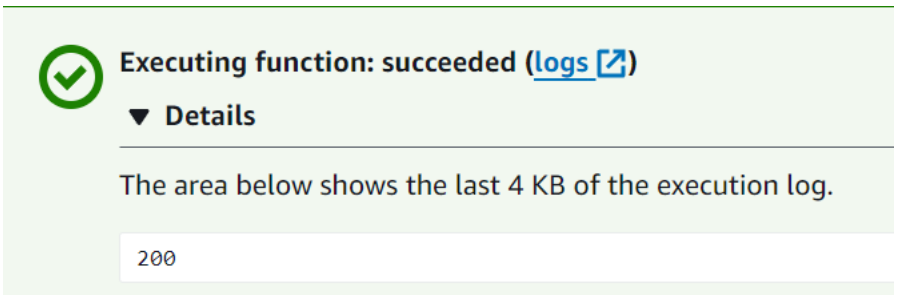
2. In the **DEPLOY** section, choose **Deploy** to update your function's code:



3. Choose the **Test** tab.



4. Choose **Test**.
5. The function returns a `200` status code. This means that the function has outbound internet access.

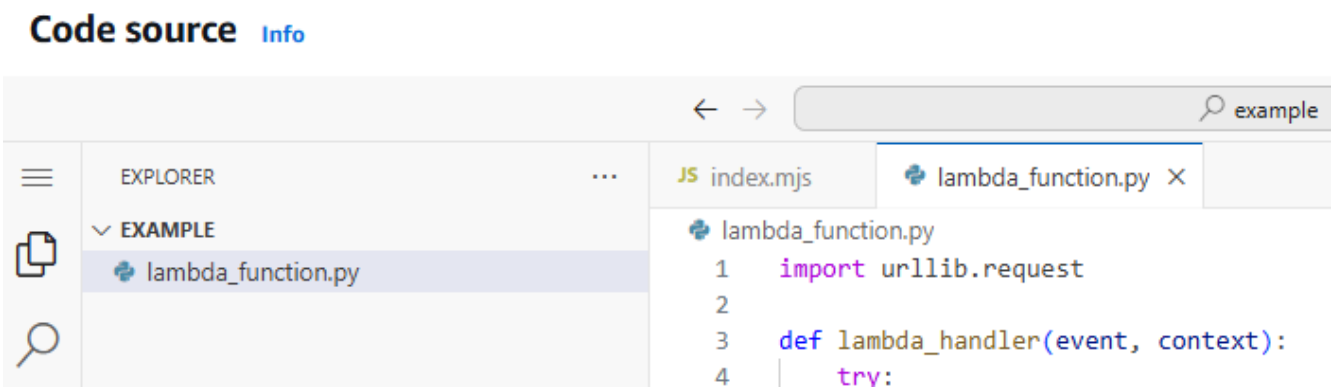


If the function can't reach the public internet, you get an error message like this:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Python

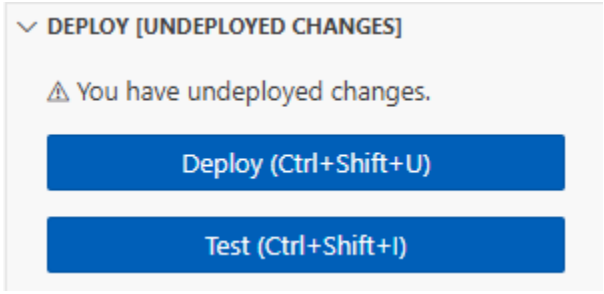
1. In the **Code source** pane on the Lambda console, paste the following code into the **lambda_function.py** file. The function makes an HTTP GET request to a public endpoint and returns the HTTP response code to test if the function has access to the public internet.



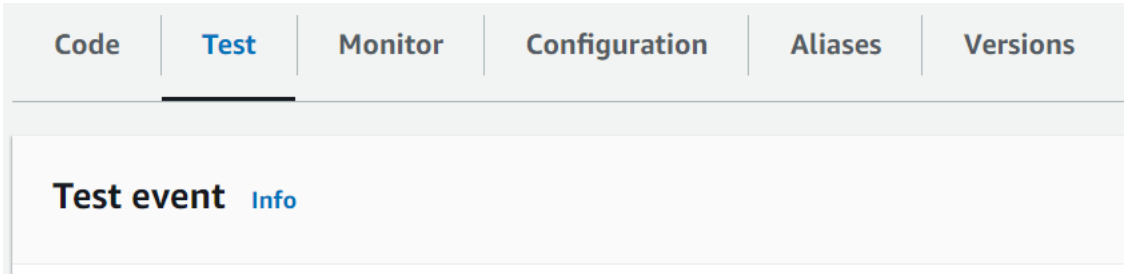

```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. In the **DEPLOY** section, choose **Deploy** to update your function's code:

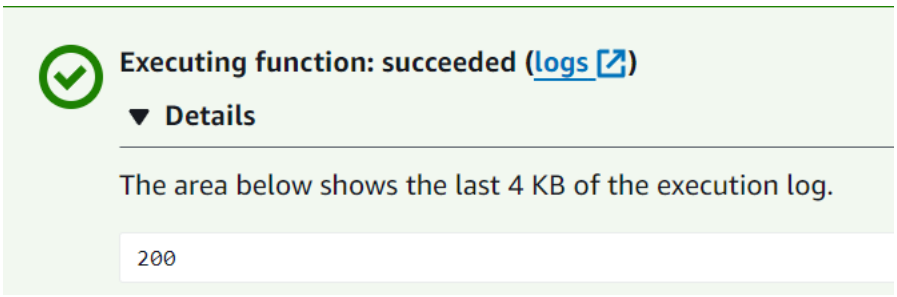


3. Choose the **Test** tab.



4. Choose **Test**.

5. The function returns a `200` status code. This means that the function has outbound internet access.



If the function can't reach the public internet, you get an error message like this:

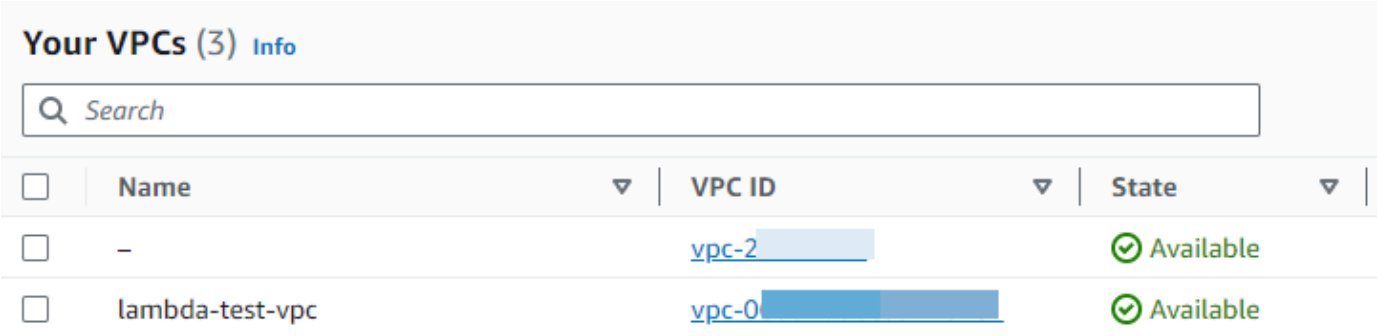
```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12j1c-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

I already have a VPC

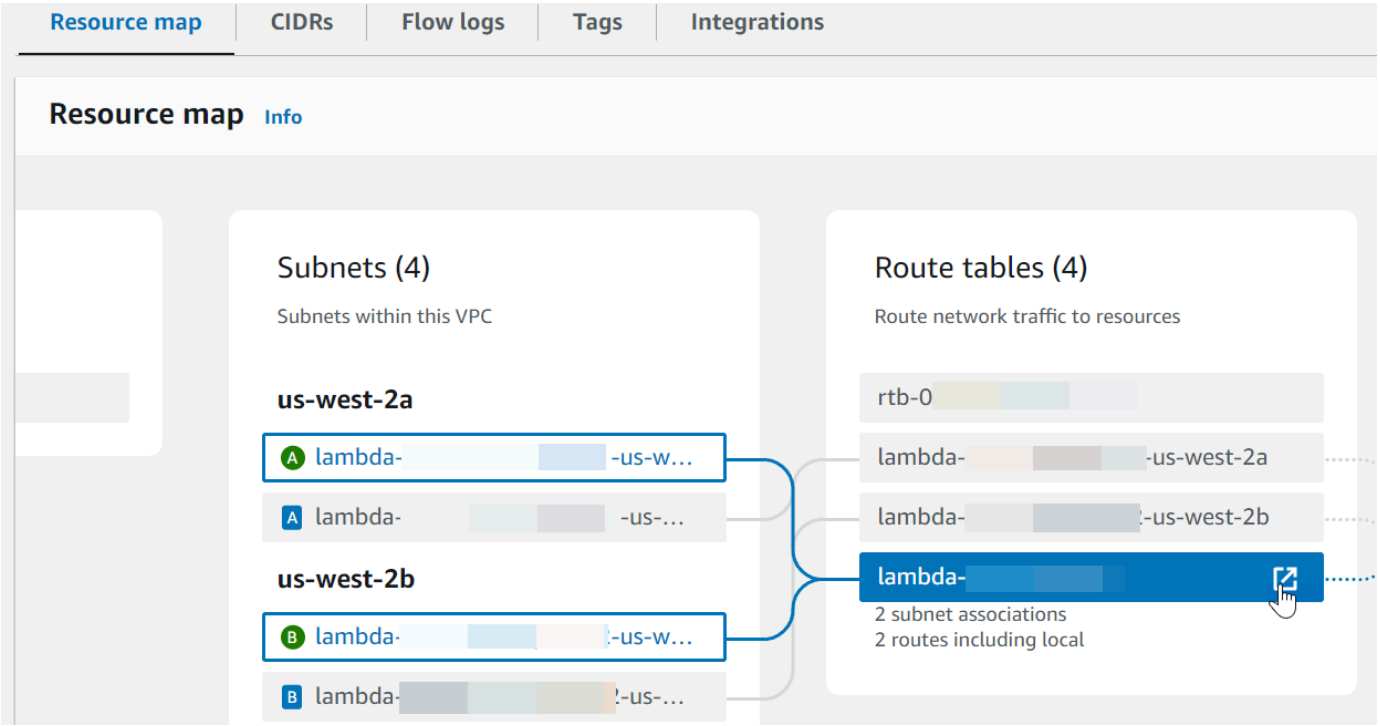
If you already have a VPC but you need to configure public internet access for a Lambda function, follow these steps. This procedure assumes that your VPC has at least two subnets. If you don't have two subnets, see Create a subnet in the *Amazon VPC User Guide*.

Verify the route table configuration

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Choose the **VPC ID**.



3. Scroll down to the **Resource map** section. Note the route table mappings. Open each route table that is mapped to a subnet.

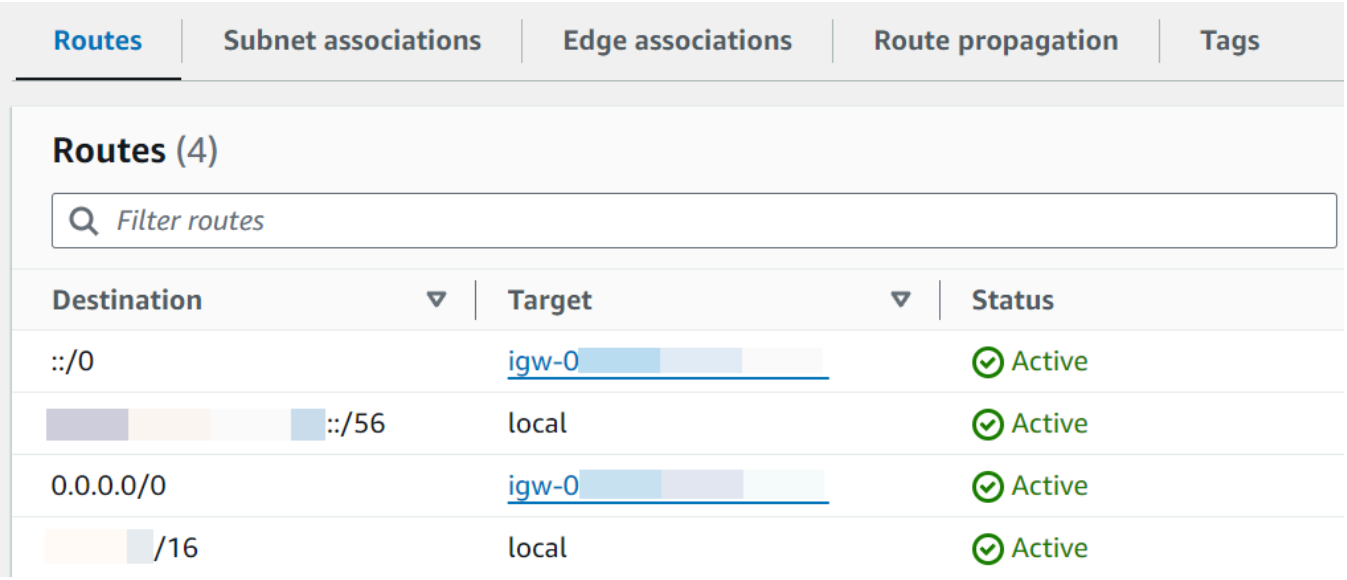


4. Scroll down to the **Routes** tab. Review the routes to determine if your VPC has both of the following route tables. Each of these requirements must be satisfied by a separate route table.

   - Internet-bound traffic (`0.0.0.0/0` for IPv4, `::/0` for IPv6) is routed to an internet gateway (`igw-xxxxxxxxxx`). This means that the subnet associated with the route table is a public subnet.

     > **Note**
     >
     > If your subnet doesn't have an IPv6 CIDR block, you will only see the IPv4 route (`0.0.0.0/0`).

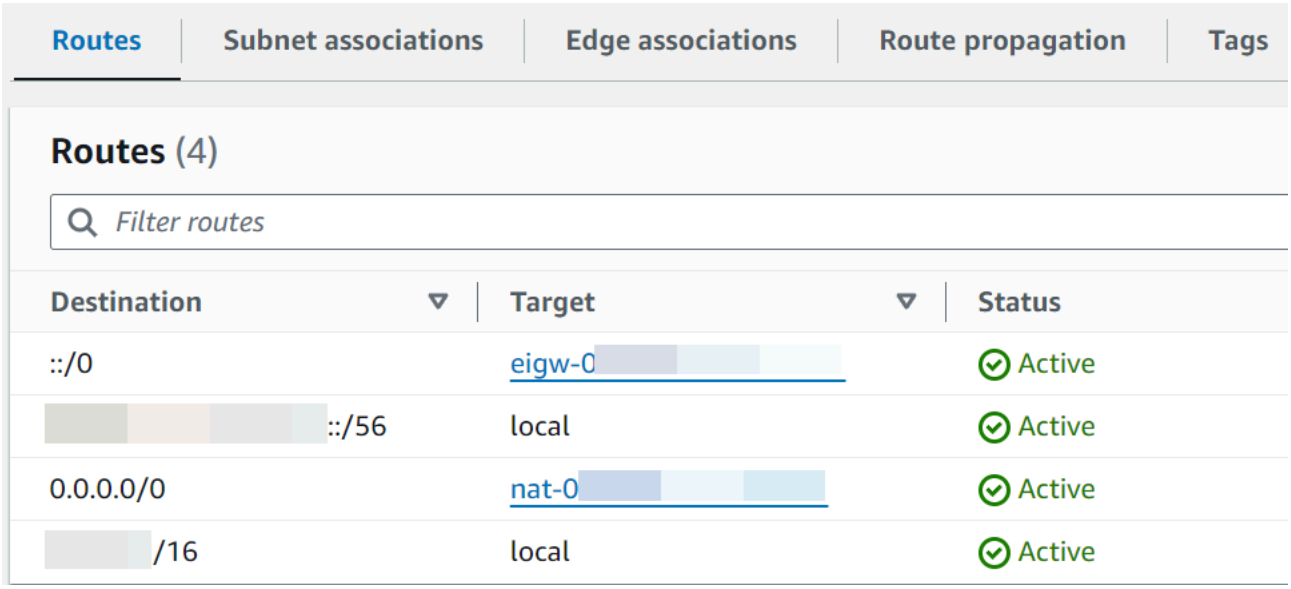
Example public subnet route table



- Internet-bound traffic for IPv4 (`0.0.0.0/0`) is routed to a NAT gateway (`nat-xxxxxxxxxx`) that is associated with a public subnet. This means that the subnet is a private subnet that can access the internet through the NAT gateway.

> **Note**
>
> If your subnet has an IPv6 CIDR block, the route table must also route internet-bound IPv6 traffic (`::/0`) to an egress-only internet gateway (`eigw-xxxxxxxxxx`). If your subnet doesn't have an IPv6 CIDR block, you will only see the IPv4 route (`0.0.0.0/0`).

Example private subnet route table



5. Repeat the previous step until you have reviewed each route table associated with a subnet in your VPC and confirmed that you have a route table with an internet gateway and a route table with a NAT gateway.

   If you don't have two route tables, one with a route to an internet gateway and one with a route to a NAT gateway, follow these steps to create the missing resources and route table entries.

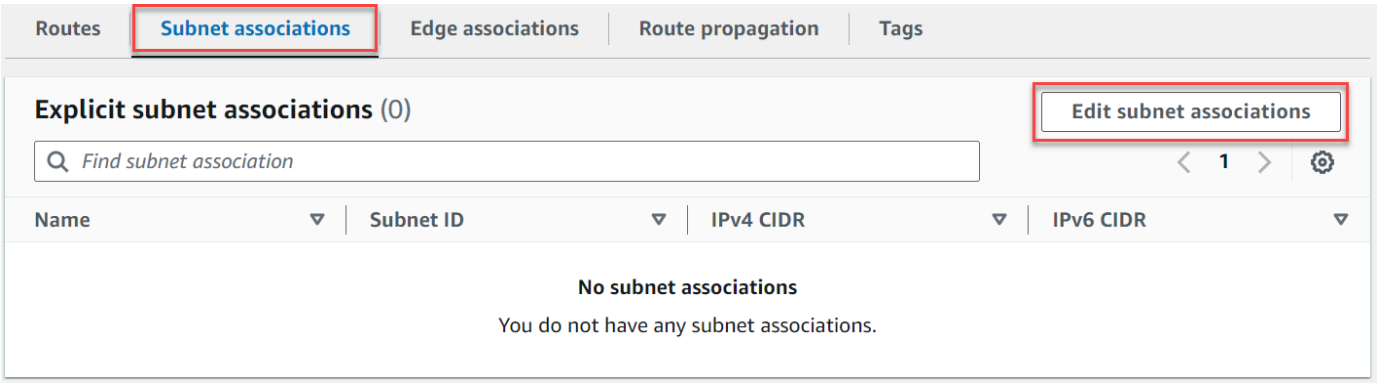
Create a route table

Follow these steps to create a route table and associate it with a subnet.

To create a custom route table using the Amazon VPC console

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Route tables**.
3. Choose **Create route table**.
4. (Optional) For **Name**, enter a name for your route table.
5. For **VPC**, choose your VPC.
6. (Optional) To add a tag, choose **Add new tag** and enter the tag key and tag value.
7. Choose **Create route table**.

8. On the **Subnet associations** tab, choose **Edit subnet associations**.



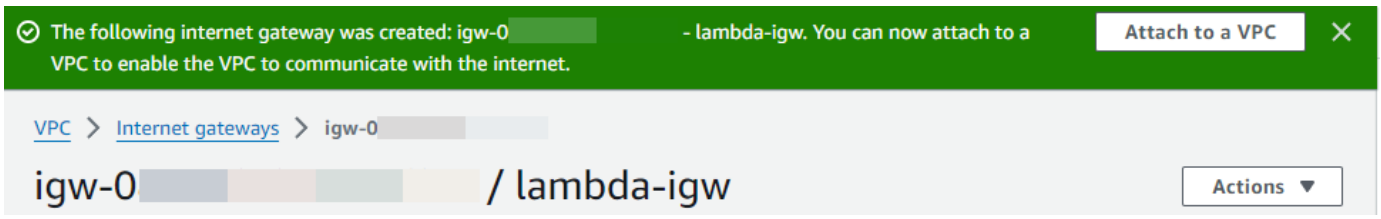
9. Select the check box for the subnet to associate with the route table.
10. Choose **Save associations**.

Create an internet gateway

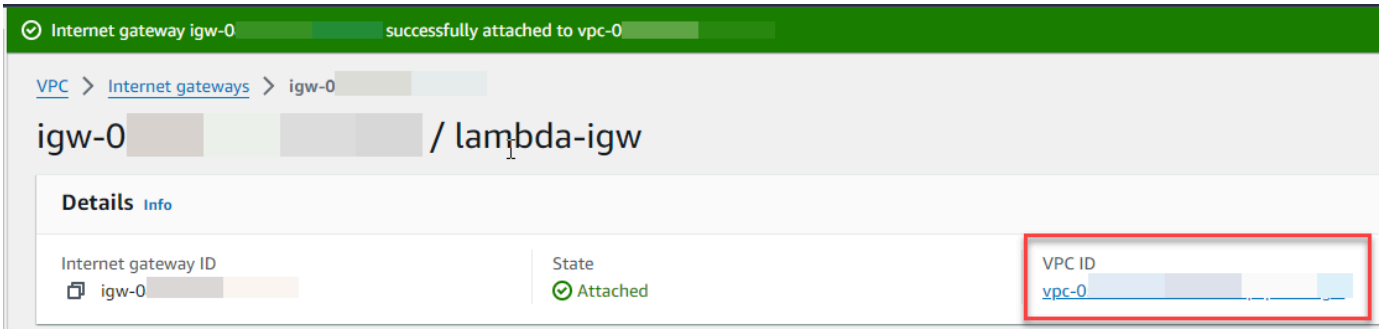
Follow these steps to create an internet gateway, attach it to your VPC, and add it to your public subnet's route table.

To create an internet gateway

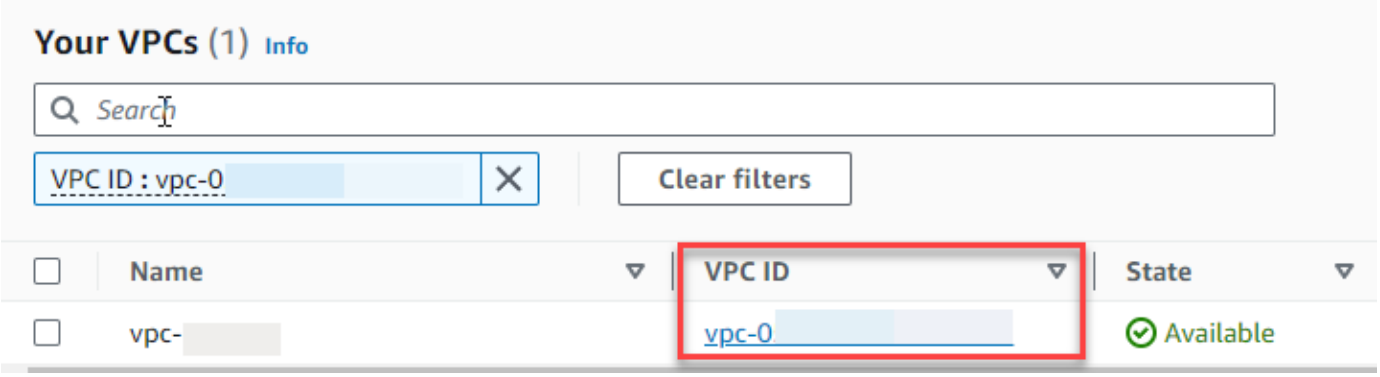
1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the navigation pane, choose **Internet gateways**.
3. Choose **Create internet gateway**.
4. (Optional) Enter a name for your internet gateway.
5. (Optional) To add a tag, choose **Add new tag** and enter the tag key and value.
6. Choose **Create internet gateway**.
7. Choose **Attach to a VPC** from the banner at the top of the screen, select an available VPC, and then choose **Attach internet gateway**.



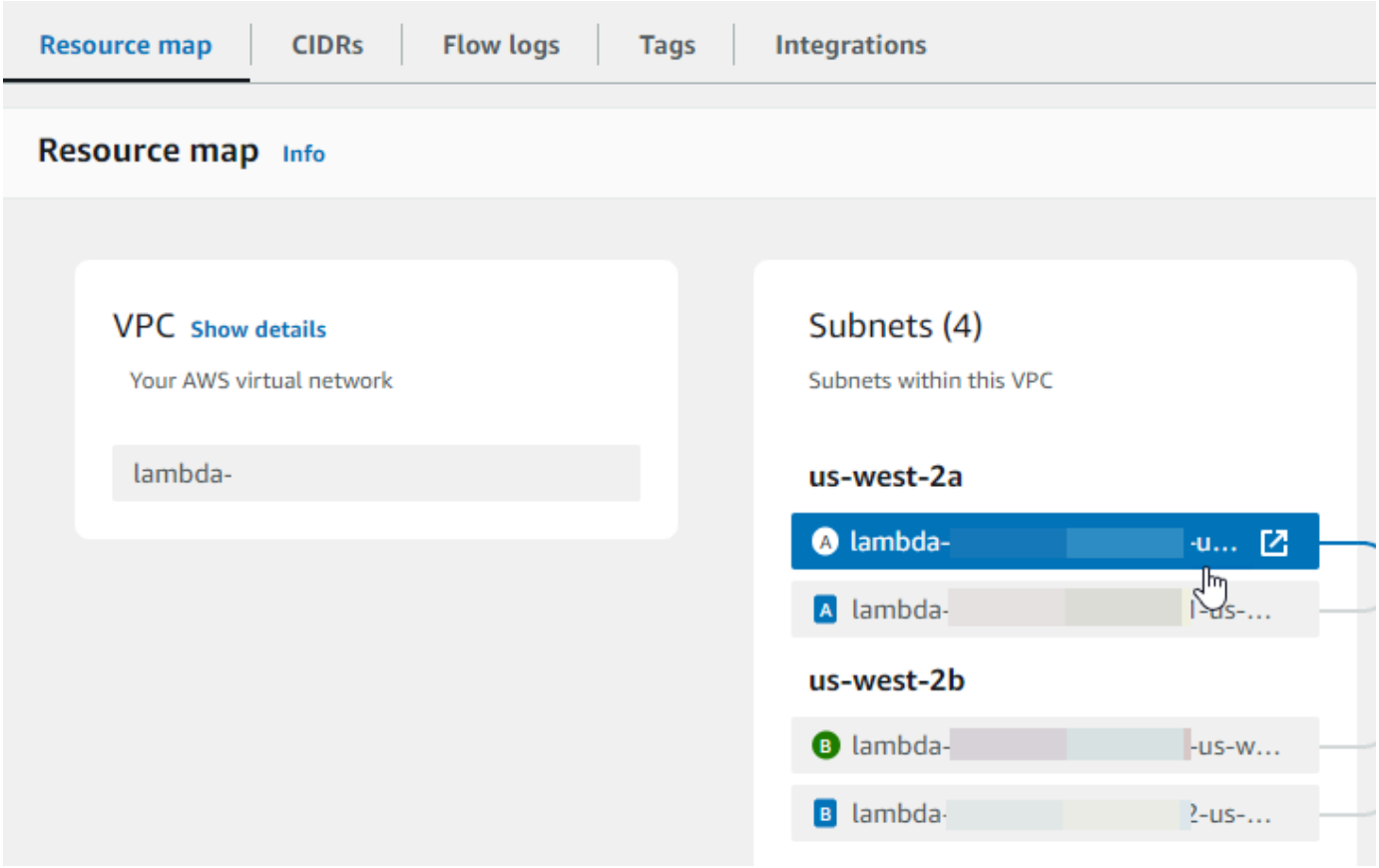
8. Choose the **VPC ID**.



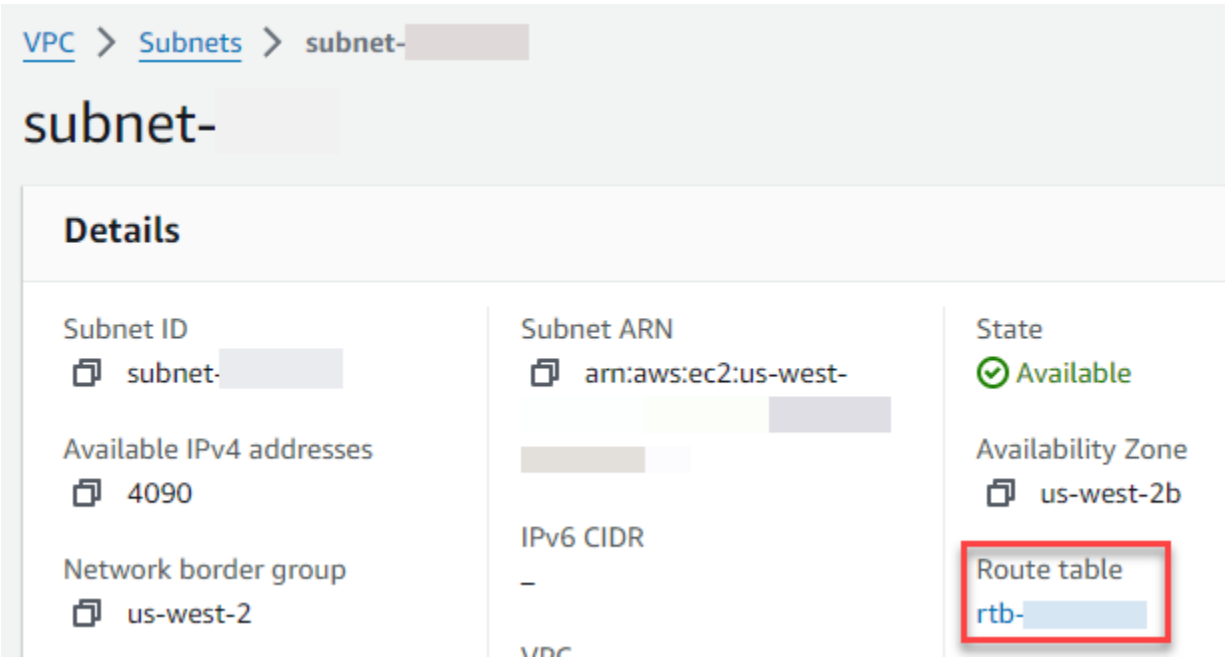
9. Choose the **VPC ID** again to open the VPC details page.



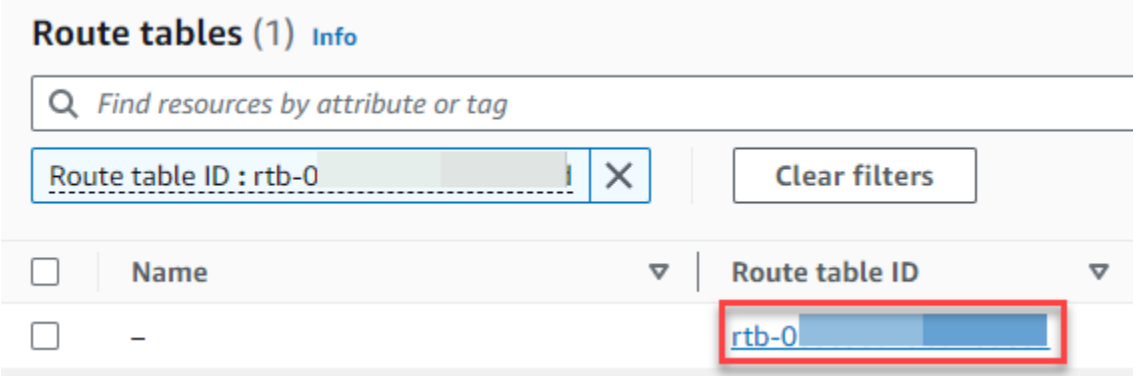
10. Scroll down to the **Resource map** section and then choose a subnet. The subnet details are displayed in a new tab.



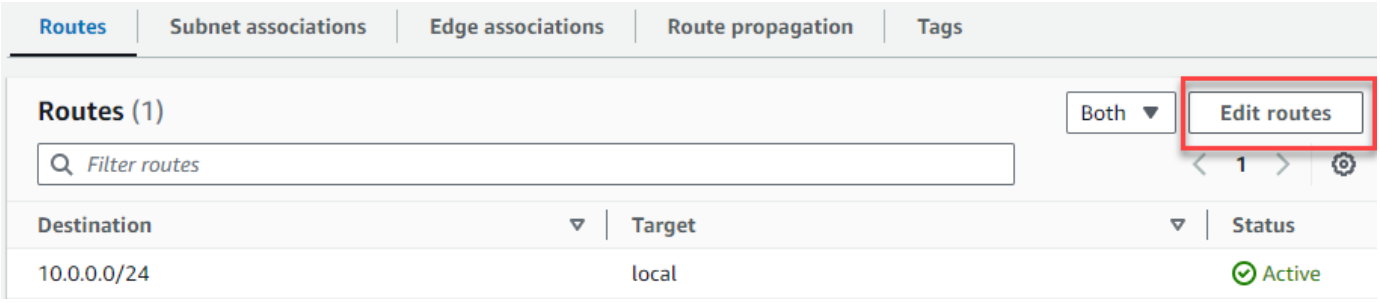
11. Choose the link under **Route table**.



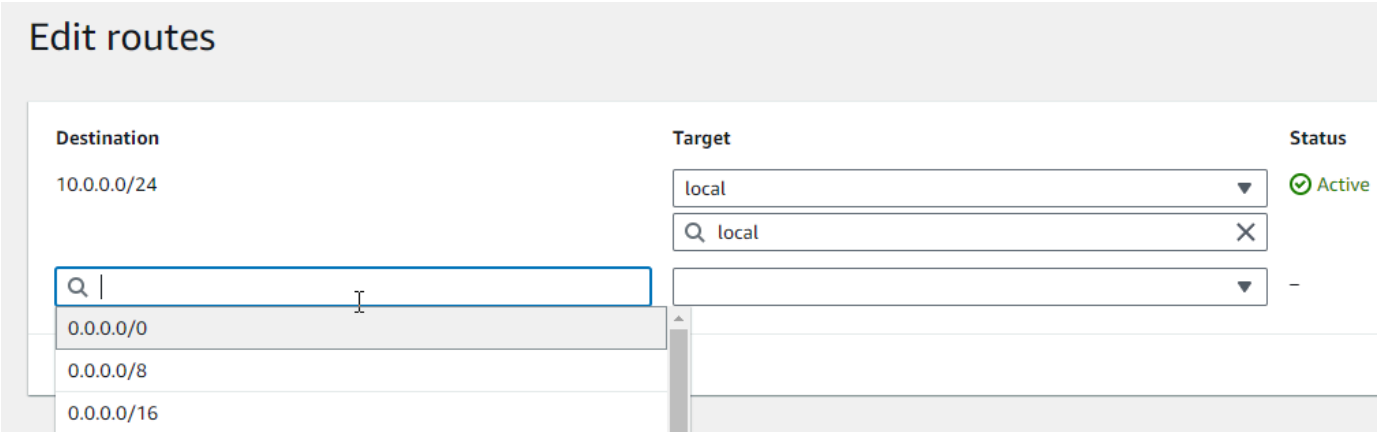
12. Choose the **Route table ID** to open the route table details page.



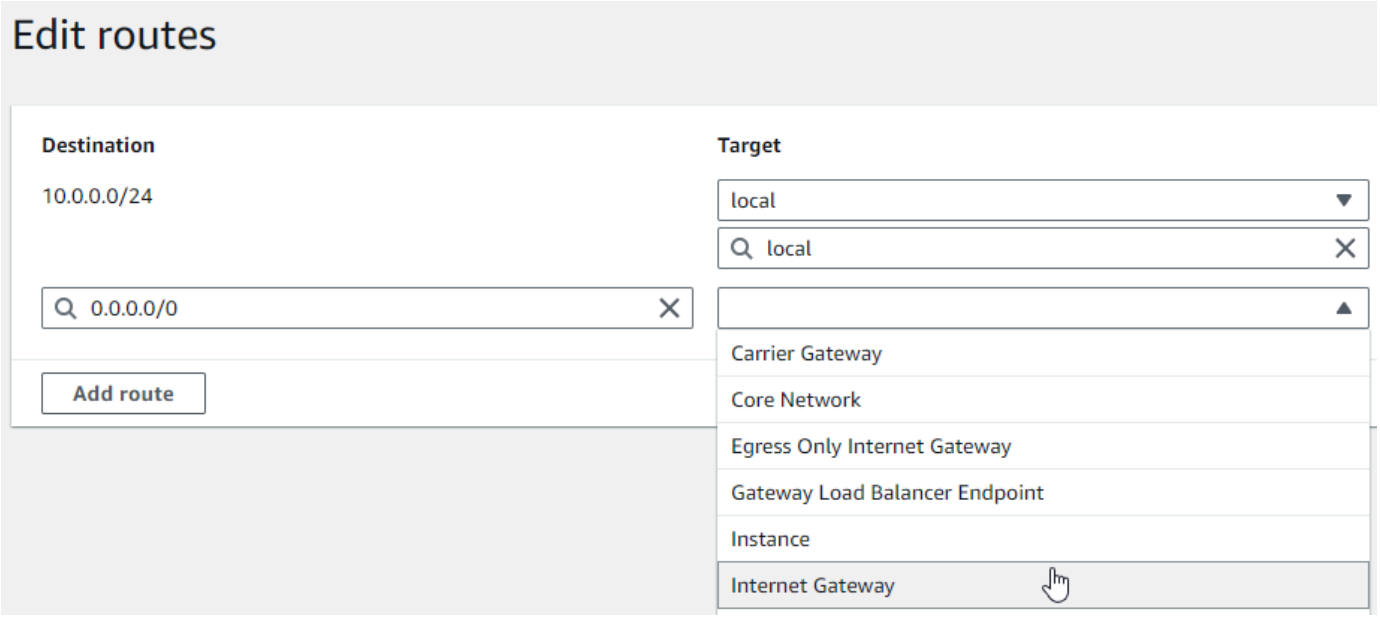
13. Under **Routes**, choose **Edit routes**.



14. Choose **Add route**, and then enter `0.0.0.0/0` in the **Destination** box.



15. For **Target**, select **Internet gateway**, and then choose the internet gateway that you created earlier. If your subnet has an IPv6 CIDR block, you must also add a route for `::/0` to the same internet gateway.



16. Choose **Save changes**.

Create a NAT gateway

Follow these steps to create a NAT gateway, associate it with a public subnet, and then add it to your private subnet's route table.

To create a NAT gateway and associate it with a public subnet

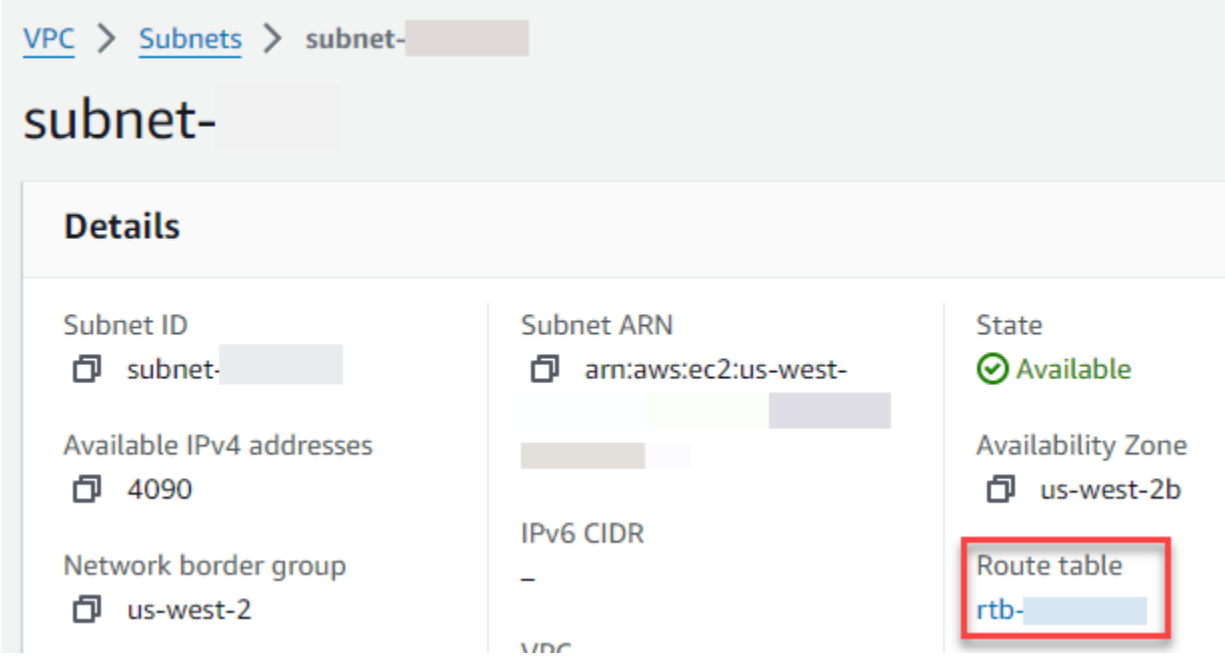
1. In the navigation pane, choose **NAT gateways**.
2. Choose **Create NAT gateway**.
3. (Optional) Enter a name for your NAT gateway.
4. For **Subnet**, select a public subnet in your VPC. (A public subnet is a subnet that has a direct route to an internet gateway in its route table.)

> **Note**
>
> NAT gateways are associated with a public subnet, but the route table entry is in the private subnet.

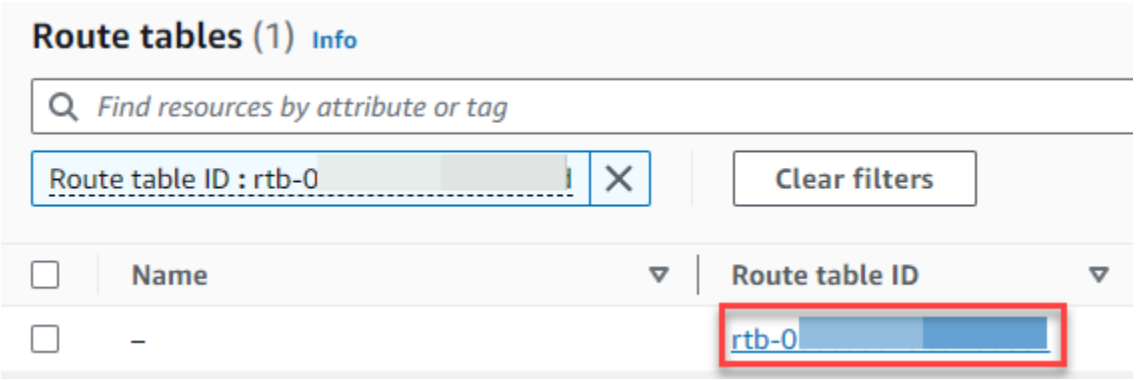
5. For **Elastic IP allocation ID**, select an elastic IP address or choose **Allocate Elastic IP**.
6. Choose **Create NAT gateway**.

To add a route to the NAT gateway in the private subnet's route table

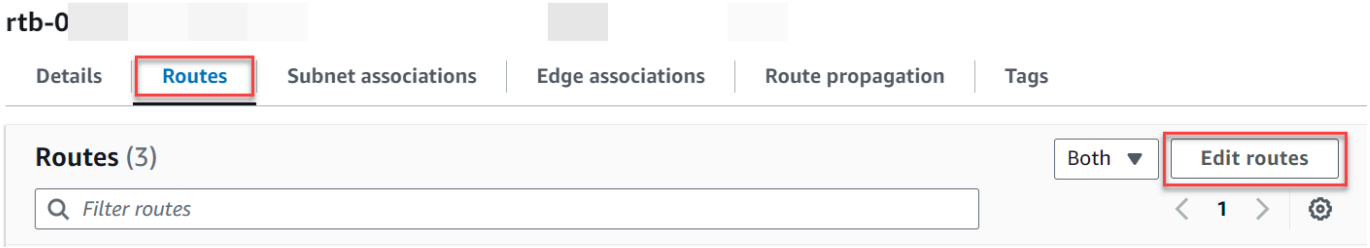
1. In the navigation pane, choose **Subnets**.
2. Select a private subnet in your VPC. (A private subnet is a subnet that doesn't have a route to an internet gateway in its route table.)
3. Choose the link under **Route table**.



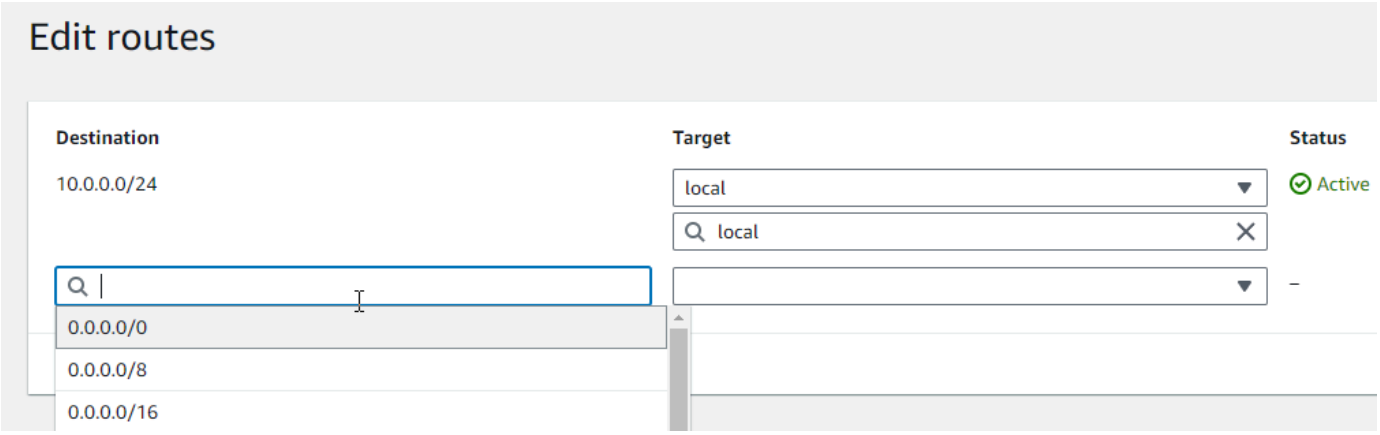
4. Choose the **Route table ID** to open the route table details page.



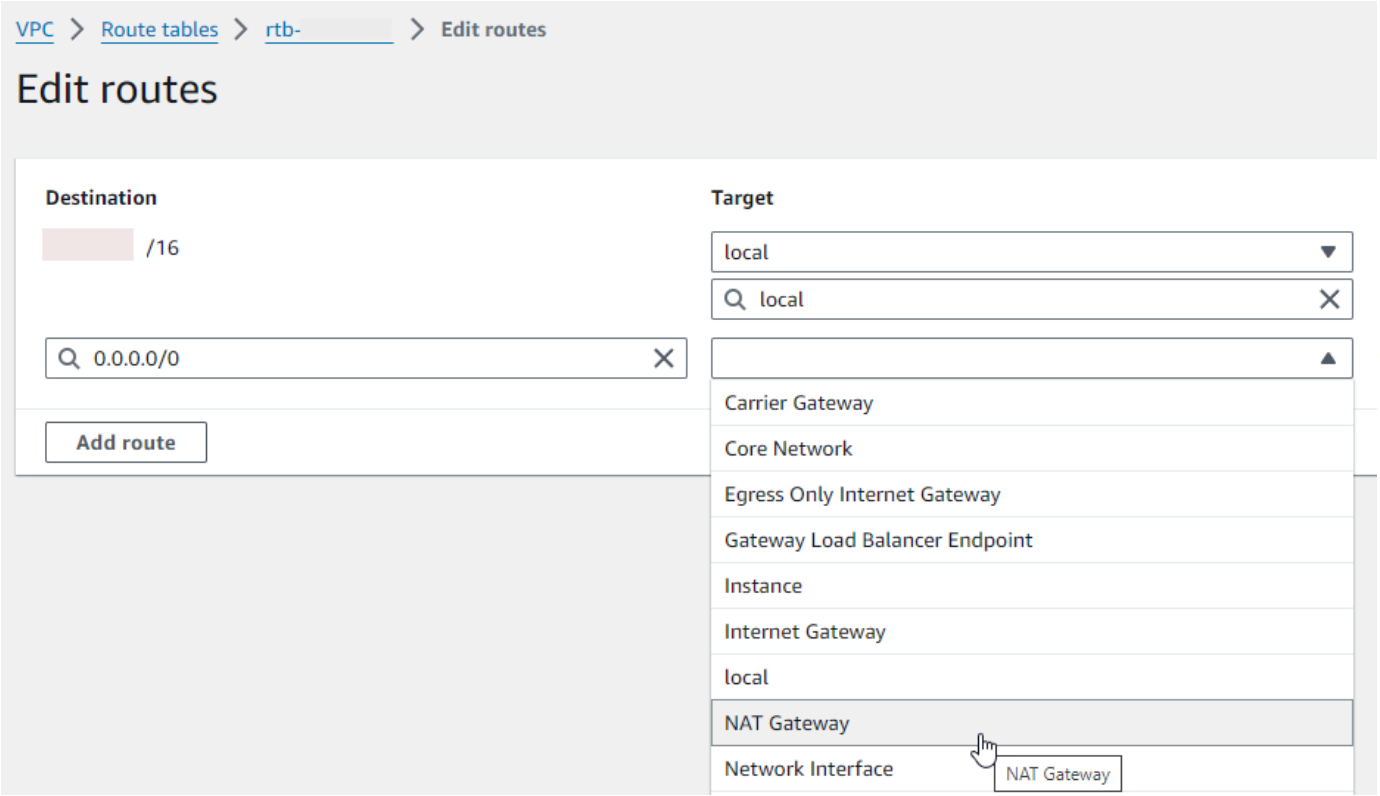
5. Scroll down and choose the **Routes** tab, then choose **Edit routes**



6. Choose **Add route**, and then enter `0.0.0.0/0` in the **Destination** box.



7. For **Target**, select **NAT gateway**, and then choose the NAT gateway that you created earlier.



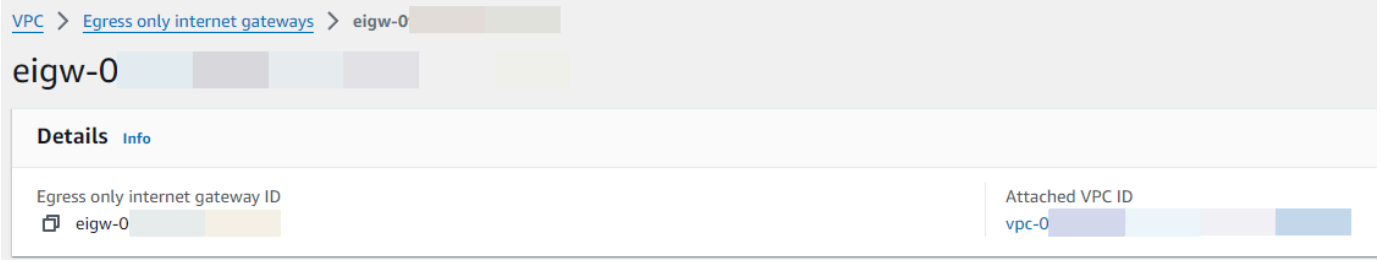
8. Choose **Save changes**.

Create an egress-only internet gateway (IPv6 only)

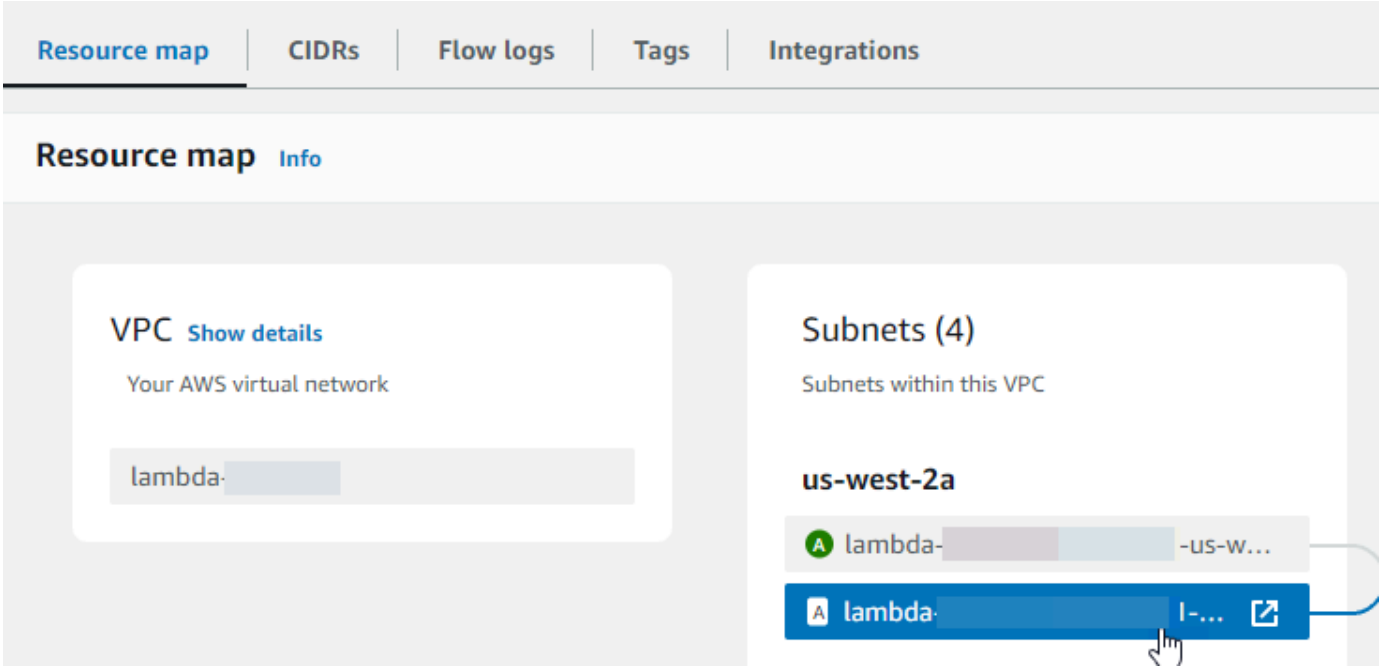
Follow these steps to create an egress-only internet gateway and add it to your private subnet's route table.

To create an egress-only internet gateway

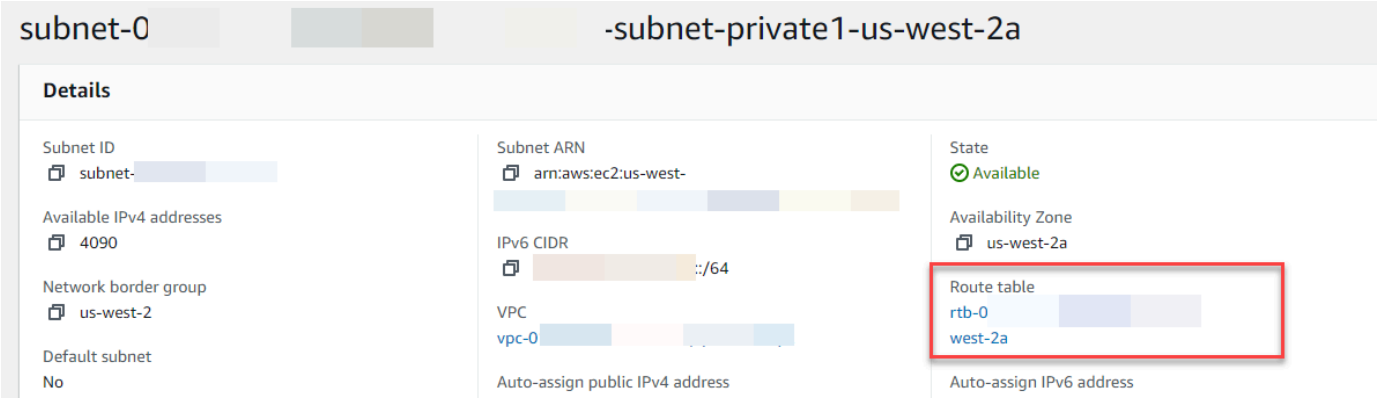
1. In the navigation pane, choose **Egress-only internet gateways**.
2. Choose **Create egress only internet gateway**.
3. (Optional) Enter a name.
4. Select the VPC in which to create the egress-only internet gateway.
5. Choose **Create egress only internet gateway**.
6. Choose the link under **Attached VPC ID**.



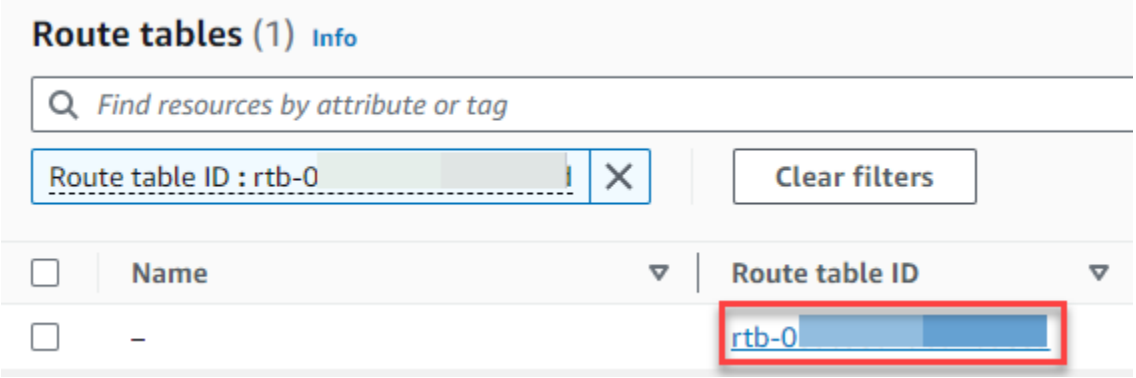
7. Choose the link under **VPC ID** to open the VPC details page.
8. Scroll down to the **Resource map** section and then choose a private subnet. (A private subnet is a subnet that doesn't have a route to an internet gateway in its route table.) The subnet details are displayed in a new tab.



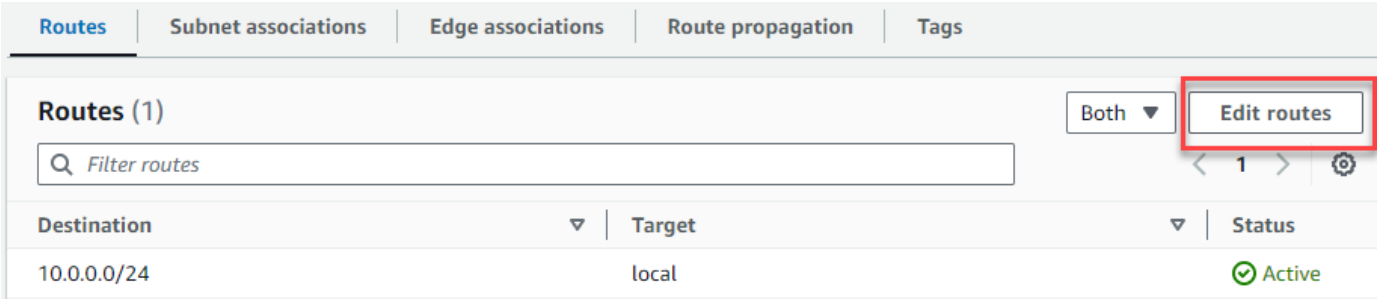
9. Choose the link under **Route table**.



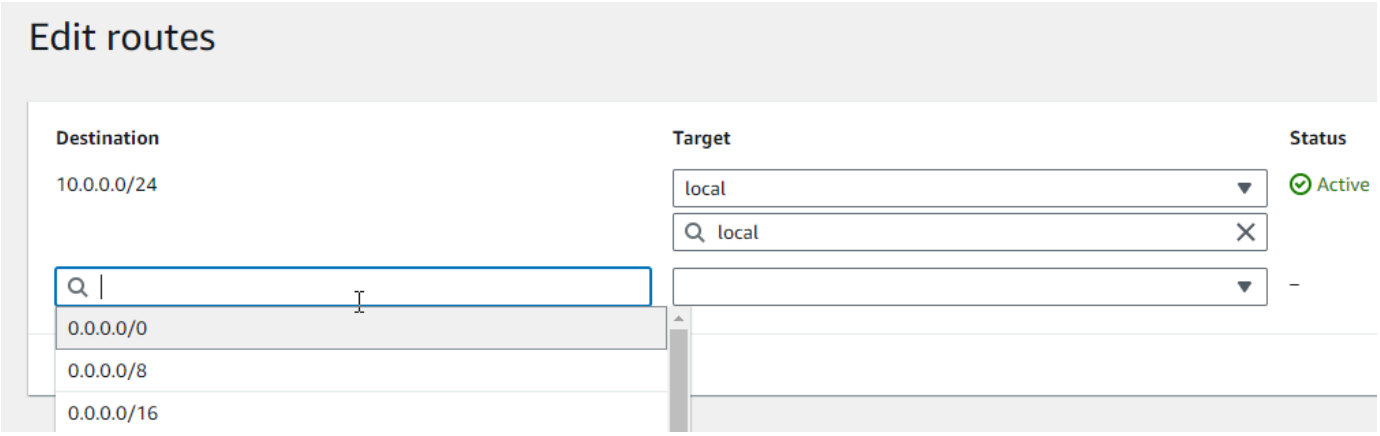
10. Choose the **Route table ID** to open the route table details page.



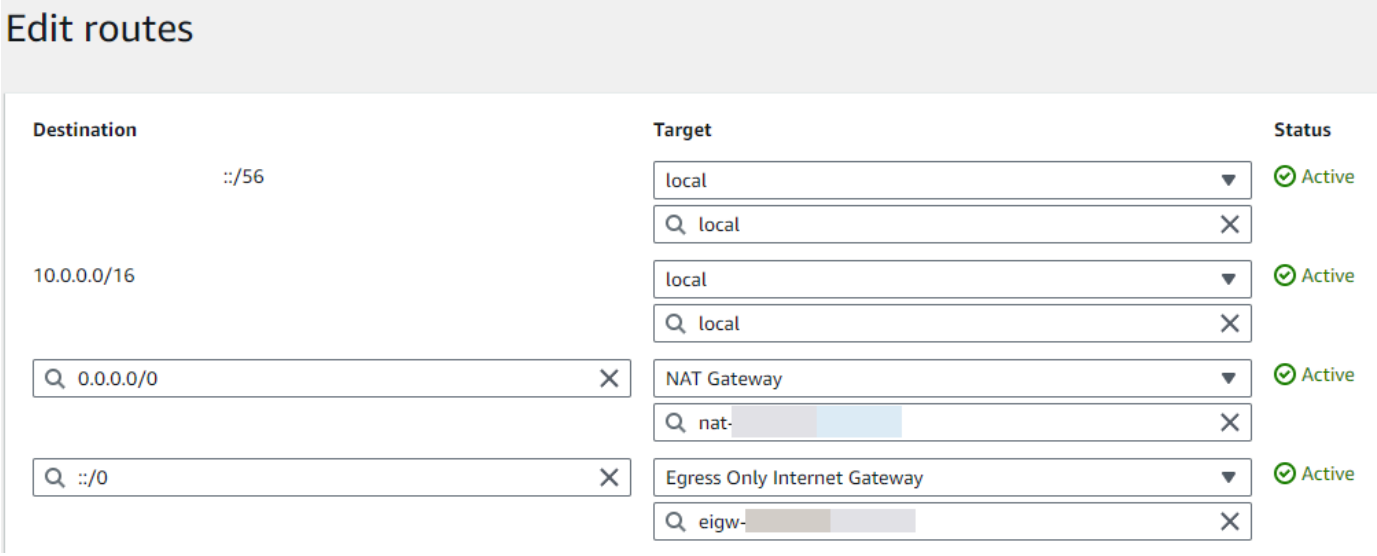
11. Under **Routes**, choose **Edit routes**.



12. Choose **Add route**, and then enter `::/0` in the **Destination** box.



13. For **Target**, select **Egress Only Internet Gateway**, and then choose the gateway that you created earlier.



14. Choose **Save changes**.

Configure the Lambda function

To configure a VPC when you create a function

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Under **Basic information**, for **Function name**, enter a name for your function.
4. Expand **Advanced settings**.
5. Select **Enable VPC**, and then choose a VPC.
6. (Optional) To allow outbound IPv6 traffic, select **Allow IPv6 traffic for dual-stack subnets**.
7. For **Subnets**, select all private subnets. The private subnets can access the internet through the NAT gateway. Connecting a function to a public subnet doesn't give it internet access.

> **Note**
>
> If you selected **Allow IPv6 traffic for dual-stack subnets**, all selected subnets must have an IPv4 CIDR block and an IPv6 CIDR block.

8. For **Security groups**, select a security group that allows outbound traffic.
9. Choose **Create function**.

Lambda automatically creates an execution role with the AWSLambdaVPCAccessExecutionRole AWS managed policy. The permissions in this policy are required only to create elastic network interfaces for the VPC configuration, not to invoke your function. To apply least-privilege permissions, you can remove the **AWSLambdaVPCAccessExecutionRole** policy from your execution role after you create the function and VPC configuration. For more information, see Required IAM permissions.

To configure a VPC for an existing function

To add a VPC configuration to an existing function, the function's execution role must have permission to create and manage elastic network interfaces. The AWSLambdaVPCAccessExecutionRole AWS managed policy includes the required permissions. To apply least-privilege permissions, you can remove the **AWSLambdaVPCAccessExecutionRole** policy from your execution role after you create the VPC configuration.

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the **Configuration** tab, and then choose **VPC**.
4. Under **VPC**, choose **Edit**.
5. Select the VPC.
6. (Optional) To allow outbound IPv6 traffic, select **Allow IPv6 traffic for dual-stack subnets**.
7. For **Subnets**, select all private subnets. The private subnets can access the internet through the NAT gateway. Connecting a function to a public subnet doesn't give it internet access.

   > **Note**
   >
   > If you selected **Allow IPv6 traffic for dual-stack subnets**, all selected subnets must have an IPv4 CIDR block and an IPv6 CIDR block.

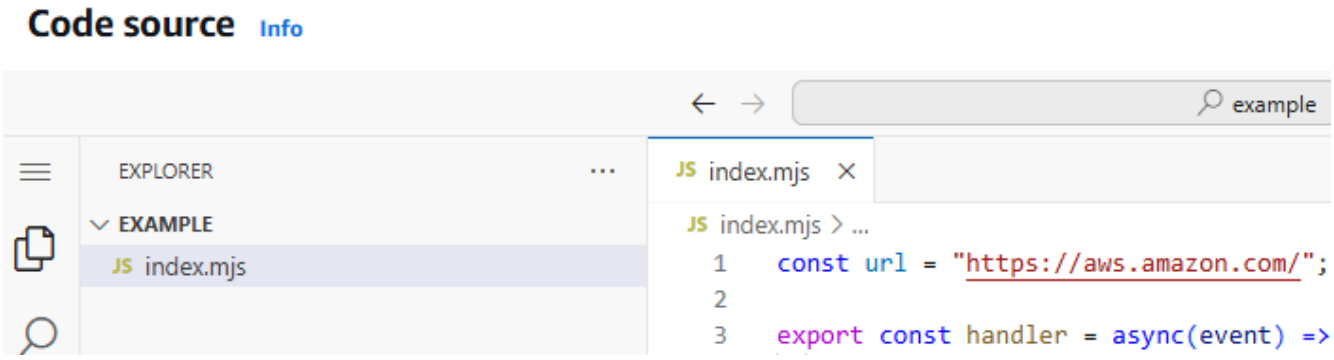
8. For **Security groups**, select a security group that allows outbound traffic.
9. Choose **Save**.

Test the function

Use the following sample code to confirm that your VPC-connected function can reach the public internet. If successful, the code returns a `200` status code. If unsuccessful, the function times out.

Node.js

1. In the **Code source** pane on the Lambda console, paste the following code into the **index.mjs** file. The function makes an HTTP GET request to a public endpoint and returns the HTTP response code to test if the function has access to the public internet.

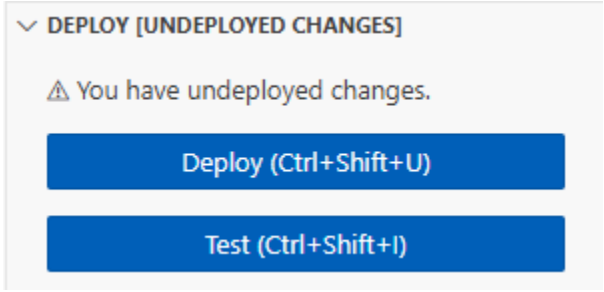


Example— HTTP request with async/await

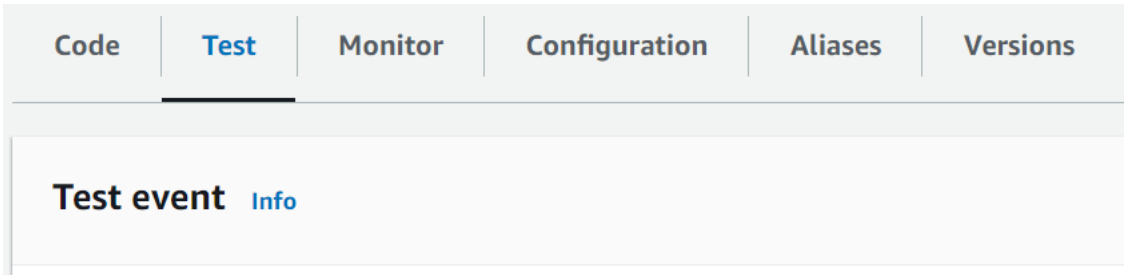
```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

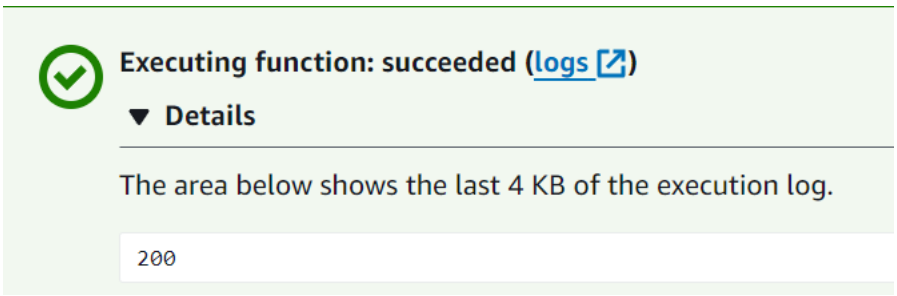
2. In the **DEPLOY** section, choose **Deploy** to update your function's code:



3. Choose the **Test** tab.



4. Choose **Test**.
5. The function returns a `200` status code. This means that the function has outbound internet access.

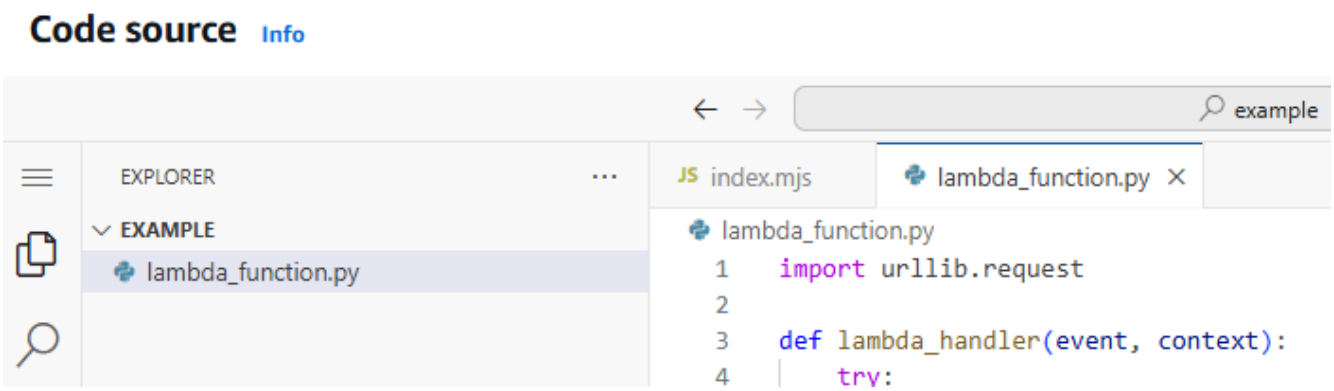


If the function can't reach the public internet, you get an error message like this:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Python

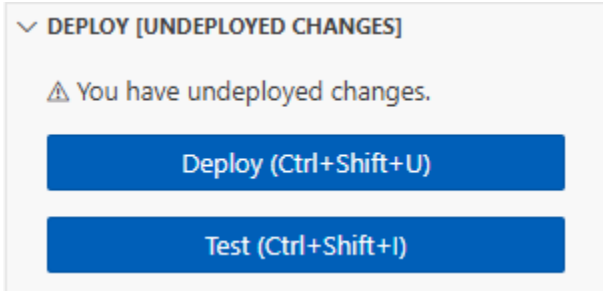
1. In the **Code source** pane on the Lambda console, paste the following code into the **lambda_function.py** file. The function makes an HTTP GET request to a public endpoint and returns the HTTP response code to test if the function has access to the public internet.



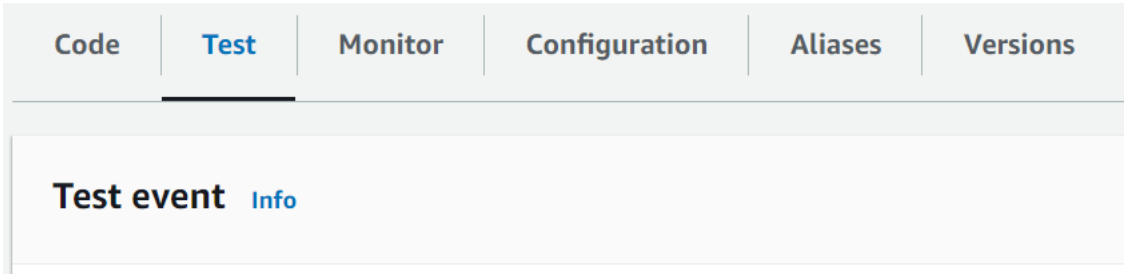

```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

2. In the **DEPLOY** section, choose **Deploy** to update your function's code:

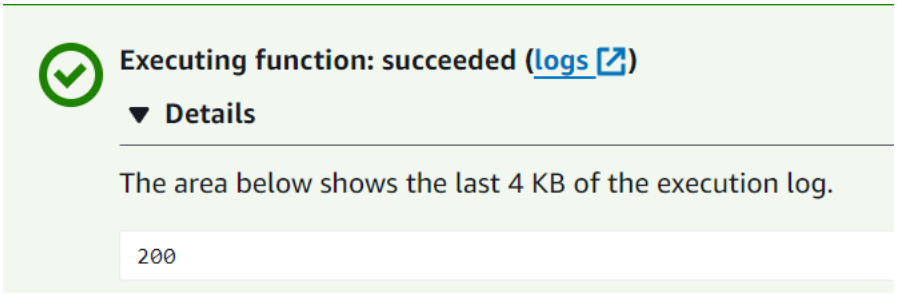


3. Choose the **Test** tab.



4. Choose **Test**.

5. The function returns a `200` status code. This means that the function has outbound internet access.



If the function can't reach the public internet, you get an error message like this:

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Connecting inbound interface VPC endpoints for Lambda

If you use Amazon Virtual Private Cloud (Amazon VPC) to host your AWS resources, you can establish a connection between your VPC and Lambda. You can use this connection to invoke your Lambda function without crossing the public internet.

To establish a private connection between your VPC and Lambda, create an interface VPC endpoint. Interface endpoints are powered by AWS PrivateLink, which enables you to privately access Lambda APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with Lambda APIs. Traffic between your VPC and Lambda does not leave the AWS network.

Each interface endpoint is represented by one or more elastic network interfaces in your subnets. A network interface provides a private IP address that serves as an entry point for traffic to Lambda.

Sections

- Considerations for Lambda interface endpoints
- Creating an interface endpoint for Lambda
- Creating an interface endpoint policy for Lambda

Considerations for Lambda interface endpoints

Before you set up an interface endpoint for Lambda, be sure to review Interface endpoint properties and limitations in the *Amazon VPC User Guide*.

You can call any of the Lambda API operations from your VPC. For example, you can invoke the Lambda function by calling the `Invoke` API from within your VPC. For the full list of Lambda APIs, see Actions in the Lambda API reference.

`use1-az3` is a limited capacity Region for Lambda VPC functions. You shouldn't use subnets in this availability zone with your Lambda functions because this can result in reduced zonal redundancy in the event of an outage.

Keep-alive for persistent connections

Lambda purges idle connections over time, so you must use a keep-alive directive to maintain persistent connections. Attempting to reuse an idle connection when invoking a function results in a connection error. To maintain your persistent connection, use the keep-alive directive associated

with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js in the *AWS SDK for JavaScript Developer Guide*.

Billing Considerations

There is no additional cost to access a Lambda function through an interface endpoint. For more Lambda pricing information, see AWS Lambda Pricing.

Standard pricing for AWS PrivateLink applies to interface endpoints for Lambda. Your AWS account is billed for every hour an interface endpoint is provisioned in each Availability Zone and for data processed through the interface endpoint. For more interface endpoint pricing information, see AWS PrivateLink pricing.

VPC Peering Considerations

You can connect other VPCs to the VPC with interface endpoints using VPC peering. VPC peering is a networking connection between two VPCs. You can establish a VPC peering connection between your own two VPCs, or with a VPC in another AWS account. The VPCs can also be in two different AWS Regions.

Traffic between peered VPCs stays on the AWS network and does not traverse the public internet. Once VPCs are peered, resources like Amazon Elastic Compute Cloud (Amazon EC2) instances, Amazon Relational Database Service (Amazon RDS) instances, or VPC-enabled Lambda functions in both VPCs can access the Lambda API through interface endpoints created in the one of the VPCs.

Creating an interface endpoint for Lambda

You can create an interface endpoint for Lambda using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Creating an interface endpoint in the *Amazon VPC User Guide*.

To create an interface endpoint for Lambda (console)

1. Open the Endpoints page of the Amazon VPC console.
2. Choose **Create Endpoint**.
3. For **Service category**, verify that **AWS services** is selected.
4. For **Service Name**, choose **com.amazonaws.*region*.lambda**. Verify that the **Type** is **Interface**.
5. Choose a VPC and subnets.

6. To enable private DNS for the interface endpoint, select the **Enable DNS Name** check box. We recommend that you enable private DNS names for your VPC endpoints for AWS services. This ensures that requests that use the public service endpoints, such as requests made through an AWS SDK, resolve to your VPC endpoint.
7. For **Security group**, choose one or more security groups.
8. Choose **Create endpoint**.

To use the private DNS option, you must set the `enableDnsHostnames` and `enableDnsSupportattributes` of your VPC. For more information, see Viewing and updating DNS support for your VPC in the *Amazon VPC User Guide*. If you enable private DNS for the interface endpoint, you can make API requests to Lambda using its default DNS name for the Region, for example, `lambda.us-east-1.amazonaws.com`. For more service endpoints, see Service endpoints and quotas in the *AWS General Reference*.

For more information, see Accessing a service through an interface endpoint in the *Amazon VPC User Guide*.

For information about creating and configuring an endpoint using CloudFormation, see the AWS::EC2::VPCEndpoint resource in the *AWS CloudFormation User Guide*.

To create an interface endpoint for Lambda (AWS CLI)

Use the create-vpc-endpoint command and specify the VPC ID, VPC endpoint type (interface), service name, subnets that will use the endpoint, and security groups to associate with the endpoint's network interfaces. For example:

```
aws ec2 create-vpc-endpoint
  --vpc-id vpc-ec43eb89
  --vpc-endpoint-type Interface
  --service-name com.amazonaws.us-east-1.lambda
  --subnet-id subnet-abababab
  --security-group-id sg-1a2b3c4d
```

Creating an interface endpoint policy for Lambda

To control who can use your interface endpoint and which Lambda functions the user can access, you can attach an endpoint policy to your endpoint. The policy specifies the following information:

- The principal that can perform actions.

- The actions that the principal can perform.
- The resources on which the principal can perform actions.

For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

Example: Interface endpoint policy for Lambda actions

The following is an example of an endpoint policy for Lambda. When attached to an endpoint, this policy allows user `MyUser` to invoke the function `my-function`.

> **Note**
>
> You need to include both the qualified and the unqualified function ARN in the resource.

```
{
   "Statement":[
      {
         "Principal":
         {
             "AWS": "arn:aws:iam::111122223333:user/MyUser"
         },
         "Effect":"Allow",
         "Action":[
            "lambda:InvokeFunction"
         ],
         "Resource": [
               "arn:aws:lambda:us-east-2:123456789012:function:my-function",
               "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
            ]
      }
   ]
}
```

Configuring file system access for Lambda functions

You can configure a function to mount an Amazon Elastic File System (Amazon EFS) file system to a local directory. With Amazon EFS, your function code can access and modify shared resources safely and at high concurrency.

Supported Regions

Amazon EFS for Lambda is available in all commercial Regions except Asia Pacific (New Zealand), Asia Pacific (Taipei), Asia Pacific (Malaysia), Mexico (Central), Asia Pacific (Thailand), and Canada West (Calgary).

Sections

- Execution role and user permissions
- Configuring a file system and access point
- Connecting to a file system (console)

Execution role and user permissions

If the file system doesn't have a user-configured AWS Identity and Access Management (IAM) policy, EFS uses a default policy that grants full access to any client that can connect to the file system using a file system mount target. If the file system has a user-configured IAM policy, your function's execution role must have the correct `elasticfilesystem` permissions.

Execution role permissions

- **elasticfilesystem:ClientMount**
- **elasticfilesystem:ClientWrite (not required for read-only connections)**

These permissions are included in the **AmazonElasticFileSystemClientReadWriteAccess** managed policy. Additionally, your execution role must have the permissions required to connect to the file system's VPC.

When you configure a file system, Lambda uses your permissions to verify mount targets. To configure a function to connect to a file system, your user needs the following permissions:

User permissions

- **elasticfilesystem:DescribeMountTargets**

Configuring a file system and access point

Create a file system in Amazon EFS with a mount target in every Availability Zone that your function connects to. For performance and resilience, use at least two Availability Zones. For example, in a simple configuration you could have a VPC with two private subnets in separate Availability Zones. The function connects to both subnets and a mount target is available in each. Ensure that NFS traffic (port 2049) is allowed by the security groups used by the function and mount targets.

> **Note**
>
> When you create a file system, you choose a performance mode that can't be changed later. **General purpose** mode has lower latency, and **Max I/O** mode supports a higher maximum throughput and IOPS. For help choosing, see Amazon EFS performance in the *Amazon Elastic File System User Guide*.

An access point connects each instance of the function to the right mount target for the Availability Zone it connects to. For best performance, create an access point with a non-root path, and limit the number of files that you create in each directory. The following example creates a directory named `my-function` on the file system and sets the owner ID to 1001 with standard directory permissions (755).

Example access point configuration

- **Name** – `files`
- **User ID** – `1001`
- **Group ID** – `1001`
- **Path** – `/my-function`
- **Permissions** – `755`
- **Owner user ID** – `1001`
- **Group user ID** – `1001`

When a function uses the access point, it is given user ID 1001 and has full access to the directory.

For more information, see the following topics in the *Amazon Elastic File System User Guide*:

- Creating resources for Amazon EFS
- Working with users, groups, and permissions

Connecting to a file system (console)

A function connects to a file system over the local network in a VPC. The subnets that your function connects to can be the same subnets that contain mount points for your file system, or subnets in the same Availability Zone that can route NFS traffic (port 2049) to the file system.

> **Note**
>
> If your function is not already connected to a VPC, see Giving Lambda functions access to resources in an Amazon VPC.

To configure file system access

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **File systems**.
4. Under **File system**, choose **Add file system**.
5. Configure the following properties:
   - **EFS file system** – The access point for a file system in the same VPC.
   - **Local mount path** – The location where the file system is mounted on the Lambda function, starting with `/mnt/`.

> **Pricing**
>
> Amazon EFS charges for storage and throughput, with rates that vary by storage class. For details, see Amazon EFS pricing.

Lambda charges for data transfer between VPCs. This only applies if your function's VPC is peered to another VPC with a file system. The rates are the same as for Amazon EC2 data transfer between VPCs in the same Region. For details, see Lambda pricing.

Create an alias for a Lambda function

You can create aliases for your Lambda function. A Lambda alias is a pointer to a function version that you can update. The function's users can access the function version using the alias Amazon Resource Name (ARN). When you deploy a new version, you can update the alias to use the new version, or split traffic between two versions.

Console

To create an alias using the console

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Aliases** and then choose **Create alias**.
4. On the **Create alias** page, do the following:
   a. Enter a **Name** for the alias.
   b. (Optional) Enter a **Description** for the alias.
   c. For **Version**, choose a function version that you want the alias to point to.
   d. (Optional) To configure routing on the alias, expand **Weighted alias**. For more information, see Implement Lambda canary deployments using a weighted alias.
   e. Choose **Save**.

AWS CLI

To create an alias using the AWS Command Line Interface (AWS CLI), use the create-alias command.

```
aws lambda create-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number \
  --description " "
```

To change an alias to point a new version of the function, use the update-alias command.

```
aws lambda update-alias \
  --function-name my-function \
```

```
  --name alias-name \
  --function-version version-number
```

To delete an alias, use the delete-alias command.

```
aws lambda delete-alias \
  --function-name my-function \
  --name alias-name
```

The AWS CLI commands in the preceding steps correspond to the following Lambda API operations:

- CreateAlias
- UpdateAlias
- DeleteAlias

Using Lambda aliases in event sources and permissions policies

Each alias has a unique ARN. An alias can point only to a function version, not to another alias. You can update an alias to point to a new version of the function.

Event sources such as Amazon Simple Storage Service (Amazon S3) invoke your Lambda function. These event sources maintain a mapping that identifies the function to invoke when events occur. If you specify a Lambda function alias in the mapping configuration, you don't need to update the mapping when the function version changes. For more information, see How Lambda processes records from stream and queue-based event sources.

In a resource policy, you can grant permissions for event sources to use your Lambda function. If you specify an alias ARN in the policy, you don't need to update the policy when the function version changes.

Resource policies

You can use a resource-based policy to give a service, resource, or account access to your function. The scope of that permission depends on whether you apply it to an alias, a version, or the entire function. For example, if you use an alias name (such as `helloworld:PROD`), the permission allows you to invoke the `helloworld` function using the alias ARN (`helloworld:PROD`).

If you attempt to invoke the function without an alias or a specific version, then you get a permission error. This permission error still occurs even if you attempt to directly invoke the function version associated with the alias.

For example, the following AWS CLI command grants Amazon S3 permissions to invoke the PROD alias of the `helloworld` function when Amazon S3 is acting on behalf of `amzn-s3-demo-bucket`.

```
aws lambda add-permission \
  --function-name helloworld \
  --qualifier PROD \
  --statement-id 1 \
  --principal s3.amazonaws.com \
  --action lambda:InvokeFunction \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

For more information about using resource names in policies, see Fine-tuning the Resources and Conditions sections of policies.

Implement Lambda canary deployments using a weighted alias

You can use a weighted alias to split traffic between two different versions of the same function. With this approach, you can test new versions of your functions with a small percentage of traffic and quickly roll back if necessary. This is known as a canary deployment. Canary deployments differ from blue/green deployments by exposing the new version to only a portion of requests rather than switching all traffic at once.

You can point an alias to a maximum of two Lambda function versions. The versions must meet the following criteria:

- Both versions must have the same execution role.
- Both versions must have the same dead-letter queue configuration, or no dead-letter queue configuration.
- Both versions must be published. The alias cannot point to $LATEST.

> **Note**
>
> Lambda uses a simple probabilistic model to distribute the traffic between the two function versions. At low traffic levels, you might see a high variance between the configured and actual percentage of traffic on each version. If your function uses provisioned concurrency, you can avoid spillover invocations by configuring a higher number of provisioned concurrency instances during the time that alias routing is active.

Create a weighted alias

Console

To configure routing on an alias using the console

> **Note**
>
> Verify that the function has at least two published versions. To create additional versions, follow the instructions in Creating function versions.

1. Open the Functions page of the Lambda console.

2. Choose a function.
3. Choose **Aliases** and then choose **Create alias**.
4. On the **Create alias** page, do the following:

    a. Enter a **Name** for the alias.

    b. (Optional) Enter a **Description** for the alias.

    c. For **Version**, choose the first function version that you want the alias to point to.

    d. Expand **Weighted alias**.

    e. For **Additional version**, choose the second function version that you want the alias to point to.

    f. For **Weight (%)**, enter a weight value for the function. *Weight* is the percentage of traffic that is assigned to that version when the alias is invoked. The first version receives the residual weight. For example, if you specify 10 percent to **Additional version**, the first version is assigned 90 percent automatically.

    g. Choose **Save**.

AWS CLI

Use the create-alias and update-alias AWS CLI commands to configure the traffic weights between two function versions. When you create or update the alias, you specify the traffic weight in the `routing-config` parameter.

The following example creates a Lambda function alias named **routing-alias** that points to version 1 of the function. Version 2 of the function receives 3 percent of the traffic. The remaining 97 percent of traffic is routed to version 1.

```
aws lambda create-alias \
  --name routing-alias \
  --function-name my-function \
  --function-version 1  \
  --routing-config AdditionalVersionWeights={"2"=0.03}
```

Use the `update-alias` command to increase the percentage of incoming traffic to version 2. In the following example, you increase the traffic to 5 percent.

```
aws lambda update-alias \
  --name routing-alias \
```

```
  --function-name my-function \
  --routing-config AdditionalVersionWeights={"2"=0.05}
```

To route all traffic to version 2, use the `update-alias` command to change the `function-version` property to point the alias to version 2. The command also resets the routing configuration.

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function  \
  --function-version 2 \
  --routing-config AdditionalVersionWeights={}
```

The AWS CLI commands in the preceding steps correspond to the following Lambda API operations:

- CreateAlias
- UpdateAlias

Determining which version was invoked

When you configure traffic weights between two function versions, there are two ways to determine the Lambda function version that has been invoked:

- **CloudWatch Logs** – Lambda automatically emits a START log entry that contains the invoked version ID for every function invocation. Example:

  `START RequestId: 1dh194d3759ed-4v8b-a7b4-1e541f60235f Version: 2`

  For alias invocations, Lambda uses the `ExecutedVersion` dimension to filter the metric data by the invoked version. For more information, see Viewing metrics for Lambda functions.
- **Response payload (synchronous invocations)** – Responses to synchronous function invocations include an `x-amz-executed-version` header to indicate which function version has been invoked.

Create a rolling deployment with weighted aliases

Use AWS CodeDeploy and AWS Serverless Application Model (AWS SAM) to create a rolling deployment that automatically detects changes to your function code, deploys a new version of

your function, and gradually increase the amount of traffic flowing to the new version. The amount of traffic and rate of increase are parameters that you can configure.

In a rolling deployment, AWS SAM performs these tasks:

- Configures your Lambda function and creates an alias. The weighted alias routing configuration is the underlying capability that implements the rolling deployment.
- Creates a CodeDeploy application and deployment group. The deployment group manages the rolling deployment and the rollback, if needed.
- Detects when you create a new version of your Lambda function.
- Triggers CodeDeploy to start the deployment of the new version.

Example AWS SAM template

The following example shows an AWS SAM template for a simple rolling deployment.

```
AWSTemplateFormatVersion : '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A sample SAM template for deploying Lambda functions

Resources:
# Details about the myDateTimeFunction Lambda function
  myDateTimeFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: myDateTimeFunction.handler
      Runtime: nodejs24.x
# Creates an alias named "live" for the function, and automatically publishes when you
 update the function.
      AutoPublishAlias: live
      DeploymentPreference:
# Specifies the deployment configuration
          Type: Linear10PercentEvery2Minutes
```

This template defines a Lambda function named `myDateTimeFunction` with the following properties.

AutoPublishAlias

The `AutoPublishAlias` property creates an alias named `live`. In addition, the AWS SAM framework automatically detects when you save new code for the function. The framework then publishes a new function version and updates the `live` alias to point to the new version.

DeploymentPreference

The `DeploymentPreference` property determines the rate at which the CodeDeploy application shifts traffic from the original version of the Lambda function to the new version. The value `Linear10PercentEvery2Minutes` shifts an additional ten percent of the traffic to the new version every two minutes.

For a list of the predefined deployment configurations, see Deployment configurations.

For more information on how to create rolling deployments with CodeDeploy and AWS SAM, see the following:

- Tutorial: Deploy an updated Lambda function with CodeDeploy and the AWS Serverless Application Model
- Deploying serverless applications gradually with AWS SAM

Manage Lambda function versions

You can use versions to manage the deployment of your functions. For example, you can publish a new version of a function for beta testing without affecting users of the stable production version. Lambda creates a new version of your function each time that you publish the function. The new version is a copy of the unpublished version of the function. The unpublished version is named $LATEST.

Importantly, any time you deploy your function code, you overwrite the current code in $LATEST. To save the current iteration of $LATEST, create a new function version. If $LATEST is identical to a previously published version, you won't be able to create a new version until you deploy changes to $LATEST. These changes can include updating the code, or modifying the function configuration settings.

After you publish a function version, its code, runtime, architecture, memory, layers, and most other configuration settings are immutable. This means that you can't change these settings without publishing a new version from $LATEST. You can configure the following items for a published function version:

- Triggers
- Destinations
- Provisioned concurrency
- Asynchronous invocation
- Database connections and proxies

> **Note**
>
> When using runtime management controls with **Auto** mode, the runtime version used by the function version is updated automatically. When using **Function update** or **Manual** mode, the runtime version is not updated. For more information, see the section called "Runtime version updates".

Sections

- Creating function versions
- Using versions

- Granting permissions

Creating function versions

You can change the function code and settings only on the unpublished version of a function. When you publish a version, Lambda locks the code and most of the settings to maintain a consistent experience for users of that version.

You can create a function version using the Lambda console.

To create a new function version

1. Open the Functions page of the Lambda console.
2. Choose a function and then choose the **Versions** tab.
3. On the versions configuration page, choose **Publish new version**.
4. (Optional) Enter a version description.
5. Choose **Publish**.

Alternatively, you can publish a version of a function using the PublishVersion API operation.

The following AWS CLI command publishes a new version of a function. The response returns configuration information about the new version, including the version number and the function ARN with the version suffix.

```
aws lambda publish-version --function-name my-function
```

You should see the following output:

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
  "Version": "1",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs24.x",
  ...
}
```

> **Note**
>
> Lambda assigns monotonically increasing sequence numbers for versioning. Lambda never reuses version numbers, even after you delete and recreate a function.

Using versions

You can reference your Lambda function using either a qualified ARN or an unqualified ARN.

- **Qualified ARN** – The function ARN with a version suffix. The following example refers to version 42 of the `helloworld` function.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:42
```

- **Unqualified ARN** – The function ARN without a version suffix.

```
arn:aws:lambda:aws-region:acct-id:function:helloworld
```

You can use a qualified or an unqualified ARN in all relevant API operations. However, you can't use an unqualified ARN to create an alias.

If you decide not to publish function versions, you can invoke the function using either the qualified or unqualified ARN in your event source mapping. When you invoke a function using an unqualified ARN, Lambda implicitly invokes $LATEST.

The qualified ARN for each Lambda function version is unique. After you publish a version, you can't change the ARN or the function code.

Lambda publishes a new function version only if the code has never been published, or if the code has changed from the last published version. If there is no change, the function version remains at the last published version.

When you publish a version, Lambda creates an immutable snapshot of your function's code and configuration. Not all configuration changes trigger the publication of a new version. The following configuration changes qualify a function for version publication:

- Function code
- Environment variables

- Runtime
- Handler
- Layers
- Memory size
- Timeout
- VPC configuration
- Dead Letter Queue (DLQ) configuration
- IAM role
- Description
- Architecture (x86_64 or arm64)
- Ephemeral storage size
- Package type
- Logging configuration
- File system configuration
- SnapStart
- Tracing configuration

Operational settings such as reserved concurrency don't trigger the publication of a new version when changed.

Granting permissions

You can use a resource-based policy or an identity-based policy to grant access to your function. The scope of the permission depends on whether you apply the policy to a function or to one version of a function. For more information about function resource names in policies, see Fine-tuning the Resources and Conditions sections of policies.

You can simplify the management of event sources and AWS Identity and Access Management (IAM) policies by using function aliases. For more information, see Create an alias for a Lambda function.

Using tags on Lambda functions

You can tag functions to organize and manage your resources. Tags are free-form key-value pairs associated with your resources that are supported across AWS services. For more information about use cases for tags, see Common tagging strategies in the *Tagging AWS Resources and Tag Editor Guide*.

Tags apply at the function level, not to versions or aliases. Tags are not part of the version-specific configuration that AWS Lambda creates a snapshot of when you publish a version. You can use the Lambda API to view and update tags. You can also view and update tags while managing a specific function in the Lambda console.

Sections

- Permissions required for working with tags
- Using tags with the Lambda console
- Using tags with the AWS CLI

Permissions required for working with tags

To allow an AWS Identity and Access Management (IAM) identity (user, group, or role) to read or set tags on a resource, grant it the corresponding permissions:

- **lambda:ListTags**–When a resource has tags, grant this permission to anyone who needs to call `ListTags` on it. For tagged functions, this permission is also necessary for `GetFunction`.
- **lambda:TagResource**–Grant this permission to anyone who needs to call `TagResource` or perform a tag on create.

Optionally, consider granting the **lambda:UntagResource** permission as well to allow `UntagResource` calls to the resource.

For more information, see Identity-based IAM policies for Lambda.

Using tags with the Lambda console

You can use the Lambda console to create functions that have tags, add tags to existing functions, and filter functions by tags that you add.

To add tags when you create a function

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch** or **Container image**.
4. Under **Basic information**, set up your function. For more information about configuring functions, see *Configuring functions*.
5. Expand **Advanced settings**, and then select **Enable tags**.
6. Choose **Add new tag**, and then enter a **Key** and an optional **Value**. To add more tags, repeat this step.
7. Choose **Create function**.

To add tags to an existing function

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose **Configuration**, and then choose **Tags**.
4. Under **Tags**, choose **Manage tags**.
5. Choose **Add new tag**, and then enter a **Key** and an optional **Value**. To add more tags, repeat this step.
6. Choose **Save**.

To filter functions with tags

1. Open the Functions page of the Lambda console.
2. Choose the search box to see a list of function properties and tag keys.
3. Choose a tag key to see a list of values that are in use in the current AWS Region.
4. Select **Use: "tag-name"** to see all functions tagged with this key, or choose an **Operator** to further filter by value.
5. Select your tag value to filter by a combination of tag key and value.

The search bar also supports searching for tag keys. Enter `tag` to see only a list of tag keys, or enter the name of a key to find it in the list.

Using tags with the AWS CLI

You can add and remove tags on existing Lambda resources, including functions, with the Lambda API. You can also add tags when creating a function, which allows you to keep a resource tagged through its entire lifecycle.

Updating tags with the Lambda tag APIs

You can add and remove tags for supported Lambda resources through the TagResource and UntagResource API operations.

You can call these operations using the AWS CLI. To add tags to an existing resource, use the `tag-resource` command. This example adds two tags, one with the key *Department* and one with the key *CostCenter*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

To remove tags, use the `untag-resource` command. This example removes the tag with the key *Department*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Adding tags when creating a function

To create a new Lambda function with tags, use the CreateFunction API operation. Specify the `Tags` parameter. You can call this operation with the `create-function` CLI command and the --tags option. Before using the tags parameter with `CreateFunction`, ensure that your role has permission to tag resources alongside the usual permissions needed for this operation. For more information about permissions for tagging, see the section called “Permissions required for working with tags”. This example adds two tags, one with the key *Department* and one with the key *CostCenter*.

```
aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs24.x \
--role arn:aws:iam::123456789012:role/lambda-role \
```

```
--tags Department=Marketing,CostCenter=1234ABCD
```

Viewing tags on a function

To view the tags that are applied to a specific Lambda resource, use the `ListTags` API operation. For more information, see ListTags.

You can call this operation with the `list-tags` AWS CLI command by providing an ARN (Amazon Resource Name).

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

You can view the tags that are applied to a specific resource with the GetFunction API operation. Comparable functionality is not available for other resource types.

You can call this operation with the `get-function` CLI command:

```
aws lambda get-function --function-name my-function
```

Filtering resources by tag

You can use the AWS Resource Groups Tagging API GetResources API operation to filter your resources by tags. The `GetResources` operation receives up to 10 filters, with each filter containing a tag key and up to 10 tag values. You provide `GetResources` with a `ResourceType` to filter by specific resource types.

You can call this operation using the `get-resources` AWS CLI command. For examples of using `get-resources`, see get-resources in the *AWS CLI Command Reference*.

Response streaming for Lambda functions

Lambda functions can natively stream response payloads back to clients through Lambda function URLs or by using the InvokeWithResponseStream API (via the AWS SDK or direct API calls). Your Lambda function can also stream response payloads through the Amazon API Gateway proxy integration, which uses the InvokeWithResponseStream API to invoke your function. Response streaming can benefit latency sensitive applications by improving time to first byte (TTFB) performance. This is because you can send partial responses back to the client as they become available. Additionally, response streaming functions can return payloads up to 200 MB, compared to the 6 MB maximum for buffered responses. Streaming a response also means that your function doesn't need to fit the entire response in memory. For very large responses, this can reduce the amount of memory you need to configure for your function.

> **Note**
>
> Lambda response streaming is not yet available in all AWS Regions. Please refer to Builder Center's AWS Capabilities by Region for feature availability by Region.

The speed at which Lambda streams your responses depends on the response size. The streaming rate for the first 6 MB of your function's response is uncapped. For responses larger than 6 MB, the remainder of the response is subject to a bandwidth cap. For more information on streaming bandwidth, see Bandwidth limits for response streaming.

Streaming responses incur cost and streamed responses are not interrupted or stopped when the invoking client connection is broken. Customers will be billed for the full function duration, so customers should exercise caution when configuring long function timeouts.

Lambda supports response streaming on Node.js managed runtimes. For other languages, including Python, you can use a custom runtime with a custom Runtime API integration to stream responses or use the Lambda Web Adapter.

> **Note**
>
> When testing your function through the Lambda console, you'll always see responses as buffered.

Topics

- Bandwidth limits for response streaming
- VPC compatibility with response streaming
- Writing response streaming-enabled Lambda functions
- Invoking a response streaming enabled function using Lambda function URLs
- Tutorial: Creating a response streaming Lambda function with a function URL

Bandwidth limits for response streaming

The first 6 MB of your function's response payload has uncapped bandwidth. After this initial burst, Lambda streams your response at a maximum rate of 2 MBps. If your function responses never exceed 6 MB, then this bandwidth limit never applies.

> **Note**
>
> Bandwidth limits only apply to your function's response payload, and not to network access by your function.

The rate of uncapped bandwidth varies depending on a number of factors, including your function's processing speed. You can normally expect a rate higher than 2 MBps for the first 6 MB of your function's response. If your function is streaming a response to a destination outside of AWS, the streaming rate also depends on the speed of the external internet connection.

VPC compatibility with response streaming

When using Lambda functions in a VPC environment, there are important considerations for response streaming:

- Lambda function URLs do not support response streaming within a VPC environment.
- You can use response streaming within a VPC by invoking your Lambda function through the AWS SDK using the `InvokeWithResponseStream` API. This requires setting up the appropriate VPC endpoints for Lambda.
- For VPC environments, you'll need to create an interface VPC endpoint for Lambda to enable communication between your resources in the VPC and the Lambda service.

A typical architecture for response streaming in a VPC might include:

```
Client in VPC -> Interface VPC endpoint for Lambda -> Lambda function -> Response
 streaming back through the same path
```

Writing response streaming-enabled Lambda functions

Writing the handler for response streaming functions is different than typical handler patterns. When writing streaming functions, be sure to do the following:

- Wrap your function with the `awslambda.streamifyResponse()` decorator. The `awslambda` global object is provided by Lambda's Node.js runtime environment.
- End the stream gracefully to ensure that all data processing is complete.

Configuring a handler function to stream responses

To indicate to the runtime that Lambda should stream your function's responses, you must wrap your function with the `streamifyResponse()` decorator. This tells the runtime to use the proper logic path for streaming responses and enables the function to stream responses.

The `streamifyResponse()` decorator accepts a function that accepts the following parameters:

- `event` – Provides information about the function URL's invocation event, such as the HTTP method, query parameters, and the request body.
- `responseStream` – Provides a writable stream.
- `context` – Provides methods and properties with information about the invocation, function, and execution environment.

The `responseStream` object is a Node.js `writableStream`. As with any such stream, you should use the `pipeline()` method.

> **Note**
>
> The `awslambda` global object is automatically provided by Lambda's Node.js runtime and no import is required.

Example response streaming-enabled handler

```
import { pipeline } from 'node:stream/promises';
import { Readable } from 'node:stream';

export const echo = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  // As an example, convert event to a readable stream.
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));

  await pipeline(requestStream, responseStream);
});
```

While `responseStream` offers the `write()` method to write to the stream, we recommend that you use `pipeline()` wherever possible. Using `pipeline()` ensures that the writable stream is not overwhelmed by a faster readable stream.

Ending the stream

Make sure that you properly end the stream before the handler returns. The `pipeline()` method handles this automatically.

For other use cases, call the `responseStream.end()` method to properly end a stream. This method signals that no more data should be written to the stream. This method isn't required if you write to the stream with `pipeline()` or `pipe()`.

Starting with Node.js 24, Lambda no longer waits for unresolved promises to complete after your handler returns or the response stream ends. If your function depends on additional asynchronous operations, such as timers or fetches, you should `await` them in your handler.

Example ending a stream with pipeline()

```
import { pipeline } from 'node:stream/promises';

export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  await pipeline(requestStream, responseStream);
});
```

Example ending a stream without pipeline()

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

Invoking a response streaming enabled function using Lambda function URLs

> **Note**
>
> Your Lambda function can now stream response payloads through the Amazon API Gateway proxy integration.

You can invoke response streaming enabled functions by changing the invoke mode of your function's URL. The invoke mode determines which API operation Lambda uses to invoke your function. The available invoke modes are:

- `BUFFERED` – This is the default option. Lambda invokes your function using the `Invoke` API operation. Invocation results are available when the payload is complete. The maximum payload size is 6 MB.
- `RESPONSE_STREAM` – Enables your function to stream payload results as they become available. Lambda invokes your function using the `InvokeWithResponseStream` API operation. The maximum response payload size is 200 MB.

You can still invoke your function without response streaming by directly calling the `Invoke` API operation. However, Lambda streams all response payloads for invocations that come through the function's URL until you change the invoke mode to `BUFFERED`.

Console

To set the invoke mode of a function URL (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to set the invoke mode for.
3. Choose the **Configuration** tab, and then choose **Function URL**.
4. Choose **Edit**, then choose **Additional settings**.
5. Under **Invoke mode**, choose your desired invoke mode.
6. Choose **Save**.

AWS CLI

To set the invoke mode of a function's URL (AWS CLI)

```
aws lambda update-function-url-config \
  --function-name my-function \
  --invoke-mode RESPONSE_STREAM
```

CloudFormation

To set the invoke mode of a function's URL (CloudFormation)

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

For more information about configuring function URLs, see Lambda function URLs.

Tutorial: Creating a response streaming Lambda function with a function URL

In this tutorial, you create a Lambda function defined as a .zip file archive with a function URL endpoint that returns a response stream. For more information about configuring function URLs, see Function URLs.

Prerequisites

This tutorial assumes that you have some knowledge of basic Lambda operations and the Lambda console. If you haven't already, follow the instructions in Create a Lambda function with the console to create your first Lambda function.

To complete the following steps, you need the AWS CLI version 2. Commands and the expected output are listed in separate blocks:

```
aws --version
```

You should see the following output:

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

For long commands, an escape character (\) is used to split a command over multiple lines.

On Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux. Example CLI commands in this guide use Linux formatting. Commands which include inline JSON documents must be reformatted if you are using the Windows CLI.

Create an execution role

Create the execution role that gives your Lambda function permission to access AWS resources.

To create an execution role

1. Open the Roles page of the AWS Identity and Access Management (IAM) console.
2. Choose **Create role**.
3. Create a role with the following properties:
   - **Trusted entity type – AWS service**
   - **Use case – Lambda**

- **Permissions** – **AWSLambdaBasicExecutionRole**
- **Role name** – **`response-streaming-role`**

The **AWSLambdaBasicExecutionRole** policy has the permissions that the function needs to write logs to Amazon CloudWatch Logs. After you create the role, note down the its Amazon Resource Name (ARN). You'll need it in the next step.

Create a response streaming function (AWS CLI)

Create a response streaming Lambda function with a function URL endpoint using the AWS Command Line Interface (AWS CLI).

To create a function that can stream responses

1. Copy the following code example into a file named `index.js`. This function streams three responses, separated by 1 second.

```
exports.handler = awslambda.streamifyResponse(
 async (event, responseStream, _context) => {
  // Metadata is a JSON serializable JS object. Its shape is not defined here.
  const metadata = {
  statusCode: 200,
  headers: {
   "Content-Type": "application/json",
   "CustomHeader": "outerspace"
  }
  };

  // Assign to the responseStream parameter to prevent accidental reuse of the non-
wrapped stream.
  responseStream = awslambda.HttpResponseStream.from(responseStream, metadata);

  responseStream.write("Streaming with Helper \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 0 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 1 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 2 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.end();
```

```
  await responseStream.finished();
 }
  );
```

2. Create a deployment package.

```
zip function.zip index.js
```

3. Create a Lambda function with the `create-function` command. Replace the value of `--role` with the role ARN from the previous step. This command sets the function timeout to 10 seconds, which allows the function to stream three responses.

```
aws lambda create-function \
  --function-name my-streaming-function \
  --runtime nodejs24.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
  --timeout 10 \
  --role arn:aws:iam::123456789012:role/response-streaming-role
```

To create a function URL

1. Add a resource-based policy to your function that grants `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` permissions. Each statement must be added in a separate command. Replace the value of `--principal` with your AWS account ID.

```
aws lambda add-permission \
  --function-name my-streaming-function \
  --action lambda:InvokeFunctionUrl \
  --statement-id UrlPolicyInvokeURL \
  --principal 123456789012 \
  --function-url-auth-type AWS_IAM
```

```
aws lambda add-permission \
    --function-name my-streaming-function \
    --action lambda:InvokeFunction \
    --statement-id UrlPolicyInvokeFunction \
    --principal 123456789012
```

2. Create a URL endpoint for the function with the `create-function-url-config` command.

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

> **Note**
>
> If you get an error about `--invoke-mode`, you might need to upgrade to a newer version of the AWS CLI.

Test the function URL endpoint

Test your integration by invoking your function. You can open your function's URL in a browser, or you can use curl.

```
curl --request GET "https://abcdefghijklm7nop7qrs740abcd.lambda-url.us-east-1.on.aws/"
 --user "AKIAIOSFODNN7EXAMPLE" --aws-sigv4 "aws:amz:us-east-1:lambda" --no-buffer
```

Our function URL uses the `IAM_AUTH` authentication type. This means that you need to sign requests with both your AWS access key and secret key. In the previous command, replace `AKIAIOSFODNN7EXAMPLE` with the AWS access key ID. Enter your AWS secret key when prompted. If you don't have your AWS secret key, you can use temporary AWS credentials instead.

You should see a response like this:

```
Streaming with Helper
Hello 0
Hello 1
Hello 2
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

Using the Lambda metadata endpoint

The Lambda metadata endpoint lets your functions discover which Availability Zone (AZ) they are running in, enabling you to optimize latency by routing to same-AZ resources like Amazon ElastiCache and Amazon RDS endpoints, and to implement AZ-aware resilience patterns.

The endpoint returns metadata in a simple JSON format through a localhost HTTP API within the execution environment and is accessible to both runtimes and extensions.

Sections

- Getting started
- Understanding Availability Zone IDs
- API reference

Getting started

Powertools for AWS Lambda provides a utility for accessing the Lambda metadata endpoint in Python, TypeScript, Java, and .NET. The utility caches the response after the first call and handles SnapStart cache invalidation automatically.

Use the Powertools for AWS Lambda metadata utility or call the metadata endpoint directly

Python

Install the Powertools package:

```
pip install "aws-lambda-powertools"
```

Use the metadata utility in your handler:

Example Retrieving AZ ID with Powertools (Python)

```
from aws_lambda_powertools.utilities.lambda_metadata import get_lambda_metadata

def handler(event, context):
    metadata = get_lambda_metadata()
    az_id = metadata.availability_zone_id  # e.g., "use1-az1"

    return {"az_id": az_id}
```

TypeScript

Install the Powertools package:

```
npm install @aws-lambda-powertools/commons
```

Use the metadata utility in your handler:

Example Retrieving AZ ID with Powertools (TypeScript)

```
import { getMetadata } from '@aws-lambda-powertools/commons/utils/metadata';

const metadata = await getMetadata();

export const handler = async () => {
  const { AvailabilityZoneID: azId } = metadata;
  return azId;
};
```

Java

Add the Powertools dependency to your `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>software.amazon.lambda</groupId>
        <artifactId>powertools-lambda-metadata</artifactId>
        <version>2.10.0</version>
    </dependency>
</dependencies>
```

Use the metadata client in your handler:

Example Retrieving AZ ID with Powertools (Java)

```
import software.amazon.lambda.powertools.metadata.LambdaMetadata;
import software.amazon.lambda.powertools.metadata.LambdaMetadataClient;

public class App implements RequestHandler<Object, String> {

    @Override
    public String handleRequest(Object input, Context context) {
```

```
        LambdaMetadata metadata = LambdaMetadataClient.get();
        String azId = metadata.getAvailabilityZoneId(); // e.g., "use1-az1"

        return "{\"azId\": \"" + azId + "\"}";
    }
}
```

.NET

Install the Powertools package:

```
dotnet add package AWS.Lambda.Powertools.Metadata
```

Use the metadata class in your handler:

Example Retrieving AZ ID with Powertools (.NET)

```
using AWS.Lambda.Powertools.Metadata;

public class Function
{
    public string Handler(object input, ILambdaContext context)
    {
        var azId = LambdaMetadata.AvailabilityZoneId;
        return $"Running in AZ: {azId}";
    }
}
```

All Runtimes

All Lambda runtimes support the metadata endpoint, including custom runtimes and container images. Use the following example to access the metadata API directly from your function using the environment variables that Lambda automatically sets in the execution environment.

Example Accessing the metadata endpoint directly

```
# Variables are automatically set by Lambda
METADATA_ENDPOINT="http://${AWS_LAMBDA_METADATA_API}/2026-01-15/metadata/execution-
environment"

# Make the request
```

```
RESPONSE=$(curl -s -H "Authorization: Bearer ${AWS_LAMBDA_METADATA_TOKEN}"
 "$METADATA_ENDPOINT")

# Parse the AZ ID
AZ_ID=$(echo "$RESPONSE" | jq -r '.AvailabilityZoneID')

echo "Function is running in AZ ID: $AZ_ID"
```

Understanding Availability Zone IDs

AZ IDs (for example, `use1-az1`) always refer to the same physical location across all AWS accounts, while AZ names (for example, `us-east-1a`) may map to different physical infrastructure in each AWS account in certain regions. For more information, see AZ IDs for cross-account consistency.

Converting AZ ID to AZ name:

To convert an AZ ID to an AZ name, use the Amazon EC2 DescribeAvailabilityZones API. To use this API, add the `ec2:DescribeAvailabilityZones` permission to your function's execution role.

API reference

Environment variables

Lambda automatically sets the following environment variables in every execution environment:

- `AWS_LAMBDA_METADATA_API` – The metadata server address in the format `{ipv4_address}:{port}` (for example, `169.254.100.1:9001`).
- `AWS_LAMBDA_METADATA_TOKEN` – A unique authentication token for the current execution environment. Lambda generates this token automatically at initialization. Include it in all metadata API requests.

Endpoint

`GET http://${AWS_LAMBDA_METADATA_API}/2026-01-15/metadata/execution-environment`

Request

Required headers:

- `Authorization` – The token value from the `AWS_LAMBDA_METADATA_TOKEN` environment variable with the Bearer scheme: `Bearer <token>`. This token-based authentication provides defense in depth protection against Server-Side Request Forgery (SSRF) vulnerabilities. Each execution environment receives a unique, randomly generated token at initialization.

Response

Status: `200 OK`

Content-Type: `application/json`

Cache-Control: `private, max-age=43200, immutable`

The response is immutable within an execution environment. Clients should cache the response and respect the `Cache-Control` TTL. For SnapStart functions, the TTL is reduced during initialization so that clients refresh metadata after restore when the execution environment may be in a different AZ. If you use Powertools, caching and SnapStart invalidation are handled automatically.

Body:

```
{
  "AvailabilityZoneID": "use1-az1"
}
```

The `AvailabilityZoneID` field contains the unique identifier for the Availability Zone where the execution environment is running.

> **Note**
>
> Additional fields may be added to the response in future updates. Clients should ignore unknown fields and not fail if new fields appear.

Error responses

- **401 Unauthorized** – The `Authorization` header is missing or contains an invalid token. Verify you are passing `Bearer ${AWS_LAMBDA_METADATA_TOKEN}`.
- **405 Method Not Allowed** – Request method is not `GET`.

- **500 Internal Server Error** – Server-side processing error.

Understanding Lambda function invocation methods

After you deploy your Lambda function, you can invoke it in several ways:

- The Lambda console – Use the Lambda console to quickly create a test event to invoke your function.
- The AWS SDK – Use the AWS SDK to programmatically invoke your function.
- The Invoke API – Use the Lambda Invoke API to directly invoke your function.
- The AWS Command Line Interface (AWS CLI) – Use the `aws lambda invoke` AWS CLI command to directly invoke your function from the command line.
- A function URL HTTP(S) endpoint – Use function URLs to create a dedicated HTTP(S) endpoint that you can use to invoke your function.

All of these methods are *direct* ways to invoke your function. In Lambda, a common use case is to invoke your function based on an event that occurs elsewhere in your application. Some services can invoke a Lambda function with each new event. This is called a trigger. For stream and queue-based services, Lambda invokes the function with batches of records. This is called an event source mapping.

When you invoke a function, you can choose to invoke it synchronously or asynchronously. With synchronous invocation, you wait for the function to process the event and return a response. With asynchronous invocation, Lambda queues the event for processing and returns a response immediately. The `InvocationType` request parameter in the Invoke API determines how Lambda invokes your function. A value of `RequestResponse` indicates synchronous invocation, and a value of `Event` indicates asynchronous invocation.

To invoke your function over IPv6, use Lambda's public dual-stack endpoints. Dual-stack endpoints support both IPv4 and IPv6. Lambda dual-stack endpoints use the following syntax:

```
protocol://lambda.us-east-1.api.aws
```

You can also use Lambda function URLs to invoke functions over IPv6. Function URL endpoints have the following format:

```
https://url-id.lambda-url.us-east-1.on.aws
```

If the function invocation results in an error, for synchronous invocations, view the error message in the response and retry the invocation manually. For asynchronous invocations, Lambda handles retries automatically and can send invocation records to a destination.

Invoke a Lambda function synchronously

When you invoke a function synchronously, Lambda runs the function and waits for a response. When the function completes, Lambda returns the response from the function's code with additional data, such as the version of the function that was invoked. To invoke a function synchronously with the AWS CLI, use the `invoke` command.

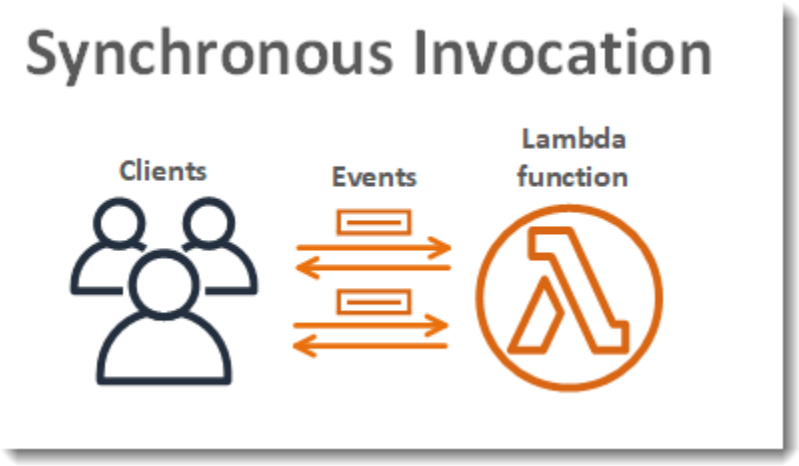
```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

The following diagram shows clients invoking a Lambda function synchronously. Lambda sends the events directly to the function and sends the function's response back to the invoker.



The `payload` is a string that contains an event in JSON format. The name of the file where the AWS CLI writes the response from the function is `response.json`. If the function returns an

object or error, the response body is the object or error in JSON format. If the function exits without error, the response body is `null`.

> **Note**
>
> Lambda does not wait for external extensions to complete before sending the response. External extensions run as independent processes in the execution environment and continue to run after the function invocation is complete. For more information, see Augment Lambda functions using Lambda extensions.

The output from the command, which is displayed in the terminal, includes information from headers in the response from Lambda. This includes the version that processed the event (useful when you use aliases), and the status code returned by Lambda. If Lambda was able to run the function, the status code is 200, even if the function returned an error.

> **Note**
>
> For functions with a long timeout, your client might be disconnected during synchronous invocation while it waits for a response. Configure your HTTP client, SDK, firewall, proxy, or operating system to allow for long connections with timeout or keep-alive settings.

If Lambda isn't able to run the function, the error is displayed in the output.

```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload value response.json
```

You should see the following output:

```
An error occurred (InvalidRequestContentException) when calling the Invoke operation:
 Could not parse request body into json: Unrecognized token 'value': was expecting
 ('true', 'false' or 'null')
 at [Source: (byte[])"value"; line: 1, column: 11]
```

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
```

```
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

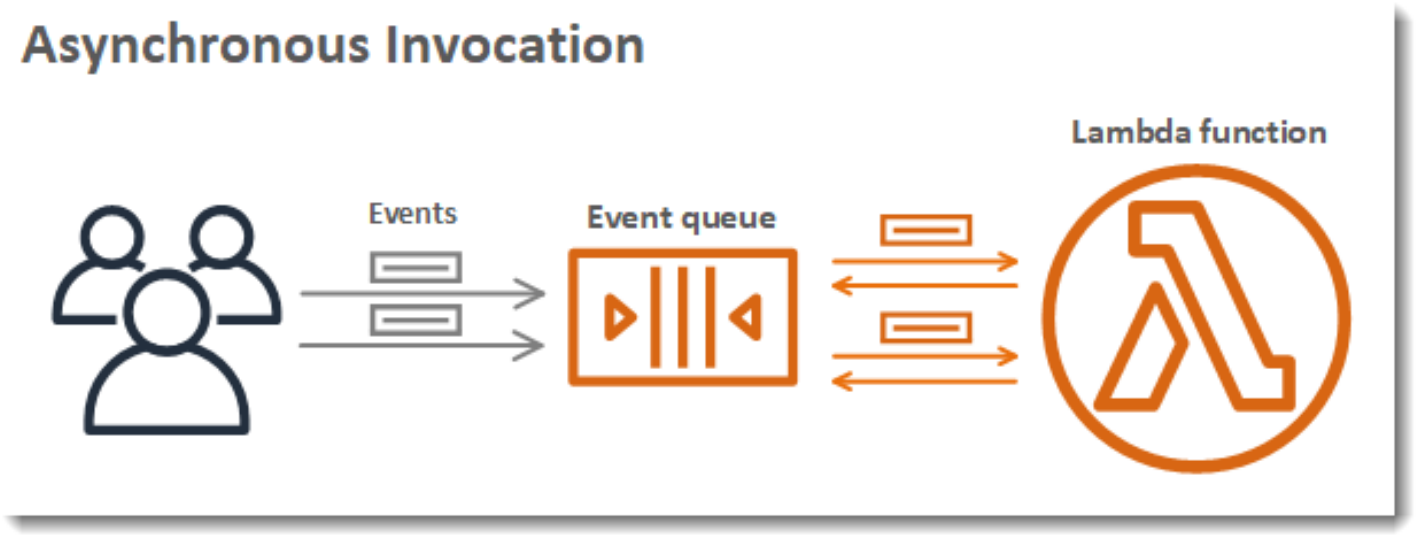
For more information about the `Invoke` API, including a full list of parameters, headers, and errors, see Invoke.

When you invoke a function directly, you can check the response for errors and retry. The AWS CLI and AWS SDK also automatically retry on client timeouts, throttling, and service errors. For more information, see Understanding retry behavior in Lambda.

Invoking a Lambda function asynchronously

Several AWS services, such as Amazon Simple Storage Service (Amazon S3) and Amazon Simple Notification Service (Amazon SNS), invoke functions asynchronously to process events. You can also invoke a Lambda function asynchronously using the AWS Command Line Interface (AWS CLI) or one of the AWS SDKs. When you invoke a function asynchronously, you don't wait for a response from the function code. You hand off the event to Lambda and Lambda handles the rest. You can configure how Lambda handles errors, and can send invocation records to a downstream resource such as Amazon Simple Queue Service (Amazon SQS) or Amazon EventBridge (EventBridge) to chain together components of your application.

The following diagram shows clients invoking a Lambda function asynchronously. Lambda queues the events before sending them to the function.



For asynchronous invocation, Lambda places the event in a queue and returns a success response without additional information. A separate process reads events from the queue and sends them to your function.

To invoke a Lambda function asynchronously using the AWS Command Line Interface (AWS CLI) or one of the AWS SDKs, set the InvocationType parameter to `Event`. The following example shows an AWS CLI command to invoke a function.

```
aws lambda invoke \
  --function-name my-function  \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
```

```
  --payload '{ "key": "value" }' response.json
```

You should see the following output:

```
{
    "StatusCode": 202
}
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

The output file (`response.json`) doesn't contain any information, but is still created when you run this command. If Lambda isn't able to add the event to the queue, the error message appears in the command output.

How Lambda handles errors and retries with asynchronous invocation

Lambda manages your function's asynchronous event queue and attempts to retry on errors. If the function returns an error, by default Lambda attempts to run it two more times, with a one-minute wait between the first two attempts, and two minutes between the second and third attempts. Function errors include errors returned by the function's code and errors returned by the function's runtime, such as timeouts.

If the function doesn't have enough concurrency available to process all events, additional requests are throttled. For throttling errors (429) and system errors (500-series), Lambda returns the event to the queue and attempts to run the function again for up to 6 hours by default. The retry interval increases exponentially from 1 second after the first attempt to a maximum of 5 minutes. If the queue contains many entries, Lambda increases the retry interval and reduces the rate at which it reads events from the queue.

Even if your function doesn't return an error, it's possible for it to receive the same event from Lambda multiple times because the queue itself is eventually consistent. If the function can't keep up with incoming events, events might also be deleted from the queue without being sent to the function. Ensure that your function code gracefully handles duplicate events, and that you have enough concurrency available to handle all invocations.

When the queue is very long, new events might age out before Lambda has a chance to send them to your function. When an event expires or fails all processing attempts, Lambda discards it. You can configure error handling for a function to reduce the number of retries that Lambda performs, or to discard unprocessed events more quickly. To capture discarded events, configure a dead-letter queue for the function. To capture records of failed invocations (such as timeouts or runtime errors), create an on-failure destination.

Configuring error handling settings for Lambda asynchronous invocations

Use the following settings to configure how Lambda handles errors and retries for asynchronous function invocations:

- MaximumEventAgeInSeconds: The maximum amount of time, in seconds, that Lambda keeps an event in the asynchronous event queue before discarding it.
- MaximumRetryAttempts: The maximum number of times that Lambda retries events when the function returns an error.

Use the Lambda console or AWS CLI to configure error handling settings on a function, a version, or an alias.

Console

To configure error handling

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Asynchronous invocation**.
4. Under **Asynchronous invocation**, choose **Edit**.
5. Configure the following settings.
   - **Maximum age of event** – The maximum amount of time Lambda retains an event in the asynchronous event queue, up to 6 hours.
   - **Retry attempts** – The number of times Lambda retries when the function returns an error, between 0 and 2.
6. Choose **Save**.

AWS CLI

To configure asynchronous invocation with the AWS CLI, use the put-function-event-invoke-config command. The following example configures a function with a maximum event age of 1 hour and no retries.

```
aws lambda put-function-event-invoke-config \
  --function-name error \
  --maximum-event-age-in-seconds 3600 \
  --maximum-retry-attempts 0
```

The `put-function-event-invoke-config` command overwrites any existing configuration on the function, version, or alias. To configure an option without resetting others, use update-function-event-invoke-config. The following example configures Lambda to send a record to a standard SQS queue named `destination` when an event can't be processed.

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

You should see the following output:

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

When an invocation event exceeds the maximum age or fails all retry attempts, Lambda discards it. To retain a copy of discarded events, configure a failed-event destination.

Capturing records of Lambda asynchronous invocations

Lambda can send records of asynchronous invocations to one of the following AWS services.

- **Amazon SQS** – A standard SQS queue
- **Amazon SNS** – A standard SNS topic
- **Amazon S3** – An Amazon S3 bucket (on failure only)
- **AWS Lambda** – A Lambda function
- **Amazon EventBridge** – An EventBridge event bus

The invocation record contains details about the request and response in JSON format. You can configure separate destinations for events that are processed successfully, and events that fail all processing attempts. Alternatively, you can configure a standard Amazon SQS queue or standard Amazon SNS topic as a dead-letter queue for discarded events. For dead-letter queues, Lambda only sends the content of the event, without details about the response.

If Lambda can't send a record to a destination you have configured, it sends a `DestinationDeliveryFailures` metric to Amazon CloudWatch. This can happen if your configuration includes an unsupported destination type, such as an Amazon SQS FIFO queue or an Amazon SNS FIFO topic. Delivery errors can also occur due to permissions errors and size limits. For more information on Lambda invocation metrics, see the section called “Invocation metrics”.

> **Note**
>
> To prevent a function from triggering, you can set the function's reserved concurrency to zero. When you set reserved concurrency to zero for an asynchronously invoked function, Lambda begins sending new events to the configured dead-letter queue or the on-failure event destination, without any retries. To process events that were sent while reserved concurrency was set to zero, you must consume the events from the dead-letter queue or the on-failure event destination.

Adding a destination

To retain records of asynchronous invocations, add a destination to your function. You can choose to send either successful or failed invocations to a destination. Each function can have multiple destinations, so you can configure separate destinations for successful and failed events. Each

record sent to the destination is a JSON document with details about the invocation. Like error handling settings, you can configure destinations on a function, function version, or alias.

> **Tip**
>
> You can also retain records of failed invocations for the following event source mapping types: Amazon Kinesis, Amazon DynamoDB, and Apache Kafka (Amazon MSK and self-managed Apache Kafka).

The following table lists supported destinations for asynchronous invocation records. For Lambda to successfully send records to your chosen destination, ensure that your function's execution role also contains the relevant permissions. The table also describes how each destination type receives the JSON invocation record.

Destination type	Required permission	Destination-specific JSON format
Amazon SQS queue	sqs:SendMessage	Lambda passes the invocation record as the `Message` to the destination.
Amazon SNS topic	sns:Publish	Lambda passes the invocation record as the `Message` to the destination.
Amazon S3 bucket (on failure only)	s3:PutObject s3:ListBucket	• Lambda stores the invocation record as a JSON object in the destination bucket. • The S3 object name uses the following naming convention: `aws/lambda/async/<function-name>/YYYY/MM/DD/YYYY-MM-DD`

Destination type	Required permission	Destination-specific JSON format
		`THH.MM.SS-<Random UUID>`
Lambda function	lambda:InvokeFunction	Lambda passes the invocation record as the payload to the function.
EventBridge	events:PutEvents	• Lambda passes the invocation record as the `detail` in the PutEvents call. • The value for the `source` event field is `lambda`. • The value for the `detail-type` event field is either "Lambda Function Invocation Result - Success" or "Lambda Function Invocation Result - Failure". • The `resource` event field contains the function and destination Amazon Resource Names (ARNs). • For other event fields, see Amazon EventBridge events.

Note

For Amazon S3 destinations, if you have enabled encryption on the bucket using a KMS key, your function also needs the kms:GenerateDataKey permission.

> **Important**
>
> When using Amazon SNS as a destination, be aware that Amazon SNS has a maximum message size limit of 256 KB. If your async invocation payload approaches 1 MB, the invocation record (which includes the original payload plus additional metadata) may exceed the Amazon SNS limit and cause delivery failures. Consider using Amazon SQS or Amazon S3 destinations for larger payloads.

The following steps describe how to configure a destination for a function using the Lambda console and the AWS CLI.

Console

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add destination**.
4. For **Source**, choose **Asynchronous invocation**.
5. For **Condition**, choose from the following options:
   - **On failure** – Send a record when the event fails all processing attempts or exceeds the maximum age.
   - **On success** – Send a record when the function successfully processes an asynchronous invocation.
6. For **Destination type**, choose the type of resource that receives the invocation record.
7. For **Destination**, choose a resource.
8. Choose **Save**.

AWS CLI

To configure a destination using the AWS CLI, run the update-function-event-invoke-config command. The following example configures Lambda to send a record to a standard SQS queue named `destination` when an event can't be processed.

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
```

```
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

Security best practices for Amazon S3 destinations

Deleting an S3 bucket that's configured as a destination without removing the destination from your function's configuration can create a security risk. If another user knows your destination bucket's name, they can recreate the bucket in their AWS account. Records of failed invocations will be sent to their bucket, potentially exposing data from your function.

> **Warning**
>
> To ensure that invocation records from your function can't be sent to an S3 bucket in another AWS account, add a condition to your function's execution role that limits `s3:PutObject` permissions to buckets in your account.

The following example shows an IAM policy that limits your function's `s3:PutObject` permissions to buckets in your account. This policy also gives Lambda the `s3:ListBucket` permission it needs to use an S3 bucket as a destination.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
```

```
        }
    ]
}
```

To add a permissions policy to your function's execution role using the AWS Management Console or AWS CLI, refer to the instructions in the following procedures:

Console

To add a permissions policy to a function's execution role (console)

1. Open the Functions page of the Lambda console.
2. Select the Lambda function whose execution role you want to modify.
3. In the **Configuration** tab, select **Permissions**.
4. In the **Execution role** tab, select your function's **Role name** to open the role's IAM console page.
5. Add a permissions policy to the role by doing the following:
    a. In the **Permissions policies** pane, choose **Add permissions** and select **Create inline policy**.
    b. In **Policy editor**, select **JSON**.
    c. Paste the policy you want to add into the editor (replacing the existing JSON), and then choose **Next**.
    d. Under **Policy details**, enter a **Policy name**.
    e. Choose **Create policy**.

AWS CLI

To add a permissions policy to a function's execution role (CLI)

1. Create a JSON policy document with the required permissions and save it in a local directory.
2. Use the IAM `put-role-policy` CLI command to add the permissions to your function's execution role. Run the following command from the directory you saved your JSON policy document in and replace the role name, policy name, and policy document with your own values.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Example invocation record

When an invocation matches the condition, Lambda sends a JSON document with details about the invocation to the destination. The following example shows an invocation record for an event that failed three processing attempts due to a function error.

Example

```
{
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

The invocation record contains details about the event, the response, and the reason that the record was sent.

Tracing requests to destinations

You can use AWS X-Ray to see a connected view of each request as it's queued, processed by a Lambda function, and passed to the destination service. When you activate X-Ray tracing for a function or a service that invokes a function, Lambda adds an X-Ray header to the request and passes the header to the destination service. Traces from upstream services are automatically linked to traces from downstream Lambda functions and destination services, creating an end-to-end view of the entire application. For more information about tracing, see Visualize Lambda function invocations using AWS X-Ray.

Adding a dead-letter queue

As an alternative to an on-failure destination, you can configure your function with a dead-letter queue to save discarded events for further processing. A dead-letter queue acts the same as an on-failure destination in that it is used when an event fails all processing attempts or expires without being processed. However, you can only add or remove a dead-letter queue at the function level. Function versions use the same dead-letter queue settings as the unpublished version ($LATEST). On-failure destinations also support additional targets and include details about the function's response in the invocation record.

To reprocess events in a dead-letter queue, you can set it as an event source for your Lambda function. Alternatively, you can manually retrieve the events.

You can choose an Amazon SQS standard queue or Amazon SNS standard topic for your dead-letter queue. FIFO queues and Amazon SNS FIFO topics are not supported.

- Amazon SQS queue – A queue holds failed events until they're retrieved. Choose an Amazon SQS standard queue if you expect a single entity, such as a Lambda function or CloudWatch alarm, to process the failed event. For more information, see Using Lambda with Amazon SQS.
- Amazon SNS topic – A topic relays failed events to one or more destinations. Choose an Amazon SNS standard topic if you expect multiple entities to act on a failed event. For example, you can configure a topic to send events to an email address, a Lambda function, and/or an HTTP endpoint. For more information, see Invoking Lambda functions with Amazon SNS notifications.

To send events to a queue or topic, your function needs additional permissions. Add a policy with the required permissions to your function's execution role. If the target queue or topic is encrypted

with a customer managed AWS KMS key, ensure that both your function's execution role and the key's resource-based policy contains the relevant permissions.

After creating the target and updating your function's execution role, add the dead-letter queue to your function. You can configure multiple functions to send events to the same target.

Console

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Asynchronous invocation**.
4. Under **Asynchronous invocation**, choose **Edit**.
5. Set **Dead-letter queue service** to **Amazon SQS** or **Amazon SNS**.
6. Choose the target queue or topic.
7. Choose **Save**.

AWS CLI

To configure a dead-letter queue with the AWS CLI, use the update-function-configuration command.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --dead-letter-config TargetArn=arn:aws:sns:us-east-1:123456789012:my-topic
```

Lambda sends the event to the dead-letter queue as-is, with additional information in attributes. You can use this information to identify the error that the function returned, or to correlate the event with logs or an AWS X-Ray trace.

Dead-letter queue message attributes

- **RequestID** (String) – The ID of the invocation request. Request IDs appear in function logs. You can also use the X-Ray SDK to record the request ID on an attribute in the trace. You can then search for traces by request ID in the X-Ray console.
- **ErrorCode** (Number) – The HTTP status code.
- **ErrorMessage** (String) – The first 1 KB of the error message.

If Lambda can't send a message to the dead-letter queue, it deletes the event and emits the DeadLetterErrors metric. This can happen because of lack of permissions, or if the total size of the message exceeds the limit for the target queue or topic. For example, say that an Amazon SNS notification with a body close to 1 MB in size triggers a function that results in an error. In that case, the event data that Amazon SNS adds, combined with the attributes that Lambda adds, can cause the message to exceed the maximum size allowed in the dead-letter queue.

If you're using Amazon SQS as an event source, configure a dead-letter queue on the Amazon SQS queue itself and not on the Lambda function. For more information, see Using Lambda with Amazon SQS.

Invoking durable Lambda functions

Durable Lambda functions can be invoked using the same methods as default Lambda functions, but with important considerations for long-running executions. This section covers invocation patterns, execution management, and best practices for durable functions.

Synchronous invocation limits

Synchronous invocations of durable Lambda functions are limited to 15 minutes, the same as default Lambda functions. If your durable function needs to run longer than 15 minutes, it must be invoked asynchronously.

When to use synchronous invocation: Use for durable functions that complete within 15 minutes and when you need immediate results, such as quick approval workflows or short data processing tasks.

Asynchronous invocation for long-running workflows

For durable functions that may run longer than 15 minutes, use asynchronous invocation. This allows the function to continue running while your client receives an immediate acknowledgment.

TypeScript

```
import { LambdaClient, InvokeCommand } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

// Asynchronous invocation
const command = new InvokeCommand({
  FunctionName: "my-durable-function",
  InvocationType: "Event", // Asynchronous
  Payload: JSON.stringify({ orderId: "12345" })
});

await client.send(command);
```

Python

```
import boto3
import json

client = boto3.client('lambda')

# Asynchronous invocation
response = client.invoke(
    FunctionName='my-durable-function',
    InvocationType='Event',  # Asynchronous
    Payload=json.dumps({'order_id': '12345'})
)
```

Execution management APIs

Lambda provides APIs to manage and monitor durable function executions, including listing executions, getting execution status, and stopping running executions.

TypeScript

```
// Get execution status
const statusCommand = new InvokeCommand({
  FunctionName: "my-durable-function",
  InvocationType: "RequestResponse",
  Payload: JSON.stringify({
    action: "getStatus",
    executionId: "exec-123"
  })
});

const result = await client.send(statusCommand);
```

Python

```
# Get execution status
response = client.invoke(
    FunctionName='my-durable-function',
    InvocationType='RequestResponse',
    Payload=json.dumps({
```

```
        'action': 'get_status',
        'execution_id': 'exec-123'
    })
)
```

How Lambda processes records from stream and queue-based event sources

An *event source mapping* is a Lambda resource that reads items from stream and queue-based services and invokes a function with batches of records. Within an event source mapping, resources called *event pollers* actively poll for new messages and invoke functions. By default, Lambda automatically scales event pollers, but for certain event source types, you can use provisioned mode to control the minimum and maximum number of event pollers dedicated to your event source mapping.

The following services use event source mappings to invoke Lambda functions:

- Amazon DocumentDB (with MongoDB compatibility) (Amazon DocumentDB)
- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

> **Warning**
>
> Lambda event source mappings process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

How event source mappings differ from direct triggers

Some AWS services can directly invoke Lambda functions using *triggers*. These services push events to Lambda, and the function is invoked immediately when the specified event occurs. Triggers are suitable for discrete events and real-time processing. When you create a trigger using the Lambda console, the console interacts with the corresponding AWS service to configure the event notification on that service. The trigger is actually stored and managed by the service that

generates the events, not by Lambda. Here are some examples of services that use triggers to invoke Lambda functions:

- **Amazon Simple Storage Service (Amazon S3):** Invokes a function when an object is created, deleted, or modified in a bucket. For more information, see Tutorial: Using an Amazon S3 trigger to invoke a Lambda function.
- **Amazon Simple Notification Service (Amazon SNS):** Invokes a function when a message is published to an SNS topic. For more information, see Tutorial: Using AWS Lambda with Amazon Simple Notification Service.
- **Amazon API Gateway:** Invokes a function when an API request is made to a specific endpoint. For more information, see Invoking a Lambda function using an Amazon API Gateway endpoint.

Event source mappings are Lambda resources created and managed within the Lambda service. Event source mappings are designed for processing high-volume streaming data or messages from queues. Processing records from a stream or queue in batches is more efficient than processing records individually.

Batching behavior

By default, an event source mapping batches records together into a single payload that Lambda sends to your function. To fine-tune batching behavior, you can configure a batching window (MaximumBatchingWindowInSeconds) and a batch size (BatchSize). A batching window is the maximum amount of time to gather records into a single payload. A batch size is the maximum number of records in a single batch. Lambda invokes your function when one of the following three criteria is met:

- **The batching window reaches its maximum value.** Default batching window behavior varies depending on the specific event source.
  - **For Kinesis, DynamoDB, and Amazon SQS event sources:** The default batching window is 0 seconds. This means that Lambda invokes your function as soon as records are available. To set a batching window, configure `MaximumBatchingWindowInSeconds`. You can set this parameter to any value from 0 to 300 seconds in increments of 1 second. If you configure a batching window, the next window begins as soon as the previous function invocation completes.
  - **For Amazon MSK, self-managed Apache Kafka, Amazon MQ, and Amazon DocumentDB event sources:** The default batching window is 500 ms. You can configure `MaximumBatchingWindowInSeconds` to any value from 0 seconds to 300 seconds in

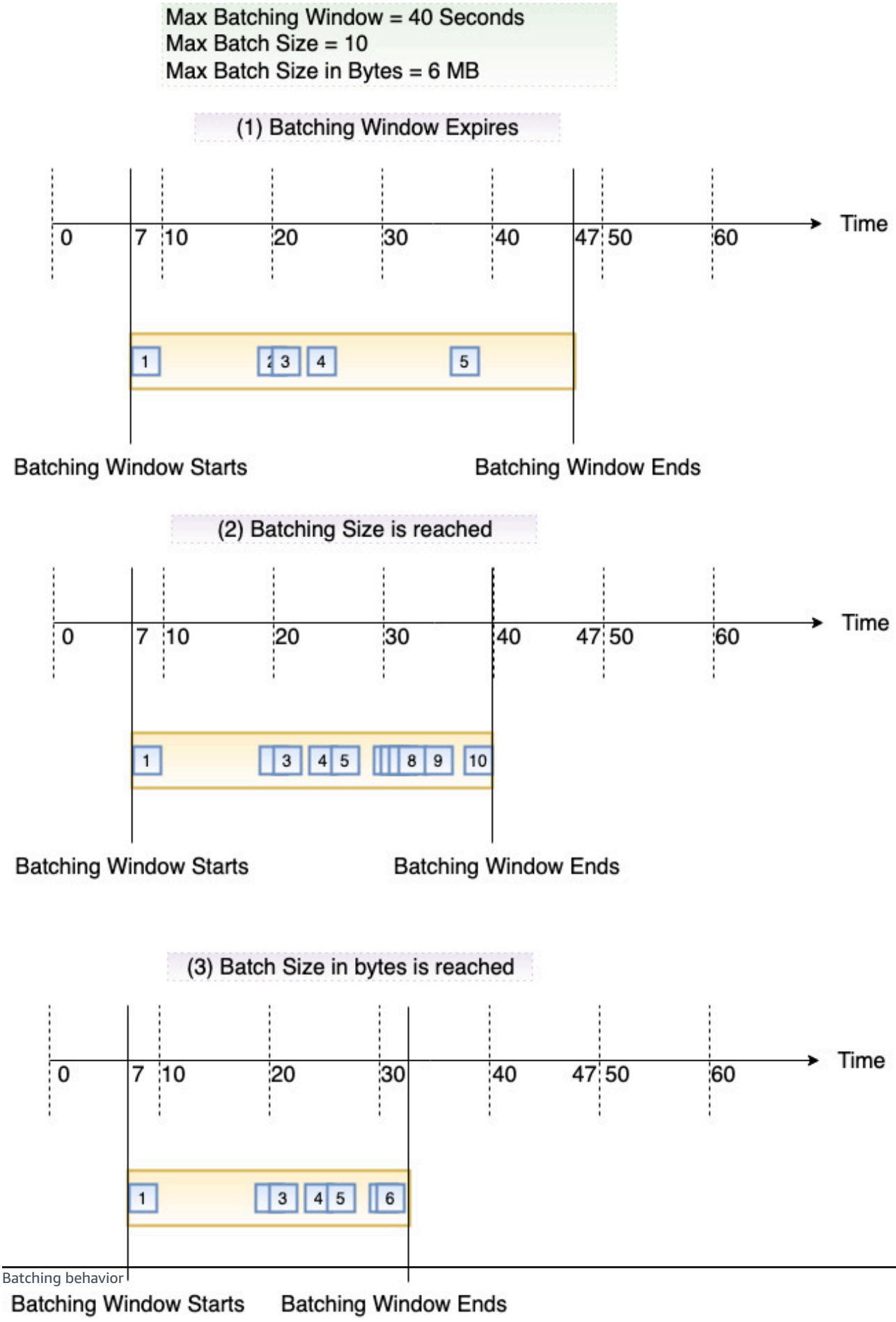
increments of seconds. In provisioned mode for Kafka event source mappings, when you configure a batching window, the next window begins as soon as the previous batch is completed. In non-provisioned Kafka event source mappings, when you configure a batching window, the next window begins as soon as the previous function invocation completes. To minimize latency when using Kafka event source mappings in provisioned mode, set `MaximumBatchingWindowInSeconds` to 0. This setting ensures that Lambda will start processing the next batch immediately after completing the current function invocation. For additional information on low latency processing, see Low latency Apache Kafka.

- **For Amazon MQ and Amazon DocumentDB event sources:** The default batching window is 500 ms. You can configure `MaximumBatchingWindowInSeconds` to any value from 0 seconds to 300 seconds in increments of seconds. A batching window begins as soon as the first record arrives.

  > **Note**
  >
  > Because you can only change `MaximumBatchingWindowInSeconds` in increments of seconds, you cannot revert to the 500 ms default batching window after you have changed it. To restore the default batching window, you must create a new event source mapping.

- **The batch size is met.** The minimum batch size is 1. The default and maximum batch size depend on the event source. For details about these values, see the BatchSize specification for the `CreateEventSourceMapping` API operation.
- **The payload size reaches 6 MB.** You cannot modify this limit.

The following diagram illustrates these three conditions. Suppose a batching window begins at `t = 7` seconds. In the first scenario, the batching window reaches its 40 second maximum at `t = 47` seconds after accumulating 5 records. In the second scenario, the batch size reaches 10 before the batching window expires, so the batching window ends early. In the third scenario, the maximum payload size is reached before the batching window expires, so the batching window ends early.


Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB
(1) Batching Window Expires
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
Batching Window Starts
Batching Window Ends
(2) Batching Size is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
8
9
10
Batching Window Starts
Batching Window Ends
(3) Batch Size in bytes is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
6
Batching Window Starts
Batching Window Ends

We recommend that you test with different batch and record sizes so that the polling frequency of each event source is tuned to how quickly your function is able to complete its task. The CreateEventSourceMapping `BatchSize` parameter controls the maximum number of records that can be sent to your function with each invoke. A larger batch size can often more efficiently absorb the invoke overhead across a larger set of records, increasing your throughput.

Lambda doesn't wait for any configured extensions to complete before sending the next batch for processing. In other words, your extensions may continue to run as Lambda processes the next batch of records. This can cause throttling issues if you breach any of your account's concurrency settings or limits. To detect whether this is a potential issue, monitor your functions and check whether you're seeing higher concurrency metrics than expected for your event source mapping. Due to short times in between invokes, Lambda may briefly report higher concurrency usage than the number of shards. This can be true even for Lambda functions without extensions.

By default, if your function returns an error, the event source mapping reprocesses the entire batch until the function succeeds, or the items in the batch expire. To ensure in-order processing, the event source mapping pauses processing for the affected shard until the error is resolved. For stream sources (DynamoDB and Kinesis), you can configure the maximum number of times that Lambda retries when your function returns an error. Service errors or throttles where the batch does not reach your function do not count toward retry attempts. You can also configure the event source mapping to send an invocation record to a destination when it discards an event batch.

Provisioned mode

Lambda event source mappings use event pollers to poll your event source for new messages. By default, Lambda manages the autoscaling of these pollers based on message volume. When message traffic increases, Lambda automatically increases the number of event pollers to handle the load, and reduces them when traffic decreases.

In provisioned mode, you can fine-tune the throughput of your event source mapping by defining minimum and maximum limits for dedicated polling resources that remain ready to handle expected traffic patterns. These resources auto-scale 3 times faster to handle sudden spikes in event traffic and provide 16 times higher capacity to process millions of events. This helps you build highly responsive event-driven workloads with stringent performance requirements.

In Lambda, an event poller is a compute unit with throughput capabilities that vary by event source type. For Amazon MSK and self-managed Apache Kafka, each event poller can handle up to 5 MB/sec of throughput or up to 5 concurrent invocations. For example, if your event source produces

an average payload of 1 MB and the average duration of your function is 1 second, a single Kafka event poller can support 5 MB/sec throughput and 5 concurrent Lambda invocations (assuming no payload transformation). For Amazon SQS, each event poller can handle up to 1 MB/sec of throughput or up to 10 concurrent invocations. Using provisioned mode incurs additional costs based on your event poller usage. For pricing details, see AWS Lambda pricing.

Provisioned mode is available for Amazon MSK, self-managed Apache Kafka, and Amazon SQS event sources. While concurrency settings give you control over the scaling of your function, provisioned mode gives you control over the throughput of your event source mapping. To ensure maximum performance, you might need to adjust both settings independently.

Provisioned mode is ideal for real-time applications requiring consistent event processing latency, such as financial services firms processing market data feeds, e-commerce platforms providing real-time personalized recommendations, and gaming companies managing live player interactions.

Each event poller supports different throughput capacity:

- For Amazon MSK and self-managed Apache Kafka: up to 5 MB/sec of throughput or up to 5 concurrent invokes
- For Amazon SQS: up to 1 MB/sec of throughput or up to 10 concurrent invokes or up to 10 SQS polling API calls per second.

For Amazon SQS event source mappings, you can set the minimum number of pollers between 2 and 200 with a default of 2, and the maximum number between 2 and 2,000 with a default of 200. Lambda scales the number of event pollers between your configured minimum and maximum, quickly adding up to 1,000 concurrency per minute to provide consistent, low-latency processing of your events.

For Kafka event source mappings, you can set the minimum number of pollers between 1 and 200 with a default of 1, and the maximum number between 1 and 2,000 with a default of 200. Lambda scales the number of event pollers between your configured minimum and maximum based on your event backlog in your topic to provide low-latency processing of your events.

Note that for Amazon SQS event sources, the maximum concurrency setting cannot be used with provisioned mode. When using provisioned mode, you control concurrency through the maximum event pollers setting.

For details about configuring provisioned mode, see the following sections:

- Configuring provisioned mode for Amazon MSK event source mappings
- Configuring provisioned mode for self-managed Apache Kafka event source mappings
- Using provisioned mode with Amazon SQS event source mappings

To minimize latency in provisioned mode, set `MaximumBatchingWindowInSeconds` to 0. This setting ensures that Lambda will start processing the next batch immediately after completing the current function invocation. For additional information on low latency processing, see Low latency Apache Kafka.

After configuring provisioned mode, you can observe the usage of event pollers for your workload by monitoring the `ProvisionedPollers` metric. For more information, see the section called "Event source mapping metrics".

Event source mapping API

To manage an event source with the AWS Command Line Interface (AWS CLI) or an AWS SDK, you can use the following API operations:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

Using tags on event source mappings

You can tag event source mappings to organize and manage your resources. Tags are free-form key-value pairs associated with your resources that are supported across AWS services. For more information about use cases for tags, see Common tagging strategies in the *Tagging AWS Resources and Tag Editor Guide*.

Event source mappings are associated with functions, which can have their own tags. Event source mappings do not automatically inherit tags from functions. You can use the AWS Lambda API to view and update tags. You can also view and update tags while managing a specific event source mapping in the Lambda console, including those using Provisioned Mode for Amazon SQS.

Permissions required for working with tags

To allow an AWS Identity and Access Management (IAM) identity (user, group, or role) to read or set tags on a resource, grant it the corresponding permissions:

- **lambda:ListTags**–When a resource has tags, grant this permission to anyone who needs to call `ListTags` on it. For tagged functions, this permission is also necessary for `GetFunction`.
- **lambda:TagResource**–Grant this permission to anyone who needs to call `TagResource` or perform a tag on create.

Optionally, consider granting the **lambda:UntagResource** permission as well to allow `UntagResource` calls to the resource.

For more information, see Identity-based IAM policies for Lambda.

Using tags with the Lambda console

You can use the Lambda console to create event source mappings that have tags, add tags to existing event source mappings, and filter event source mappings by tag, including those configured in Provisioned Mode for Amazon SQS.

When you add a trigger for supported stream and queue-based services using the Lambda console, Lambda automatically creates an event source mapping. For more information about these event sources, see the section called “Event source mappings”. To create an event source mapping in the console, you will need the following prerequisites:

- A function.
- An event source from an affected service.

You can add the tags as part of the same user interface you use to create or update triggers.

To add a tag when you create a event source mapping

1. Open the Functions page of the Lambda console.
2. Choose the name of your function.
3. Under **Function overview**, choose **Add trigger**.
4. Under **Trigger configuration**, in the dropdown list, choose the name of the service your event source comes from.

5. Provide the core configuration for your event source. For more information about configuring your event source, consult the section for the related service in *Integrating other services*.
6. Under **Event source mapping configuration**, choose **Additional settings**.
7. Under **Tags**, choose **Add new tag**
8. In the **Key** field, enter your tag key. For information about tagging restrictions, see Tag naming limits and requirements in the *Tagging AWS Resources and Tag Editor Guide*.
9. Choose **Add**.

To add tags to an existing event source mapping

1. Open Event source mappings in the Lambda console.
2. From the resource list, choose the **UUID** for the event source mapping corresponding to your **Function** and **Event source ARN**.
3. From the tab list below the **General configuration pane**, choose **Tags**.
4. Choose **Manage tags**.
5. Choose **Add new tag**.
6. In the **Key** field, enter your tag key. For information about tagging restrictions, see Tag naming limits and requirements in the *Tagging AWS Resources and Tag Editor Guide*.
7. Choose **Save**.

To filter event source mappings by tag

1. Open Event source mappings in the Lambda console.
2. Choose the search box.
3. From the dropdown list, select your tag key from below the **Tags** subheading.
4. Select **Use: "tag-name"** to see all event source mappings tagged with this key, or choose an **Operator** to further filter by value.
5. Select your tag value to filter by a combination of tag key and value.

The search box also supports searching for tag keys. Enter the name of a key to find it in the list.

Using tags with the AWS CLI

You can add and remove tags on existing Lambda resources, including event source mappings, with the Lambda API. You can also add tags when creating an event source mapping, which allows you to keep a resource tagged through its entire lifecycle.

Updating tags with the Lambda tag APIs

You can add and remove tags for supported Lambda resources through the TagResource and UntagResource API operations.

You can call these operations using the AWS CLI. To add tags to an existing resource, use the `tag-resource` command. This example adds two tags, one with the key *`Department`* and one with the key *`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

To remove tags, use the `untag-resource` command. This example removes the tag with the key *`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier \
--tag-keys Department
```

Adding tags when you create an event source mapping

To create a new Lambda event source mapping with tags, use the CreateEventSourceMapping API operation. Specify the `Tags` parameter. You can call this operation with the `create-event-source-mapping` AWS CLI command and the `--tags` option. For more information about the CLI command, see create-event-source-mapping in the *AWS CLI Command Reference*.

Before using the `Tags` parameter with `CreateEventSourceMapping`, ensure that your role has permission to tag resources alongside the usual permissions needed for this operation. For more information about permissions for tagging, see the section called “Permissions required for working with tags”.

Viewing tags with the Lambda tag APIs

To view the tags that are applied to a specific Lambda resource, use the `ListTags` API operation. For more information, see ListTags.

You can call this operation with the `list-tags` AWS CLI command by providing an ARN (Amazon Resource Name).

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

Filtering resources by tag

You can use the AWS Resource Groups Tagging API GetResources API operation to filter your resources by tags. The `GetResources` operation receives up to 10 filters, with each filter containing a tag key and up to 10 tag values. You provide `GetResources` with a `ResourceType` to filter by specific resource types.

You can call this operation using the `get-resources` AWS CLI command. For examples of using `get-resources`, see get-resources in the *AWS CLI Command Reference*.

Control which events Lambda sends to your function

You can use event filtering to control which records from a stream or queue Lambda sends to your function. For example, you can add a filter so that your function only processes Amazon SQS messages containing certain data parameters. Event filtering works only with certain event source mappings. You can add filters to event source mappings for the following AWS services:

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- Self-managed Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

For specific information about filtering with specific event sources, see the section called "Using filters with different AWS services". Lambda doesn't support event filtering for Amazon DocumentDB.

By default, you can define up to five different filters for a single event source mapping. Your filters are logically ORed together. If a record from your event source satisfies one or more of your filters, Lambda includes the record in the next event it sends to your function. If none of your filters are satisfied, Lambda discards the record.

> **Note**
>
> If you need to define more than five filters for an event source, you can request a quota increase for up to 10 filters for each event source. If you attempt to add more filters than your current quota permits, Lambda will return an error when you try to create the event source.

Topics

- Understanding event filtering basics
- Handling records that don't meet filter criteria
- Filter rule syntax

- Attaching filter criteria to an event source mapping (console)
- Attaching filter criteria to an event source mapping (AWS CLI)
- Attaching filter criteria to an event source mapping (AWS SAM)
- Encryption of filter criteria
- Using filters with different AWS services

Understanding event filtering basics

A filter criteria (`FilterCriteria`) object is a structure that consists of a list of filters (`Filters`). Each filter is a structure that defines an event filtering pattern (`Pattern`). A pattern is a string representation of a JSON filter rule. The structure of a `FilterCriteria` object is as follows.

```
{
   "Filters": [
        {
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

Your filter pattern can include metadata properties, data properties, or both. The available metadata parameters and the format of the data parameters vary according to the AWS service which is acting as the event source. For example, suppose your event source mapping receives the following record from an Amazon SQS queue:

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
```

```
    "body": "{\\n \"City\": \"Seattle\",\\n \"State\": \"WA\",\\n \"Temperature\":
 \"46\"\\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
}
```

- **Metadata properties** are the fields containing information about the event that created the record. In the example Amazon SQS record, the metadata properties include fields such as `messageID`, `eventSourceArn`, and `awsRegion`.
- **Data properties** are the fields of the record containing the data from your stream or queue. In the Amazon SQS event example, the key for the data field is `body`, and the data properties are the fields `City` `State`, and `Temperature`.

Different types of event source use different key values for their data fields. To filter on data properties, make sure that you use the correct key in your filter's pattern. For a list of data filtering keys, and to see examples of filter patterns for each supported AWS service, refer to Using filters with different AWS services.

Event filtering can handle multi-level JSON filtering. For example, consider the following fragment of a record from a DynamoDB stream:

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
            "N": "1234"
    },
    ...
}
```

Suppose you want to process only those records where the value of the sort key `Number` is 4567. In this case, your `FilterCriteria` object would look like this:

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

Handling records that don't meet filter criteria

How Lambda handles records that don't meet your filter criteria depends on the event source.

- For **Amazon SQS**, if a message doesn't satisfy your filter criteria, Lambda automatically removes the message from the queue. You don't have to manually delete these messages in Amazon SQS.
- For **Kinesis** and **DynamoDB**, after your filter criteria evaluates a record, the streams iterator advances past this record. If the record doesn't satisfy your filter criteria, you don't have to manually delete the record from your event source. After the retention period, Kinesis and DynamoDB automatically delete these old records. If you want records to be deleted sooner, see Changing the Data Retention Period.
- For **Amazon MSK**, **self-managed Apache Kafka**, and **Amazon MQ** messages, Lambda drops messages that don't match all fields included in the filter. For Amazon MSK and self-managed Apache Kafka, Lambda commits offsets for matched and unmatched messages after successfully invoking the function. For Amazon MQ, Lambda acknowledges matched messages after successfully invoking the function, and acknowledges unmatched messages when filtering them.

Filter rule syntax

For filter rules, Lambda supports the Amazon EventBridge rules and uses the same syntax as EventBridge. For more information, see Amazon EventBridge event patterns in the *Amazon EventBridge User Guide*.

The following is a summary of all the comparison operators available for Lambda event filtering.

Comparison operator	Example	Rule syntax
Null	UserID is null	"UserID": [null]
Empty	LastName is empty	"LastName": [""]
Equals	Name is "Alice"	"Name": ["Alice"]
Equals (ignore case)	Name is "Alice"	"Name": [{ "equals-ignore-case": "alice" }]
And	Location is "New York" and Day is "Monday"	"Location": ["New York"], "Day": ["Monday"]
Or	PaymentType is "Credit" or "Debit"	"PaymentType": ["Credit", "Debit"]
Or (multiple fields)	Location is "New York", or Day is "Monday".	"$or": [{ "Location": ["New York"] }, { "Day": ["Monday"] }]
Not	Weather is anything but "Raining"	"Weather": [{ "anything-but": ["Raining"] }]
Numeric (equals)	Price is 100	"Price": [{ "numeric": ["=", 100] }]
Numeric (range)	Price is more than 10, and less than or equal to 20	"Price": [{ "numeric": [">", 10, "<=", 20] }]
Exists	ProductName exists	"ProductName": [{ "exists": true }]

Comparison operator	Example	Rule syntax
Does not exist	ProductName does not exist	"ProductName": [{ "exists": false }]
Begins with	Region is in the US	"Region": [{"prefix": "us-" }]
Ends with	FileName ends with a .png extension.	"FileName": [{ "suffix": ".png" }]

Note

Like EventBridge, for strings, Lambda uses exact character-by-character matching without case-folding or any other string normalization. For numbers, Lambda also uses string representation. For example, 300, 300.0, and 3.0e2 are not considered equal.

Note that the Exists operator only works on leaf nodes in your event source JSON. It doesn't match intermediate nodes. For example, with the following JSON, the filter pattern `{ "person": { "address": [ { "exists": true } ] } }` wouldn't find a match because `"address"` is an intermediate node.

```
{
  "person": {
    "name": "John Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
  }
}
```

Attaching filter criteria to an event source mapping (console)

Follow these steps to create a new event source mapping with filter criteria using the Lambda console.

To create a new event source mapping with filter criteria (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of a function to create an event source mapping for.
3. Under **Function overview**, choose **Add trigger**.
4. For **Trigger configuration**, choose a trigger type that supports event filtering. For a list of supported services, refer to the list at the beginning of this page.
5. Expand **Additional settings**.
6. Under **Filter criteria**, choose **Add**, and then define and enter your filters. For example, you can enter the following.

   ```
   { "Metadata" : [ 1, 2 ] }
   ```

   This instructs Lambda to process only the records where field `Metadata` is equal to 1 or 2. You can continue to select **Add** to add more filters up to the maximum allowed amount.
7. When you have finished adding your filters, choose **Save**.

When you enter filter criteria using the console, you enter only the filter pattern and don't need to provide the `Pattern` key or escape quotes. In step 6 of the preceding instructions, `{ "Metadata" : [ 1, 2 ] }` corresponds to the following `FilterCriteria`.

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

After creating your event source mapping in the console, you can see the formatted `FilterCriteria` in the trigger details. For more examples of creating event filters using the console, see Using filters with different AWS services.

Attaching filter criteria to an event source mapping (AWS CLI)

Suppose you want an event source mapping to have the following `FilterCriteria`:

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

This create-event-source-mapping command creates a new Amazon SQS event source mapping for function `my-function` with the specified `FilterCriteria`.

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

Note that to update an event source mapping, you need its UUID. You can get the UUID from a list-event-source-mappings call. Lambda also returns the UUID in the create-event-source-mapping CLI response.

To remove filter criteria from an event source, you can run the following update-event-source-mapping command with an empty `FilterCriteria` object.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

For more examples of creating event filters using the AWS CLI, see Using filters with different AWS services.

Attaching filter criteria to an event source mapping (AWS SAM)

Suppose you want to configure an event source in AWS SAM to use the following filter criteria:

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

To add these filter criteria to your event source mapping, insert the following snippet into the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

For more information on creating and configuring an AWS SAM template for an event source mapping, see the EventSource section of the AWS SAM Developer Guide. Fore more examples of creating event filters using AWS SAM templates, see Using filters with different AWS services.

Encryption of filter criteria

By default, Lambda doesn't encrypt your filter criteria object. For use cases where you may include sensitive information in your filter criteria object, you can use your own KMS key to encrypt it.

After you encrypt your filter criteria object, you can view its plaintext version using a GetEventSourceMapping API call. You must have `kms:Decrypt` permissions to be able to successfully view the filter criteria in plaintext.

> **Note**
>
> If your filter criteria object is encrypted, Lambda redacts the value of the `FilterCriteria` field in the response of ListEventSourceMappings calls. Instead, this field displays as `null`.
> To see the true value of `FilterCriteria`, use the GetEventSourceMapping API.
> To view the decrypted value of `FilterCriteria` in the console, ensure that your IAM role contains permissions for GetEventSourceMapping.

You can specify your own KMS key via the console, API/CLI, or CloudFormation.

To encrypt filter criteria with a customer-owned KMS key (console)

1. Open the Functions page of the Lambda console.
2. Choose **Add trigger**. If you already have an existing trigger, choose the **Configuration** tab, and then choose **Triggers**. Select the existing trigger, and choose **Edit**.
3. Select the checkbox next to **Encrypt with customer managed KMS key**.
4. For **Choose a customer managed KMS encryption key**, select an existing enabled key or create a new key. Depending on the operation, you need some or all of the following permissions: `kms:DescribeKey`, `kms:GenerateDataKey`, and `kms:Decrypt`. Use the KMS key policy to grant these permissions.

If you use your own KMS key, the following API operations must be permitted in the key policy:

- `kms:Decrypt` – Must be granted to the regional Lambda service principal (`lambda.`*`AWS_region`*`.amazonaws.com`). This allows Lambda to decrypt data with this KMS key.
  - To prevent a cross-service confused deputy problem, the key policy uses the `aws:SourceArn` global condition key. The correct value of the `aws:SourceArn` key is the ARN of your event source mapping resource, so you can add this to your policy only after you know its ARN. Lambda also forwards the `aws:lambda:FunctionArn` and `aws:lambda:EventSourceArn` keys and their respective values in the encryption context when making a decryption request to KMS. These values must match the specified conditions in the key policy for the decryption request to succeed. You don't need to include EventSourceArn for Self-managed Kafka event sources since they don't have an EventSourceArn.
- `kms:Decrypt` – Must also be granted to the principal that intends to use the key to view the plaintext filter criteria in GetEventSourceMapping or DeleteEventSourceMapping API calls.
- `kms:DescribeKey` – Provides the customer managed key details to allow the specified principal to use the key.
- `kms:GenerateDataKey` – Provides permissions for Lambda to generate a data key to encrypt the filter criteria, on behalf of the specified principal (envelope encryption).

You can use AWS CloudTrail to track AWS KMS requests that Lambda makes on your behalf. For sample CloudTrail events, see ???.

We also recommend using the kms:ViaService condition key to limit the use of the KMS key to requests from Lambda only. The value of this key is the regional Lambda service principal (lambda.*AWS_region*.amazonaws.com). The following is a sample key policy that grants all the relevant permissions:

Example AWS KMS key policy

JSON

```
{
    "Version":"2012-10-17",
    "Id": "example-key-policy-1",
    "Statement": [
        {
            "Sid": "Allow Lambda to decrypt using the key",
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.us-east-1.amazonaws.com"
            },
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "*",
            "Condition": {
                "ArnEquals" : {
                    "aws:SourceArn": [
                        "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:<esm_uuid>"
                    ]
                },
                "StringEquals": {

 "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:test-function",

 "kms:EncryptionContext:aws:lambda:EventSourceArn": "arn:aws:sqs:us-
east-1:123456789012:test-queue"
                }
            }
        },
        {
            "Sid": "Allow actions by an AWS account on the key",
```

```
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Sid": "Allow use of the key to specific roles",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/ExampleRole"
            },
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals" : {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com"
                }
            }
        }
    ]
}
```

To use your own KMS key to encrypt filter criteria, you can also use the following CreateEventSourceMapping AWS CLI command. Specify the KMS key ARN with the `--kms-key-arn` flag.

```
aws lambda create-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

If you have an existing event source mapping, use the UpdateEventSourceMapping AWS CLI command instead. Specify the KMS key ARN with the `--kms-key-arn` flag.

```
aws lambda update-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

This operation overwrites any KMS key that was previously specified. If you specify the `--kms-key-arn` flag along with an empty argument, Lambda stops using your KMS key to encrypt filter criteria. Instead, Lambda defaults back to using an Amazon-owned key.

To specify your own KMS key in a CloudFormation template, use the `KMSKeyArn` property of the `AWS::Lambda::EventSourceMapping` resource type. For example, you can insert the following snippet into the YAML template for your event source.

```
MyEventSourceMapping:
  Type: AWS::Lambda::EventSourceMapping
  Properties:
    ...
    FilterCriteria:
      Filters:
        - Pattern: '{"a": [1, 2]}'
    KMSKeyArn: "arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599"
    ...
```

To be able to view your encrypted filter criteria in plaintext in a GetEventSourceMapping or DeleteEventSourceMapping API call, you must have `kms:Decrypt` permissions.

Starting August 6, 2024, the `FilterCriteria` field no longer shows up in AWS CloudTrail logs from CreateEventSourceMapping, UpdateEventSourceMapping, and DeleteEventSourceMapping API calls if your function doesn't use event filtering. If your function does use event filtering, the `FilterCriteria` field shows up as empty ({}). You can still view your filter criteria in plaintext in the response of GetEventSourceMapping API calls if you have `kms:Decrypt` permissions for the correct KMS key.

Sample CloudTrail log entry for Create/Update/DeleteEventSourceMapping calls

In the following AWS CloudTrail sample log entry for a CreateEventSourceMapping call, `FilterCriteria` shows up as empty ({}) because the function uses event filtering. This is the

case even if `FilterCriteria` object contains valid filter criteria that your function is actively using. If the function doesn't use event filtering, CloudTrail won't display the `FilterCriteria` field at all in log entries.


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:userid1",
        "arn": "arn:aws:sts::123456789012:assumed-role/Example/example-role",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA987654321EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/User1",
                "accountId": "123456789012",
                "userName": "User1"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2024-05-09T20:35:01Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "AWS Internal"
    },
    "eventTime": "2024-05-09T21:05:41Z",
    "eventSource": "lambda.amazonaws.com",
    "eventName": "CreateEventSourceMapping20150331",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "AWS Internal",
    "userAgent": "AWS Internal",
    "requestParameters": {
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "functionName": "example-function",
        "enabled": true,
        "batchSize": 10,
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "scalingConfig": {},
```

```
        "maximumBatchingWindowInSeconds": 0,
        "sourceAccessConfigurations": []
    },
    "responseElements": {
        "uUID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
        "batchSize": 10,
        "maximumBatchingWindowInSeconds": 0,
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:example-
function",
        "lastModified": "May 9, 2024, 9:05:41 PM",
        "state": "Creating",
        "stateTransitionReason": "USER_INITIATED",
        "functionResponseTypes": [],
        "eventSourceMappingArn": "arn:aws:lambda:us-east-2:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb"
    },
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```


Using filters with different AWS services

Different types of event source use different key values for their data fields. To filter on data properties, make sure that you use the correct key in your filter's pattern. The following table gives the filtering keys for each supported AWS service.

AWS service	Filtering key
DynamoDB	dynamodb
Kinesis	data

AWS service	Filtering key
Amazon MQ	`data`
Amazon MSK	`value`
Self-managed Apache Kafka	`value`
Amazon SQS	`body`

The following sections give examples of filter patterns for different types of event sources. They also provide definitions of supported incoming data formats and filter pattern body formats for each supported service.

- the section called “Event filtering”
- the section called “Event filtering”
- the section called “Event filtering”
- the section called “Event filtering”
- the section called “Event filtering”

Testing Lambda functions in the console

You can test your Lambda function in the console by invoking your function with a test event. A *test event* is a JSON input to your function. If your function doesn't require input, the event can be an empty document (`{}`).

When you run a test in the console, Lambda synchronously invokes your function with the test event. The function runtime converts the event JSON into an object and passes it to your code's handler method for processing.

Create a test event

Before you can test in the console, you need to create a private or shareable test event.

Invoking functions with test events

To test a function

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the **Test** tab.
4. Under **Test event**, choose **Create new event** or **Edit saved event** and then choose the saved event that you want to use.
5. Optionally - choose a **Template** for the event JSON.
6. Choose **Test**.
7. To review the test results, under **Execution result**, expand **Details**.

To invoke your function without saving your test event, choose **Test** before saving. This creates an unsaved test event that Lambda preserves only for the duration of the session.

For the Node.js, Python, and Ruby runtimes, you can also access your existing saved and unsaved test events on the **Code** tab. Use the **TEST EVENTS** section to create, edit, and run tests.

Creating private test events

Private test events are available only to the event creator, and they require no additional permissions to use. You can create and save up to 10 private test events per function.

To create a private test event

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the **Test** tab.
4. Under **Test event**, do the following:
    a. Choose a **Template**.
    b. Enter a **Name** for the test.
    c. In the text entry box, enter the JSON test event.
    d. Under **Event sharing settings**, choose **Private**.
5. Choose **Save changes**.

For the Node.js, Python, and Ruby runtimes, you can also create test events on the **Code** tab. Use the **TEST EVENTS** section to create, edit, and run tests.

Creating shareable test events

Shareable test events are test events that you can share with other users in the same AWS account. You can edit other users' shareable test events and invoke your function with them.

Lambda saves shareable test events as schemas in an Amazon EventBridge (CloudWatch Events) schema registry named `lambda-testevent-schemas`. As Lambda utilizes this registry to store and call shareable test events you create, we recommend that you do not edit this registry or create a registry using the `lambda-testevent-schemas` name.

To see, share, and edit shareable test events, you must have permissions for all of the following EventBridge (CloudWatch Events) schema registry API operations:

- `schemas.CreateRegistry`
- `schemas.CreateSchema`
- `schemas.DeleteSchema`

- schemas.DeleteSchemaVersion
- schemas.DescribeRegistry
- schemas.DescribeSchema
- schemas.GetDiscoveredSchema
- schemas.ListSchemaVersions
- schemas.UpdateSchema

Note that saving edits made to a shareable test event overwrites that event.

If you cannot create, edit, or see shareable test events, check that your account has the required permissions for these operations. If you have the required permissions but still cannot access shareable test events, check for any resource-based policies that might limit access to the EventBridge (CloudWatch Events) registry.

To create a shareable test event

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to test.
3. Choose the **Test** tab.
4. Under **Test event**, do the following:
    a. Choose a **Template**.
    b. Enter a **Name** for the test.
    c. In the text entry box, enter the JSON test event.
    d. Under **Event sharing settings**, choose **Shareable**.
5. Choose **Save changes**.

Use shareable test events with AWS Serverless Application Model.

You can use AWS SAM to invoke shareable test events. See `sam remote test-event` in the AWS Serverless Application Model Developer Guide

Deleting shareable test event schemas

When you delete shareable test events, Lambda removes them from the `lambda-testevent-schemas` registry. If you remove the last shareable test event from the registry, Lambda deletes the registry.

If you delete the function, Lambda does not delete any associated shareable test event schemas. You must clean up these resources manually from the EventBridge (CloudWatch Events) console.

Lambda function states

Lambda includes a State field in the function configuration for all functions to indicate when your function is ready to invoke. `State` provides information about the current status of the function, including whether you can successfully invoke the function. Function states do not change the behavior of function invocations or how your function runs the code.

> **Note**
>
> Function state definitions differ slightly for SnapStart functions. For more information, see Lambda SnapStart and function states.

In many cases, a DynamoDB table is an ideal way to retain state between invocations since it provides low-latency data access and can scale with the Lambda service. You can also store data in Amazon EFS for Lambda if you are using this service, and this provides low-latency access to file system storage.

Function states include:

- `Pending` – After Lambda creates the function, it sets the state to pending. While in pending state, Lambda attempts to create or configure resources for the function, such as VPC or EFS resources. Lambda does not invoke a function during pending state. Any invocations or other API actions that operate on the function will fail.
- `Active` – Your function transitions to active state after Lambda completes resource configuration and provisioning. Functions can only be successfully invoked while active.
- `Failed` – Indicates that resource configuration or provisioning encountered an error. When function creation fails, Lambda sets the function state to failed, and you must delete and recreate the function.
- `Inactive` – A function becomes inactive when it has been idle long enough for Lambda to reclaim the external resources that were configured for it. When you try to invoke a function that is inactive, the invocation fails and Lambda sets the function to pending state until the function resources are recreated. If Lambda fails to recreate the resources, the function returns to the inactive state. You might need to resolve any errors and redeploy your function to restore it to the active state.

If you are using SDK-based automation workflows or calling Lambda's service APIs directly, ensure that you check a function's state before invocation to verify that it is active. You can do this with the Lambda API action GetFunction, or by configuring a waiter using the AWS SDK for Java 2.0.

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

You should see the following output:

```
[
 "Active",
 "Successful"
]
```

The following operations fail while function creation is pending:

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Function states during updates

Lambda has two operations for updating functions:

- UpdateFunctionCode: Updates the function's deployment package
- UpdateFunctionConfiguration: Updates the function's configuration

Lambda uses the LastUpdateStatus attribute to track the progress of these update operations. While an update is in progress (when `"LastUpdateStatus": "InProgress"`):

- The function's State remains `Active`.
- Invocations continue to use the function's previous code and configuration until the update completes.
- The following operations fail:
  - UpdateFunctionCode

- UpdateFunctionConfiguration
- PublishVersion
- TagResource

When an update fails (when `"LastUpdateStatus": "Failed"`):

- The function's State remains `Active`.
- Invocations continue to use the function's previous code and configuration.

Example GetFunctionConfiguration response

The following example is the result of GetFunctionConfiguration request on a function undergoing an update.

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "VpcConfig": {
        "SubnetIds": [
            "subnet-071f712345678e7c8",
            "subnet-07fd123456788a036",
            "subnet-0804f77612345cacf"
        ],
        "SecurityGroupIds": [
            "sg-085912345678492fb"
        ],
        "VpcId": "vpc-08e1234569e011e83"
    },
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    ...
}
```

Understanding retry behavior in Lambda

When you invoke a function directly, you determine the strategy for handling errors related to your function code. Lambda does not automatically retry these types of errors on your behalf. To retry, you can manually re-invoke your function, send the failed event to a queue for debugging, or ignore the error. Your function's code might have run completely, partially, or not at all. If you retry, ensure that your function's code can handle the same event multiple times without causing duplicate transactions or other unwanted side effects.

When you invoke a function indirectly, you need to be aware of the retry behavior of the invoker and any service that the request encounters along the way. This includes the following scenarios.

- **Asynchronous invocation** – Lambda retries function errors twice. If the function doesn't have enough capacity to handle all incoming requests, events might wait in the queue for hours to be sent to the function. You can configure a dead-letter queue on the function to capture events that weren't successfully processed. For more information, see the section called “Dead-letter queues”.
- **Event source mappings** – Event source mappings that read from streams retry the entire batch of items. Repeated errors block processing of the affected shard until the error is resolved or the items expire. To detect stalled shards, you can monitor the Iterator Age metric.

  For event source mappings that read from a queue, you determine the length of time between retries and destination for failed events by configuring the visibility timeout and redrive policy on the source queue. For more information, see How Lambda processes records from stream and queue-based event sources and the service-specific topics under Invoking Lambda with events from other AWS services.
- **AWS services** – AWS services can invoke your function synchronously or asynchronously. For synchronous invocation, the service decides whether to retry. For example, Amazon S3 batch operations retries the operation if the Lambda function returns a `TemporaryFailure` response code. Services that proxy requests from an upstream user or client may have a retry strategy or may relay the error response back to the requester. For example, API Gateway always relays the error response back to the requester.

  For asynchronous invocation, the retry logic is the same regardless of the invocation source. By default, Lambda retries a failed asynchronous invocation up to two times. For more information, see How Lambda handles errors and retries with asynchronous invocation.

- **Other accounts and clients** – When you grant access to other accounts, you can use resource-based policies to restrict the services or resources they can configure to invoke your function. To protect your function from being overloaded, consider putting an API layer in front of your function with Amazon API Gateway.

To help you deal with errors in Lambda applications, Lambda integrates with services like Amazon CloudWatch and AWS X-Ray. You can use a combination of logs, metrics, alarms, and tracing to quickly detect and identify issues in your function code, API, or other resources that support your application. For more information, see Monitoring, debugging, and troubleshooting Lambda functions.

Use Lambda recursive loop detection to prevent infinite loops

When you configure a Lambda function to output to the same service or resource that invokes the function, it's possible to create an infinite recursive loop. For example, a Lambda function might write a message to an Amazon Simple Queue Service (Amazon SQS) queue, which then invokes the same function. This invocation causes the function to write another message to the queue, which in turn invokes the function again.

Unintentional recursive loops can result in unexpected charges being billed to your AWS account. Loops can also cause Lambda to scale and use all of your account's available concurrency. To help reduce the impact of unintentional loops, Lambda detects certain types of recursive loops shortly after they occur. By default, when Lambda detects a recursive loop, it stops your function being invoked and notifies you. If your design intentionally uses recursive patterns, you can a change a function's default configuration to allow it to be invoked recursively. See the section called "Allowing a Lambda function to run in a recursive loop" for more information.

Sections

- Understanding recursive loop detection
- Supported AWS services and SDKs
- Recursive loop notifications
- Responding to recursive loop detection notifications
- Allowing a Lambda function to run in a recursive loop
- Supported Regions for Lambda recursive loop detection

Understanding recursive loop detection

Recursive loop detection in Lambda works by tracking events. Lambda is an event-driven compute service that runs your function code when certain events occur. For example, when an item is added to an Amazon SQS queue or Amazon Simple Notification Service (Amazon SNS) topic. Lambda passes events to your function as JSON objects, which contain information about the change in the system state. When an event causes your function to run, this is called an *invocation*.

To detect recursive loops, Lambda uses AWS X-Ray tracing headers. When AWS services that support recursive loop detection send events to Lambda, those events are automatically annotated with metadata. When your Lambda function writes one of these events to another supported AWS

service using a supported version of an AWS SDK, it updates this metadata. The updated metadata includes a count of the number of times that the event has invoked the function.

> **Note**
>
> You don't need to enable X-Ray active tracing for this feature to work. Recursive loop detection is turned on by default for all AWS customers. There is no charge to use the feature.

A *chain of requests* is a sequence of Lambda invocations caused by the same triggering event. For example, imagine that an Amazon SQS queue invokes your Lambda function. Your Lambda function then sends the processed event back to the same Amazon SQS queue, which invokes your function again. In this example, each invocation of your function falls in the same chain of requests.

If your function is invoked approximately 16 times in the same chain of requests, then Lambda automatically stops the next function invocation in that request chain and notifies you. If your function is configured with multiple triggers, then invocations from other triggers aren't affected.

> **Note**
>
> Even when the `maxReceiveCount` setting on the source queue's redrive policy is higher than 16, Lambda recursion protection does not prevent Amazon SQS from retrying the message after a recursive loop is detected and terminated. When Lambda detects a recursive loop and drops subsequent invocations, it returns a `RecursiveInvocationException` to the event source mapping. This increments the `receiveCount` value on the message. Lambda continues to retry the message, and continues to block function invocations, until Amazon SQS determines that the `maxReceiveCount` is exceeded and sends the message to the configured dead-letter queue.

If you have an on-failure destination or dead-letter queue configured for your function, then Lambda also sends the event from the stopped invocation to your destination or dead-letter queue. When configuring a destination or dead-letter queue for your function, be sure not to use an event trigger or event source mapping that your function also uses. If you send events to the

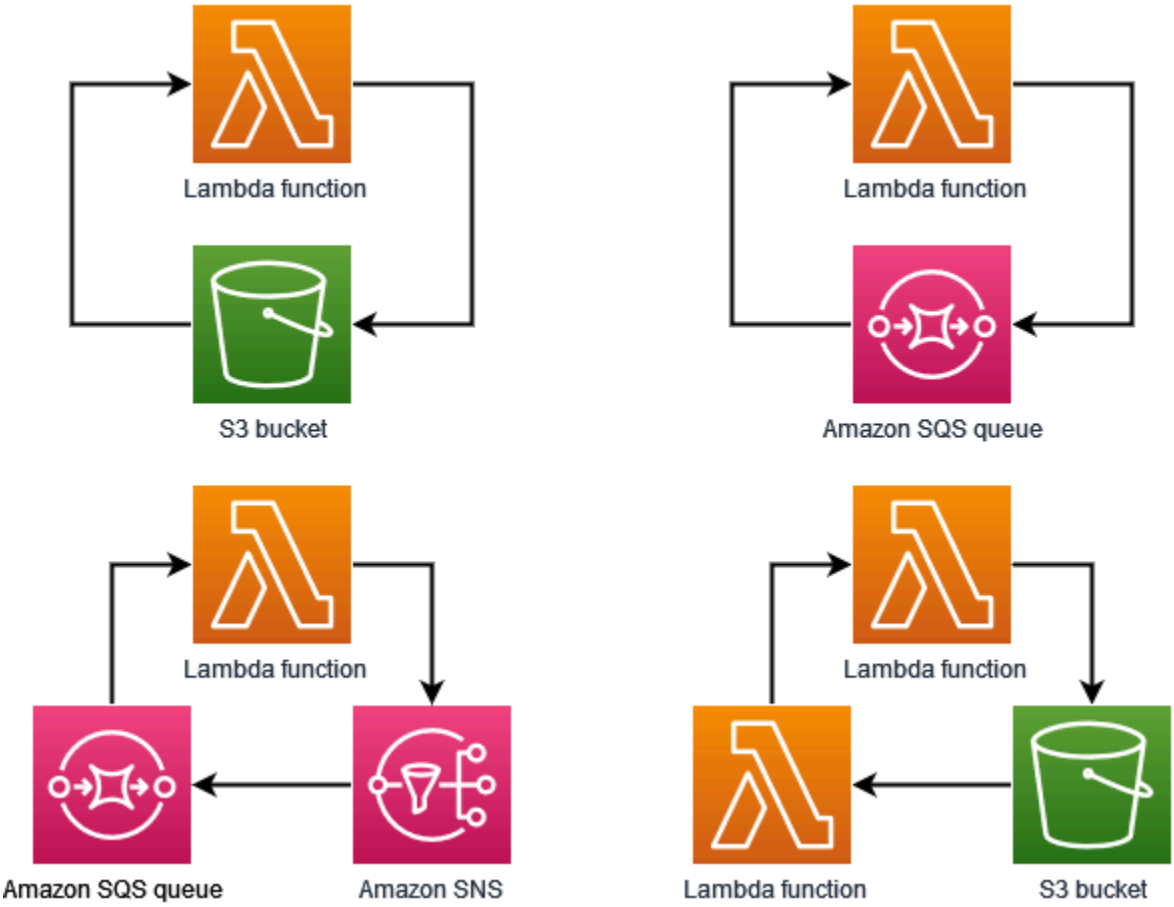
same resource that invokes your function, then you can create another recursive loop and this loop will also be terminated. If you opt out of recursion loop detection, this loop will not be terminated.

Supported AWS services and SDKs

Lambda can detect only recursive loops that include certain supported AWS services. For recursive loops to be detected, your function must also use one of the supported AWS SDKs.

Supported AWS services

Lambda currently detects recursive loops between your functions, Amazon SQS, Amazon S3, and Amazon SNS. Lambda also detects loops comprised only of Lambda functions, which may invoke each other synchronously or asynchronously. The following diagrams show some examples of loops that Lambda can detect:



When another AWS service such as Amazon DynamoDB forms part of the loop, Lambda can't currently detect and stop it.

Because Lambda currently detects only recursive loops involving Amazon SQS, Amazon S3, and Amazon SNS, it's still possible that loops involving other AWS services can result in unintended usage of your Lambda functions.

To guard against unexpected charges being billed to your AWS account, we recommend that you configure Amazon CloudWatch alarms to alert you to unusual usage patterns. For example, you can configure CloudWatch to notify you about spikes in Lambda function concurrency or invocations. You can also configure a billing alarm to notify you when spending in your account exceeds a threshold that you specify. Or, you can use AWS Cost Anomaly Detection to alert you to unusual billing patterns.

Supported AWS SDKs

For Lambda to detect recursive loops, your function must use one of the following SDK versions or higher:

Runtime	Minimum required AWS SDK version
Node.js	2.1147.0 (SDK version 2) 3.105.0 (SDK version 3)
Python	1.24.46 (boto3) 1.27.46 (botocore)
Java 8 and Java 11	2.17.135
Java 17	2.20.81
Java 21	2.21.24
.NET	3.7.293.0
Ruby	3.134.0
PHP	3.232.0
Go	V2 SDK 1.57.0

Some Lambda runtimes such as Python and Node.js include a version of the AWS SDK. If the SDK version included in your function's runtime is lower than the minimum required, then you can add a supported version of the SDK to your function's deployment package. You can also add a supported SDK version to your function using a Lambda layer. For a list of the SDKs included with each Lambda runtime, see Lambda runtimes.

Recursive loop notifications

When Lambda stops a recursive loop, you receive notifications through the Health Dashboard and through email. You can also use CloudWatch metrics to monitor the number of recursive invocations that Lambda has stopped.

Health Dashboard notifications

When Lambda stops a recursive invocation, the Health Dashboard displays a notification on the **Your account health** page, under Open and recent issues. Note that it can take up to 3.5 hours after Lambda stops a recursive invocation before this notification is displayed. For more information about viewing account events in the Health Dashboard, see Getting started with your AWS Health Dashboard – Your account health in the *AWS Health User Guide*.

Email alerts

When Lambda first stops a recursive invocation of your function, it sends you an email alert. Lambda sends a maximum of one email every 24 hours for each function in your AWS account. After Lambda sends an email notification, you won't receive any more emails for that function for another 24 hours, even if Lambda stops further recursive invocations of the function. Note that it can take up to 3.5 hours after Lambda stops a recursive invocation before you receive this email alert.

Lambda sends recursive loop email alerts to your AWS account's primary account contact and alternate operations contact. For information about viewing or updating the email addresses in your account, see Updating contact information in the *AWS General Reference*.

Amazon CloudWatch metrics

The CloudWatch metric `RecursiveInvocationsDropped` records the number of function invocations that Lambda has stopped because your function has been invoked more than approximately 16 times in a single chain of requests. Lambda emits this metric as soon as it stops a recursive invocation. To view this metric, follow the instructions for Viewing metrics on the CloudWatch console and choose the metric `RecursiveInvocationsDropped`.

Responding to recursive loop detection notifications

When your function is invoked more than approximately 16 times by the same triggering event, Lambda stops the next function invocation for that event to break the recursive loop. To prevent a reoccurrence of a recursive loop that Lambda has broken, do the following:

- Reduce your function's available concurrency to zero, which throttles all future invocations.
- Remove or disable the trigger or event source mapping that's invoking your function.
- Identify and fix code defects that write events back to the AWS resource that's invoking your function. A common source of defects occurs when you use variables to define a function's event source and target. Check that you're not using the same value for both variables.

Additionally, if the event source for your Lambda function is an Amazon SQS queue, then consider configuring a dead-letter queue on the source queue.

> **Note**
>
> Make sure that you configure the dead-letter queue on the source queue, not on the Lambda function. The dead-letter queue that you configure on a function is used for the function's asynchronous invocation queue, not for event source queues.

If the event source is an Amazon SNS topic, then consider adding an on-failure destination for your function.

To reduce your function's available concurrency to zero (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of your function.
3. Choose **Throttle**.
4. In the **Throttle your function** dialog box, choose **Confirm**.

To remove a trigger or event source mapping for your function (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of your function.

3. Choose the **Configuration** tab, then choose **Triggers**.
4. Under **Triggers**, select the trigger or event source mapping that you want to delete, then choose **Delete**.
5. In the **Delete triggers** dialog box, choose **Delete**.

To disable an event source mapping for your function (AWS CLI)

1. To find the UUID for the event source mapping that you want to disable, run the AWS Command Line Interface (AWS CLI) list-event-source-mappings command.

```
aws lambda list-event-source-mappings
```

2. To disable the event source mapping, run the following AWS CLI update-event-source-mapping command.

```
aws lambda update-event-source-mapping --function-name MyFunction \
--uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
```

Allowing a Lambda function to run in a recursive loop

If your design intentionally uses a recursive loop, you can configure a Lambda function to allow it to be invoked recursively. We recommend that you avoid using recursive loops in your design. Implementation errors can lead to recursive invocations using all of your AWS account's available concurrency and to unexpected charges being billed to your account.

> **Important**
>
> If you use recursive loops, treat them with caution. Implement best practice guard rails to minimize the risks of implementation errors. To learn more about best practices for using recursive patterns, see Recursive patterns that cause run-away Lambda functions in Serverless Land.

You can configure functions to allow recursive loops using the Lambda console, the AWS Command Line Interface (AWS CLI), and the PutFunctionRecursionConfig API. You can also configure a function's recursive loop detection setting in AWS SAM and CloudFormation.

By default, Lambda detects and terminates recursive loops. Unless your design intentionally uses a recursive loop, we recommend that you don't change your functions' default configuration.

Note that when you configure a function to allow recursive loops, the CloudWatch metric `RecursiveInvocationsDropped` isn't emitted.

Console

To allow a function to run in a recursive loop (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of your function to open the function detail page.
3. Choose the **Configuration** tab, then choose **Concurrency and recursion detection**.
4. Beside **Recursive loop detection**, choose **Edit**.
5. Select **Allow recursive loops**.
6. Choose **Save**.

AWS CLI

You can use the PutFunctionRecursionConfig API to allow your function to be invoked in a recursive loop. Specify `Allow` for the recursive loop parameter. For example, you can call this API with the `put-function-recursion-config` AWS CLI command:

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Allow
```

You can change your function's configuration back to the default setting so that Lambda terminates recursive loops when it detects them. Edit your function's configuration using the Lambda console or the AWS CLI.

Console

To configure a function so that recursive loops are terminated (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of your function to open the function detail page.
3. Choose the **Configuration** tab, then choose **Concurrency and recursion detection**.

4. Beside **Recursive loop detection**, choose **Edit**.
5. Select **Terminate recursive loops**.
6. Choose **Save**.

AWS CLI

You can use the PutFunctionRecursionConfig API to configure your function so that Lambda terminates recursive loops when it detects them. Specify `Terminate` for the recursive loop parameter. For example, you can call this API with the `put-function-recursion-config` AWS CLI command:

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Terminate
```

Supported Regions for Lambda recursive loop detection

Lambda recursive loop detection is supported in all commercial Regions except Mexico (Central) and Asia Pacific (New Zealand).

Creating and managing Lambda function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API.

> **Tip**
>
> Lambda offers two ways to invoke your function through an HTTP endpoint: function URLs and Amazon API Gateway. If you're not sure which is the best method for your use case, see the section called “Function URLs vs Amazon API Gateway”.

When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> Function URLs are not supported in the following AWS Regions: Asia Pacific (Hyderabad) (`ap-south-2`), Asia Pacific (Melbourne) (`ap-southeast-4`), Asia Pacific (Malaysia) (`ap-southeast-5`), Asia Pacific (New Zealand) (`ap-southeast-6`), Asia Pacific (Thailand) (`ap-southeast-7`), Asia Pacific (Taipei) (`ap-east-2`), Canada West (Calgary) (`ca-west-1`), Europe (Spain) (`eu-south-2`), Europe (Zurich) (`eu-central-2`), Israel (Tel Aviv) (`il-central-1`), and Middle East (UAE) (`me-central-1`).

Function URLs are dual stack-enabled, supporting IPv4 and IPv6. After you configure a function URL for your function, you can invoke your function through its HTTP(S) endpoint via a web browser, curl, Postman, or any HTTP client.

> **Note**
>
> You can access your function URL through the public Internet only. While Lambda functions do support AWS PrivateLink, function URLs do not.

Lambda function URLs use resource-based policies for security and access control. Function URLs also support cross-origin resource sharing (CORS) configuration options.

You can apply function URLs to any function alias, or to the $LATEST unpublished function version. You can't add a function URL to any other function version.

The following section show how to create and manage a function URL using the Lambda console, AWS CLI, and CloudFormation template

Topics

- Creating a function URL (console)
- Creating a function URL (AWS CLI)
- Adding a function URL to a CloudFormation template
- Cross-origin resource sharing (CORS)
- Throttling function URLs
- Deactivating function URLs
- Deleting function URLs
- Control access to Lambda function URLs
- Invoking Lambda function URLs
- Monitoring Lambda function URLs
- Select a method to invoke your Lambda function using an HTTP request
- Tutorial: Creating a webhook endpoint using a Lambda function URL

Creating a function URL (console)

Follow these steps to create a function URL using the console.

To create a function URL for an existing function

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to create the function URL for.
3. Choose the **Configuration** tab, and then choose **Function URL**.
4. Choose **Create function URL**.

5. For **Auth type**, choose **AWS_IAM** or **NONE**. For more information about function URL authentication, see Access control.
6. (Optional) Select **Configure cross-origin resource sharing (CORS)**, and then configure the CORS settings for your function URL. For more information about CORS, see Cross-origin resource sharing (CORS).
7. Choose **Save**.

This creates a function URL for the $LATEST unpublished version of your function. The function URL appears in the **Function overview** section of the console.

To create a function URL for an existing alias

1. Open the Functions page of the Lambda console.
2. Choose the name of the function with the alias that you want to create the function URL for.
3. Choose the **Aliases** tab, and then choose the name of the alias that you want to create the function URL for.
4. Choose the **Configuration** tab, and then choose **Function URL**.
5. Choose **Create function URL**.
6. For **Auth type**, choose **AWS_IAM** or **NONE**. For more information about function URL authentication, see Access control.
7. (Optional) Select **Configure cross-origin resource sharing (CORS)**, and then configure the CORS settings for your function URL. For more information about CORS, see Cross-origin resource sharing (CORS).
8. Choose **Save**.

This creates a function URL for your function alias. The function URL appears in the console's **Function overview** section for your alias.

To create a new function with a function URL

To create a new function with a function URL (console)

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Under **Basic information**, do the following:

a. For **Function name**, enter a name for your function, such as **`my-function`**.

b. For **Runtime**, choose the language runtime that you prefer, such as **Node.js 24**.

c. For **Architecture**, choose either **x86_64** or **arm64**.

d. Expand **Permissions**, then choose whether to create a new execution role or use an existing one.

4. Expand **Advanced settings**, and then select **Function URL**.
5. For **Auth type**, choose **AWS_IAM** or **NONE**. For more information about function URL authentication, see Access control.
6. (Optional) Select **Configure cross-origin resource sharing (CORS)**. By selecting this option during function creation, your function URL allows requests from all origins by default. You can edit the CORS settings for your function URL after creating the function. For more information about CORS, see Cross-origin resource sharing (CORS).
7. Choose **Create function**.

This creates a new function with a function URL for the $LATEST unpublished version of the function. The function URL appears in the **Function overview** section of the console.

Creating a function URL (AWS CLI)

To create a function URL for an existing Lambda function using the AWS Command Line Interface (AWS CLI), run the following command:

```
aws lambda create-function-url-config \
    --function-name my-function \
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

This adds a function URL to the **`prod`** qualifier for the function **`my-function`**. For more information about these configuration parameters, see CreateFunctionUrlConfig in the API reference.

> **Note**
>
> To create a function URL via the AWS CLI, the function must already exist.

Adding a function URL to a CloudFormation template

To add an `AWS::Lambda::Url` resource to your CloudFormation template, use the following syntax:

JSON

```
{
  "Type" : "AWS::Lambda::Url",
  "Properties" : {
      "AuthType" : String,
      "Cors" : Cors,
      "Qualifier" : String,
      "TargetFunctionArn" : String
    }
}
```

YAML

```
Type: AWS::Lambda::Url
Properties:
  AuthType: String
  Cors:
    Cors
  Qualifier: String
  TargetFunctionArn: String
```

Parameters

- (Required) `AuthType` – Defines the type of authentication for your function URL. Possible values are either `AWS_IAM` or `NONE`. To restrict access to authenticated users only, set to `AWS_IAM`. To bypass IAM authentication and allow any user to make requests to your function, set to `NONE`.
- (Optional) `Cors` – Defines the CORS settings for your function URL. To add `Cors` to your `AWS::Lambda::Url` resource in CloudFormation, use the following syntax.

  Example AWS::Lambda::Url.Cors (JSON)

  ```
  {
    "AllowCredentials" : Boolean,
    "AllowHeaders" : [ String, ... ],
  ```

```
  "AllowMethods" : [ String, ... ],
  "AllowOrigins" : [ String, ... ],
  "ExposeHeaders" : [ String, ... ],
  "MaxAge" : Integer
}
```

Example AWS::Lambda::Url.Cors (YAML)

```
  AllowCredentials: Boolean
  AllowHeaders:
    - String
  AllowMethods:
    - String
  AllowOrigins:
    - String
  ExposeHeaders:
    - String
  MaxAge: Integer
```

- (Optional) `Qualifier` – The alias name.
- (Required) `TargetFunctionArn` – The name or Amazon Resource Name (ARN) of the Lambda function. Valid name formats include the following:
  - **Function name** – `my-function`
  - **Function ARN** – `arn:aws:lambda:us-west-2:123456789012:function:my-function`
  - **Partial ARN** – `123456789012:function:my-function`

Cross-origin resource sharing (CORS)

To define how different origins can access your function URL, use cross-origin resource sharing (CORS). We recommend configuring CORS if you intend to call your function URL from a different domain. Lambda supports the following CORS headers for function URLs.

CORS header	CORS configuration property	Example values
Access-Control-Allow-Origin	`AllowOrigins`	`*` (allow all origins) `https://www.example.com`

CORS header	CORS configuration property	Example values
		`http://localhost:60905`
Access-Control-Allow-Methods	`AllowMethods`	`GET`, `POST`, `DELETE`, `*`
Access-Control-Allow-Headers	`AllowHeaders`	`Date`, `Keep-Alive` , `X-Custom-Header`
Access-Control-Expose-Headers	`ExposeHeaders`	`Date`, `Keep-Alive` , `X-Custom-Header`
Access-Control-Allow-Credentials	`AllowCredentials`	`TRUE`
Access-Control-Max-Age	`MaxAge`	`5` (default), `300`

When you configure CORS for a function URL using the Lambda console or the AWS CLI, Lambda automatically adds the CORS headers to all responses through the function URL. Alternatively, you can manually add CORS headers to your function response. If there are conflicting headers, the expected behavior depends on the type of request:

- For preflight requests such as OPTIONS requests, the configured CORS headers on the function URL take precedence. Lambda returns only these CORS headers in the response.
- For non-preflight requests such as GET or POST requests, Lambda returns both the configured CORS headers on the function URL, as well as the CORS headers returned by the function. This can result in duplicate CORS headers in the response. You may see an error similar to the following: `The 'Access-Control-Allow-Origin' header contains multiple values '*, *', but only one is allowed.`

In general, we recommend configuring all CORS settings on the function URL, rather than sending CORS headers manually in the function response.

Throttling function URLs

Throttling limits the rate at which your function processes requests. This is useful in many situations, such as preventing your function from overloading downstream resources, or handling a sudden surge in requests.

You can throttle the rate of requests that your Lambda function processes through a function URL by configuring reserved concurrency. Reserved concurrency limits the number of maximum concurrent invocations for your function. Your function's maximum request rate per second (RPS) is equivalent to 10 times the configured reserved concurrency. For example, if you configure your function with a reserved concurrency of 100, then the maximum RPS is 1,000.

Whenever your function concurrency exceeds the reserved concurrency, your function URL returns an HTTP 429 status code. If your function receives a request that exceeds the 10x RPS maximum based on your configured reserved concurrency, you also receive an HTTP 429 error. For more information about reserved concurrency, see Configuring reserved concurrency for a function.

Deactivating function URLs

In an emergency, you might want to reject all traffic to your function URL. To deactivate your function URL, set the reserved concurrency to zero. This throttles all requests to your function URL, resulting in HTTP 429 status responses. To reactivate your function URL, delete the reserved concurrency configuration, or set the configuration to an amount greater than zero.

Deleting function URLs

When you delete a function URL, you can't recover it. Creating a new function URL will result in a different URL address.

> **Note**
>
> If you delete a function URL with auth type `NONE`, Lambda doesn't automatically delete the associated resource-based policy. If you want to delete this policy, you must manually do so.

1. Open the Functions page of the Lambda console.
2. Choose the name of the function.
3. Choose the **Configuration** tab, and then choose **Function URL**.

4. Choose **Delete**.
5. Enter the word *delete* into the field to confirm the deletion.
6. Choose **Delete**.

> **Note**
>
> When you delete a function that has a function URL, Lambda asynchronously deletes the function URL. If you immediately create a new function with the same name in the same account, it is possible that the original function URL will be mapped to the new function instead of deleted.

Control access to Lambda function URLs

> **Note**
>
> Starting in October 2025, new function URLs will require both `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` permissions.

You can control access to your Lambda function URLs using the AuthType parameter combined with resource-based policies attached to your specific function. The configuration of these two components determines who can invoke or perform other administrative actions on your function URL.

The `AuthType` parameter determines how Lambda authenticates or authorizes requests to your function URL. When you configure your function URL, you must specify one of the following `AuthType` options:

- `AWS_IAM` – Lambda uses AWS Identity and Access Management (IAM) to authenticate and authorize requests based on the IAM principal's identity policy and the function's resource-based policy. Choose this option if you want only authenticated users and roles to invoke your function using the function URL.

- `NONE` – Lambda doesn't perform any authentication before invoking your function. However, your function's resource-based policy is always in effect and must grant public access before your

function URL can receive requests. Choose this option to allow public, unauthenticated access to your function URL.

For additional insights into security, you can use AWS Identity and Access Management Access Analyzer to get a comprehensive analysis of external access to your function URL. IAM Access Analyzer also monitors for new or updated permissions on your Lambda functions to help you identify permissions that grant public and cross-account access. You can use IAM Access Analyzer at no charge. To get started with IAM Access Analyzer, see Using AWS IAM Access Analyzer.

This page contains examples of resource-based policies for both auth types, and how to create these policies using the AddPermission API operation or the Lambda console. For information about how to invoke your function URL after you've set up permissions, see Invoking Lambda function URLs.

Topics

- Using the AWS_IAM auth type
- Using the NONE auth type
- Governance and access control

Using the `AWS_IAM` auth type

If you choose the `AWS_IAM` auth type, users who need to invoke your Lambda function URL must have the `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` permissions. Depending on who makes the invocation request, you might have to grant this permission using a resource-based policy.

If the principal making the request is in the same AWS account as the function URL, then the principal must **either** have `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` permissions in their identity-based policy, **or** have permissions granted to them in the function's resource-based policy. In other words, a resource-based policy is optional if the user already has `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` permissions in their identity-based policy. Policy evaluation follows the rules outlined in Policy evaluation logic.

If the principal making the request is in a different account, then the principal must have **both** an identity-based policy that gives them `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` permissions **and** permissions granted to them in a resource-based

policy on the function that they are trying to invoke. Policy evaluation follows the rules outlined in Determining whether a cross-account request is allowed.

The following resource-based policy allows the `example` role in AWS account `444455556666` to invoke the function URL associated with function `my-function`. The lambda:InvokedViaFunctionUrl context key restricts the `lambda:InvokeFunction` action to function URL calls. This means that the principal must use the function URL to invoke the function. If you don't include `lambda:InvokedViaFunctionUrl`, the principal can invoke your function through other invocation methods, in addition to the function URL.

Example— Cross-account resource-based policy

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
      },
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "AWS_IAM"
        }
      }
    },
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Condition": {
        "Bool": {
          "lambda:InvokedViaFunctionUrl": "true"
        }
      }
```

```
    }
  ]
}
```

You can create this resource-based policy through the console using the following steps:

To grant URL invocation permissions to another account (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of the function that you want to grant URL invocation permissions for.
3. Choose the **Configuration** tab, and then choose **Permissions**.
4. Under **Resource-based policy**, choose **Add permissions**.
5. Choose **Function URL**.
6. For **Auth type**, choose **AWS_IAM**.
7. Enter a **Statement ID** for your policy statement.
8. For **Principal**, enter the account ID or the Amazon Resource Name (ARN) of the user or role that you want to grant permissions to. For example: **444455556666**.
9. Choose **Save**.

Alternatively, you can create this policy using the following add-permission AWS Command Line Interface (AWS CLI) commands. When you use the AWS CLI, you must add the `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` statements separately. For example:

```
aws lambda add-permission --function-name my-function \
  --statement-id UrlPolicyInvokeURL \
  --action lambda:InvokeFunctionUrl \
  --principal 444455556666 \
  --function-url-auth-type AWS_IAM
```

```
aws lambda add-permission --function-name my-function \
  --statement-id UrlPolicyInvokeFunction \
  --action lambda:InvokeFunction \
  --principal 444455556666 \
  --invoked-via-function-url
```

Using the NONE auth type

> **Important**
>
> When your function URL auth type is NONE and you have a resource-based policy that grants public access, any unauthenticated user with your function URL can invoke your function.

In some cases, you might want your function URL to be public. For example, you might want to serve requests made directly from a web browser. To allow public access to your function URL, choose the NONE auth type.

If you choose the NONE auth type, Lambda doesn't use IAM to authenticate requests to your function URL. However, your function must have a resource-based policy that allows `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction`. When you create a function URL with auth type NONE using the console or AWS Serverless Application Model (AWS SAM), Lambda automatically creates the resource-based policy for you. If you're using the AWS CLI, AWS CloudFormation, or the Lambda API directly, you must add the policy yourself.

We recommend that you include the lambda:InvokedViaFunctionUrl context key in your resource-based policies when using the NONE auth type. This context key ensures that the function can only be invoked through the function URL and not through other invocation methods.

Note the following about this policy:

- All entities can call `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction`. This means that anyone who has your function URL can invoke your function.
- The `lambda:FunctionUrlAuthType` condition key value is NONE. This means that the policy statement allows access only when your function URL's auth type is also NONE.
- The `lambda:InvokedViaFunctionUrl` condition ensures that the function can only be invoked through the function URL and not through other invocation methods.

Example— Default resource-based policy for NONE auth type

JSON

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "FunctionURLAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "NONE"
        }
      }
    },
    {
      "Sid": "FunctionURLInvokeAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
      "Condition": {
        "Bool": {
          "lambda:InvokedViaFunctionUrl": "true"
        }
      }
    }
  ]
}
```

Create the resource-based policy using the AWS CLI

Unless you use the console or AWS SAM to create a function URL with auth type `NONE`, you must add the resource-based policy yourself. Use the following commands to create statements for the `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` permissions. Each statement must be added in a separate command.

```
aws lambda add-permission \
  --function-name UrlTestFunction \
  --statement-id UrlPolicyInvokeURL \
  --action lambda:InvokeFunctionUrl \
  --principal * \
```

```
  --function-url-auth-type NONE
```

```
aws lambda add-permission \
  --function-name UrlTestFunction \
  --statement-id UrlPolicyInvokeFunction \
  --action lambda:InvokeFunction \
  --principal * \
  --invoked-via-function-url
```

> **Note**
>
> If you delete a function URL with auth type NONE, Lambda doesn't automatically delete the associated resource-based policy. If you want to delete this policy, you must manually do so.

If a function's resource-based policy doesn't grant `lambda:invokeFunctionUrl` and `lambda:InvokeFunction` permissions, users will get a 403 Forbidden error code when they try to invoke your function URL. This will occur even if the function URL uses the NONE auth type.

Governance and access control

In addition to function URL invocation permissions, you can also control access on actions used to configure function URLs. Lambda supports the following IAM policy actions for function URLs:

- `lambda:InvokeFunctionUrl` – Invoke a Lambda function using the function URL.
- `lambda:CreateFunctionUrlConfig` – Create a function URL and set its `AuthType`.
- `lambda:UpdateFunctionUrlConfig` – Update a function URL configuration and its `AuthType`.
- `lambda:GetFunctionUrlConfig` – View the details of a function URL.
- `lambda:ListFunctionUrlConfigs` – List function URL configurations.
- `lambda:DeleteFunctionUrlConfig` – Delete a function URL.

To allow or deny function URL access to other AWS entities, include these actions in IAM policies. For example, the following policy grants the `example` role in AWS account `444455556666` permissions to update the function URL for function **`my-function`** in account `123456789012`.

Example cross-account function URL policy

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:UpdateFunctionUrlConfig",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-
function"
        }
    ]
}
```

Condition keys

For fine-grained access control over your function URLs, use condition context keys. Lambda supports the following context keys for function URLs:

- `lambda:FunctionUrlAuthType` – Defines an enum value describing the auth type that your function URL uses. The value can be either `AWS_IAM` or `NONE`.
- `lambda:InvokedViaFunctionUrl` – Restricts the `lambda:InvokeFunction` action to calls made through the function URL. This ensures that the function can only be invoked using the function URL and not through other invocation methods. For examples of resource-based policies that use the `lambda:InvokedViaFunctionUrl` context key, see the examples in Using the `AWS_IAM` auth type and Using the `NONE` auth type.

You can use these context keys in policies associated with your function. For example, you might want to restrict who can make configuration changes to your function URLs. To deny all `UpdateFunctionUrlConfig` requests to any function with URL auth type `NONE`, you can define the following policy:

Example function URL policy with explicit deny

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action":[
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

To grant the `example` role in AWS account `444455556666` permissions to make `CreateFunctionUrlConfig` and `UpdateFunctionUrlConfig` requests on functions with URL auth type `AWS_IAM`, you can define the following policy:

Example function URL policy with explicit allow

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action":[
```

```
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

You can also use this condition key in a service control policy (SCP). Use SCPs to manage permissions across an entire organization in AWS Organizations. For example, to deny users from creating or updating function URLs that use anything other than the `AWS_IAM` auth type, use the following service control policy:

Example function URL SCP with explicit deny

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:*:123456789012:function:*",
            "Condition": {
                "StringNotEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

Invoking Lambda function URLs

A function URL is a dedicated HTTP(S) endpoint for your Lambda function. You can create and configure a function URL through the Lambda console or the Lambda API.

> **Tip**
>
> Lambda offers two ways to invoke your function through an HTTP endpoint: function URLs and Amazon API Gateway. If you're not sure which is the best method for your use case, see the section called “Function URLs vs Amazon API Gateway”.

When you create a function URL, Lambda automatically generates a unique URL endpoint for you. Once you create a function URL, its URL endpoint never changes. Function URL endpoints have the following format:

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> Function URLs are not supported in the following AWS Regions: Asia Pacific (Hyderabad) (`ap-south-2`), Asia Pacific (Melbourne) (`ap-southeast-4`), Asia Pacific (Malaysia) (`ap-southeast-5`), Asia Pacific (New Zealand) (`ap-southeast-6`), Asia Pacific (Thailand) (`ap-southeast-7`), Asia Pacific (Taipei) (`ap-east-2`), Canada West (Calgary) (`ca-west-1`), Europe (Spain) (`eu-south-2`), Europe (Zurich) (`eu-central-2`), Israel (Tel Aviv) (`il-central-1`), and Middle East (UAE) (`me-central-1`).

Function URLs are dual stack-enabled, supporting IPv4 and IPv6. After configuring your function URL, you can invoke your function through its HTTP(S) endpoint via a web browser, curl, Postman, or any HTTP client. To invoke a function URL, you must have `lambda:InvokeFunctionUrl` and `lambda:InvokeFunction` permissions. For more information, see Access control.

Topics

- Function URL invocation basics
- Request and response payloads

Function URL invocation basics

If your function URL uses the `AWS_IAM` auth type, you must sign each HTTP request using AWS Signature Version 4 (SigV4). Tools such as Postman offer built-in ways to sign your requests with SigV4.

If you don't use a tool to sign HTTP requests to your function URL, you must manually sign each request using SigV4. When your function URL receives a request, Lambda also calculates the SigV4 signature. Lambda processes the request only if the signatures match. For instructions on how to manually sign your requests with SigV4, see Signing AWS requests with Signature Version 4 in the *Amazon Web Services General Reference Guide*.

If your function URL uses the `NONE` auth type, you don't have to sign your requests using SigV4. You can invoke your function using a web browser, curl, Postman, or any HTTP client.

To test simple `GET` requests to your function, use a web browser. For example, if your function URL is `https://abcdefg.lambda-url.us-east-1.on.aws`, and it takes in a string parameter `message`, your request URL could look like this:

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

To test other HTTP requests, such as a `POST` request, you can use a tool such as curl. For example, if you want to include some JSON data in a `POST` request to your function URL, you could use the following curl command:

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

Request and response payloads

When a client calls your function URL, Lambda maps the request to an event object before passing it to your function. Your function's response is then mapped to an HTTP response that Lambda sends back to the client through the function URL.

The request and response event formats follow the same schema as the Amazon API Gateway payload format version 2.0.

Request payload format

A request payload has the following structure:

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<urlid>",
    "authentication": null,
    "authorizer": {
        "iam": {
                "accessKey": "AKIA...",
                "accountId": "111122223333",
                "callerId": "AIDA...",
                "cognitoIdentity": null,
                "principalOrgId": null,
                "userArn": "arn:aws:iam::111122223333:user/example-user",
                "userId": "AIDA..."
        }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
```

```
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from client!",
  "pathParameters": null,
  "isBase64Encoded": false,
  "stageVariables": null
}
```

Parameter	Description	Example
`version`	The payload format version for this event. Lambda function URLs currently support payload format version 2.0.	`2.0`
`routeKey`	Function URLs don't use this parameter. Lambda sets this to `$default` as a placeholder.	`$default`
`rawPath`	The request path. For example, if the request URL is `https://{url-id}.lambda-url.{region}.on.aws/example/test/demo`, then the raw path value is `/example/test/demo`.	`/example/test/demo`
`rawQueryString`	The raw string containing the request's query string parameters. Supported characters include `a-z`, `A-Z`, `0-9`, `.`, `_`, `-`, `%`, `&`, `=`, and `+`.	`"?parameter1=value1&parameter2=value2"`

Parameter	Description	Example
`cookies`	An array containing all cookies sent as part of the request.	`["Cookie_1=Value_1", "Cookie_2=Value_2"]`
`headers`	The list of request headers, presented as key-value pairs.	`{"header1": "value1", "header2": "value2"}`
`queryStringParameters`	The query parameters for the request. For example, if the request URL is `https://{url-id}.lambda-url.{region}.on.aws/example?name=Jane` , then the `queryStringParameters` value is a JSON object with a key of `name` and a value of `Jane`.	`{"name": "Jane"}`
`requestContext`	An object that contains additional information about the request, such as the `requestId` , the time of the request, and the identity of the caller if authorized via AWS Identity and Access Management (IAM).	
`requestContext.accountId`	The AWS account ID of the function owner.	`"123456789012"`
`requestContext.apiId`	The ID of the function URL.	`"33anwqw8fj"`
`requestContext.authentication`	Function URLs don't use this parameter. Lambda sets this to `null`.	`null`

Parameter	Description	Example
`requestContext.authorizer`	An object that contains information about the caller identity, if the function URL uses the `AWS_IAM` auth type. Otherwise, Lambda sets this to `null`.	
`requestContext.authorizer.iam.accessKey`	The access key of the caller identity.	`"AKIAIOSFODNN7EXAMPLE"`
`requestContext.authorizer.iam.accountId`	The AWS account ID of the caller identity.	`"111122223333"`
`requestContext.authorizer.iam.callerId`	The ID (user ID) of the caller.	`"AIDACKCEVSQ6C2EXAMPLE"`
`requestContext.authorizer.iam.cognitoIdentity`	Function URLs don't use this parameter. Lambda sets this to `null` or excludes this from the JSON.	`null`
`requestContext.authorizer.iam.principalOrgId`	The principal org ID associated with the caller identity.	`"AIDACKCEVSQORGEXAMPLE"`
`requestContext.authorizer.iam.userArn`	The user Amazon Resource Name (ARN) of the caller identity.	`"arn:aws:iam::111122223333:user/example-user"`
`requestContext.authorizer.iam.userId`	The user ID of the caller identity.	`"AIDACOSFODNN7EXAMPLE2"`

Parameter	Description	Example
`requestContext.domainName`	The domain name of the function URL.	`"<url-id>.lambda-url.us-west-2.on.aws"`
`requestContext.domainPrefix`	The domain prefix of the function URL.	`"<url-id>"`
`requestContext.http`	An object that contains details about the HTTP request.	
`requestContext.http.method`	The HTTP method used in this request. Valid values include `GET`, `POST`, `PUT`, `HEAD`, `OPTIONS`, `PATCH`, and `DELETE`.	`GET`
`requestContext.http.path`	The request path. For example, if the request URL is `https://{url-id}.lambda-url.{region}.on.aws/example/test/demo` , then the path value is `/example/test/demo` .	`/example/test/demo`
`requestContext.http.protocol`	The protocol of the request.	`HTTP/1.1`
`requestContext.http.sourceIp`	The source IP address of the immediate TCP connection making the request.	`123.123.123.123`

Parameter	Description	Example
`requestContext.http.userAgent`	The User-Agent request header value.	`Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0`
`requestContext.requestId`	The ID of the invocation request. You can use this ID to trace invocation logs related to your function.	`e1506fd5-9e7b-434f-bd42-4f8fa224b599`
`requestContext.routeKey`	Function URLs don't use this parameter. Lambda sets this to `$default` as a placeholder.	`$default`
`requestContext.stage`	Function URLs don't use this parameter. Lambda sets this to `$default` as a placeholder.	`$default`
`requestContext.time`	The timestamp of the request.	`"07/Sep/2021:22:50:22 +0000"`
`requestContext.timeEpoch`	The timestamp of the request, in Unix epoch time.	`"1631055022677"`
body	The body of the request. If the content type of the request is binary, the body is base64-encoded.	`{"key1": "value1", "key2": "value2"}`
`pathParameters`	Function URLs don't use this parameter. Lambda sets this to `null` or excludes this from the JSON.	`null`

Parameter	Description	Example
isBase64Encoded	TRUE if the body is a binary payload and base64-encoded. FALSE otherwise.	FALSE
stageVariables	Function URLs don't use this parameter. Lambda sets this to null or excludes this from the JSON.	null

Response payload format

When your function returns a response, Lambda parses the response and converts it into an HTTP response. Function response payloads have the following format:

```
{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/json",
        "My-Custom-Header": "Custom Value"
    },
    "body": "{ \"message\": \"Hello, world!\" }",
    "cookies": [
        "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=78000"
    ],
    "isBase64Encoded": false
}
```

Lambda infers the response format for you. If your function returns valid JSON and doesn't return a statusCode, Lambda assumes the following:

- statusCode is 200.

> **Note**
>
> The valid statusCode are within the range of 100 to 599.

- content-type is application/json.

- body is the function response.
- `isBase64Encoded` is `false`.

The following examples show how the output of your Lambda function maps to the response payload, and how the response payload maps to the final HTTP response. When the client invokes your function URL, they see the HTTP response.

Example output for a string response

Lambda function output	Interpreted response output	HTTP response (what the client sees)

`"Hello, world!"`	`{ "statusCode": 200, "body": "Hello, world!", "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 15`
		
		`"Hello, world!"`

Example output for a JSON response

Lambda function output	Interpreted response output	HTTP response (what the client sees)

`{ "message": "Hello, world!" }`	`{ "statusCode": 200, "body": { "message": "Hello, world!" }, "headers": { "content-type": "application/json"`	`HTTP/2 200`
		`date: Wed, 08 Sep 2021 18:02:24 GMT`
		`content-type: application/json`
		`content-length: 34`
		
		`{`

Lambda function output	Interpreted response output	HTTP response (what the client sees)
	<pre> }, "isBase64Encoded": false }</pre>	<pre> "message": "Hello, world!" }</pre>

Example output for a custom response

Lambda function output	Interpreted response output	HTTP response (what the client sees)

<pre>{
   "statusCode": 201,
    "headers": {
        "Content-Type":
 "application/json",
        "My-Custom-
Header": "Custom Value"
    },
    "body": JSON.stri
ngify({
        "message":
 "Hello, world!"
    }),
    "isBase64Encoded":
 false
}</pre>
	<pre>{
	   "statusCode": 201,
	    "headers": {
	        "Content-Type":
	 "application/json",
	        "My-Custom-
	Header": "Custom Value"
	    },
	    "body": JSON.stri
	ngify({
	        "message":
	 "Hello, world!"
	    }),
	    "isBase64Encoded":
	 false
	}</pre>
		<pre>HTTP/2 201
		date: Wed, 08 Sep 2021
		 18:02:24 GMT
		content-type: applicati
		on/json
		content-length: 27
		my-custom-header:
		 Custom Value
		
		{
		  "message": "Hello,
		 world!"
		}</pre>

Cookies

To return cookies from your function, don't manually add `set-cookie` headers. Instead, include the cookies in your response payload object. Lambda automatically interprets this and adds them as `set-cookie` headers in your HTTP response, as in the following example.

Lambda function output	HTTP response (what the client sees)

```
{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/
json",
        "My-Custom-Header": "Custom
 Value"
    },
    "body": JSON.stringify({
        "message": "Hello, world!"
    }),
    "cookies": [
        "Cookie_1=Value1; Expires=21
 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=7
8000"
    ],
    "isBase64Encoded": false
}
```

```
HTTP/2 201
date: Wed, 08 Sep 2021 18:02:24 GMT
content-type: application/json
content-length: 27
my-custom-header: Custom Value
set-cookie: Cookie_1=Value2;
 Expires=21 Oct 2021 07:48 GMT
set-cookie: Cookie_2=Value2; Max-
Age=78000

{
  "message": "Hello, world!"
}
```

Monitoring Lambda function URLs

You can use AWS CloudTrail and Amazon CloudWatch to monitor your function URLs.

Topics

- Monitoring function URLs with CloudTrail
- CloudWatch metrics for function URLs

Monitoring function URLs with CloudTrail

For function URLs, Lambda automatically supports logging the following API operations as events in CloudTrail log files:

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

Each log entry contains information about the caller identity, when the request was made, and other details. You can see all events within the last 90 days by viewing your CloudTrail **Event history**. To retain records past 90 days, you can create a trail.

By default, CloudTrail doesn't log `InvokeFunctionUrl` requests, which are considered data events. However, you can turn on data event logging in CloudTrail. For more information, see Logging data events for trails in the *AWS CloudTrail User Guide*.

CloudWatch metrics for function URLs

Lambda sends aggregated metrics about function URL requests to CloudWatch. With these metrics, you can monitor your function URLs, build dashboards, and configure alarms in the CloudWatch console.

Function URLs support the following invocation metrics. We recommend viewing these metrics with the `Sum` statistic.

- `UrlRequestCount` – The number of requests made to this function URL.

- `Url4xxCount` – The number of requests that returned a 4XX HTTP status code. 4XX series codes indicate client-side errors, such as bad requests.
- `Url5xxCount` – The number of requests that returned a 5XX HTTP status code. 5XX series codes indicate server-side errors, such as function errors and timeouts.

Function URLs also support the following performance metric. We recommend viewing this metric with the `Average` or `Max` statistics.

- `UrlRequestLatency` – The time between when the function URL receives a request and when the function URL returns a response.

Each of these invocation and performance metrics supports the following dimensions:

- `FunctionName` – View aggregate metrics for function URLs assigned to a function's `$LATEST` unpublished version, or to any of the function's aliases. For example, `hello-world-function`.
- `Resource` – View metrics for a specific function URL. This is defined by a function name, along with either the function's `$LATEST` unpublished version or one of the function's aliases. For example, `hello-world-function:$LATEST`.
- `ExecutedVersion` – View metrics for a specific function URL based on the executed version. You can use this dimension primarily to track the function URL assigned to the `$LATEST` unpublished version.

Select a method to invoke your Lambda function using an HTTP request

Many common use cases for Lambda involve invoking your function using an HTTP request. For example, you might want a web application to invoke your function through a browser request. Lambda functions can also be used to create full REST APIs, handle user interactions from mobile apps, process data from external services via HTTP calls, or create custom webhooks.

The following sections explain what your choices are for invoking Lambda through HTTP and provide information to help you make the right decision for your particular use case.

What are your choices when selecting an HTTP invoke method?

Lambda offers two main methods to invoke a function using an HTTP request - function URLs and API Gateway. The key differences between these two options are as follows:

- **Lambda function URLs** provide a simple, direct HTTP endpoint for a Lambda function. They are optimized for simplicity and cost-effectiveness and provide the fastest path to expose a Lambda function via HTTP.
- **API Gateway** is a more advanced service for building fully-featured APIs. API Gateway is optimized for building and managing productions APIs at scale and provides comprehensive tools for security, monitoring, and traffic management.

Recommendations if you already know your requirements

If you're already clear on your requirements, here are our basic recommendations:

We recommend **function URLs** for simple applications or prototyping where you only need basic authentication methods and request/response handling and where you want to keep costs and complexity to a minimum.

API Gateway is a better choice for production applications at scale or for cases where you need more advanced features like OpenAPI Description support, a choice of authentication options, custom domain names, or rich request/response handling including throttling, caching, and request/response transformation.

What to consider when selecting a method to invoke your Lambda function

When selecting between function URLs and API Gateway, you need to consider the following factors:

- Your authentication needs, such as whether you require OAuth or Amazon Cognito to authenticate users
- Your scaling requirements and the complexity of the API you want to implement
- Whether you need advanced features such as request validation and request/response formatting
- Your monitoring requirements
- Your cost goals

By understanding these factors, you can select the option that best balances your security, complexity, and cost requirements.

The following information summarizes the main differences between the two options.

Authentication

- **Function URLs** provide basic authentication options through AWS Identity and Access Management (IAM). You can configure your endpoints to be either public (no authentication) or to require IAM authentication. With IAM authentication, you can use standard AWS credentials or IAM roles to control access. While straightforward to set up, this approach provides limited options compared with other authenticaton methods.
- **API Gateway** provides access to a more comprehensive range of authentication options. As well as IAM authentication, you can use Lambda authorizers (custom authentication logic), Amazon Cognito user pools, and OAuth2.0 flows. This flexibility allows you to implement complex authentication schemes, including third-party authentication providers, token-based authentication, and multi-factor authentication.

Request/response handling

- **Function URLs** provide basic HTTP request and response handling. They support standard HTTP methods and include built-in cross-origin resource sharing (CORS) support. While they can handle JSON payloads and query parameters naturally, they don't offer request transformation or validation capabilities. Response handling is similarly straightforward – the client receives the response from your Lambda function exactly as Lambda returns it.
- **API Gateway** provides sophisticated request and response handling capabilities. You can define request validators, transform requests and responses using mapping templates, set up request/response headers, and implement response caching. API Gateway also supports binary payloads and custom domain names and can modify responses before they reach the client. You can set up models for request/response validation and transformation using JSON Schema.

Scaling

- **Function URLs** scale directly with your Lambda function's concurrency limits and handle traffic spikes by scaling your function up to its maximum configured concurrency limit. Once that limit is reached, Lambda responds to additional requests with HTTP 429 responses. There's no built-in queuing mechanism, so handling scaling is entirely dependent on your Lambda function's

configuration. By default, Lambda functions have a limit of 1,000 concurrent executions per AWS Region.

- **API Gateway** provides additional scaling capabilities on top of Lambda's own scaling. It includes built-in request queuing and throttling controls, allowing you to manage traffic spikes more gracefully. API Gateway can handle up to 10,000 requests per second per region by default, with a burst capacity of 5,000 requests per second. It also provides tools to throttle requests at different levels (API, stage, or method) to protect your backend.

Monitoring

- **Function URLs** offer basic monitoring through Amazon CloudWatch metrics, including request count, latency, and error rates. You get access to standard Lambda metrics and logs, which show the raw requests coming into your function. While this provides essential operational visibility, the metrics are focused mainly on function execution.
- **API Gateway** provides comprehensive monitoring capabilities including detailed metrics, logging, and tracing options. You can monitor API calls, latency, error rates, and cache hit/miss rates through CloudWatch. API Gateway also integrates with AWS X-Ray for distributed tracing and provides customizable logging formats.

Cost

- **Function URLs** follow the standard Lambda pricing model – you only pay for function invocations and compute time. There are no additional charges for the URL endpoint itself. This makes it a cost-effective choice for simple APIs or low-traffic applications if you don't need the additional features of API Gateway.
- **API Gateway** offers a free tier that includes one million API calls received for REST APIs and one million API calls received for HTTP APIs. After this, API Gateway charges for API calls, data transfer, and caching (if enabled). Refer to the API Gateway pricing page to understand the costs for your own use case.

Other features

- **Function URLs** are designed for simplicity and direct Lambda integration. They support both HTTP and HTTPS endpoints, offer built-in CORS support, and provide dual-stack (IPv4 and IPv6) endpoints. While they lack advanced features, they excel in scenarios where you need a quick, straightforward way to expose Lambda functions via HTTP.

- **API Gateway** includes numerous additional features such as API versioning, stage management, API keys for usage plans, API documentation through Swagger/OpenAPI, WebSocket APIs, private APIs within a VPC, and WAF integration for additional security. It also supports canary deployments, mock integrations for testing, and integration with other AWS services beyond Lambda.

Select a method to invoke your Lambda function

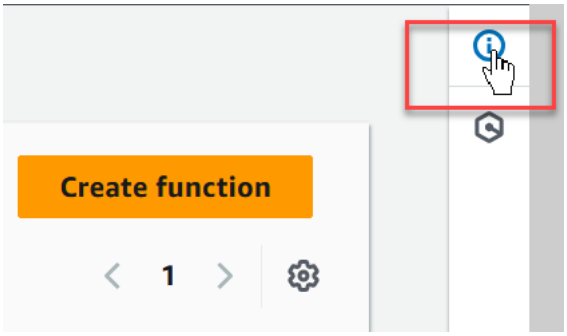
Now that you've read about the criteria for selecting between Lambda function URLs and API Gateway and the key differences between them, you can select the option that best meets your needs and use the following resources to help you get started using it.

Function URLs

Get started with function URLs with the following resources

- Follow the tutorial Creating a Lambda function with a function URL
- Learn more about function URLs in the the section called “Function URLs” chapter of this guide
- Try the in-console guided tutorial **Create a simple web app** by doing the following:

1. Open the functions page of the Lambda console.
2. Open the help panel by choosing the icon in the top right corner of the screen.



3. Select **Tutorials**.
4. In **Create a simple web app**, choose **Start tutorial**.

API Gateway

Get started with Lambda and API Gateway with the following resources

- Follow the tutorial Using Lambda with API Gateway to create a REST API integrated with a backend Lambda function.
- Learn more about the different kinds of API offered by API Gateway in the following sections of the *Amazon API Gateway Developer Guide*:
  - API Gateway REST APIs
  - API Gateway HTTP APIs
  - API Gateway WebSocket APIs
- Try one or more of the examples in the Tutorials and workshops section of the *Amazon API Gateway Developer Guide*.

Tutorial: Creating a webhook endpoint using a Lambda function URL

In this tutorial, you create a Lambda function URL to implement a webhook endpoint. A webhook is a lightweight, event-driven communication that automatically sends data between applications using HTTP. You can use a webhook to receive immediate updates about events happening in another system, such as when a new customer signs up on a website, a payment is processed, or a file is uploaded.

With Lambda, webhooks can be implemented using either Lambda function URLs or API Gateway. Function URLs are a good choice for simple webhooks that don't require features like advanced authorization or request validation.

> **Tip**
>
> If you're not sure which solution is best for your particular use case, see the section called "Function URLs vs Amazon API Gateway".

Prerequisites

To complete this tutorial, you must have either Python (version 3.8 or later) or Node.js (version 18 or later) installed on your local machine.

To test the endpoint using an HTTP request, the tutorial uses curl, a command line tool you can use to transfer data using various network protocols. Refer to the curl documentation to learn how to install the tool if you don't already have it.

Create the Lambda function

First create the Lambda function that runs when an HTTP request is sent to your webhook endpoint. In this example, the sending application sends an update whenever a payment is submitted and indicates in the body of the HTTP request whether the payment was successful. The Lambda function parses the request and takes action according to the status of the payment. In this example, the code just prints the order ID for the payment, but in a real application, you might add the order to a database or send a notification.

The function also implements the most common authentication method used for webhooks, hash-based message authentication (HMAC). With this method, both the sending and receiving applications share a secret key. The sending application uses a hashing algorithm to generate a unique signature using this key together with the message content, and includes the signature

in the webhook request as an HTTP header. The receiving application then repeats this step, generating the signature using the secret key, and compares the resulting value with the signature sent in the request header. If the result matches, the request is considered legitimate.

Create the function using the Lambda console with either the Python or Node.js runtime.

Python

Create the Lambda function

1. Open the Functions page of the Lambda console.
2. Create a basic 'Hello world' function by doing the following:
   a. Choose **Create function**.
   b. Select **Author from scratch**.
   c. For **Function name**, enter **`myLambdaWebhook`**.
   d. For **Runtime**, select **python3.14**.
   e. Choose **Create function**.
3. In the **Code source** pane, replace the existing code by copying and pasting the following:

```
import json
import hmac
import hashlib
import os

def lambda_handler(event, context):

    # Get the webhook secret from environment variables
    webhook_secret = os.environ['WEBHOOK_SECRET']

    # Verify the webhook signature
    if not verify_signature(event, webhook_secret):
        return {
            'statusCode': 401,
            'body': json.dumps({'error': 'Invalid signature'})
        }

    try:
        # Parse the webhook payload
        payload = json.loads(event['body'])
```

```
        # Handle different event types
        event_type = payload.get('type')

        if event_type == 'payment.success':
            # Handle successful payment
            order_id = payload.get('orderId')
            print(f"Processing successful payment for order {order_id}")

            # Add your business logic here
            # For example, update database, send notifications, etc.

        elif event_type == 'payment.failed':
            # Handle failed payment
            order_id = payload.get('orderId')
            print(f"Processing failed payment for order {order_id}")

            # Add your business logic here

        else:
            print(f"Received unhandled event type: {event_type}")

        # Return success response
        return {
            'statusCode': 200,
            'body': json.dumps({'received': True})
        }

    except json.JSONDecodeError:
        return {
            'statusCode': 400,
            'body': json.dumps({'error': 'Invalid JSON payload'})
        }
    except Exception as e:
        print(f"Error processing webhook: {e}")
        return {
            'statusCode': 500,
            'body': json.dumps({'error': 'Internal server error'})
        }

def verify_signature(event, webhook_secret):
    """
    Verify the webhook signature using HMAC
    """
    try:
```

```
        # Get the signature from headers
        signature = event['headers'].get('x-webhook-signature')

        if not signature:
            print("Error: Missing webhook signature in headers")
            return False

        # Get the raw body (return an empty string if the body key doesn't
exist)
        body = event.get('body', '')

        # Create HMAC using the secret key
        expected_signature = hmac.new(
            webhook_secret.encode('utf-8'),
            body.encode('utf-8'),
            hashlib.sha256
        ).hexdigest()

        # Compare the expected signature with the received signature to
authenticate the message
        is_valid = hmac.compare_digest(signature, expected_signature)
        if not is_valid:
            print(f"Error: Invalid signature. Received: {signature}, Expected:
{expected_signature}")
            return False

        return True
    except Exception as e:
        print(f"Error verifying signature: {e}")
        return False
```

4. In the **DEPLOY** section, choose **Deploy** to update your function's code.

Node.js

Create the Lambda function

1. Open the Functions page of the Lambda console.
2. Create a basic 'Hello world' function by doing the following:
   a. Choose **Create function**.
   b. Select **Author from scratch**.

c. For **Function name**, enter **myLambdaWebhook**.

d. For **Runtime**, select **nodejs24.x**.

e. Choose **Create function**.

3. In the **Code source** pane, replace the existing code by copying and pasting the following:

```
import crypto from 'crypto';

export const handler = async (event, context) => {
    // Get the webhook secret from environment variables
    const webhookSecret = process.env.WEBHOOK_SECRET;

    // Verify the webhook signature
    if (!verifySignature(event, webhookSecret)) {
        return {
            statusCode: 401,
            body: JSON.stringify({ error: 'Invalid signature' })
        };
    }

    try {
        // Parse the webhook payload
        const payload = JSON.parse(event.body);

        // Handle different event types
        const eventType = payload.type;

        switch (eventType) {
            case 'payment.success': {
                // Handle successful payment
                const orderId = payload.orderId;
                console.log(`Processing successful payment for order
 ${orderId}`);

                // Add your business logic here
                // For example, update database, send notifications, etc.
                break;
            }

            case 'payment.failed': {
                // Handle failed payment
                const orderId = payload.orderId;
                console.log(`Processing failed payment for order ${orderId}`);
```

```
                // Add your business logic here
                break;
            }

            default:
                console.log(`Received unhandled event type: ${eventType}`);
        }

        // Return success response
        return {
            statusCode: 200,
            body: JSON.stringify({ received: true })
        };

    } catch (error) {
        if (error instanceof SyntaxError) {
            // Handle JSON parsing errors
            return {
                statusCode: 400,
                body: JSON.stringify({ error: 'Invalid JSON payload' })
            };
        }

        // Handle all other errors
        console.error('Error processing webhook:', error);
        return {
            statusCode: 500,
            body: JSON.stringify({ error: 'Internal server error' })
        };
    }
};

// Verify the webhook signature using HMAC

const verifySignature = (event, webhookSecret) => {
    try {
        // Get the signature from headers
        const signature = event.headers['x-webhook-signature'];

        if (!signature) {
            console.log('No signature found in headers:', event.headers);
            return false;
        }
```

```
        // Get the raw body (return an empty string if the body key doesn't
exist)
        const body = event.body || '';

        // Create HMAC using the secret key
        const hmac = crypto.createHmac('sha256', webhookSecret);
        const expectedSignature = hmac.update(body).digest('hex');

        // Compare expected and received signatures
        const isValid = signature === expectedSignature;
        if (!isValid) {
            console.log(`Invalid signature. Received: ${signature}, Expected:
${expectedSignature}`);
            return false;
        }

        return true;
    } catch (error) {
        console.error('Error during signature verification:', error);
        return false;
    }
 };
```

4. In the **DEPLOY** section, choose **Deploy** to update your function's code.

Create the secret key

For the Lambda function to authenticate the webhook request, it uses a secret key which it shares with the calling application. In this example, the key is stored in an environment variable. In a production application, don't include sensitive information like passwords in your function code. Instead, create an AWS Secrets Manager secret and then use the AWS Parameters and Secrets Lambda extension to retrieve your credentials in your Lambda function.

Create and store the webhook secret key

1. Generate a long, random string using a cryptographically secure random number generator. You can use the following code snippets in Python or Node.js to generate and print a 32-character secret, or use your own preferred method.

Python

Example code to generate a secret

```
import secrets
webhook_secret = secrets.token_urlsafe(32)
print(webhook_secret)
```

Node.js

Example code to generate a secret (ES module format)

```
import crypto from 'crypto';
let webhookSecret = crypto.randomBytes(32).toString('base64');
console.log(webhookSecret)
```

2. Store your generated string as an environment variable for your function by doing the following:

   a. In the **Configuration** tab for your function, select **Environment variables**.

   b. Choose **Edit**.

   c. Choose **Add environment variable**.

   d. For **Key**, enter **`WEBHOOK_SECRET`**, then for **Value**, enter the secret you generated in the previous step.

   e. Choose **Save**.

You'll need to use this secret again later in the tutorial to test your function, so make a note of it now.

Create the function URL endpoint

Create an endpoint for your webhook using a Lambda function URL. Beacuse you use the auth type of `NONE` to create an endpoint with public access, anyone with the URL can invoke your function. To learn more about controlling access to function URLs, see the section called “Access control”. If you need more advanced authentication options for your webhook, consider using API Gateway.

Create the function URL endpoint

1. In the **Configuration** tab for your function, select **Function URL**.
2. Choose **Create function URL**.
3. For **Auth type**, select **NONE**.
4. Choose **Save**.

The endpoint for the function URL you just created is displayed in the **Function URL** pane. Copy the endpoint to use later in the tutorial.

Test the function in the console

Before using an HTTP request to invoke your function using the URL endpoint, test it in the console to confirm your code is working as expected.

To verify the function in the console, you first calculate a webhook signature using the secret you generated earlier in the tutorial with the following test JSON payload:

```
{
    "type": "payment.success",
    "orderId": "1234",
    "amount": "99.99"
}
```

Use either of the following Python or Node.js code examples to calculate the webhook signature using your own secret.

Python

Calculate the webhook signature

1. Save the following code as a file named `calculate_signature.py`. Replace the webhook secret in the code with your own value.

```
import secrets
import hmac
import json
import hashlib
```

```
webhook_secret = "arlbSDCP86n_1H90s0fL_Qb2NAHBIBQOyGI0X4Zay4M"

body = json.dumps({"type": "payment.success", "orderId": "1234", "amount":
 "99.99"})

signature = hmac.new(
            webhook_secret.encode('utf-8'),
            body.encode('utf-8'),
            hashlib.sha256
        ).hexdigest()

print(signature)
```

2. Calculate the signature by running the following command from the same directory you saved the code in. Copy the signature the code outputs.

```
python calculate_signature.py
```

Node.js

Calculate the webhook signature

1. Save the following code as a file named `calculate_signature.mjs`. Replace the webhook secret in the code with your own value.

```
import crypto from 'crypto';

const webhookSecret = "arlbSDCP86n_1H90s0fL_Qb2NAHBIBQOyGI0X4Zay4M"
const body = "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}";

let hmac = crypto.createHmac('sha256', webhookSecret);
let signature = hmac.update(body).digest('hex');

console.log(signature);
```

2. Calculate the signature by running the following command from the same directory you saved the code in. Copy the signature the code outputs.

```
node calculate_signature.mjs
```

You can now test your function code using a test HTTP request in the console.

Test the function in the console

1. Select the **Code** tab for your function.
2. In the **TEST EVENTS** section, choose **Create new test event**
3. For **Event Name**, enter **`myEvent`**.
4. Replace the existing JSON by copying and pasting the following into the **Event JSON** pane. Replace the webhook signature with the value you calculated in the previous step.

```
{
  "headers": {
    "Content-Type": "application/json",
    "x-webhook-signature":
 "2d672e7a0423fab740fbc040e801d1241f2df32d2ffd8989617a599486553e2a"
  },
  "body": "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}"
}
```

5. Choose **Save**.
6. Choose **Invoke**.

   You should see output similar to the following:

   Python

```
Status: Succeeded
Test Event Name: myEvent

Response:
{
  "statusCode": 200,
  "body": "{\"received\": true}"
}

Function Logs:
START RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6 Version: $LATEST
Processing successful payment for order 1234
END RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6
```

```
REPORT RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6 Duration: 1.55 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 36 MB Init Duration: 136.32
 ms
```

Node.js

```
Status: Succeeded
Test Event Name: myEvent

Response:
{
  "statusCode": 200,
  "body": "{\"received\":true}"
}

Function Logs:
START RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 Version: $LATEST
2025-01-10T18:05:42.062Z e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 INFO Processing
 successful payment for order 1234
END RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4
REPORT RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 Duration: 60.10 ms Billed
 Duration: 61 ms Memory Size: 128 MB Max Memory Used: 72 MB Init Duration:
 174.46 ms

Request ID: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4
```

Test the function using an HTTP request

Use the curl command line tool to test your webhook endpoint.

Test the function using HTTP requests

1. In a terminal or shell program, run the following curl command. Replace the URL with the value for your own function URL endpoint and replace the webhook signature with the signature you calculated using your own secret key.

```
curl -X POST https://ryqgmbx5xjzxahif6frvzikpre0bpvpf.lambda-url.us-west-2.on.aws/
 \
-H "Content-Type: application/json" \
-H "x-webhook-
signature: d5f52b76ffba65ff60ea73da67bdf1fc5825d4db56b5d3ffa0b64b7cb85ef48b" \
```

```
-d '{"type": "payment.success", "orderId": "1234", "amount": "99.99"}'
```

You should see the following output:

```
{"received": true}
```

2. Inspect the CloudWatch logs for your function to confirm it parsed the payload correctly by doing the following:

   a. Open the Logs group page in the Amazon CloudWatch console.

   b. Select your function's log group (`/aws/lambda/myLambdaWebhook`).

   c. Select the most recent log stream.

      You should see output similar to the following in your function's logs:

      Python

      ```
      Processing successful payment for order 1234
      ```

      Node.js

      ```
      2025-01-10T18:05:42.062Z e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 INFO
       Processing successful payment for order 1234
      ```

3. Confirm that your code detects an invalid signature by running the following curl command. Replace the URL with your own function URL endpoint.

   ```
   curl -X POST https://ryqgmbx5xjzxahif6frvzikpre0bpvpf.lambda-url.us-west-2.on.aws/
    \
   -H "Content-Type: application/json" \
   -H "x-webhook-signature: abcdefg" \
   -d '{"type": "payment.success", "orderId": "1234", "amount": "99.99"}'
   ```

   You should see the following output:

   ```
   {"error": "Invalid signature"}
   ```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

When you created the Lambda function in the console, Lambda also created an execution role for your function.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that Lambda created. The role has the name format `myLambdaWebhook-role-<random string>`.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

Understanding Lambda function scaling

Concurrency is the number of in-flight requests that your AWS Lambda function is handling at the same time. For each concurrent request, Lambda provisions a separate instance of your execution environment. As your functions receive more requests, Lambda automatically handles scaling the number of execution environments until you reach your account's concurrency limit. By default, Lambda provides your account with a total concurrency limit of 1,000 concurrent executions across all functions in an AWS Region. To support your specific account needs, you can request a quota increase and configure function-level concurrency controls so that your critical functions don't experience throttling.

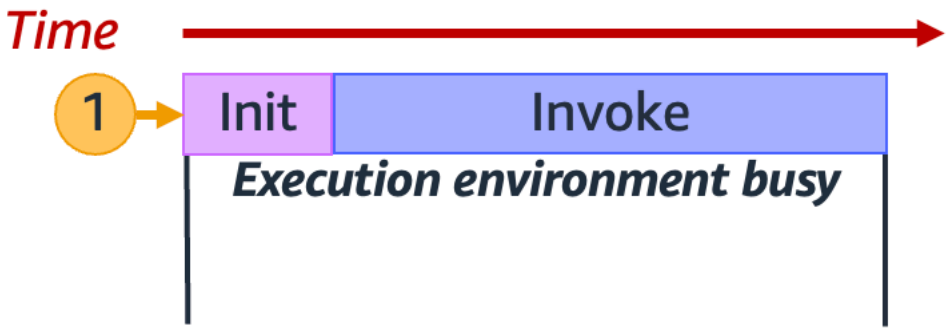
This topic explains concurrency concepts and function scaling in Lambda. By the end of this topic, you'll be able to understand how to calculate concurrency, visualize the two main concurrency control options (reserved and provisioned), estimate appropriate concurrency control settings, and view metrics for further optimization.

Sections

- Understanding and visualizing concurrency
- Calculating concurrency for a function
- Understanding reserved concurrency and provisioned concurrency
- Understanding concurrency and requests per second
- Concurrency quotas
- Configuring reserved concurrency for a function
- Configuring provisioned concurrency for a function
- Lambda scaling behavior
- Monitoring concurrency

Understanding and visualizing concurrency

Lambda invokes your function in a secure and isolated execution environment. To handle a request, Lambda must first initialize an execution environment (the Init phase), before using it to invoke your function (the Invoke phase):

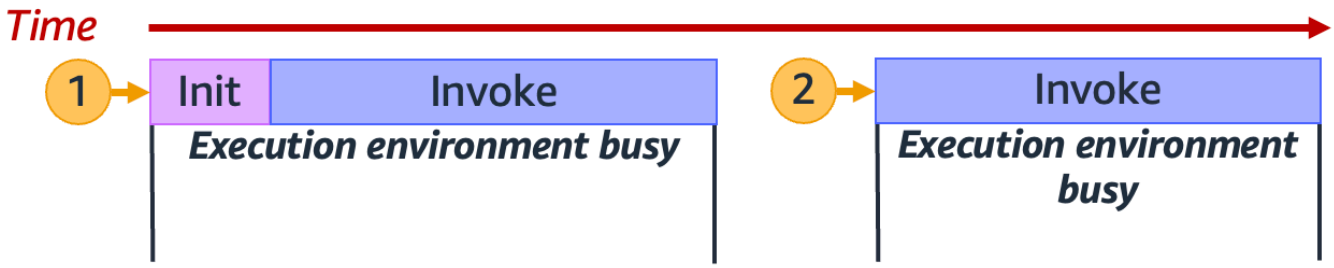


Note

Actual Init and Invoke durations can vary depending on many factors, such as the runtime you choose and the Lambda function code. The previous diagram isn't meant to represent the exact proportions of Init and Invoke phase durations.

The previous diagram uses a rectangle to represent a single execution environment. When your function receives its very first request (represented by the yellow circle with label 1), Lambda creates a new execution environment and runs the code outside your main handler during the Init phase. Then, Lambda runs your function's main handler code during the Invoke phase. During this entire process, this execution environment is busy and cannot process other requests.

When Lambda finishes processing the first request, this execution environment can then process additional requests for the same function. For subsequent requests, Lambda doesn't need to re-initialize the environment.

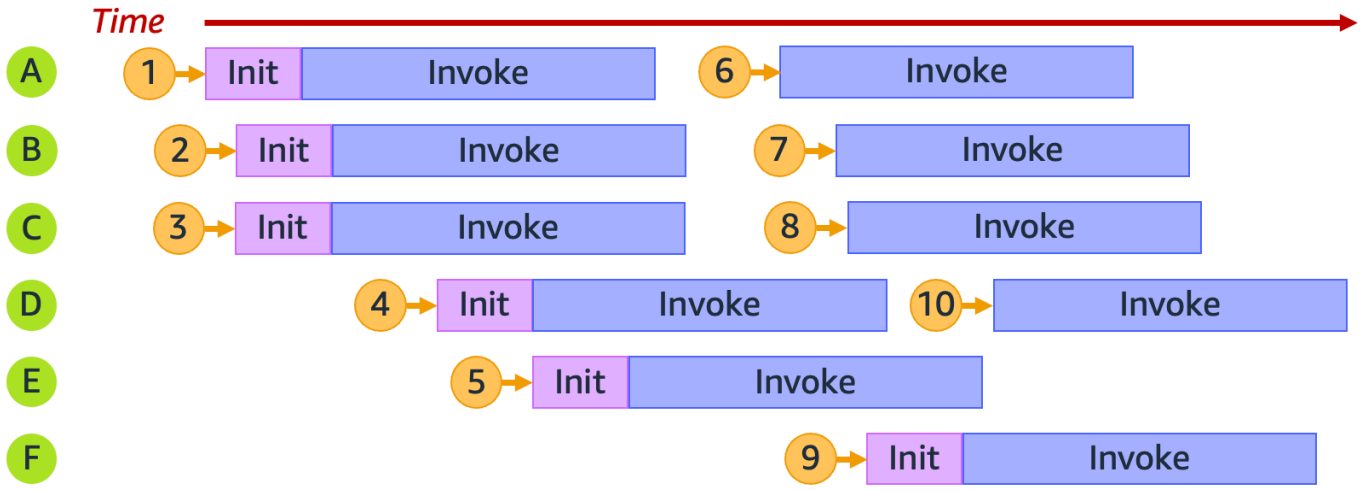


In the previous diagram, Lambda reuses the execution environment to handle the second request (represented by the yellow circle with label 2).

So far, we've focused on just a single instance of your execution environment (that is, a concurrency of 1). In practice, Lambda may need to provision multiple execution environment instances in parallel to handle all incoming requests. When your function receives a new request, one of two things can happen:

- If a pre-initialized execution environment instance is available, Lambda uses it to process the request.
- Otherwise, Lambda creates a new execution environment instance to process the request.

For example, let's explore what happens when your function receives 10 requests:

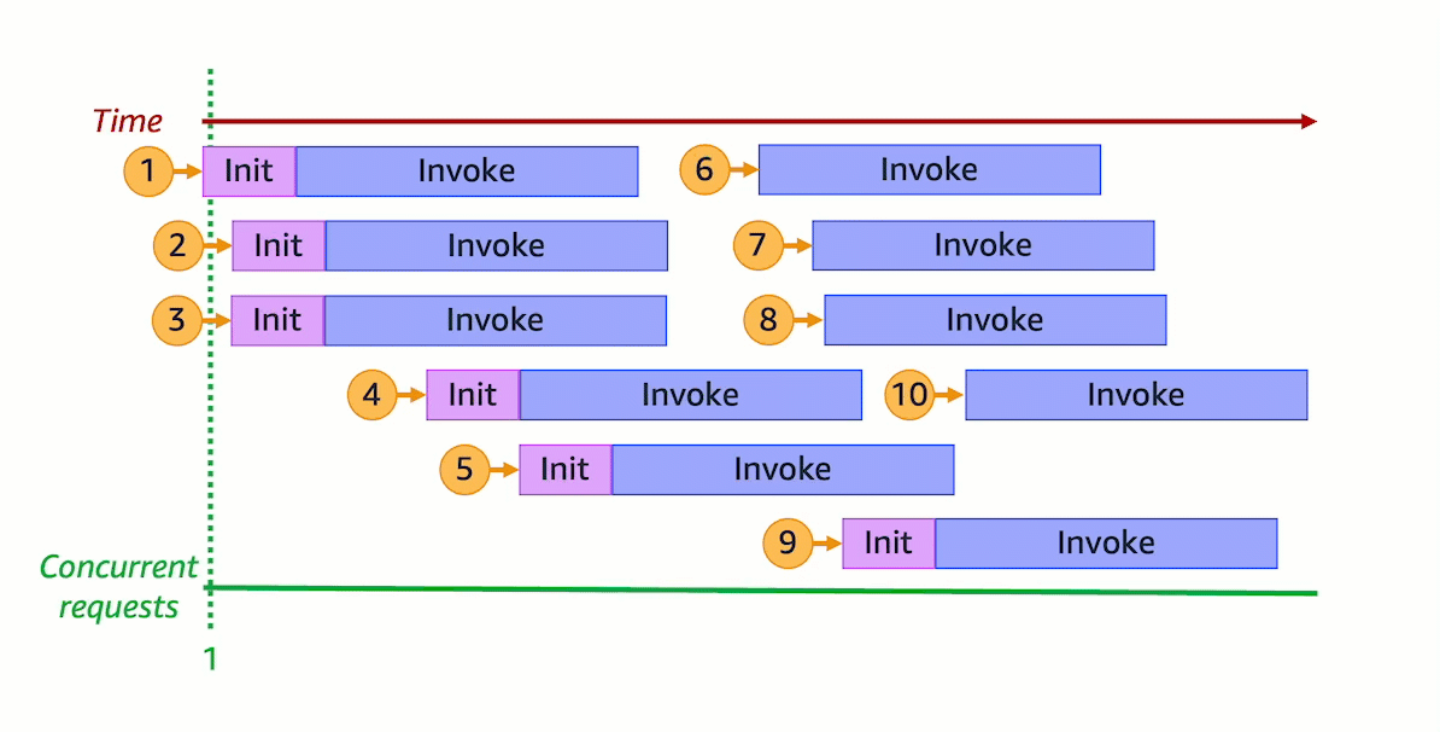


In the previous diagram, each horizontal plane represents a single execution environment instance (labeled from A through F). Here's how Lambda handles each request:

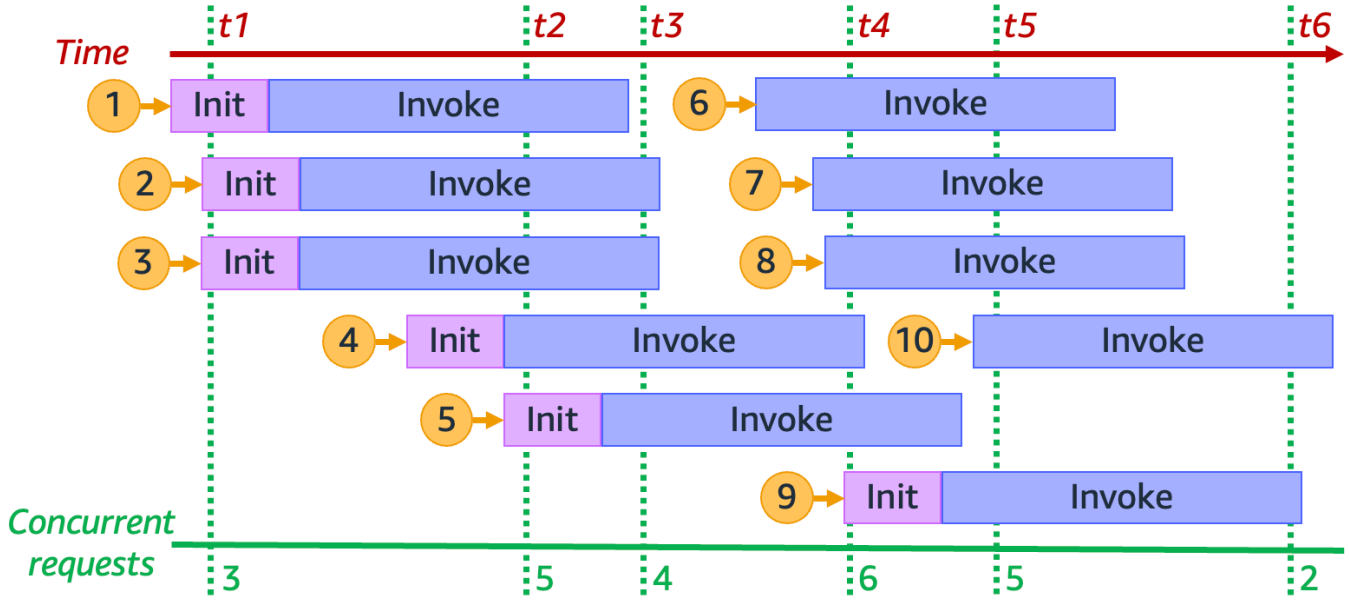
Request	Lambda behavior	Reasoning
1	Provisions new environment **A**	This is the first request; no execution environment instances are available.
2	Provisions new environment **B**	Existing execution environment instance **A** is busy.

Request	Lambda behavior	Reasoning
3	Provisions new environment **C**	Existing execution environment instances **A** and **B** are both busy.
4	Provisions new environment **D**	Existing execution environment instances **A**, **B**, and **C** are all busy.
5	Provisions new environment **E**	Existing execution environment instances **A**, **B**, **C**, and **D** are all busy.
6	Reuses environment **A**	Execution environment instance **A** has finished processing request **1** and is now available.
7	Reuses environment **B**	Execution environment instance **B** has finished processing request **2** and is now available.
8	Reuses environment **C**	Execution environment instance **C** has finished processing request **3** and is now available.
9	Provisions new environment **F**	Existing execution environment instances **A**, **B**, **C**, **D**, and **E** are all busy.
10	Reuses environment **D**	Execution environment instance **D** has finished processing request **4** and is now available.

As your function receives more concurrent requests, Lambda scales up the number of execution environment instances in response. The following animation tracks the number of concurrent requests over time:



By freezing the previous animation at six distinct points in time, we get the following diagram:



In the previous diagram, we can draw a vertical line at any point in time and count the number of environments that intersect this line. This gives us the number of concurrent requests at that point in time. For example, at time `t1`, there are three active environments serving three concurrent requests. The maximum number of concurrent requests in this simulation occurs at time `t4`, when there are six active environments serving six concurrent requests.

To summarize, your function's concurrency is the number of concurrent requests that it's handling at the same time. In response to an increase in your function's concurrency, Lambda provisions more execution environment instances to meet request demand.

Calculating concurrency for a function

In general, concurrency of a system is the ability to process more than one task simultaneously. In Lambda, concurrency is the number of in-flight requests that your function is handling at the same time. A quick and practical way of measuring concurrency of a Lambda function is to use the following formula:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Concurrency differs from requests per second. For example, suppose your function receives 100 requests per second on average. If the average request duration is one second, then it's true that the concurrency is also 100:

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

However, if the average request duration is 500 ms, then the concurrency is 50:

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

What does a concurrency of 50 mean in practice? If the average request duration is 500 ms, then you can think of an instance of your function as being able to handle two requests per second. Then, it takes 50 instances of your function to handle a load of 100 requests per second. A concurrency of 50 means that Lambda must provision 50 execution environment instances to efficiently handle this workload without any throttling. Here's how to express this in equation form:

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

If your function receives double the number of requests (200 requests per second), but only requires half the time to process each request (250 ms), then the concurrency is still 50:

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

Test your understanding of concurrency

Suppose you have a function that takes, on average, 200 ms to run. During peak load, you observe 5,000 requests per second. What is the concurrency of your function during peak load?

Answer

The average function duration is 200 ms, or 0.2 seconds. Using the concurrency formula, you can plug in the numbers to get a concurrency of 1,000:

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

Alternatively, an average function duration of 200 ms means that your function can process 5 requests per second. To handle the 5,000 request per second workload, you need 1,000 execution environment instances. Thus, the concurrency is 1,000:

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

Understanding reserved concurrency and provisioned concurrency

By default, your account has a concurrency limit of 1,000 concurrent executions across all functions in a Region. Your functions share this pool of 1,000 concurrency on an on-demand basis. Your functions experience throttling (that is, they start to drop requests) if you run out of available concurrency.

Some of your functions might be more critical than others. As a result, you might want to configure concurrency settings to ensure that critical functions get the concurrency that they need. There are two types of concurrency controls available: reserved concurrency and provisioned concurrency.

- Use **reserved concurrency** to set both the maximum and minimum number of concurrent instances to reserve a portion of your account's concurrency for a function. This is useful if you

don't want other functions taking up all the available unreserved concurrency. When a function has reserved concurrency, no other function can use that concurrency.

- Use **provisioned concurrency** to pre-initialize a number of environment instances for a function. This is useful for reducing cold start latencies.

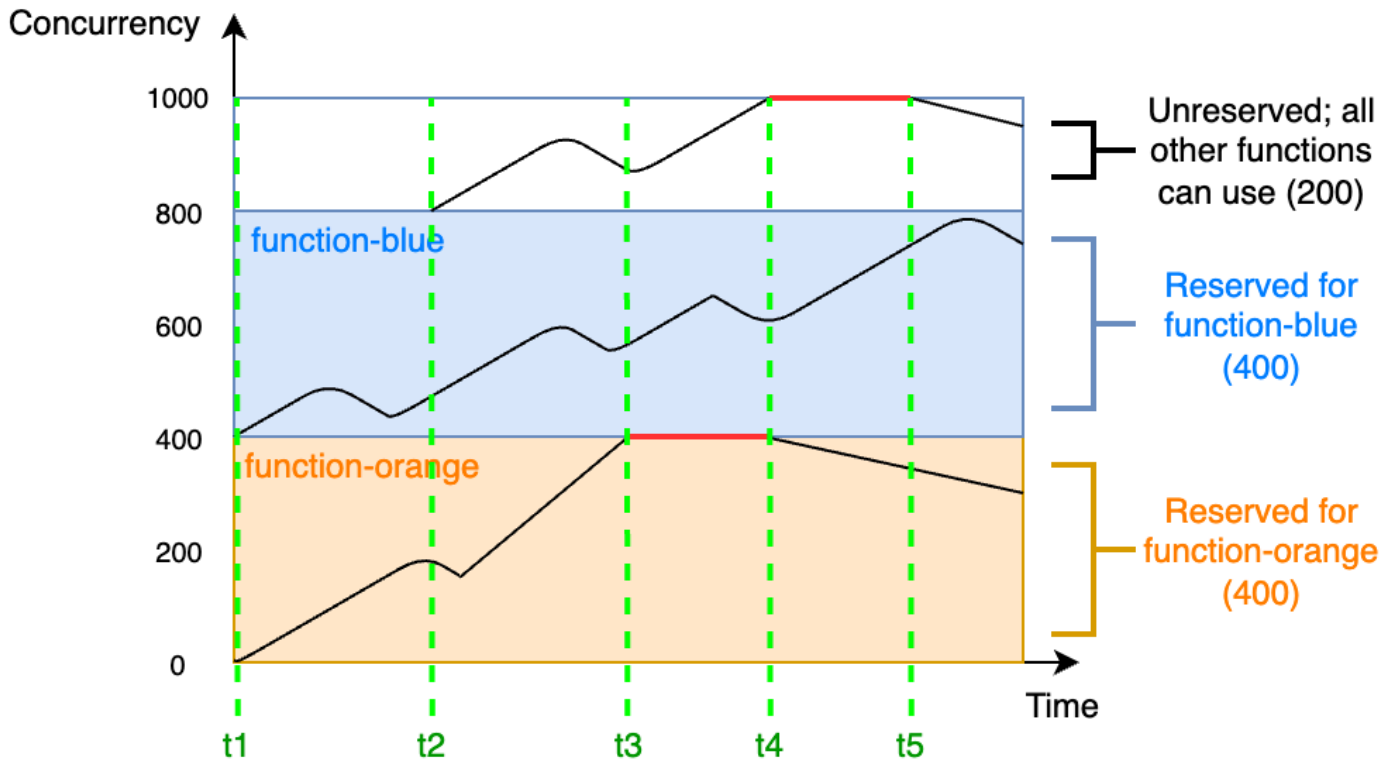
Reserved concurrency

If you want to guarantee that a certain amount of concurrency is available for your function at any time, use reserved concurrency.

Reserved concurrency sets the maximum and minimum number of concurrent instances that you want to allocate to your function. When you dedicate reserved concurrency to a function, no other function can use that concurrency. In other words, setting reserved concurrency can impact the concurrency pool that's available to other functions. Functions that don't have reserved concurrency share the remaining pool of unreserved concurrency.

Configuring reserved concurrency counts towards your overall account concurrency limit. There is no charge for configuring reserved concurrency for a function.

To better understand reserved concurrency, consider the following diagram:



In this diagram, your account concurrency limit for all the functions in this Region is at the default limit of 1,000. Suppose you have two critical functions, `function-blue` and `function-orange`, that routinely expect to get high invocation volumes. You decide to give 400 units of reserved concurrency to `function-blue`, and 400 units of reserved concurrency to `function-orange`. In this example, all other functions in your account must share the remaining 200 units of unreserved concurrency.

The diagram has five points of interest:

- At `t1`, both `function-orange` and `function-blue` begin receiving requests. Each function begins to use up its allocated portion of reserved concurrency units.
- At `t2`, `function-orange` and `function-blue` steadily receive more requests. At the same time, you deploy some other Lambda functions, which begin receiving requests. You don't allocate reserved concurrency to these other functions. They begin using the remaining 200 units of unreserved concurrency.
- At `t3`, `function-orange` hits the max concurrency of 400. Although there is unused concurrency elsewhere in your account, `function-orange` cannot access it. The red line indicates that `function-orange` is experiencing throttling, and Lambda may drop requests.
- At `t4`, `function-orange` starts to receive fewer requests and is no longer throttling. However, your other functions experience a spike in traffic and begin throttling. Although there is unused concurrency elsewhere in your account, these other functions cannot access it. The red line indicates that your other functions are experiencing throttling.
- At `t5`, other functions start to receive fewer requests and are no longer throttling.

From this example, notice that reserving concurrency has the following effects:

- **Your function can scale independently of other functions in your account.** All of your account's functions in the same Region that don't have reserved concurrency share the pool of unreserved concurrency. Without reserved concurrency, other functions can potentially use up all of your available concurrency. This prevents critical functions from scaling up if needed.
- **Your function can't scale out of control.** Reserved concurrency caps your function's maximum and minimum concurrency. This means that your function can't use concurrency reserved for other functions, or concurrency from the unreserved pool. Additionally, reserved concurrency acts as both a lower and upper bound - it reserves the specified capacity exclusively for your function while also preventing it from scaling beyond that limit. You can reserve concurrency

to prevent your function from using all the available concurrency in your account, or from overloading downstream resources.

- **You may not be able to use all of your account's available concurrency.** Reserving concurrency counts towards your account concurrency limit, but this also means that other functions cannot use that chunk of reserved concurrency. If your function doesn't use up all of the concurrency that you reserve for it, you're effectively wasting that concurrency. This isn't an issue unless other functions in your account could benefit from the wasted concurrency.

To learn how to manage reserved concurrency settings for your functions, see Configuring reserved concurrency for a function.

Provisioned concurrency

You use reserved concurrency to define the maximum number of execution environments reserved for a Lambda function. However, none of these environments come pre-initialized. As a result, your function invocations may take longer because Lambda must first initialize the new environment before being able to use it to invoke your function. When Lambda has to initialize a new environment in order to carry out an invocation, this is known as a cold start. To mitigate cold starts, you can use provisioned concurrency.

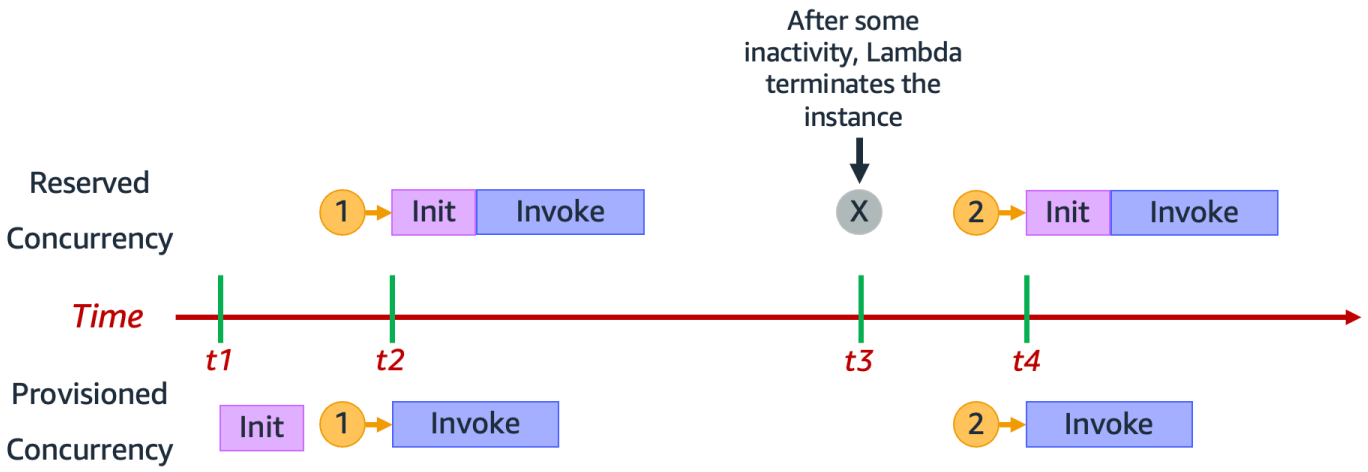
Provisioned concurrency is the number of pre-initialized execution environments that you want to allocate to your function. If you set provisioned concurrency on a function, Lambda initializes that number of execution environments so that they are prepared to respond immediately to function requests.

> **Note**
>
> Using provisioned concurrency incurs additional charges to your account. If you're working with the Java 11 or Java 17 runtimes, you can also use Lambda SnapStart to mitigate cold start issues at no additional cost. SnapStart uses cached snapshots of your execution environment to significantly improve startup performance. You cannot use both SnapStart and provisioned concurrency on the same function version. For more information about SnapStart features, limitations, and supported Regions, see Improving startup performance with Lambda SnapStart.

When using provisioned concurrency, Lambda still recycles execution environments in the background. For example, this can occur after an invocation failure. However, at any given time,

Lambda always ensures that the number of pre-initialized environments is equal to the value of your function's provisioned concurrency setting. Importantly, even if you're using provisioned concurrency, you can still experience a cold start delay if Lambda has to reset the execution environment.

In contrast, when using reserved concurrency, Lambda may completely terminate an environment after a period of inactivity. The following diagram illustrates this by comparing the lifecycle of a single execution environment when you configure your function using reserved concurrency compared to provisioned concurrency.

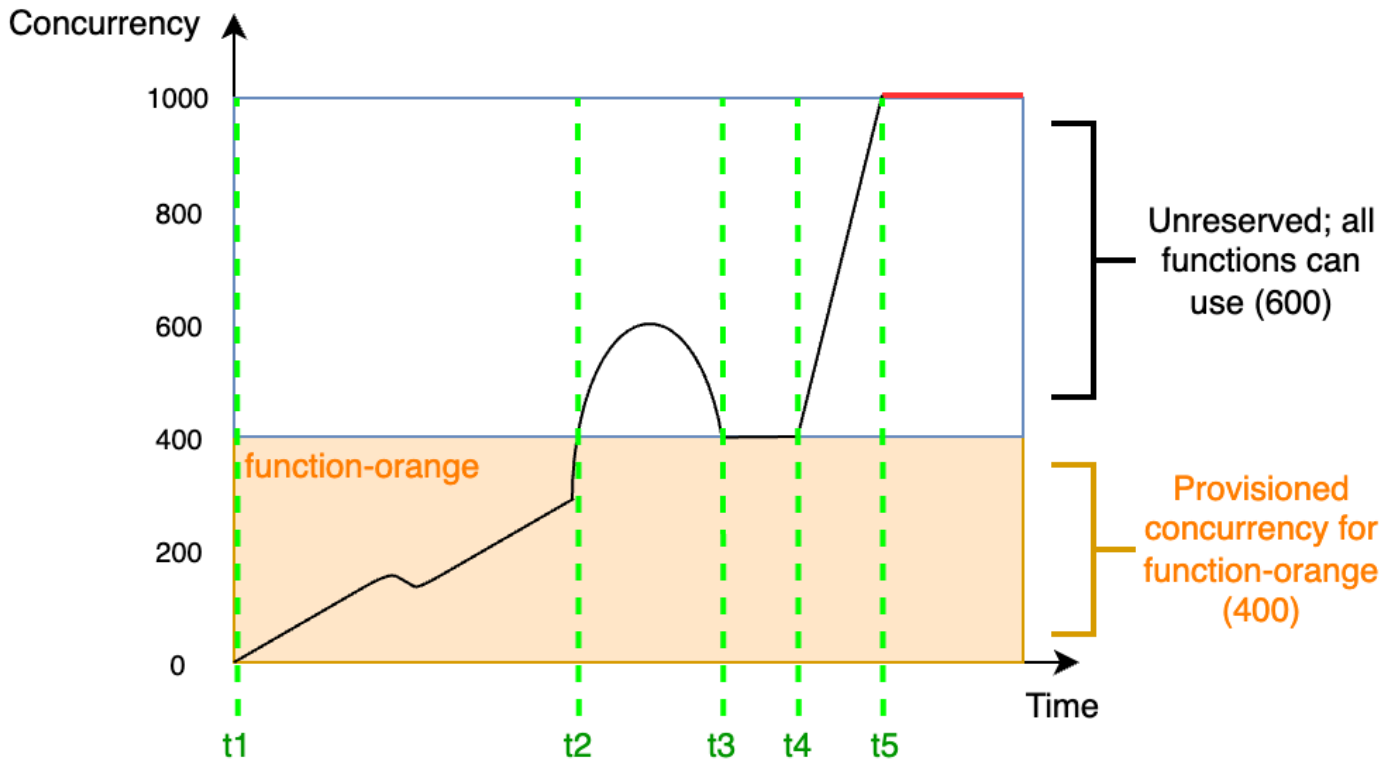


The diagram has four points of interest:

Time	Reserved concurrency	Provisioned concurrency
t1	Nothing happens.	Lambda pre-initializes an execution environment instance.
t2	Request 1 comes in. Lambda must initialize a new execution environment instance.	Request 1 comes in. Lambda uses the pre-initialized environment instance.
t3	After some inactivity, Lambda terminates the active environment instance.	Nothing happens.

Time	Reserved concurrency	Provisioned concurrency
t4	Request 2 comes in. Lambda must initialize a new execution environment instance.	Request 2 comes in. Lambda uses the pre-initialized environment instance.

To better understand provisioned concurrency, consider the following diagram:



In this diagram, you have an account concurrency limit of 1,000. You decide to give 400 units of provisioned concurrency to `function-orange`. All functions in your account, *including* `function-orange`, can use the remaining 600 units of unreserved concurrency.

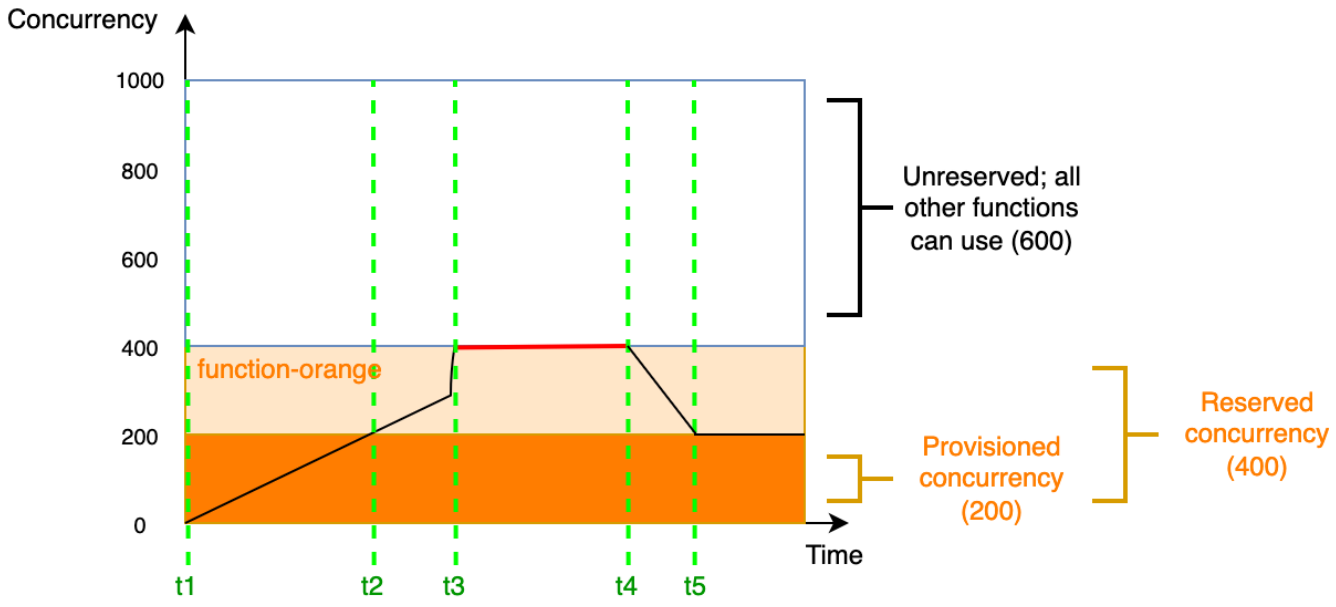
The diagram has five points of interest:

- At `t1`, `function-orange` begins receiving requests. Since Lambda has pre-initialized 400 execution environment instances, `function-orange` is ready for immediate invocation.
- At `t2`, `function-orange` reaches 400 concurrent requests. As a result, `function-orange` runs out of provisioned concurrency. However, since there's still unreserved concurrency available,

Lambda can use this to handle additional requests to `function-orange` (there's no throttling). Lambda must create new instances to serve these requests, and your function may experience cold start latencies.

- At `t3`, `function-orange` returns to 400 concurrent requests after a brief spike in traffic. Lambda is again able to handle all requests without cold start latencies.
- At `t4`, functions in your account experience a burst in traffic. This burst can come from `function-orange` or any other function in your account. Lambda uses unreserved concurrency to handle these requests.
- At `t5`, functions in your account reach the maximum concurrency limit of 1,000, and experience throttling.

The previous example considered only provisioned concurrency. In practice, you can set both provisioned concurrency and reserved concurrency on a function. You might do this if you had a function that handles a consistent load of invocations on weekdays, but routinely sees spikes of traffic on weekends. In this case, you could use provisioned concurrency to set a baseline amount of environments to handle request during weekdays, and use reserved concurrency to handle the weekend spikes. Consider the following diagram:



In this diagram, suppose that you configure 200 units of provisioned concurrency and 400 units of reserved concurrency for `function-orange`. Because you configured reserved concurrency, `function-orange` cannot use any of the 600 units of unreserved concurrency.

This diagram has five points of interest:

- At `t1`, `function-orange` begins receiving requests. Since Lambda has pre-initialized 200 execution environment instances, `function-orange` is ready for immediate invocation.
- At `t2`, `function-orange` uses up all its provisioned concurrency. `function-orange` can continue serving requests using reserved concurrency, but these requests may experience cold start latencies.
- At `t3`, `function-orange` reaches 400 concurrent requests. As a result, `function-orange` uses up all its reserved concurrency. Since `function-orange` cannot use unreserved concurrency, requests begin to throttle.
- At `t4`, `function-orange` starts to receive fewer requests, and no longer throttles.
- At `t5`, `function-orange` drops down to 200 concurrent requests, so all requests are again able to use provisioned concurrency (that is, no cold start latencies).

Both reserved concurrency and provisioned concurrency count towards your account concurrency limit and Regional quotas. In other words, allocating reserved and provisioned concurrency can impact the concurrency pool that's available to other functions. Configuring provisioned concurrency incurs charges to your AWS account.

> **Note**
>
> If the amount of provisioned concurrency on a function's versions and aliases adds up to the function's reserved concurrency, then all invocations run on provisioned concurrency. This configuration also has the effect of throttling the unpublished version of the function ($LATEST), which prevents it from executing. You can't allocate more provisioned concurrency than reserved concurrency for a function.

To manage provisioned concurrency settings for your functions, see Configuring provisioned concurrency for a function. To automate provisioned concurrency scaling based on a schedule or application utilization, see Using Application Auto Scaling to automate provisioned concurrency management.

How Lambda allocates provisioned concurrency

Provisioned concurrency doesn't come online immediately after you configure it. Lambda starts allocating provisioned concurrency after a minute or two of preparation. For each function,

Lambda can provision up to 6,000 execution environments every minute, regardless of AWS Region. This is exactly the same as the concurrency scaling rate for functions.

When you submit a request to allocate provisioned concurrency, you can't access any of those environments until Lambda completely finishes allocating them. For example, if you request 5,000 provisioned concurrency, none of your requests can use provisioned concurrency until Lambda completely finishes allocating the 5,000 execution environments.

Comparing reserved concurrency and provisioned concurrency

The following table summarizes and compares reserved and provisioned concurrency.

Topic	Reserved concurrency	Provisioned concurrency
Definition	Maximum number of execution environment instances for your function.	Set number of pre-provisioned execution environment instances for your function.
Provisioning behavior	Lambda provisions new instances on an on-demand basis.	Lambda pre-provisions instances (that is, before your function starts receiving requests).
Cold start behavior	Cold start latency possible, since Lambda must create new instances on-demand.	Cold start latency not possible, since Lambda doesn't have to create instances on-demand.
Throttling behavior	Function throttled when reserved concurrency limit reached.	If reserved concurrency not set: function uses unreserved concurrency when provisioned concurrency limit reached. If reserved concurrency set: function throttled when reserved concurrency limit reached.

Topic	Reserved concurrency	Provisioned concurrency
Default behavior if not set	Function uses unreserved concurrency available in your account.	Lambda doesn't pre-provision any instances. Instead, if reserved concurrency not set: function uses unreserved concurrency available in your account. If reserved concurrency set: function uses reserved concurrency.
Pricing	No additional charge.	Incurs additional charges.

Understanding concurrency and requests per second

As mentioned in the previous section, concurrency differs from requests per second. This is an especially important distinction to make when working with functions that have an average request duration of less than 100 ms.

Across all functions in your account, Lambda enforces a requests per second limit that's equal to 10 times your account concurrency. For example, since the default account concurrency limit is 1,000, functions in your account can handle a maximum of 10,000 requests per second.

For example, consider a function with an average request duration of 50 ms. At 20,000 requests per second, here's the concurrency of this function:

```
Concurrency = (20,000 requests/second) * (0.05 second/request) = 1,000
```

Based on this result, you might expect that the account concurrency limit of 1,000 is sufficient to handle this load. However, because of the 10,000 requests per second limit, your function can only handle 10,000 requests per second out of the 20,000 total requests. This function experiences throttling.

The lesson is that you must consider both concurrency and requests per second when configuring concurrency settings for your functions. In this case, you need to request an account concurrency limit increase to 2,000, since this would increase your total requests per second limit to 20,000.

> **Note**
>
> Based on this request per second limit, it's incorrect to say that each Lambda execution environment can handle only a maximum of 10 requests per second. Instead of observing the load on any individual execution environment, Lambda only considers overall concurrency and overall requests per second when calculating your quotas.

Test your understanding of concurrency (sub-100 ms functions)

Suppose that you have a function that takes, on average, 20 ms to run. During peak load, you observe 30,000 requests per second. What is the concurrency of your function during peak load?

Answer

The average function duration is 20 ms, or 0.02 seconds. Using the concurrency formula, you can plug in the numbers to get a concurrency of 600:

```
Concurrency = (30,000 requests/second) * (0.02 seconds/request) = 600
```

By default, the account concurrency limit of 1,000 seems sufficient to handle this load. However, the requests per second limit of 10,000 isn't enough to handle the incoming 30,000 requests per second. To fully accommodate the 30,000 requests, you need to request an account concurrency limit increase to 3,000 or higher.

The requests per second limit applies to all quotas in Lambda that involve concurrency. In other words, it applies to synchronous on-demand functions, functions that use provisioned concurrency, and concurrency scaling behavior. For example, here are a few scenarios where you must carefully consider both your concurrency and request per second limits:

- A function using on-demand concurrency can experience a burst increase of 500 concurrency every 10 seconds, or by 5,000 requests per second every 10 seconds, whichever happens first.
- Suppose you have a function that has a provisioned concurrency allocation of 10. This function spills over into on-demand concurrency after 10 concurrency or 100 requests per second, whichever happens first.

Concurrency quotas

Lambda sets quotas for the total amount of concurrency that you can use across all functions in a Region. These quotas exist on two levels:

- **At the account level**, your functions can have up to 1,000 units of concurrency by default. To increase this limit, see Requesting a quota increase in the *Service Quotas User Guide*.
- **At the function level**, you can reserve up to 900 units of concurrency across all your functions by default. Regardless of your total account concurrency limit, Lambda always reserves 100 units of concurrency for your functions that don't explicitly reserve concurrency. For example, if you increased your account concurrency limit to 2,000, then you can reserve up to 1,900 units of concurrency at the function level.
- At both the account level and the function level, Lambda also enforces a requests per second limit of equal to 10 times the corresponding concurrency quota. For instance, this applies to account-level concurrency, functions using on-demand concurrency, functions using provisoned concurrency, and concurrency scaling behavior. For more information, see the section called "Understanding concurrency and requests per second".

To check your current account level concurrency quota, use the AWS Command Line Interface (AWS CLI) to run the following command:

```
aws lambda get-account-settings
```

You should see output that looks like the following:

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

`ConcurrentExecutions` is your total account-level concurrency quota. `UnreservedConcurrentExecutions` is the amount of reserved concurrency that you can still allocate to your functions.

As your function receives more requests, Lambda automatically scales up the number of execution environments to handle these requests until your account reaches its concurrency quota. However, to protect against over-scaling in response to sudden bursts of traffic, Lambda limits how fast your functions can scale. This **concurrency scaling rate** is the maximum rate at which functions in your account can scale in response to increased requests. (That is, how quickly Lambda can create new execution environments.) The concurrency scaling rate differs from the account-level concurrency limit, which is the total amount of concurrency available to your functions.

In each AWS Region, and for each function, your concurrency scaling rate is 1,000 execution environment instances every 10 seconds (or 10,000 requests per second every 10 seconds). In other words, every 10 seconds, Lambda can allocate at most 1,000 additional execution environment instances, or accommodate 10,000 additional requests per second, to each of your functions.

Usually, you don't need to worry about this limitation. Lambda's scaling rate is sufficient for most use cases.

Importantly, the concurrency scaling rate is a function-level limit. This means that each function in your account can scale independently of other functions.

For more information about scaling behavior, see Lambda scaling behavior.

Configuring reserved concurrency for a function

In Lambda, concurrency is the number of in-flight requests that your function is currently handling. There are two types of concurrency controls available:

- Reserved concurrency – This sets both the maximum and minimum number of concurrent instances allocated to your function. When a function has reserved concurrency, no other function can use that concurrency. Reserved concurrency is useful for ensuring that your most critical functions always have enough concurrency to handle incoming requests. Additionally, reserved concurrency can be used for limiting concurrency to prevent overwhelming downstream resources, like database connections. Reserved concurrency acts as both a lower and upper bound - it reserves the specified capacity exclusively for your function while also preventing it from scaling beyond that limit. Configuring reserved concurrency for a function incurs no additional charges.
- Provisioned concurrency – This is the number of pre-initialized execution environments allocated to your function. These execution environments are ready to respond immediately to incoming function requests. Provisioned concurrency is useful for reducing cold start latencies for functions and designed to make functions available with double-digit millisecond response times. Generally, interactive workloads benefit the most from the feature. Those are applications with users initiating requests, such as web and mobile applications, and are the most sensitive to latency. Asynchronous workloads, such as data processing pipelines, are often less latency sensitive and so do not usually need provisioned concurrency. Configuring provisioned concurrency incurs additional charges to your AWS account.

This topic details how to manage and configure reserved concurrency. For a conceptual overview of these two types of concurrency controls, see Reserved concurrency and provisioned concurrency. For information on configuring provisioned concurrency, see the section called “Configuring provisioned concurrency”.

> **Note**
>
> Lambda functions linked to an Amazon MQ event source mapping have a default maximum concurrency. For Apache Active MQ, the maximum number of concurrent instances is 5. For Rabbit MQ, the maximum number of concurrent instances is 1. Setting reserved or provisioned concurrency for your function doesn't change these limits. To request an increase in the default maximum concurrency when using Amazon MQ, contact Support.

Sections

- Configuring reserved concurrency
- Accurately estimating required reserved concurrency for a function

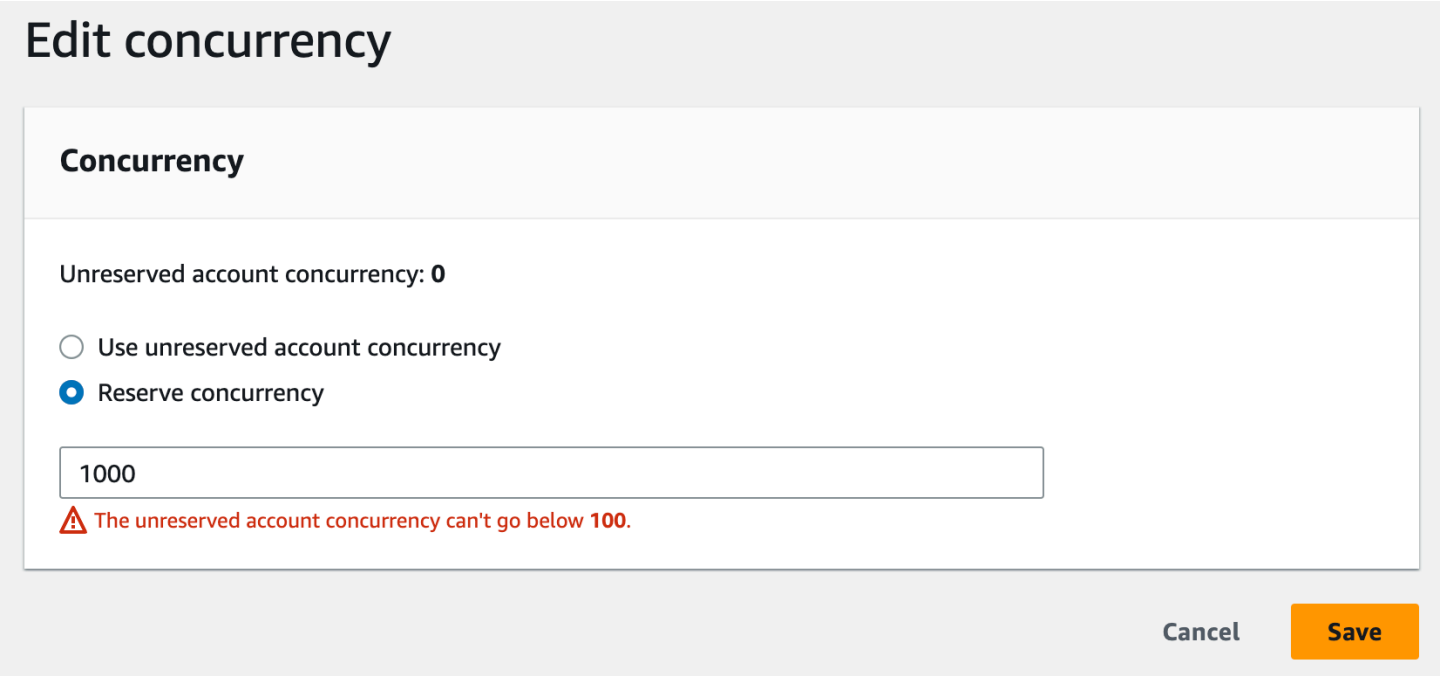
Configuring reserved concurrency

You can configure reserved concurrency settings for a function using the Lambda console or the Lambda API.

To reserve concurrency for a function (console)

1. Open the Functions page of the Lambda console.
2. Choose the function you want to reserve concurrency for.
3. Choose **Configuration** and then choose **Concurrency**.
4. Under **Concurrency**, choose **Edit**.
5. Choose **Reserve concurrency**. Enter the amount of concurrency to reserve for the function.
6. Choose **Save**.

You can reserve up to the **Unreserved account concurrency** value minus 100. The remaining 100 units of concurrency are for functions that aren't using reserved concurrency. For example, if your account has a concurrency limit of 1,000, you cannot reserve all 1,000 units of concurrency to a single function.



Reserving concurrency for a function impacts the concurrency pool that's available to other functions. For example, if you reserve 100 units of concurrency for `function-a`, other functions in your account must share the remaining 900 units of concurrency, even if `function-a` doesn't use all 100 reserved concurrency units.

To intentionally throttle a function, set its reserved concurrency to 0. This stops your function from processing any events until you remove the limit.

To configure reserved concurrency with the Lambda API, use the following API operations.

- PutFunctionConcurrency
- GetFunctionConcurrency
- DeleteFunctionConcurrency

For example, to configure reserved concurrency with the AWS Command Line Interface (CLI), use the `put-function-concurrency` command. The following command reserves 100 concurrency units for a function named `my-function`:

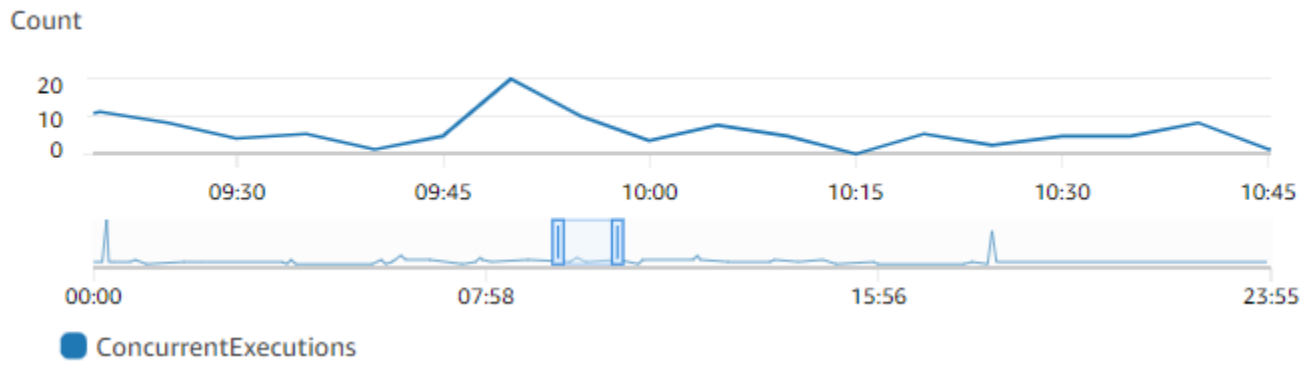
```
aws lambda put-function-concurrency --function-name my-function \
    --reserved-concurrent-executions 100
```

You should see output that looks like the following:

```
{
    "ReservedConcurrentExecutions": 100
}
```

Accurately estimating required reserved concurrency for a function

If your function is currently serving traffic, you can easily view its concurrency metrics using CloudWatch metrics. Specifically, the `ConcurrentExecutions` metric shows you the number of concurrent invocations for each function in your account.



The previous graph suggests that this function serves an average of 5 to 10 concurrent requests at any given time, and peaks at 20 requests on a typical day. Suppose that there are many other functions in your account. **If this function is critical to your application and you don't want to drop any requests**, use a number greater than or equal to 20 as your reserved concurrency setting.

Alternatively, recall that you can also calculate concurrency using the following formula:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

Multiplying average requests per second with the average request duration in seconds gives you a rough estimate of how much concurrency you need to reserve. You can estimate average requests per second using the `Invocation` metric, and the average request duration in seconds using the `Duration` metric. See Using CloudWatch metrics with Lambda for more details.

You should also be familiar with your upstream and downstream throughput constraints. While Lambda functions scale seamlessly with load, upstream and downstream dependencies may not have the same throughput capabilities. If you need to limit how high your function can scale, configure reserved concurrency on your function.

Configuring provisioned concurrency for a function

In Lambda, concurrency is the number of in-flight requests that your function is currently handling. There are two types of concurrency controls available:

- Reserved concurrency – This sets both the maximum and minimum number of concurrent instances allocated to your function. When a function has reserved concurrency, no other function can use that concurrency. Reserved concurrency is useful for ensuring that your most critical functions always have enough concurrency to handle incoming requests. Additionally, reserved concurrency can be used for limiting concurrency to prevent overwhelming downstream resources, like database connections. Reserved concurrency acts as both a lower and upper bound - it reserves the specified capacity exclusively for your function while also preventing it from scaling beyond that limit. Configuring reserved concurrency for a function incurs no additional charges.
- Provisioned concurrency – This is the number of pre-initialized execution environments allocated to your function. These execution environments are ready to respond immediately to incoming function requests. Provisioned concurrency is useful for reducing cold start latencies for functions and designed to make functions available with double-digit millisecond response times. Generally, interactive workloads benefit the most from the feature. Those are applications with users initiating requests, such as web and mobile applications, and are the most sensitive to latency. Asynchronous workloads, such as data processing pipelines, are often less latency sensitive and so do not usually need provisioned concurrency. Configuring provisioned concurrency incurs additional charges to your AWS account.

This topic details how to manage and configure provisioned concurrency. For a conceptual overview of these two types of concurrency controls, see Reserved concurrency and provisioned concurrency. For more information on configuring reserved concurrency, see the section called "Configuring reserved concurrency".

> **Note**
>
> Lambda functions linked to an Amazon MQ event source mapping have a default maximum concurrency. For Apache Active MQ, the maximum number of concurrent instances is 5. For Rabbit MQ, the maximum number of concurrent instances is 1. Setting reserved or provisioned concurrency for your function doesn't change these limits. To request an increase in the default maximum concurrency when using Amazon MQ, contact Support.

Sections

- Configuring provisioned concurrency
- Accurately estimating required provisioned concurrency for a function
- Optimizing function code when using provisioned concurrency
- Using environment variables to view and control provisioned concurrency behavior
- Understanding logging and billing behavior with provisioned concurrency
- Using Application Auto Scaling to automate provisioned concurrency management

Configuring provisioned concurrency

You can configure provisioned concurrency settings for a function using the Lambda console or the Lambda API.

To allocate provisioned concurrency for a function (console)

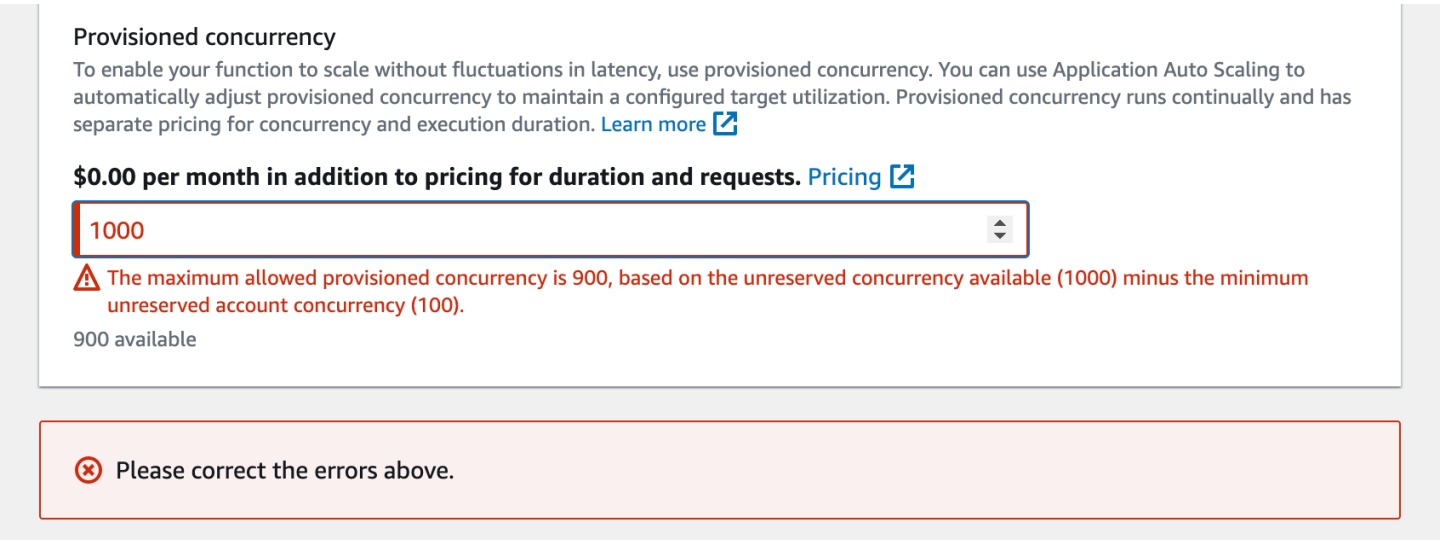
1. Open the Functions page of the Lambda console.
2. Choose the function you want to allocate provisioned concurrency for.
3. Choose **Configuration** and then choose **Concurrency**.
4. Under **Provisioned concurrency configurations**, choose **Add configuration**.
5. Choose the qualifier type, and alias or version.

> **Note**
>
> You cannot use provisioned concurrency with the $LATEST version of any function. If your function has an event source, make sure that event source points to the correct function alias or version. Otherwise, your function won't use provisioned concurrency environments.

6. Enter a number under **Provisioned concurrency**.
7. Choose **Save**.

You can configure up to the **Unreserved account concurrency** in your account, minus 100. The remaining 100 units of concurrency are for functions that aren't using reserved concurrency. For example, if your account has a concurrency limit of 1,000, and you haven't assigned any reserved

or provisioned concurrency to any of your other functions, you can configure a maximum of 900 provisioned concurrency units for a single function.



Configuring provisioned concurrency for a function has an impact on the concurrency pool available to other functions. For instance, if you configure 100 units of provisioned concurrency for `function-a`, other functions in your account must share the remaining 900 units of concurrency. This is true even if `function-a` doesn't use all 100 units.

It's possible to allocate both reserved concurrency and provisioned concurrency for the same function. In such cases, the provisioned concurrency cannot exceed the reserved concurrency.

This limitation extends to function versions. The maximum provisioned concurrency you can assign to a specific function version is the function's reserved concurrency minus the provisioned concurrency on other function versions.

To configure provisioned concurrency with the Lambda API, use the following API operations.

- PutProvisionedConcurrencyConfig
- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

For example, to configure provisioned concurrency with the AWS Command Line Interface (CLI), use the `put-provisioned-concurrency-config` command. The following command allocates 100 units of provisioned concurrency for the `BLUE` alias of a function named `my-function`:

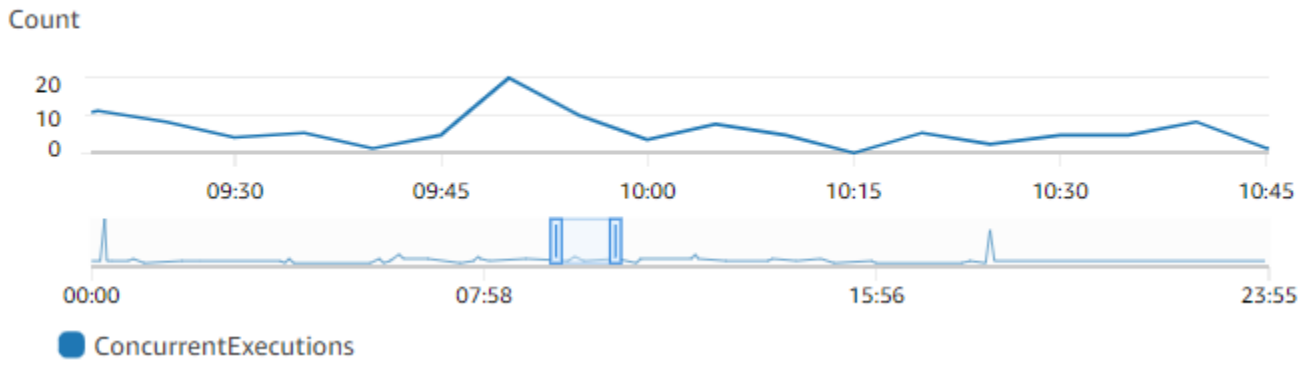
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

You should see output that looks like the following:

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

Accurately estimating required provisioned concurrency for a function

You can view any active function's concurrency metrics using CloudWatch metrics. Specifically, the `ConcurrentExecutions` metric shows you the number of concurrent invocations for functions in your account.



The previous graph suggests that this function serves an average of 5 to 10 concurrent requests at any given time, and peaks at 20 requests. Suppose that there are many other functions in your account. **If this function is critical to your application and you need a low-latency response on every invocation**, configure at least 20 units of provisioned concurrency.

Recall that you can also calculate concurrency using the following formula:

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

To estimate how much concurrency you need, multiply average requests per second with the average request duration in seconds. You can estimate average requests per second using the `Invocation` metric, and the average request duration in seconds using the `Duration` metric.

When configuring provisioned concurrency, Lambda suggests adding a 10% buffer on top of the amount of concurrency your function typically needs. For example, if your function usually peaks at 200 concurrent requests, set the provisioned concurrency to 220 (200 concurrent requests + 10% = 220 provisioned concurrency).

Optimizing function code when using provisioned concurrency

If you're using provisioned concurrency, consider restructuring your function code to optimize for low latency. For functions using provisioned concurrency, Lambda runs any initialization code, such as loading libraries and instantiating clients, during allocation time. Therefore, it's advisable to move as much initialization outside of the main function handler to avoid impacting latency during actual function invocations. In contrast, initializing libraries or instantiating clients within your main handler code means your function must run this each time it's invoked (this occurs regardless of whether you're using provisioned concurrency).

For on-demand invocations, Lambda may need to rerun your initialization code every time your function experiences a cold start. For such functions, you may choose to defer initialization of a specific capability until your function needs it. For example, consider the following control flow for a Lambda handler:

```
def handler(event, context):
    ...
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

In the previous example, instead of initializing `CLIENT_A` outside of the main handler, the developer initialized it within the `if` statement. By doing this, Lambda runs this code only if `some_condition` is met. If you initialize `CLIENT_A` outside the main handler, Lambda runs that code on every cold start. This can increase overall latency.

You can measure cold starts as Lambda scales up by adding X-Ray monitoring to your function. A function using provisioned concurrency does not exhibit cold start behavior since the execution environment is prepared ahead of invocation. However, provisioned concurrency must be applied

to a specific version or alias of a function, not the $LATEST version. In cases where you continue to see cold start behavior, ensure that you are invoking the version of alias with provisioned concurrency configured.

Using environment variables to view and control provisioned concurrency behavior

It's possible for your function to use up all of its provisioned concurrency. Lambda uses on-demand instances to handle any excess traffic. To determine the type of initialization Lambda used for a specific environment, check the value of the `AWS_LAMBDA_INITIALIZATION_TYPE` environment variable. This variable has two possible values: `provisioned-concurrency` or `on-demand`. The value of `AWS_LAMBDA_INITIALIZATION_TYPE` is immutable and remains constant throughout the lifetime of the environment. To check the value of an environment variable in your function code, see Retrieving Lambda environment variables.

If you're using the .NET 8 runtime, you can configure the `AWS_LAMBDA_DOTNET_PREJIT` environment variable to improve the latency for functions, even if they don't use provisioned concurrency. The .NET runtime employs lazy compilation and initialization for each library that your code calls for the first time. As a result, the first invocation of a Lambda function may take longer than subsequent ones. To mitigate this, you can choose one of three values for `AWS_LAMBDA_DOTNET_PREJIT`:

- `ProvisionedConcurrency`: Lambda performs ahead-of-time JIT compilation for all environments using provisioned concurrency. This is the default value.
- `Always`: Lambda performs ahead-of-time JIT compilation for every environment, even if the function doesn't use provisioned concurrency.
- `Never`: Lambda disables ahead-of-time JIT compilation for all environments.

Understanding logging and billing behavior with provisioned concurrency

For provisioned concurrency environments, your function's initialization code runs during allocation, and periodically as Lambda recycles instances of your environment. Lambda bills you for initialization even if the environment instance never processes a request. Provisioned concurrency runs continually and incurs separate billing from initialization and invocation costs. For more details, see AWS Lambda Pricing.

When you configure a Lambda function with provisioned concurrency, Lambda pre-initializes that execution environment so that it's available ahead of invocation requests. Lambda logs the Init Duration field of the function in a platform-initReport log event in JSON logging format every time the environment is initialized. To see this log event, configure your JSON log level to at least `INFO`. You can also use the Telemetry API to consume platform events where the Init Duration field is reported.

Using Application Auto Scaling to automate provisioned concurrency management

You can use Application Auto Scaling to manage provisioned concurrency on a schedule or based on utilization. If your function receives predictable traffic patterns, use scheduled scaling. If you want your function to maintain a specific utilization percentage, use a target tracking scaling policy.

> **Note**
>
> If you use Application Auto Scaling to manage your function's provisioned concurrency, ensure that you configure an initial provisioned concurrency value first. If your function doesn't have an initial provisioned concurrency value, Application Auto Scaling may not handle function scaling properly.

Scheduled scaling

With Application Auto Scaling, you can set your own scaling schedule according to predictable load changes. For more information and examples, see Scheduled scaling for Application Auto Scaling in the Application Auto Scaling User Guide, and Scheduling AWS Lambda Provisioned Concurrency for recurring peak usage on the AWS Compute Blog.

Target tracking

With target tracking, Application Auto Scaling creates and manages a set of CloudWatch alarms based on how you define your scaling policy. When these alarms activate, Application Auto Scaling automatically adjusts the amount of environments allocated using provisioned concurrency. Use target tracking for applications that don't have predictable traffic patterns.

To scale provisioned concurrency using target tracking, use the `RegisterScalableTarget` and `PutScalingPolicy` Application Auto Scaling API operations. For example, if you're using the AWS Command Line Interface (CLI), follow these steps:

1. Register a function's alias as a scaling target. The following example registers the BLUE alias of a function named `my-function`:

```
aws application-autoscaling register-scalable-target --service-namespace lambda \
    --resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \
    --scalable-dimension lambda:function:ProvisionedConcurrency
```

2. Apply a scaling policy to the target. The following example configures Application Auto Scaling to adjust the provisioned concurrency configuration for an alias to keep utilization near 70 percent, but you can apply any value between 10% and 90%.

```
aws application-autoscaling put-scaling-policy \
    --service-namespace lambda \
    --scalable-dimension lambda:function:ProvisionedConcurrency \
    --resource-id function:my-function:BLUE \
    --policy-name my-policy \
    --policy-type TargetTrackingScaling \
    --target-tracking-scaling-policy-configuration '{ "TargetValue":
 0.7, "PredefinedMetricSpecification": { "PredefinedMetricType":
 "LambdaProvisionedConcurrencyUtilization" }}'
```

You should see output that looks like this:

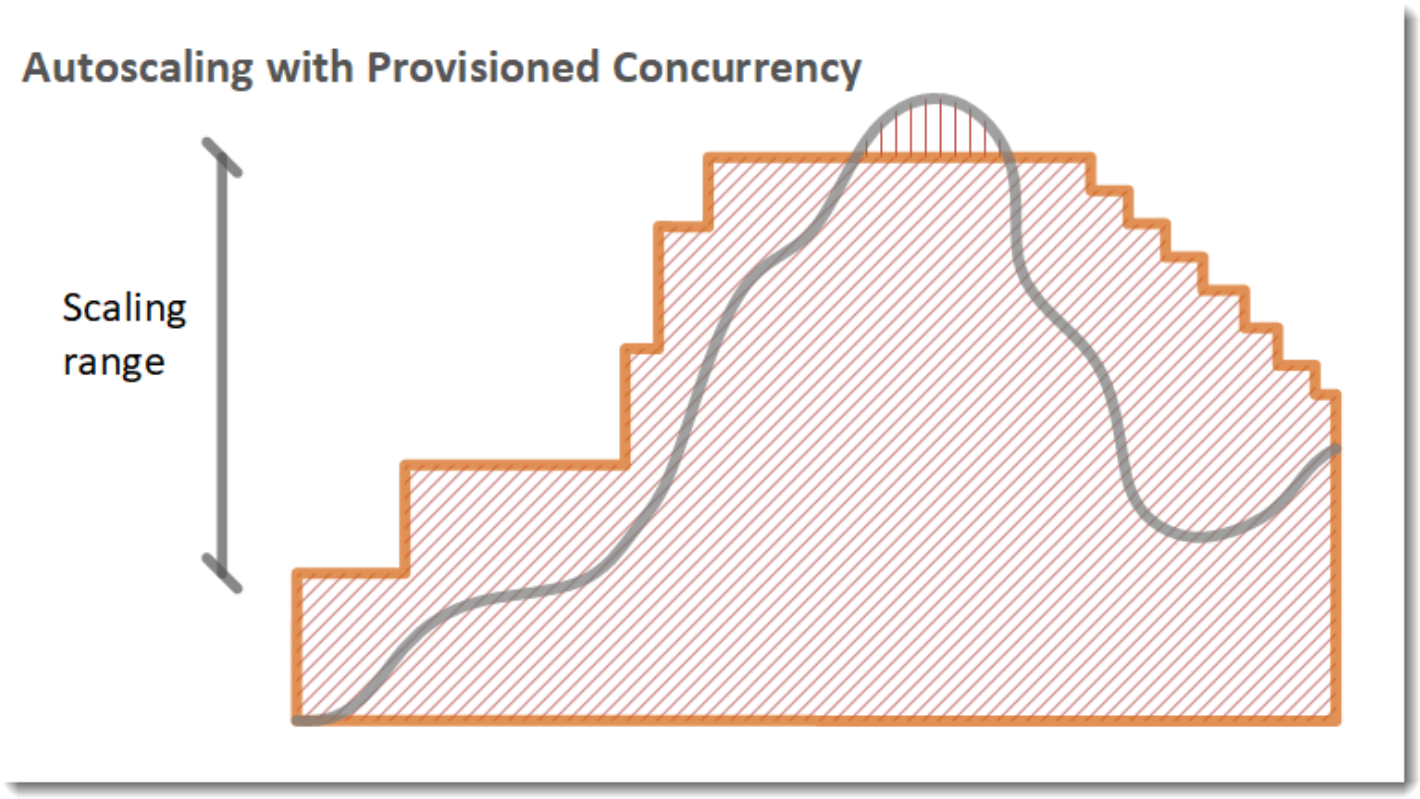
```
{
    "PolicyARN": "arn:aws:autoscaling:us-
east-2:123456789012:scalingPolicy:12266dbb-1524-xmpl-a64e-9a0a34b996fa:resource/lambda/
function:my-function:BLUE:policyName/my-policy",
    "Alarms": [
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7"
        },
        {
```

```
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66"
        }
    ]
}
```

Application Auto Scaling creates two alarms in CloudWatch. The first alarm triggers when the utilization of provisioned concurrency consistently exceeds 70%. When this happens, Application Auto Scaling allocates more provisioned concurrency to reduce utilization. The second alarm triggers when utilization is consistently less than 63% (90 percent of the 70% target). When this happens, Application Auto Scaling reduces the alias's provisioned concurrency.

> **Note**
>
> Lambda emits the `ProvisionedConcurrencyUtilization` metric only when your function is active and receiving requests. During periods of inactivity, no metrics are emitted, and your auto-scaling alarms will enter the `INSUFFICIENT_DATA` state. As a result, Application Auto Scaling won't be able to adjust your function's provisioned concurrency. This may lead to unexpected billing.

In the following example, a function scales between a minimum and maximum amount of provisioned concurrency based on utilization.



Legend

- Function instances
- Open requests
- Provisioned concurrency
- Standard concurrency

When the number of open requests increase, Application Auto Scaling increases provisioned concurrency in large steps until it reaches the configured maximum. After this, the function can continue to scale on standard, unreserved concurrency if you haven't reached your account concurrency limit. When utilization drops and stays low, Application Auto Scaling decreases provisioned concurrency in smaller periodic steps.

Both of the Application Auto Scaling alarms use the average statistic by default. Functions that experience quick bursts of traffic may not trigger these alarms. For example, suppose your Lambda function executes quickly (i.e. 20-100 ms) and your traffic comes in quick bursts. In this case, the number of requests exceeds the allocated provisioned concurrency during the burst. However, Application Auto Scaling requires the burst load to sustain for at least 3 minutes in order to provision additional environments. Additionally, both CloudWatch alarms require 3 data points that hit the target average to activate the auto scaling policy. If your function experiences quick bursts of traffic, using the **Maximum** statistic instead of the **Average** statistic can be more effective at scaling provisioned concurrency to minimize cold starts.

For more information on target tracking scaling policies, see Target tracking scaling policies for Application Auto Scaling.

Lambda scaling behavior

As your function receives more requests, Lambda automatically scales up the number of execution environments to handle these requests until your account reaches its concurrency quota. However, to protect against over-scaling in response to sudden bursts of traffic, Lambda limits how fast your functions can scale. This **concurrency scaling rate** is the maximum rate at which functions in your account can scale in response to increased requests. (That is, how quickly Lambda can create new execution environments.) The concurrency scaling rate differs from the account-level concurrency limit, which is the total amount of concurrency available to your functions.

Concurrency scaling rate

In each AWS Region, and for each function, your concurrency scaling rate is 1,000 execution environment instances every 10 seconds (or 10,000 requests per second every 10 seconds). In other words, every 10 seconds, Lambda can allocate at most 1,000 additional execution environment instances, or accommodate 10,000 additional requests per second, to each of your functions.

Usually, you don't need to worry about this limitation. Lambda's scaling rate is sufficient for most use cases.

Importantly, the concurrency scaling rate is a function-level limit. This means that each function in your account can scale independently of other functions.

> **Note**
>
> In practice, Lambda makes a best attempt to refill your concurrency scaling rate continuously over time, rather than in one single refill of 1,000 units every 10 seconds.

Lambda doesn't accrue unused portions of your concurrency scaling rate. This means that at any instant in time, your scaling rate is always 1,000 concurrency units at maximum. For example, if you don't use any of your available 1,000 concurrency units in a 10-second interval, you won't accrue 1,000 additional units in the next 10-second interval. Your concurrency scaling rate is still 1,000 in the next 10-second interval.

As long as your function continues to receive increasing numbers of requests, then Lambda scales at the fastest rate available to you, up to your account's concurrency limit. You can limit the

amount of concurrency that individual functions can use by configuring reserved concurrency. If requests come in faster than your function can scale, or if your function is at maximum concurrency, then additional requests fail with a throttling error (429 status code).

Monitoring concurrency

Lambda emits Amazon CloudWatch metrics to help you monitor concurrency for your functions. This topic explains these metrics and how to interpret them.

Sections

- General concurrency metrics
- Provisioned concurrency metrics
- Working with the ClaimedAccountConcurrency metric

General concurrency metrics

Use the following metrics to monitor concurrency for your Lambda functions. The granularity for each metric is 1 minute.

- `ConcurrentExecutions` – The number of active concurrent invocations at a given point in time. Lambda emits this metric for all functions, versions, and aliases. For any function in the Lambda console, Lambda displays the graph for `ConcurrentExecutions` natively in the **Monitoring** tab, under **Metrics**. View this metric using **MAX**.
- `UnreservedConcurrentExecutions` – The number of active concurrent invocations that are using unreserved concurrency. Lambda emits this metric across all functions in a region. View this metric using **MAX**.
- `ClaimedAccountConcurrency` – The amount of concurrency that is unavailable for on-demand invocations. `ClaimedAccountConcurrency` is equal to `UnreservedConcurrentExecutions` plus the amount of allocated concurrency (i.e. the total reserved concurrency plus total provisioned concurrency). If `ClaimedAccountConcurrency` exceeds your account concurrency limit, you can request a higher account concurrency limit. View this metric using **MAX**. For more information, see Working with the `ClaimedAccountConcurrency` metric.

Provisioned concurrency metrics

Use the following metrics to monitor Lambda functions using provisioned concurrency. The granularity for each metric is 1 minute.

- `ProvisionedConcurrentExecutions` – The number of execution environment instances that are actively processing an invocation on provisioned concurrency. Lambda emits this metric for each function version and alias with provisioned concurrency configured. View this metric using **MAX**.

`ProvisionedConcurrentExecutions` is not the same as the total number of provisioned concurrency that you allocate. For example, suppose you allocate 100 units of provisioned concurrency to a function version. During any given minute, if at most 50 out of those 100 execution environments were handling invocations simultaneously, then the value of **MAX**(`ProvisionedConcurrentExecutions`) is 50.

- `ProvisionedConcurrencyInvocations` – The number of times Lambda invokes your function code using provisioned concurrency. Lambda emits this metric for each function version and alias with provisioned concurrency configured. View this metric using **SUM**.

`ProvisionedConcurrencyInvocations` differs from `ProvisionedConcurrentExecutions` in that `ProvisionedConcurrencyInvocations` counts total number of invocations, while `ProvisionedConcurrentExecutions` counts number of active environments. To understand this distinction, consider the following scenario:



In this example, suppose that you receive 1 invocation per minute, and each invocation takes 2 minutes to complete. Each orange horizontal bar represents a single request. Suppose that you allocate 10 units of provisioned concurrency to this function, such that each request runs on provisioned concurrency.

In between minutes 0 and 1, `Request 1` comes in. **At minute 1**, the value for **MAX**(`ProvisionedConcurrentExecutions`) is 1, since at most 1 execution environment was active during the past minute. The value for **SUM**(`ProvisionedConcurrencyInvocations`) is also 1, since 1 new request came in during the past minute.

In between minutes 1 and 2, `Request 2` comes in, and `Request 1` continues to run. **At minute 2**, the value for **MAX**(`ProvisionedConcurrentExecutions`) is 2, since at most 2 execution environments were active during the past minute. However, the value for **SUM**(`ProvisionedConcurrencyInvocations`) is 1, since only 1 new request came in during the past minute. This metric behavior continues until the end of the example.

- `ProvisionedConcurrencySpilloverInvocations` – The number of times Lambda invokes your function on standard (reserved or unreserved) concurrency when all provisioned concurrency is in use. Lambda emits this metric for each function version and alias with provisioned concurrency configured. View this metric using **SUM**. The value of `ProvisionedConcurrencyInvocations` + `ProvisionedConcurrencySpilloverInvocations` should be equal to the total number of function invocations (i.e. the `Invocations` metric).

  `ProvisionedConcurrencyUtilization` – The percentage of provisioned concurrency in use (i.e. the value of `ProvisionedConcurrentExecutions` divided by the total amount of provisioned concurrency allocated). Lambda emits this metric for each function version and alias with provisioned concurrency configured. View this metric using **MAX**.

For example, suppose you provision 100 units of provisioned concurrency to a function version. During any given minute, if at most 60 out of those 100 execution environments were handling invocations simultaneously, then the value of **MAX**(`ProvisionedConcurrentExecutions`) is 60, and the value of **MAX**(`ProvisionedConcurrencyUtilization`) is 0.6.

A high value for `ProvisionedConcurrencySpilloverInvocations` may indicate that you need to allocate additional provisioned concurrency for your function. Alternatively, you can configure Application Auto Scaling to handle automatic scaling of provisioned concurrency based on pre-defined thresholds.

Conversely, consistently low values for `ProvisionedConcurrencyUtilization` may indicate that you over-allocated provisioned concurrency for your function.

Working with the ClaimedAccountConcurrency metric

Lambda uses the ClaimedAccountConcurrency metric to determine how much concurrency your account is available for on-demand invocations. Lambda calculates ClaimedAccountConcurrency using the following formula:

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

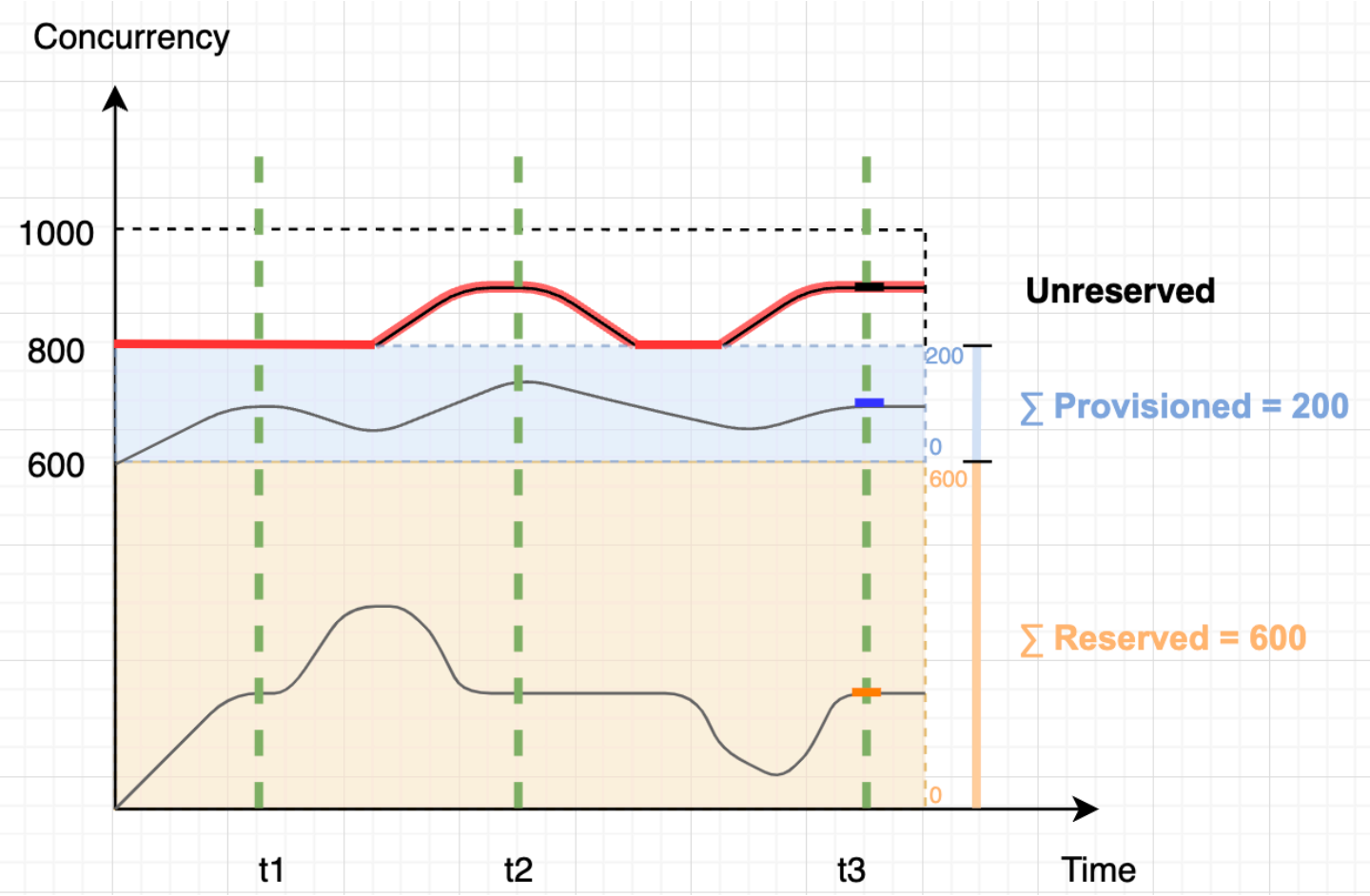
UnreservedConcurrentExecutions is the number of active concurrent invocations that are using unreserved concurrency. Allocated concurrency is the sum of the following two parts (substituting RC as "reserved concurrency" and PC as "provisioned concurrency"):

- The total RC across all functions in a Region.
- The total PC across all functions in a Region that use PC, excluding functions that use RC.

> **Note**
>
> You can't allocate more PC than RC for a function. Thus, a function's RC is always greater than or equal to its PC. To calculate the contribution to allocated concurrency for such functions with both PC and RC, Lambda considers only RC, which is the maximum of the two.

Lambda uses the ClaimedAccountConcurrency metric, rather than ConcurrentExecutions, to determine how much concurrency is available for on-demand invocations. While the ConcurrentExecutions metric is useful for tracking the number of active concurrent invocations, it doesn't always reflect your true concurrency availability. This is because Lambda also considers reserved concurrency and provisioned concurrency to determine availability.

To illustrate ClaimedAccountConcurrency, consider a scenario where you configure a lot of reserved concurrency and provisioned concurrency across your functions that go largely unused. In the following example, assume that your account concurrency limit is 1,000, and you have two main functions in your account: function-orange and function-blue. You allocate 600 units of reserved concurrency for function-orange. You allocate 200 units of provisioned concurrency for function-blue. Suppose that over time, you deploy additional functions and observe the following traffic pattern:



In the previous diagram, the black lines indicate the actual concurrency use over time, and the red line indicates the value of `ClaimedAccountConcurrency` over time. Throughout this scenario, `ClaimedAccountConcurrency` is 800 at minimum, despite low actual concurrency utilization across your functions. This is because you allocated 800 total units of concurrency for `function-orange` and `function-blue`. From Lambda's perspective, you have "claimed" this concurrency for use, so you effectively have only 200 units of concurrency remaining for other functions.

For this scenario, allocated concurrency is 800 in the `ClaimedAccountConcurrency` formula. We can then derive the value of `ClaimedAccountConcurrency` at various points in the diagram:

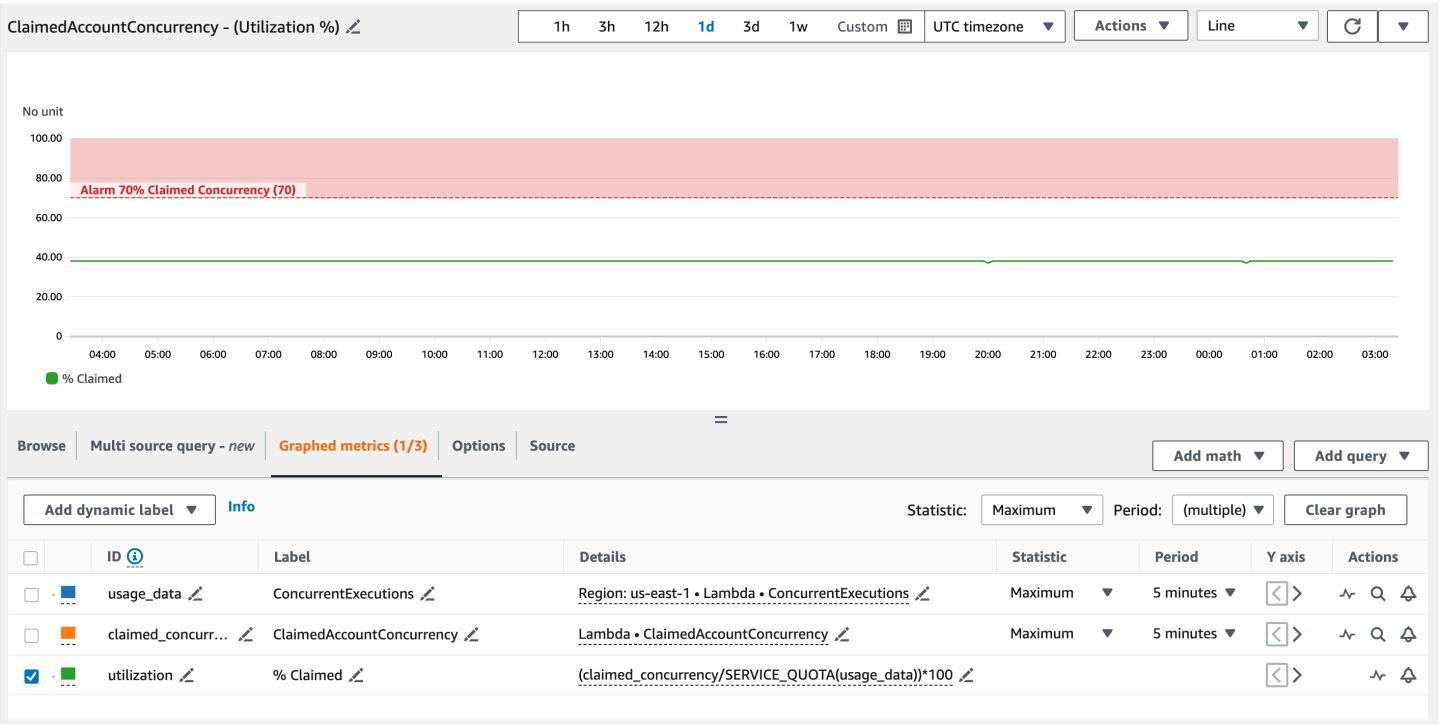
- At `t1`, `ClaimedAccountConcurrency` is 800 (800 + 0 `UnreservedConcurrentExecutions`).
- At `t2`, `ClaimedAccountConcurrency` is 900 (800 + 100 `UnreservedConcurrentExecutions`).
- At `t3`, `ClaimedAccountConcurrency` is again 900 (800 + 100 `UnreservedConcurrentExecutions`).

Setting up the `ClaimedAccountConcurrency` metric in CloudWatch

Lambda emits the `ClaimedAccountConcurrency` metric in CloudWatch. Use this metric along with the value of `SERVICE_QUOTA(ConcurrentExecutions)` to get the percent utilization of concurrency in your account, as shown in the following formula:

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

The following screenshot illustrates how you can graph this formula in CloudWatch. The green `claim_utilization` line represents the concurrency utilization in this account, which is at around 40%:



The previous screenshot also includes a CloudWatch alarm that goes into `ALARM` state when the concurrency utilization exceeds 70%. You can use the `ClaimedAccountConcurrency` metric along with similar alarms to proactively determine when you might need to request a higher account concurrency limit.

Building Lambda functions with Node.js

You can run JavaScript code with Node.js in AWS Lambda. Lambda provides runtimes for Node.js that run your code to process events. Your code runs in an environment that includes the AWS SDK for JavaScript, with credentials from an AWS Identity and Access Management (IAM) role that you manage. To learn more about the SDK versions included with the Node.js runtimes, see the section called "Runtime-included SDK versions".

Lambda supports the following Node.js runtimes.

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Node.js 24	`nodejs24.x`	Amazon Linux 2023	Apr 30, 2028	Jun 1, 2028	Jul 1, 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	Apr 30, 2027	Jun 1, 2027	Jul 1, 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2023	Apr 30, 2026	Aug 31, 2026	Sep 30, 2026

To create a Node.js function

1. Open the Lambda console.
2. Choose **Create function**.
3. Configure the following settings:
   - **Function name**: Enter a name for the function.
   - **Runtime**: Choose **Node.js 24.x**.
4. Choose **Create function**.

The console creates a Lambda function with a single source file named `index.mjs`. You can edit this file and add more files in the built-in code editor. In the **DEPLOY** section, choose **Deploy** to

update your function's code. Then, to run your code, choose **Create test event** in the **TEST EVENTS** section.

The `index.mjs` file exports a function named `handler` that takes an event object and a context object. This is the handler function that Lambda calls when the function is invoked. The Node.js function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is `index.handler`.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to create a deployment package to upload your code to the Lambda function.

The function runtime passes a context object to the handler, in addition to the invocation event. The context object contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any logs that your function outputs during invocation. If your function returns an error, Lambda formats the error and returns it to the invoker.

Topics

- Runtime-included SDK versions
- Using keep-alive for TCP connections
- CA certificate loading
- Experimental Node.js features
- Define Lambda function handler in Node.js
- Deploy Node.js Lambda functions with .zip file archives
- Deploy Node.js Lambda functions with container images
- Working with layers for Node.js Lambda functions
- Using the Lambda context object to retrieve Node.js function information
- Log and monitor Node.js Lambda functions
- Instrumenting Node.js code in AWS Lambda

Runtime-included SDK versions

All supported Lambda Node.js runtimes include a specific minor version of the AWS SDK for JavaScript v3, not the latest version. The specific minor version that's included in the runtime depends on the runtime version and your AWS Region. To find the specific version of the SDK included in the runtime that you're using, create a Lambda function with the following code.

Example index.mjs

```
import packageJson from '@aws-sdk/client-s3/package.json' with { type: 'json' };

export const handler = async () => ({ version: packageJson.version });
```

This returns a response in the following format:

```
{
  "version": "3.632.0"
}
```

For more information, see Using the SDK for JavaScript v3 in your handler.

Using keep-alive for TCP connections

The default Node.js HTTP/HTTPS agent creates a new TCP connection for every new request. To avoid the cost of establishing new connections, keep-alive is enabled by default in all supported Node.js runtimes. Keep-alive can reduce request times for Lambda functions that make multiple API calls using the SDK.

To disable keep-alive, see Reusing connections with keep-alive in Node.js in the *AWS SDK for JavaScript 3.x Developer Guide*. For more information about using keep-alive, see HTTP keep-alive is on by default in modular AWS SDK for JavaScript on the AWS Developer Tools Blog.

CA certificate loading

For Node.js runtime versions up to Node.js 18, Lambda automatically loads Amazon-specific CA (certificate authority) certificates to make it easier for you to create functions that interact with other AWS services. For example, Lambda includes the Amazon RDS certificates necessary for

validating the server identity certificate installed on your Amazon RDS database. This behavior can have a performance impact during cold starts.

Starting with Node.js 20, Lambda no longer loads additional CA certificates by default. The Node.js 20 runtime contains a certificate file with all Amazon CA certificates located at `/var/runtime/ca-cert.pem`. To restore the same behavior from Node.js 18 and earlier runtimes, set the `NODE_EXTRA_CA_CERTS` environment variable to `/var/runtime/ca-cert.pem`.

For optimal performance, we recommend bundling only the certificates that you need with your deployment package and loading them via the `NODE_EXTRA_CA_CERTS` environment variable. The certificates file should consist of one or more trusted root or intermediate CA certificates in PEM format. For example, for RDS, include the required certificates alongside your code as `certificates/rds.pem`. Then, load the certificates by setting `NODE_EXTRA_CA_CERTS` to `/var/task/certificates/rds.pem`.

Experimental Node.js features

The upstream Node.js language releases enable some experimental features by default. Lambda disables these features to ensure runtime stability and consistent performance. The following table lists the experimental features that Lambda disables.

Experimental feature	Supported Node.js versions	Node.js flag applied by Lambda	Lambda flag to re-enable
Support for importing modules using require in ES modules	Node.js 20, Node.js 22	`--no-experimental-require-module`	`--experimental-require-module`
Support for automatically detecting ES vs CommonJS modules	Node.js 22	`--no-experimental-detect-module`	`--experimental-detect-module`

To enable a disabled experimental feature, set the re-enable flag in the `NODE_OPTIONS` environment variable. For example, to enable ES module require support, set `NODE_OPTIONS` to `--`

`experimental-require-module`. Lambda detects this override and removes the corresponding disable flag.

> **Important**
>
> Using experimental features can lead to instability and performance issues. These features might be changed or removed in future Node.js versions. Functions that use experimental features aren't eligible for the Lambda Service Level Agreement (SLA) or AWS Support.

Define Lambda function handler in Node.js

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

This page describes how to work with Lambda function handlers in Node.js, including options for project setup, naming conventions, and best practices. This page also includes an example of a Node.js Lambda function that takes in information about an order, produces a text file receipt, and puts this file in an Amazon Simple Storage Service (Amazon S3) bucket. For information about how to deploy your function after writing it, see the section called “Deploy .zip file archives” or the section called “Deploy container images”.

Topics

- Setting up your Node.js handler project
- Example Node.js Lambda function code
- CommonJS and ES Modules
- Node.js initialization
- Handler naming conventions
- Defining and accessing the input event object
- Valid handler patterns for Node.js functions
- Using the SDK for JavaScript v3 in your handler
- Accessing environment variables
- Using global state
- Code best practices for Node.js Lambda functions

Setting up your Node.js handler project

There are multiple ways to initialize a Node.js Lambda project. For example, you can create a standard Node.js project using `npm`, create an AWS SAM application, or create an AWS CDK application.

To create the project using `npm`:

```
npm init
```

This command initializes your project and generates a `package.json` file that manages your project's metadata and dependencies.

Your function code lives in a `.js` or `.mjs` JavaScript file. In the following example, we name this file `index.mjs` because it uses an ES module handler. Lambda supports both ES module and CommonJS handlers. For more information, see CommonJS and ES Modules.

A typical Node.js Lambda function project follows this general structure:

```
/project-root
  ### index.mjs – Contains main handler
  ### package.json – Project metadata and dependencies
  ### package-lock.json – Dependency lock file
  ### node_modules/ – Installed dependencies
```

Example Node.js Lambda function code

The following example Lambda function code takes in information about an order, produces a text file receipt, and puts this file in an Amazon S3 bucket.

Example index.mjs Lambda function

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

// Initialize the S3 client outside the handler for reuse
const s3Client = new S3Client();

/**
 * Lambda handler for processing orders and storing receipts in S3.
 * @param {Object} event - Input event containing order details
 * @param {string} event.order_id - The unique identifier for the order
 * @param {number} event.amount - The order amount
 * @param {string} event.item - The item purchased
 * @returns {Promise<string>} Success message
 */
export const handler = async(event) => {
    try {
        // Access environment variables
        const bucketName = process.env.RECEIPT_BUCKET;
        if (!bucketName) {
            throw new Error('RECEIPT_BUCKET environment variable is not set');
        }
```

```
        // Create the receipt content and key destination
        const receiptContent = `OrderID: ${event.order_id}\nAmount: $
${event.amount.toFixed(2)}\nItem: ${event.item}`;
        const key = `receipts/${event.order_id}.txt`;

        // Upload the receipt to S3
        await uploadReceiptToS3(bucketName, key, receiptContent);

        console.log(`Successfully processed order ${event.order_id} and stored receipt
 in S3 bucket ${bucketName}`);
        return 'Success';
    } catch (error) {
        console.error(`Failed to process order: ${error.message}`);
        throw error;
    }
};

/**
 * Helper function to upload receipt to S3
 * @param {string} bucketName - The S3 bucket name
 * @param {string} key - The S3 object key
 * @param {string} receiptContent - The content to upload
 * @returns {Promise<void>}
 */
async function uploadReceiptToS3(bucketName, key, receiptContent) {
    try {
        const command = new PutObjectCommand({
            Bucket: bucketName,
            Key: key,
            Body: receiptContent
        });

        await s3Client.send(command);
    } catch (error) {
        throw new Error(`Failed to upload receipt to S3: ${error.message}`);
    }
}
```

This `index.mjs` file contains the following sections of code:

- `import` block: Use this block to include libraries that your Lambda function requires, such as AWS SDK clients.

- `const s3Client` declaration: This initializes an Amazon S3 client outside of the handler function. This causes Lambda to run this code during the initialization phase, and the client is preserved for reuse across multiple invocations.
- JSDoc comment block: Define the input and output types for your handler using JSDoc annotations.
- `export const handler`: This is the main handler function that Lambda invokes. When deploying your function, specify `index.handler` for the Handler property. The value of the `Handler` property is the file name and the name of the exported handler method, separated by a dot.
- `uploadReceiptToS3` function: This is a helper function that's referenced by the main handler function.

For this function to work properly, its execution role must allow the `s3:PutObject` action. Also, ensure that you define the `RECEIPT_BUCKET` environment variable. After a successful invocation, the Amazon S3 bucket should contain a receipt file.

CommonJS and ES Modules

Node.js supports two module systems: CommonJS and ECMAScript modules (ES modules). Lambda recommends using ES modules as it supports top-level await, which enables asynchronous tasks to be completed during execution environment initialization.

Node.js treats files with a `.cjs` file name extension as CommonJS modules while a `.mjs` extension denotes ES modules. By default, Node.js treats files with the `.js` file name extension as CommonJS modules. You can configure Node.js to treat `.js` files as ES modules by specifying the `type` as `module` in the function's `package.json` file. You can configure Node.js in Lambda to detect automatically whether a `.js` file should be treated as CommonJS or as an ES module by adding the `–experimental-detect-module` flag to the `NODE_OPTIONS` environment variable. For more information, see Experimental Node.js features.

The following examples show function handlers written using both ES modules and CommonJS modules. The remaining examples on this page all use ES modules.

ES module example

Example– ES module handler

```
const url = "https://aws.amazon.com/";
```

```
export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

CommonJS module example

Example– CommonJS module handler

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = async function (event) {
  let statusCode;
  await new Promise(function (resolve, reject) {
    https.get(url, (res) => {
        statusCode = res.statusCode;
        resolve(statusCode);
      }).on("error", (e) => {
        reject(Error(e));
      });
  });
  console.log(statusCode);
  return statusCode;
};
```

Node.js initialization

Node.js uses a non-blocking I/O model that supports efficient asynchronous operations using an event loop. For example, if Node.js makes a network call, the function continues to process other operations without blocking on a network response. When the network response is received, it is placed into the callback queue. Tasks from the queue are processed when the current task completes.

Lambda recommends using top-level await so that asynchronous tasks initiated during execution environment initialization are completed during initialization. Asynchronous tasks that are not completed during initialization will typically run during the first function invoke. This can cause unexpected behavior or errors. For example, your function initialization may make a network call to fetch a parameter from AWS Parameter Store. If this task is not completed during initialization, the value may be null during an invocation. There can also be a delay between initialization and invoke which can trigger errors in time-sensitive operations. In particular, AWS service calls can rely on time-sensitive request signatures, resulting in service call failures if the call is not completed during the initialization phase. Completing tasks during initialization typically improves cold-start performance, and first invoke performance when using Provisioned Concurrency. For more information, see our blog post Using Node.js ES modules and top-level await in AWS Lambda.

Handler naming conventions

When you configure a function, the value of the Handler setting is the file name and the name of the exported handler method, separated by a dot. The default for functions created in the console and for examples in this guide is `index.handler`. This indicates the `handler` method that's exported from the `index.js` or `index.mjs` file.

If you create a function in the console using a different file name or function handler name, you must edit the default handler name.

To change the function handler name (console)

1. Open the Functions page of the Lambda console and choose your function.
2. Choose the **Code** tab.
3. Scroll down to the **Runtime settings** pane and choose **Edit**.
4. In **Handler**, enter the new name for your function handler.
5. Choose **Save**.

Defining and accessing the input event object

JSON is the most common and standard input format for Lambda functions. In this example, the function expects an input similar to the following:

```
{
    "order_id": "12345",
```

```
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

When working with Lambda functions in Node.js, you can define the expected shape of the input event using JSDoc annotations. In this example, we define the input structure in the handler's JSDoc comment:

```
/**
 * Lambda handler for processing orders and storing receipts in S3.
 * @param {Object} event - Input event containing order details
 * @param {string} event.order_id - The unique identifier for the order
 * @param {number} event.amount - The order amount
 * @param {string} event.item - The item purchased
 * @returns {Promise<string>} Success message
 */
```

After you define these types in your JSDoc comment, you can access the fields of the event object directly in your code. For example, `event.order_id` retrieves the value of `order_id` from the original input.

Valid handler patterns for Node.js functions

We recommend that you use async/await to declare the function handler instead of using callbacks. Async/await is a concise and readable way to write asynchronous code, without the need for nested callbacks or chaining promises. With async/await, you can write code that reads like synchronous code, while still being asynchronous and non-blocking.

async function handlers (recommended)

The `async` keyword marks a function as asynchronous, and the `await` keyword pauses the execution of the function until a `Promise` is resolved. The handler accepts the following arguments:

- `event`: Contains the input data passed to your function.
- `context`: Contains information about the invocation, function, and execution environment. For more information, see Using the Lambda context object to retrieve Node.js function information.

Here are the valid signatures for the async/await pattern:

```
export const handler = async (event) => { };
```

```
export const handler = async (event, context) => { };
```

Synchronous function handlers

Where your function does not perform any asynchronous tasks, you can use a synchronous function handler, using one of the following function signatures:

```
export const handler = (event) => { };
```

```
export const handler = (event, context) => { };
```

Response streaming function handlers

Lambda supports response streaming with Node.js. Response streaming function handlers use the `awslambda.streamifyResponse()` decorator and take 3 parameters: `event`, `responseStream`, and `context`. The function signature is:

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 context) => { });
```

For more information, see Response streaming for Lambda functions.

Callback-based function handlers

> **Note**
>
> Callback-based function handlers are only supported up to Node.js 22. Starting from Node.js 24, asynchronous tasks should be implemented using async function handlers.

Callback-based function handlers must use the event, context, and callback arguments. Example:

```
export const handler = (event, context, callback) => { };
```

The callback function expects an `Error` and a response, which must be JSON-serializable. The function continues to execute until the event loop is empty or the function times out. The response

isn't sent to the invoker until all event loop tasks are finished. If the function times out, an error is returned instead. You can configure the runtime to send the response immediately by setting context.callbackWaitsForEmptyEventLoop to false.

Example– HTTP request with callback

The following example function checks a URL and returns the status code to the invoker.

```
import https from "https";
let url = "https://aws.amazon.com/";

export const handler = (event, context, callback) => {
  https.get(url, (res) => {
    callback(null, res.statusCode);
  }).on("error", (e) => {
    callback(Error(e));
  });
};
```

Using the SDK for JavaScript v3 in your handler

Often, you'll use Lambda functions to interact with or make updates to other AWS resources. The simplest way to interface with these resources is to use the AWS SDK for JavaScript. All supported Lambda Node.js runtimes include the SDK for JavaScript version 3. However, we strongly recommend that you include the AWS SDK clients that you need in your deployment package. This maximizes backward compatibility during future Lambda runtime updates. Only rely on the runtime-provided SDK when you can't include additional packages (for example, when using the Lambda console code editor or inline code in an AWS CloudFormation template).

To add SDK dependencies to your function, use the `npm install` command for the specific SDK clients that you need. In the example code, we used the Amazon S3 client. Add this dependency by running the following command in the directory that contains your `package.json` file:

```
npm install @aws-sdk/client-s3
```

In the function code, import the client and commands that you need, as the example function demonstrates:

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

Then, initialize an Amazon S3 client:

```
const s3Client = new S3Client();
```

In this example, we initialized our Amazon S3 client outside of the main handler function to avoid having to initialize it every time we invoke our function. After you initialize your SDK client, you can then use it to make API calls for that AWS service. The example code calls the Amazon S3 PutObject API action as follows:

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: receiptContent
});
```

Accessing environment variables

In your handler code, you can reference any environment variables by using `process.env`. In this example, we reference the defined `RECEIPT_BUCKET` environment variable using the following lines of code:

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

Using global state

Lambda runs your static code during the initialization phase before invoking your function for the first time. Resources created during initialization stay in memory between invocations, so you can avoid having to create them every time you invoke your function.

In the example code, the S3 client initialization code is outside the handler. The runtime initializes the client before the function handles its first event, and the client remains available for reuse across all invocations.

Code best practices for Node.js Lambda functions

Follow these guidelines when building Lambda functions:

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function.
- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries. For the Node.js and Python runtimes, these include the AWS SDKs. To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation.

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of function invocations and escalated costs. If you see an unintended volume of invocations, set the

function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

Deploy Node.js Lambda functions with .zip file archives

Your AWS Lambda function's code comprises a .js or .mjs file containing your function's handler code, together with any additional packages and modules your code depends on. To deploy this function code to Lambda, you use a *deployment package*. This package may either be a .zip file archive or a container image. For more information about using container images with Node.js, see Deploy Node.js Lambda functions with container images.

To create your deployment package as .zip file archive, you can use your command-line tool's built-in .zip file archive utility, or any other .zip file utility such as 7zip. The examples shown in the following sections assume you're using a command-line `zip` tool in a Linux or MacOS environment. To use the same commands in Windows, you can install the Windows Subsystem for Linux to get a Windows-integrated version of Ubuntu and Bash.

Note that Lambda uses POSIX file permissions, so you may need to set permissions for the deployment package folder before you create the .zip file archive.

Topics

- Runtime dependencies in Node.js
- Creating a .zip deployment package with no dependencies
- Creating a .zip deployment package with dependencies
- Creating a Node.js layer for your dependencies
- Dependency search path and runtime-included libraries
- Creating and updating Node.js Lambda functions using .zip files

Runtime dependencies in Node.js

For Lambda functions that use the Node.js runtime, a dependency can be any Node.js module. The Node.js runtime includes a number of common libraries, as well as a version of the AWS SDK for JavaScript. All supported Node.js runtimes include version 3 of the SDK. To use version 2 of the SDK, you must add the SDK to your .zip file deployment package. To find the specific version of the SDK that's included in the runtime that you're using, see the section called "Runtime-included SDK versions".

Lambda periodically updates the SDK libraries in the Node.js runtime to include the latest features and security upgrades. Lambda also applies security patches and updates to the other libraries

included in the runtime. To have full control of the dependencies in your package, you can add your preferred version of any runtime-included dependency to your deployment package. For example, if you want to use a particular version of the SDK for JavaScript, you can include it in your .zip file as a dependency. For more information on adding runtime-included dependencies to your .zip file, see Dependency search path and runtime-included libraries.

Under the AWS shared responsibility model, you are responsible for the management of any dependencies in your functions' deployment packages. This includes applying updates and security patches. To update dependencies in your function's deployment package, first create a new .zip file and then upload it to Lambda. See Creating a .zip deployment package with dependencies and Creating and updating Node.js Lambda functions using .zip files for more information.

Creating a .zip deployment package with no dependencies

If your function code has no dependencies except for libraries included in the Lambda runtime, your .zip file contains only the `index.js` or `index.mjs` file with your function's handler code. Use your preferred zip utility to create a .zip file with your `index.js` or `index.mjs` file at the root. If the file containing your handler code isn't at the root of your .zip file, Lambda won't be able to run your code.

To learn how to deploy your .zip file to create a new Lambda function or update an existing one, see Creating and updating Node.js Lambda functions using .zip files.

Creating a .zip deployment package with dependencies

If your function code depends on packages or modules that aren't included in the Lambda Node.js runtime, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. The instructions in this section show you how to include your dependencies in your .zip deployment package. For instructions on how to include your dependencies in a layer, see the section called "Creating a Node.js layer for your dependencies".

The following example CLI commands create a .zip file named `my_deployment_package.zip` containing the `index.js` or `index.mjs` file with your function's handler code and its dependencies. In the example, you install dependencies using the npm package manager.

To create the deployment package

1. Navigate to the project directory containing your `index.js` or `index.mjs` source code file. In this example, the directory is named `my_function`.

```
cd my_function
```

2. Install your function's required libraries in the `node_modules` directory using the `npm install` command. In this example you install the AWS X-Ray SDK for Node.js.

   ```
   npm install aws-xray-sdk
   ```

   This creates a folder structure similar to the following:

   ```
   ~/my_function
   ### index.mjs
   ### node_modules
       ### async
       ### async-listener
       ### atomic-batcher
       ### aws-sdk
       ### aws-xray-sdk
       ### aws-xray-sdk-core
   ```

   You can also add custom modules that you create yourself to your deployment package. Create a directory under `node_modules` with the name of your module and save your custom written packages there.

3. Create a .zip file that contains the contents of your project folder at the root. Use the `r` (recursive) option to ensure that zip compresses the subfolders.

   ```
   zip -r my_deployment_package.zip .
   ```

Creating a Node.js layer for your dependencies

The instructions in this section show you how to include your dependencies in a layer. For instructions on how to include your dependencies in your deployment package, see the section called “Creating a .zip deployment package with dependencies”.

When you add a layer to a function, Lambda loads the layer content into the `/opt` directory of that execution environment. For each Lambda runtime, the PATH variable already includes specific folder paths within the `/opt` directory. To ensure that Lambda picks up your layer content, your layer .zip file should have its dependencies in one of the following folder paths:

- `nodejs/node_modules`
- `nodejs/node18/node_modules (NODE_PATH)`
- `nodejs/node20/node_modules (NODE_PATH)`
- `nodejs/node22/node_modules (NODE_PATH)`

For example, your layer .zip file structure might look like the following:

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

In addition, Lambda automatically detects any libraries in the `/opt/lib` directory, and any binaries in the `/opt/bin` directory. To ensure that Lambda properly finds your layer content, you can also create a layer with the following structure:

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

After you package your layer, see the section called "Creating and deleting layers" and the section called "Adding layers" to complete your layer setup.

Dependency search path and runtime-included libraries

The Node.js runtime includes a number of common libraries, as well as a version of the AWS SDK for JavaScript. If you want to use a different version of a runtime-included library, you can do this by bundling it with your function or by adding it as a dependency in your deployment package. For example, you can use a different version of the SDK by adding it to your .zip deployment package. You can also include it in a Lambda layer for your function.

When you use an `import` or `require` statement in your code, the Node.js runtime searches the directories in the `NODE_PATH` path until it finds the module. By default, the first location the runtime searches is the directory into which your .zip deployment package is decompressed and mounted (`/var/task`). If you include a version of a runtime-included library in your deployment

package, this version will take precedence over the version included in the runtime. Dependencies in your deployment package also have precedence over dependencies in layers.

When you add a dependency to a layer, Lambda extracts this to `/opt/nodejs/nodexx/node_modules` where `nodexx` represents the version of the runtime you are using. In the search path, this directory has precedence over the directory containing the runtime-included libraries (`/var/lang/lib/node_modules`). Libraries in function layers therefore have precedence over versions included in the runtime.

You can see the full search path for your Lambda function by adding the following line of code.

```
console.log(process.env.NODE_PATH)
```

You can also add dependencies in a separate folder inside your .zip package. For example, you might add a custom module to a folder in your .zip package called `common`. When your .zip package is decompressed and mounted, this folder is placed inside the `/var/task` directory. To use a dependency from a folder in your .zip deployment package in your code, use an `import { } from` or `const { } = require()` statement, depending on whether you are using CJS or ESM module resolution. For example:

```
import { myModule } from './common'
```

If you bundle your code with `esbuild`, `rollup`, or similar, the dependencies used by your function are bundled together in one or more files. We recommend using this method to vend dependencies whenever possible. Compared to adding dependencies to your deployment package, bundling your code results in improved performance due to the reduction in I/O operations.

Creating and updating Node.js Lambda functions using .zip files

After you have created your .zip deployment package, you can use it to create a new Lambda function or update an existing one. You can deploy your .zip package using the Lambda console, the AWS Command Line Interface, and the Lambda API. You can also create and update Lambda functions using AWS Serverless Application Model (AWS SAM) and CloudFormation.

The maximum size for a .zip deployment package for Lambda is 250 MB (unzipped). Note that this limit applies to the combined size of all the files you upload, including any Lambda layers.

The Lambda runtime needs permission to read the files in your deployment package. In Linux permissions octal notation, Lambda needs 644 permissions for non-executable files (rw-r--r--) and 755 permissions (rwxr-xr-x) for directories and executable files.

In Linux and MacOS, use the `chmod` command to change file permissions on files and directories in your deployment package. For example, to give a non-executable file the correct permissions, run the following command.

```
chmod 644 <filepath>
```

To change file permissions in Windows, see Set, View, Change, or Remove Permissions on an Object in the Microsoft Windows documentation.

> **Note**
>
> If you don't grant Lambda the permissions it needs to access directories in your deployment package, Lambda sets the permissions for those directories to 755 (rwxr-xr-x).

Creating and updating functions with .zip files using the console

To create a new function, you must first create the function in the console, then upload your .zip archive. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip file.

If your .zip file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the AWS Management Console, see Getting started with Amazon S3. To upload files using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> You cannot change the deployment package type (.zip or container image) for an existing function. For example, you cannot convert a container image function to use a .zip file archive. You must create a new function.

To create a new function (console)

1. Open the Functions page of the Lambda console and choose **Create Function**.
2. Choose **Author from scratch**.
3. Under **Basic information**, do the following:
    a. For **Function name**, enter the name for your function.
    b. For **Runtime**, select the runtime you want to use.
    c. (Optional) For **Architecture**, choose the instruction set architecture for your function. The default architecture is x86_64. Ensure that the .zip deployment package for your function is compatible with the instruction set architecture you select.
4. (Optional) Under **Permissions**, expand **Change default execution role**. You can create a new **Execution role** or use an existing one.
5. Choose **Create function**. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **.zip file**.
5. To upload the .zip file, do the following:
    a. Select **Upload**, then select your .zip file in the file chooser.
    b. Choose **Open**.
    c. Choose **Save**.

To upload a .zip archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.

4. Choose **Amazon S3 location**.
5. Paste the Amazon S3 link URL of your .zip file and choose **Save**.

Updating .zip file functions using the console code editor

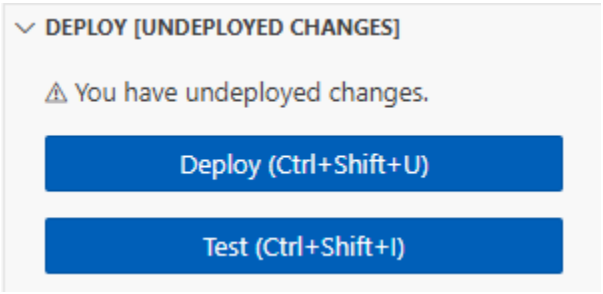
For some functions with .zip deployment packages, you can use the Lambda console's built-in code editor to update your function code directly. To use this feature, your function must meet the following criteria:

- Your function must use one of the interpreted language runtimes (Python, Node.js, or Ruby)
- Your function's deployment package must be smaller than 50 MB (unzipped).

Function code for functions with container image deployment packages cannot be edited directly in the console.

To update function code using the console code editor

1. Open the Functions page of the Lambda console and select your function.
2. Select the **Code** tab.
3. In the **Code source** pane, select your source code file and edit it in the integrated code editor.
4. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Creating and updating functions with .zip files using the AWS CLI

You can can use the AWS CLI to create a new function or to update an existing one using a .zip file. Use the create-function and update-function-code commands to deploy your .zip package. If your .zip file is smaller than 50MB, you can upload the .zip package from a file location on your local build machine. For larger files, you must upload your .zip package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> If you upload your .zip file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip file with the AWS CLI, you must specify the following:

- The name of your function (`--function-name`)
- Your function's runtime (`--runtime`)
- The Amazon Resource Name (ARN) of your function's execution role (`--role`)
- The name of the handler method in your function code (`--handler`)

You must also specify the location of your .zip file. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction \
--runtime nodejs24.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--code` option as shown in the following example command. You only need to use the `S3ObjectVersion` parameter for versioned objects.

```
aws lambda create-function --function-name myFunction \
--runtime nodejs24.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the `--function-name` parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda update-function-code --function-name myFunction \
```

```
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--s3-bucket` and `--s3-key` options as shown in the following example command. You only need to use the `--s3-object-version` parameter for versioned objects.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Creating and updating functions with .zip files using the Lambda API

To create and update functions using a .zip file archive, use the following API operations:

- CreateFunction
- UpdateFunctionCode

Creating and updating functions with .zip files using AWS SAM

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build, package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. To learn more about using AWS SAM to build and deploy Lambda functions, see Getting started with AWS SAM in the *AWS Serverless Application Model Developer Guide*.

You can also use AWS SAM to create a Lambda function using an existing .zip file archive. To create a Lambda function using AWS SAM, you can save your .zip file in an Amazon S3 bucket or in a local folder on your build machine. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

In your AWS SAM template, the `AWS::Serverless::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- `PackageType` - set to `Zip`
- `CodeUri` - set to the function code's Amazon S3 URI, path to local folder, or FunctionCode object
- `Runtime` - Set to your chosen runtime

With AWS SAM, if your .zip file is larger than 50MB, you don't need to upload it to an Amazon S3 bucket first. AWS SAM can upload .zip packages up to the maximum allowed size of 250MB (unzipped) from a location on your local build machine.

To learn more about deploying functions using .zip file in AWS SAM, see AWS::Serverless::Function in the *AWS SAM Developer Guide*.

Creating and updating functions with .zip files using CloudFormation

You can use CloudFormation to create a Lambda function using a .zip file archive. To create a Lambda function from a .zip file, you must first upload your file to an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide.*

In your CloudFormation template, the `AWS::Lambda::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- `PackageType` - Set to `Zip`
- `Code` - Enter the Amazon S3 bucket name and the .zip file name in the `S3Bucket` and `S3Key` fields
- `Runtime` - Set to your chosen runtime

The .zip file that CloudFormation generates cannot exceed 4MB. To learn more about deploying functions using .zip file in CloudFormation, see AWS::Lambda::Function in the *CloudFormation User Guide*.

Deploy Node.js Lambda functions with container images

There are three ways to build a container image for a Node.js Lambda function:

- Using an AWS base image for Node.js

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- Using an AWS OS-only base image

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the runtime interface client for Node.js in the image.

- Using a non-AWS base image

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the runtime interface client for Node.js in the image.

> **Tip**
>
> To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

This page explains how to build, test, and deploy container images for Lambda.

Topics

- AWS base images for Node.js
- Using an AWS base image for Node.js
- Using an alternative base image with the runtime interface client

AWS base images for Node.js

AWS provides the following base images for Node.js:

Tags	Runtime	Operating system	Dockerfile	Deprecation
24	Node.js 24	Amazon Linux 2023	Dockerfile for Node.js 24 on GitHub	Apr 30, 2028
22	Node.js 22	Amazon Linux 2023	Dockerfile for Node.js 22 on GitHub	Apr 30, 2027
20	Node.js 20	Amazon Linux 2023	Dockerfile for Node.js 20 on GitHub	Apr 30, 2026

Amazon ECR repository: gallery.ecr.aws/lambda/nodejs

The Node.js 20 and later base images are based on the Amazon Linux 2023 minimal container image. Earlier base images use Amazon Linux 2. AL2023 provides several advantages over Amazon Linux 2, including a smaller deployment footprint and updated versions of libraries such as `glibc`.

AL2023-based images use `microdnf` (symlinked as `dnf`) as the package manager instead of `yum`, which is the default package manager in Amazon Linux 2. `microdnf` is a standalone implementation of `dnf`. For a list of packages that are included in AL2023-based images, refer to the **Minimal Container** columns in Comparing packages installed on Amazon Linux 2023 Container Images. For more information about the differences between AL2023 and Amazon Linux 2, see Introducing the Amazon Linux 2023 runtime for AWS Lambda on the AWS Compute Blog.

> **Note**
>
> To run AL2023-based images locally, including with AWS Serverless Application Model (AWS SAM), you must use Docker version 20.10.10 or later.

Using an AWS base image for Node.js

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.
- Node.js

Creating an image from a base image

To create a container image from an AWS base image for Node.js

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
cd example
```

2. Create a new Node.js project with npm. To accept the default options provided in the interactive experience, press `Enter`.

```
npm init
```

3. Create a new file called `index.js`. You can add the following sample function code to the file for testing, or use your own.

Example CommonJS handler

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. If your function depends on libraries other than the AWS SDK for JavaScript, use npm to add them to your package.

5. Create a new Dockerfile with the following configuration:

   - Set the FROM property to the URI of the base image.
   - Use the COPY command to copy the function code and runtime dependencies to {LAMBDA_TASK_ROOT}, a Lambda-defined environment variable.
   - Set the CMD argument to the Lambda function handler.

   Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the root user when no USER instruction is provided.

   Example Dockerfile

   ```
   FROM public.ecr.aws/lambda/nodejs:22

   # Copy function code
   COPY index.js ${LAMBDA_TASK_ROOT}

   # Set the CMD to your handler (could also be done as a parameter override outside
    of the Dockerfile)
   CMD [ "index.handler" ]
   ```

6. Build the Docker image with the docker build command. The following example names the image docker-image and gives it the test tag. To make your image compatible with Lambda, you must use the --provenance=false option.

   ```
   docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
    .
   ```

   Note

   The command specifies the --platform linux/amd64 option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the --platform linux/arm64 option instead.

(Optional) Test the image locally

1. Start the Docker image with the **docker run** command. In this example, `docker-image` is the image name and `test` is the tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.

   Linux/macOS

   In Linux and macOS, run the following `curl` command:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   In PowerShell, run the following `Invoke-WebRequest` command:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Get the container ID.

```
docker ps
```

4. Use the docker kill command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Using an alternative base image with the runtime interface client

If you use an OS-only base image or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the Runtime API, which manages the interaction between Lambda and your function code.

Install the Node.js runtime interface client using the npm package manager:

```
npm install aws-lambda-ric
```

You can also download the Node.js runtime interface client from GitHub.

The following example demonstrates how to build a container image for Node.js using a non-AWS base image. The example Dockerfile uses a `bookworm` base image. The Dockerfile includes the runtime interface client.

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.

- Node.js

Creating an image from an alternative base image

To create a container image from a non-AWS base image

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
cd example
```

2. Create a new Node.js project with npm. To accept the default options provided in the interactive experience, press Enter.

```
npm init
```

3. Create a new file called index.js. You can add the following sample function code to the file for testing, or use your own.

Example CommonJS handler

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. Create a new Dockerfile. The following Dockerfile uses a bookworm base image instead of an AWS base image. The Dockerfile includes the runtime interface client, which makes the image compatible with Lambda. The Dockerfile uses a multi-stage build. The first stage creates a build image, which is a standard Node.js environment where the function's dependencies are installed. The second stage creates a slimmer image which includes the function code and its dependencies. This reduces the final image size.

   - Set the FROM property to the base image identifier.
   - Use the COPY command to copy the function code and runtime dependencies.
   - Set the ENTRYPOINT to the module that you want the Docker container to run when it starts. In this case, the module is the runtime interface client.

- Set the CMD argument to the Lambda function handler.

Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM node:20-bookworm as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Install build dependencies
RUN apt-get update && \
    apt-get install -y \
    g++ \
    make \
    cmake \
    unzip \
    libcurl4-openssl-dev

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

WORKDIR ${FUNCTION_DIR}

# Install Node.js dependencies
RUN npm install

# Install the runtime interface client
RUN npm install aws-lambda-ric

# Grab a fresh slim copy of the image to reduce the final size
FROM node:20-bookworm-slim

# Required for Node runtimes which use npm@8.6.0+ because
```

```
# by default npm writes logs under /home/.npm and Lambda fs is read-only
ENV NPM_CONFIG_CACHE=/tmp/.npm

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
# Pass the name of the function handler as an argument to the runtime
CMD ["index.handler"]
```

5. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test the image. You can build the emulator into your image or use the following procedure to install it on your local machine.

To install and run the runtime interface emulator on your local machine

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   To install the arm64 emulator, replace the `$downloadLink` with the following:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. Start the Docker image with the **docker run** command. Note the following:

   - `docker-image` is the image name and `test` is the tag.
   - `/usr/local/bin/npx aws-lambda-ric index.handler` is the `ENTRYPOINT` followed by the `CMD` from your Dockerfile.

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/npx aws-lambda-ric index.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/npx aws-lambda-ric index.handler
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

Linux/macOS

In Linux and macOS, run the following `curl` command:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

In PowerShell, run the following `Invoke-WebRequest` command:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Get the container ID.

```
docker ps
```

5. Use the docker kill command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

- `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
- Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Working with layers for Node.js Lambda functions

Use Lambda layers to package code and dependencies that you want to reuse across multiple functions. Layers usually contain library dependencies, a custom runtime, or configuration files. Creating a layer involves three general steps:

1. Package your layer content. This means creating a .zip file archive that contains the dependencies you want to use in your functions.
2. Create the layer in Lambda.
3. Add the layer to your functions.

Topics

- Package your layer content
- Create the layer in Lambda
- Add the layer to your function
- Sample app

Package your layer content

To create a layer, bundle your packages into a .zip file archive that meets the following requirements:

- Build the layer using the same Node.js version that you plan to use for the Lambda function. For example, if you build your layer using Node.js 24, use the Node.js 24 runtime for your function.
- Your layer's .zip file must use one of these directory structures:
  - `nodejs/node_modules`
  - `nodejs/node`*X*`/node_modules` (where *X* is your Node.js version, for example `node22`)

  For more information, see Layer paths for each Lambda runtime.
- The packages in your layer must be compatible with Linux. Lambda functions run on Amazon Linux.

You can create layers that contain either third-party Node.js libraries installed with `npm` (such as `axios` or `lodash`) or your own JavaScript modules.

Third-party dependencies

To create a layer using npm packages

1. Create the required directory structure and install packages directly into it:

```
mkdir -p nodejs
npm install --prefix nodejs lodash axios
```

This command installs the packages directly into the `nodejs/node_modules` directory, which is the structure that Lambda requires.

> **Note**
>
> For packages with native dependencies or binary components (such as sharp or bcrypt), ensure that they're compatible with the Lambda Linux environment and your function's architecture. You might need to use the `--platform` flag:
>
> ```
> npm install --prefix nodejs --platform=linux --arch=x64 sharp
> ```
>
> For more complex native dependencies, you might need to compile them in a Linux environment that matches the Lambda runtime. You can use Docker for this purpose.

2. Zip the layer content:

Linux/macOS

```
zip -r layer.zip nodejs/
```

PowerShell

```
Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
```

The directory structure of your .zip file should look like this:

```
nodejs/
### package.json
### package-lock.json
```

```
### node_modules/
    ### lodash/
    ### axios/
    ### (dependencies of the other packages)
```

> **Note**
>
> - Make sure your .zip file includes the `nodejs` directory at the root level with `node_modules` inside it. This structure ensures that Lambda can locate and import your packages.
> - The `package.json` and `package-lock.json` files in the `nodejs/` directory are used by npm for dependency management but are not required by Lambda for layer functionality. Each installed package already contains its own `package.json` file that defines how Lambda imports the package.

Custom JavaScript modules

To create a layer using your own code

1. Create the required directory structure for your layer:

```
mkdir -p nodejs/node_modules/validator
cd nodejs/node_modules/validator
```

2. Create a `package.json` file for your custom module to define how it should be imported:

   Example nodejs/node_modules/validator/package.json

```
{
  "name": "validator",
  "version": "1.0.0",
  "type": "module",
  "main": "index.mjs"
}
```

3. Create your JavaScript module file:

Example nodejs/node_modules/validator/index.mjs

```
export function validateOrder(orderData) {
  // Validates an order and returns formatted data
  const requiredFields = ['productId', 'quantity'];

  // Check required fields
  const missingFields = requiredFields.filter(field => !(field in orderData));
  if (missingFields.length > 0) {
    throw new Error(`Missing required fields: ${missingFields.join(', ')}`);
  }

  // Validate quantity
  const quantity = orderData.quantity;
  if (!Number.isInteger(quantity) || quantity < 1) {
    throw new Error('Quantity must be a positive integer');
  }

  // Format and return the validated data
  return {
    productId: String(orderData.productId),
    quantity: quantity,
    shippingPriority: orderData.priority || 'standard'
  };
}

export function formatResponse(statusCode, body) {
  // Formats the API response
  return {
    statusCode: statusCode,
    body: JSON.stringify(body)
  };
}
```

4. Zip the layer content:

   Linux/macOS

   ```
   zip -r layer.zip nodejs/
   ```

PowerShell

```
Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
```

The directory structure of your .zip file should look like this:

```
nodejs/
### node_modules/
    ### validator/
        ### package.json
        ### index.mjs
```

5. In your function, import and use the modules. Example:

```
import { validateOrder, formatResponse } from 'validator';

export const handler = async (event) => {
  try {
    // Parse the order data from the event body
    const orderData = JSON.parse(event.body || '{}');

    // Validate and format the order
    const validatedOrder = validateOrder(orderData);

    return formatResponse(200, {
      message: 'Order validated successfully',
      order: validatedOrder
    });
  } catch (error) {
    if (error instanceof Error && error.message.includes('Missing required
 fields')) {
      return formatResponse(400, {
        error: error.message
      });
    }

    return formatResponse(500, {
      error: 'Internal server error'
    });
  }
```

```
};
```

You can use the following test event to invoke the function:

```
{
    "body": "{\"productId\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Expected response:

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"productId\":
\"ABC123\",\"quantity\":2,\"shippingPriority\":\"express\"}}"
}
```

Create the layer in Lambda

You can publish your layer using either the AWS CLI or the Lambda console.

AWS CLI

Run the publish-layer-version AWS CLI command to create the Lambda layer:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes nodejs24.x
```

The compatible runtimes parameter is optional. When specified, Lambda uses this parameter to filter layers in the Lambda console.

Console

To create a layer (console)

1. Open the Layers page of the Lambda console.
2. Choose **Create layer**.
3. Choose **Upload a .zip file**, and then upload the .zip archive that you created earlier.

4. (Optional) For **Compatible runtimes**, choose the Node.js runtime that corresponds to the Node.js version you used to build your layer.
5. Choose **Create**.

Add the layer to your function

AWS CLI

To attach the layer to your function, run the update-function-configuration AWS CLI command. For the `--layers` parameter, use the layer ARN. The ARN must specify the version (for example, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). For more information, see Layers and layer versions.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

Console

To add a layer to a function

1. Open the Functions page of the Lambda console.
2. Choose the function.
3. Scroll down to the **Layers** section, and then choose **Add a layer**.
4. Under **Choose a layer**, select **Custom layers**, and then choose your layer.

> **Note**
>
> If you didn't add a compatible runtime when you created the layer, your layer won't be listed here. You can specify the layer ARN instead.

5. Choose **Add**.

Sample app

For more examples of how to use Lambda layers, see the layer-nodejs sample application in the AWS Lambda Developer Guide GitHub repository. This application includes a layer that contains the lodash library. After creating the layer, you can deploy and invoke the corresponding function to confirm that the layer works as expected.

Using the Lambda context object to retrieve Node.js function information

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.

Context properties

- `functionName` – The name of the Lambda function.
- `functionVersion` – The version of the function.
- `invokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memoryLimitInMB` – The amount of memory that's allocated for the function.
- `awsRequestId` – The identifier of the invocation request.
- `logGroupName` – The log group for the function.
- `logStreamName` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
  - `cognitoIdentityId` – The authenticated Amazon Cognito identity.
  - `cognitoIdentityPoolId` – The Amazon Cognito identity pool that authorized the invocation.
- `clientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`

  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`
  - `env.make`
  - `env.model`
  - `env.locale`
  - `custom` – Custom values that are set by the client application.
- `callbackWaitsForEmptyEventLoop` – By default (`true`), when using a callback-based function handler, Lambda waits for the event loop to be empty after the callback runs before ending the function invoke. Set to `false` to send the response and end the invoke immediately after the callback runs instead of waiting for the event loop to be empty. Outstanding events continue to run during the next invocation. Note that Lambda supports callback-based function handlers for Node.js 22 and earlier runtimes only.

The following example function logs context information and returns the location of the logs.

Example index.js file

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

Log and monitor Node.js Lambda functions

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see Sending Lambda function logs to CloudWatch Logs.

This page describes how to produce log output from your Lambda function's code, and access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- Creating a function that returns logs
- Using Lambda advanced logging controls with Node.js
- Viewing logs in the Lambda console
- Viewing logs in the CloudWatch console
- Viewing logs using the AWS Command Line Interface (AWS CLI)
- Deleting logs

Creating a function that returns logs

To output logs from your function code, you can use methods on the console object, or any logging library that writes to `stdout` or `stderr`. The following example logs the values of environment variables and the event object.

> **Note**
>
> We recommend that you use techniques such as input validation and output encoding when logging inputs. If you log input data directly, an attacker might be able to use your code to make tampering hard to detect, forge log entries, or bypass log monitors. For more information, see Improper Output Neutralization for Logs in the *Common Weakness Enumeration*.

Example index.js file – Logging

```
exports.handler = async function(event, context) {
```

```
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.info("EVENT\n" + JSON.stringify(event, null, 2))
  console.warn("Event not processed.")
  return context.logStreamName
}
```

Example log format

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
2019-06-07T19:11:20.562Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO ENVIRONMENT
 VARIABLES
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/my-function",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2019/06/07/[$LATEST]e6f4a0c4241adcd70c262d34c0bbc85c",
  "AWS_EXECUTION_ENV": "AWS_Lambda_nodejs12.x",
  "AWS_LAMBDA_FUNCTION_NAME": "my-function",
  "PATH": "/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin",
  "NODE_PATH": "/opt/nodejs/node10/node_modules:/opt/nodejs/node_modules:/var/runtime/
node_modules",
  ...
}
2019-06-07T19:11:20.563Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO EVENT
{
  "key": "value"
}
2019-06-07T19:11:20.564Z c793869b-ee49-115b-a5b6-4fd21e8dedac WARN Event not processed.
END RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac
REPORT RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Duration: 128.83 ms Billed
 Duration: 296 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 166.62 ms
 XRAY TraceId: 1-5d9d007f-0a8c7fd02xmpl480aed55ef0 SegmentId: 3d752xmpl1bbe37e Sampled:
 true
```

The Node.js runtime logs the `START`, `END`, and `REPORT` lines for each invocation. It adds a timestamp, request ID, and log level to each entry logged by the function. The report line provides the following details.

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.

- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function. When invocations share an execution environment, Lambda reports the maximum memory used across all invocations. This behavior might result in a higher than expected reported value.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

You can view logs in the Lambda console, in the CloudWatch Logs console, or from the command line.

Using Lambda advanced logging controls with Node.js

To give you more control over how your functions' logs are captured, processed, and consumed, you can configure the following logging options for supported Node.js runtimes:

- **Log format** - select between plain text and structured JSON format for your function's logs
- **Log level** - for logs in JSON format, choose the detail level of the logs Lambda sends to Amazon CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

For more information about these logging options, and instructions on how to configure your function to use them, see the section called "Configuring advanced logging controls for Lambda functions".

To use the log format and log level options with your Node.js Lambda functions, see the guidance in the following sections.

Using structured JSON logs with Node.js

If you select JSON for your function's log format, Lambda will send logs output using the console methods of `console.trace`, `console.debug`, `console.log`, `console.info`, `console.error`, and `console.warn` to CloudWatch as structured JSON. Each JSON log object contains at least four key value pairs with the following keys:

- `"timestamp"` - the time the log message was generated
- `"level"` - the log level assigned to the message
- `"message"` - the contents of the log message
- `"requestId"` - the unique request ID for the function invocation

Depending on the logging method that your function uses, this JSON object may also contain additional key pairs. For example, if your function uses `console` methods to log error objects using multiple arguments, the JSON object will contain extra key value pairs with the keys `errorMessage`, `errorType`, and `stackTrace`.

If your code already uses another logging library, such as Powertools for AWS Lambda, to produce JSON structured logs, you don't need to make any changes. Lambda doesn't double-encode any logs that are already JSON encoded, so your function's application logs will continue to be captured as before.

For more information about using the Powertools for AWS Lambda logging package to create JSON structured logs in the Node.js runtime, see the section called "Logging".

Example JSON formatted log outputs

The following examples shows how various log outputs generated using the `console` methods with single and multiple arguments are captured in CloudWatch Logs when you set your function's log format to JSON.

The first example uses the `console.error` method to output a simple string.

Example Node.js logging code

```
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

Example JSON log record

```
{
    "timestamp":"2025-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
```

```
}
```

You can also output more complex structured log messages using either single or multiple arguments with the `console` methods. In the next example, you use `console.log` to output two key value pairs using a single argument. Note that the `"message"` field in the JSON object Lambda sends to CloudWatch Logs is not stringified.

Example Node.js logging code

```
export const handler = async (event) => {
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON log record

```
{
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
        "flag": false
    }
}
```

In the next example, you again use the `console.log` method to create a log output. This time, the method takes two arguments, a map containing two key value pairs and an identifying string. Note that in this case, because you have supplied two arguments, Lambda stringifies the `"message"` field.

Example Node.js logging code

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON log record

```
{
```

```
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda assigns outputs generated using `console.log` the log level INFO.

The final example shows how error objects can be output to CloudWatch Logs using the `console` methods. Note that when you log error objects using multiple arguments, Lambda adds the fields `errorMessage`, `errorType`, and `stackTrace` to the log output.

Example Node.js logging code

```
export const handler = async (event) => {
  let e1 = new ReferenceError("some reference error");
  let e2 = new SyntaxError("some syntax error");
  console.log(e1);
  console.log("errors logged - ", e1, e2);
};
```

Example JSON log record

```
{
    "timestamp": "2025-12-08T23:21:04.632Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "errorType": "ReferenceError",
        "errorMessage": "some reference error",
        "stackTrace": [
            "ReferenceError: some reference error",
            "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
            "    at Runtime.handleOnceNonStreaming (file:///var/runtime/
index.mjs:1173:29)"
        ]
    }
}

{
    "timestamp": "2025-12-08T23:21:04.646Z",
    "level": "INFO",
```

```
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "errors logged -  ReferenceError: some reference error
\n    at Runtime.handler (file:///var/task/index.mjs:3:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29) SyntaxError: some syntax
 error\n    at Runtime.handler (file:///var/task/index.mjs:4:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29)",
    "errorType": "ReferenceError",
    "errorMessage": "some reference error",
    "stackTrace": [
        "ReferenceError: some reference error",
        "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
        "    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
    ]
}
```

When logging multiple error types, the extra fields `errorMessage`, `errorType`, and `stackTrace` are extracted from the first error type supplied to the `console` method.

Using embedded metric format (EMF) client libraries with structured JSON logs

AWS provides open-sourced client libraries for Node.js which you can use to create embedded metric format (EMF) logs. If you have existing functions that use these libraries and you change your function's log format to JSON, CloudWatch may no longer recognize the metrics emitted by your code.

If your code currently emits EMF logs directly using `console.log` or by using Powertools for AWS Lambda (TypeScript), CloudWatch will also be unable to parse these if you change your function's log format to JSON.

> **Important**
>
> To ensure that your functions' EMF logs continue to be properly parsed by CloudWatch, update your EMF and Powertools for AWS Lambda libraries to the latest versions. If switching to the JSON log format, we also recommend that you carry out testing to ensure compatibility with your function's embedded metrics. If your code emits EMF logs directly using `console.log`, change your code to output those metrics directly to `stdout` as shown in the following code example.

Example code emitting embedded metrics to `stdout`

```
process.stdout.write(JSON.stringify(
    {
        "_aws": {
            "Timestamp": Date.now(),
            "CloudWatchMetrics": [{
                "Namespace": "lambda-function-metrics",
                "Dimensions": [["functionVersion"]],
                "Metrics": [{
                    "Name": "time",
                    "Unit": "Milliseconds",
                    "StorageResolution": 60
                }]
            }]
        },
        "functionVersion": "$LATEST",
        "time": 100,
        "requestId": context.awsRequestId
    }
) + "\n")
```

Using log-level filtering with Node.js

For AWS Lambda to filter your application logs according to their log level, your function must use JSON formatted logs. You can achieve this in two ways:

- Create log outputs using the standard console methods and configure your function to use JSON log formatting. AWS Lambda then filters your log outputs using the “level” key value pair in the JSON object described in the section called “Using structured JSON logs with Node.js”. To learn how to configure your function’s log format, see the section called “Configuring advanced logging controls for Lambda functions”.
- Use another logging library or method to create JSON structured logs in your code that include a “level” key value pair defining the level of the log output. For example, you can use Powertools for AWS Lambda to generate JSON structured log outputs from your code. See the section called “Logging” to learn more about using Powertools with the Node.js runtime.

  For Lambda to filter your function's logs, you must also include a `"timestamp"` key value pair in your JSON log output. The time must be specified in valid RFC 3339 timestamp format. If you

don't supply a valid timestamp, Lambda will assign the log the level INFO and add a timestamp for you.

When you configure your function to use log-level filtering, you select the level of logs you want AWS Lambda to send to CloudWatch Logs from the following options:

Log level	Standard usage
TRACE (most detail)	The most fine-grained information used to trace the path of your code's execution
DEBUG	Detailed information for system debugging
INFO	Messages that record the normal operation of your function
WARN	Messages about potential errors that may lead to unexpected behavior if unaddressed
ERROR	Messages about problems that prevent the code from performing as expected
FATAL (least detail)	Messages about serious errors that cause the application to stop functioning

Lambda sends logs of the selected level and lower to CloudWatch. For example, if you configure a log level of WARN, Lambda will send logs corresponding to the WARN, ERROR, and FATAL levels.

Viewing logs in the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Viewing logs in the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/*your-function-name***).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

Viewing logs using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
```

```
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Instrumenting Node.js code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- AWS Distro for OpenTelemetry (ADOT) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- AWS X-Ray SDK for Node.js – An SDK for generating and sending trace data to X-Ray.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

> **Important**
>
> The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs.

Sections

- Using ADOT to instrument your Node.js functions
- Using the X-Ray SDK to instrument your Node.js functions
- Activating tracing with the Lambda console
- Activating tracing with the Lambda API
- Activating tracing with CloudFormation
- Interpreting an X-Ray trace
- Storing runtime dependencies in a layer (X-Ray SDK)

Using ADOT to instrument your Node.js functions

ADOT provides fully managed Lambda layers that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see Custom configuration for the ADOT Collector on Lambda in the ADOT documentation.

For Node.js runtimes, you can add the **AWS managed Lambda layer for ADOT Javascript** to automatically instrument your functions. For detailed instructions on how to add this layer, see AWS Distro for OpenTelemetry Lambda Support for JavaScript in the ADOT documentation.

Using the X-Ray SDK to instrument your Node.js functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Node.js. To get the SDK, add the `aws-xray-sdk-core` package to your application's dependencies.

Example blank-nodejs/package.json

```
{
  "name": "blank-nodejs",
  "version": "1.0.0",
  "private": true,
  "devDependencies": {
    "jest": "29.7.0"
  },
  "dependencies": {
    "@aws-sdk/client-lambda": "3.345.0",
    "aws-xray-sdk-core": "3.5.3"
  },
  "scripts": {
    "test": "jest"
  }
}
```

To instrument AWS SDK clients in the AWS SDK for JavaScript v3, wrap the client instance with the `captureAWSv3Client` method.

Example blank-nodejs/function/index.js – Tracing an AWS SDK client

```
const AWSXRay = require('aws-xray-sdk-core');
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());

// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
  ...
```

The Lambda runtime sets some environment variables to configure the X-Ray SDK. For example, Lambda sets `AWS_XRAY_CONTEXT_MISSING` to `LOG_ERROR` to avoid throwing runtime errors from the X-Ray SDK. To set a custom context missing strategy, override the environment variable in your function configuration to have no value, and then you can set the context missing strategy programmatically.

Example initialization code

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

For more information, see the section called “Environment variables”.

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.

4. Under **Additional monitoring tools**, choose **Edit**.
5. Under **CloudWatch Application Signals and AWS X-Ray**, choose **Enable** for **Lambda service traces**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an CloudFormation template, use the `TracingConfig` property.

Example function-inline.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example template.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpreting an X-Ray trace

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

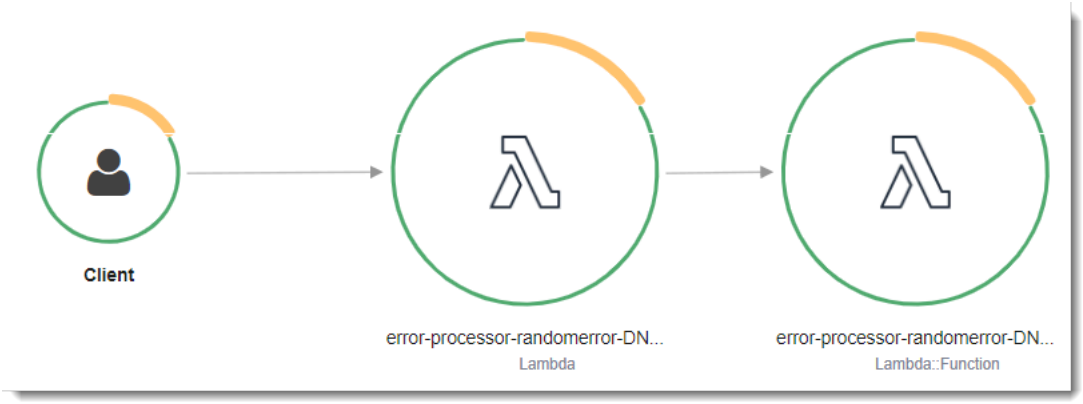
After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.



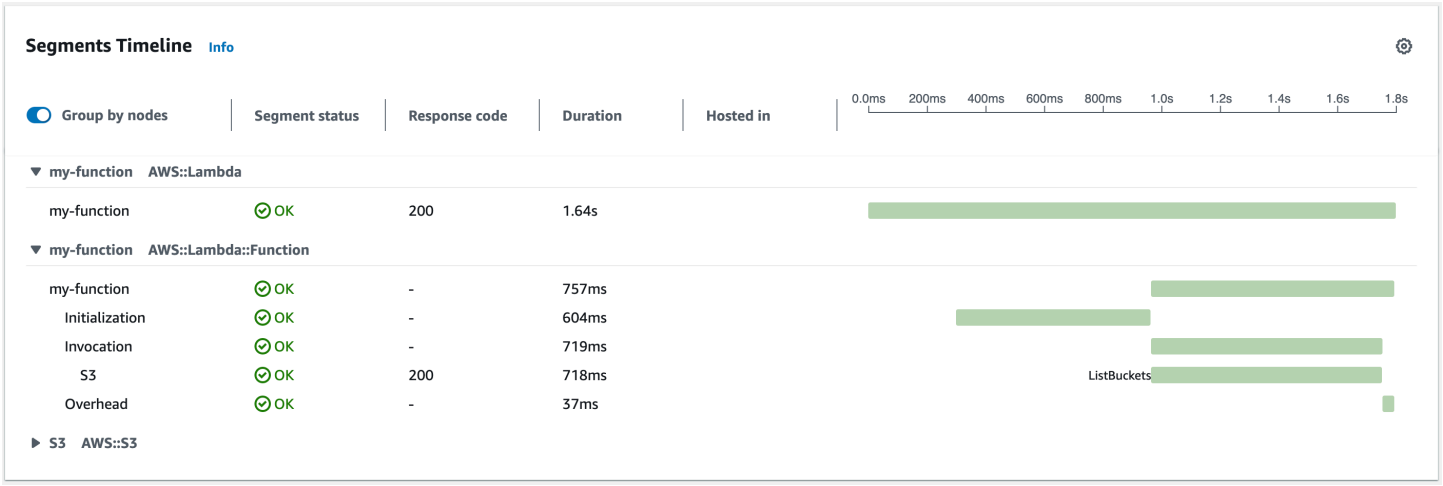
X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling

rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

In X-Ray, a *trace* records information about a request that is processed by one or more *services*. Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes:



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.



This example expands the `AWS::Lambda::Function` segment to show its three subsegments.

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
> The example trace shown here illustrates the old-style function segment. The differences between the old- and new-style segments are described in the following paragraphs.
> These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

The old-style function segment contains the following subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

The new-style function segment doesn't contain an `Invocation` subsegment. Instead, customer subsegments are attached directly to the function segment. For more information about the structure of the old- and new-style function segments, see the section called "Understanding X-Ray traces".

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the AWS X-Ray SDK for Node.js in the *AWS X-Ray Developer Guide*.

> **Pricing**
>
> You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a Lambda layer.

The following example shows an `AWS::Serverless::LayerVersion` resource that stores the AWS X-Ray SDK for Node.js.

Example template.yml – Dependencies layer

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs24.x
```

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the blank-nodejs sample application.

Building Lambda functions with TypeScript

You can use the Node.js runtime to run TypeScript code in AWS Lambda. Because Node.js doesn't run TypeScript code natively, you must first transpile your TypeScript code into JavaScript. Then, use the JavaScript files to deploy your function code to Lambda. Your code runs in an environment that includes the AWS SDK for JavaScript, with credentials from an AWS Identity and Access Management (IAM) role that you manage. To learn more about the SDK versions included with the Node.js runtimes, see the section called “Runtime-included SDK versions”.

Lambda supports the following Node.js runtimes.

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Node.js 24	`nodejs24.x`	Amazon Linux 2023	Apr 30, 2028	Jun 1, 2028	Jul 1, 2028
Node.js 22	`nodejs22.x`	Amazon Linux 2023	Apr 30, 2027	Jun 1, 2027	Jul 1, 2027
Node.js 20	`nodejs20.x`	Amazon Linux 2023	Apr 30, 2026	Aug 31, 2026	Sep 30, 2026

Topics

- Setting up a TypeScript development environment
- Type definitions for Lambda
- Define Lambda function handler in TypeScript
- Deploy transpiled TypeScript code in Lambda with .zip file archives
- Deploy transpiled TypeScript code in Lambda with container images
- Using the Lambda context object to retrieve TypeScript function information
- Log and monitor TypeScript Lambda functions
- Tracing TypeScript code in AWS Lambda

Setting up a TypeScript development environment

Use a local integrated development environment (IDE) or text editor to write your TypeScript function code. You can't create TypeScript code on the Lambda console.

You can use either esbuild or Microsoft's TypeScript compiler (`tsc`) to transpile your TypeScript code into JavaScript. The AWS Serverless Application Model (AWS SAM) and the AWS Cloud Development Kit (AWS CDK) both use esbuild.

When using esbuild, consider the following:

- There are several TypeScript caveats.
- You must configure your TypeScript transpilation settings to match the Node.js runtime that you plan to use. For more information, see Target in the esbuild documentation. For an example of a **tsconfig.json** file that demonstrates how to target a specific Node.js version supported by Lambda, refer to the TypeScript GitHub repository.
- esbuild doesn't perform type checks. To check types, use the `tsc` compiler. Run `tsc -noEmit` or add a `"noEmit"` parameter to your **tsconfig.json** file, as shown in the following example. This configures `tsc` to not emit JavaScript files. After checking types, use esbuild to convert the TypeScript files into JavaScript.

Example tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2020",
   "strict": true,
   "preserveConstEnums": true,
   "noEmit": true,
   "sourceMap": false,
   "module":"commonjs",
   "moduleResolution":"node",
   "esModuleInterop": true,
   "skipLibCheck": true,
   "forceConsistentCasingInFileNames": true,
   "isolatedModules": true,
 },
 "exclude": ["node_modules", "**/*.test.ts"]
}
```

Type definitions for Lambda

The @types/aws-lambda package provides type definitions for Lambda functions. Install this package when your function uses any of the following:

- Common AWS event sources, such as:
  - `APIGatewayProxyEvent`: For Amazon API Gateway proxy integrations
  - `SNSEvent`: For Amazon Simple Notification Service notifications
  - `SQSEvent`: For Amazon Simple Queue Service messages
  - `S3Event`: For S3 trigger events
  - `DynamoDBStreamEvent`: For Amazon DynamoDB Streams
- The Lambda Context object
- The callback handler pattern

To add the Lambda type definitions to your function, install `@types/aws-lambda` as a development dependency:

```
npm install -D @types/aws-lambda
```

Then, import the types from `aws-lambda`:

```
import { Context, S3Event, APIGatewayProxyEvent } from 'aws-lambda';

export const handler = async (event: S3Event, context: Context) => {
    // Function code
};
```

The `import ... from 'aws-lambda'` statement imports the type definitions. It does not import the `aws-lambda` npm package, which is an unrelated third-party tool. For more information, see aws-lambda in the DefinitelyTyped GitHub repository.

> **Note**
>
> You don't need @types/aws-lambda when using your own custom type definitions. For an example function that defines its own type for an event object, see Example TypeScript Lambda function code.

Define Lambda function handler in TypeScript

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

This page describes how to work with Lambda function handlers in TypeScript, including options for project setup, naming conventions, and best practices. This page also includes an example of a TypeScript Lambda function that takes in information about an order, produces a text file receipt, and puts this file in an Amazon Simple Storage Service (Amazon S3) bucket. For information about how to deploy your function after writing it, see the section called "Deploy .zip file archives" or the section called "Deploy container images".

Topics

- Setting up your TypeScript project
- Example TypeScript Lambda function code
- CommonJS and ES Modules
- Node.js initialization
- Handler naming conventions
- Defining and accessing the input event object
- Valid handler patterns for TypeScript functions
- Using the SDK for JavaScript v3 in your handler
- Accessing environment variables
- Using global state
- Code best practices for TypeScript Lambda functions

Setting up your TypeScript project

Use a local integrated development environment (IDE) or text editor to write your TypeScript function code. You can't create TypeScript code on the Lambda console.

There are multiple ways to initialize a TypeScript Lambda project. For example, you can create a project using `npm`, create an AWS SAM application, or create an AWS CDK application. To create the project using `npm`:

```
npm init
```

Your function code lives in a `.ts` file, which you transpile into a JavaScript file at build time. You can use either esbuild or Microsoft's TypeScript compiler (`tsc`) to transpile your TypeScript code into JavaScript. To use esbuild, add it as a development dependency:

```
npm install -D esbuild
```

A typical TypeScript Lambda function project follows this general structure:

```
/project-root
  ### index.ts - Contains main handler
  ### dist/ - Contains compiled JavaScript
  ### package.json - Project metadata and dependencies
  ### package-lock.json - Dependency lock file
  ### tsconfig.json - TypeScript configuration
  ### node_modules/ - Installed dependencies
```

Example TypeScript Lambda function code

The following example Lambda function code takes in information about an order, produces a text file receipt, and puts this file in an Amazon S3 bucket. This example defines a custom event type (`OrderEvent`). To learn how to import type definitions for AWS event sources, see Type definitions for Lambda.

> **Note**
>
> This example uses an ES module handler. Lambda supports both ES module and CommonJS handlers. For more information, see CommonJS and ES Modules.

Example index.ts Lambda function

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

// Initialize the S3 client outside the handler for reuse
const s3Client = new S3Client();

// Define the shape of the input event
type OrderEvent = {
```

```
    order_id: string;
    amount: number;
    item: string;
}

/**
 * Lambda handler for processing orders and storing receipts in S3.
 */
export const handler = async (event: OrderEvent): Promise<string> => {
    try {
        // Access environment variables
        const bucketName = process.env.RECEIPT_BUCKET;
        if (!bucketName) {
            throw new Error('RECEIPT_BUCKET environment variable is not set');
        }

        // Create the receipt content and key destination
        const receiptContent = `OrderID: ${event.order_id}\nAmount: $
${event.amount.toFixed(2)}\nItem: ${event.item}`;
        const key = `receipts/${event.order_id}.txt`;

        // Upload the receipt to S3
        await uploadReceiptToS3(bucketName, key, receiptContent);

        console.log(`Successfully processed order ${event.order_id} and stored receipt
 in S3 bucket ${bucketName}`);
        return 'Success';
    } catch (error) {
        console.error(`Failed to process order: ${error instanceof Error ?
 error.message : 'Unknown error'}`);
        throw error;
    }
};

/**
 * Helper function to upload receipt to S3
 */
async function uploadReceiptToS3(bucketName: string, key: string, receiptContent:
 string): Promise<void> {
    try {
        const command = new PutObjectCommand({
            Bucket: bucketName,
            Key: key,
            Body: receiptContent
```

```
        });

        await s3Client.send(command);
    } catch (error) {
        throw new Error(`Failed to upload receipt to S3: ${error instanceof Error ?
 error.message : 'Unknown error'}`);
    }
}
```

This `index.ts` file contains the following sections of code:

- `import` block: Use this block to include libraries that your Lambda function requires, such as AWS SDK clients.
- `const s3Client` declaration: This initializes an Amazon S3 client outside of the handler function. This causes Lambda to run this code during the initialization phase, and the client is preserved for reuse across multiple invocations.
- `type OrderEvent`: Defines the structure of the expected input event.
- `export const handler`: This is the main handler function that Lambda invokes. When deploying your function, specify `index.handler` for the Handler property. The value of the `Handler` property is the file name and the name of the exported handler method, separated by a dot.
- `uploadReceiptToS3` function: This is a helper function that's referenced by the main handler function.

For this function to work properly, its execution role must allow the `s3:PutObject` action. Also, ensure that you define the `RECEIPT_BUCKET` environment variable. After a successful invocation, the Amazon S3 bucket should contain a receipt file.

CommonJS and ES Modules

Node.js supports two module systems: CommonJS and ECMAScript modules (ES modules). Lambda recommends using ES modules as it supports top-level await, which enables asynchronous tasks to be completed during execution environment initialization.

Node.js treats files with a `.cjs` file name extension as CommonJS modules while a `.mjs` extension denotes ES modules. By default, Node.js treats files with the `.js` file name extension as CommonJS modules. You can configure Node.js to treat `.js` files as ES modules by specifying the `type` as `module` in the function's `package.json` file. You can configure Node.js in Lambda

to detect automatically whether a `.js` file should be treated as CommonJS or as an ES module by adding the `–experimental-detect-module` flag to the `NODE_OPTIONS` environment variable. For more information, see Experimental Node.js features.

The following examples show function handlers written using both ES modules and CommonJS modules. The remaining examples on this page all use ES modules.

Node.js initialization

Node.js uses a non-blocking I/O model that supports efficient asynchronous operations using an event loop. For example, if Node.js makes a network call, the function continues to process other operations without blocking on a network response. When the network response is received, it is placed into the callback queue. Tasks from the queue are processed when the current task completes.

Lambda recommends using top-level await so that asynchronous tasks initiated during execution environment initialization are completed during initialization. Asynchronous tasks that are not completed during initialization will typically run during the first function invoke. This can cause unexpected behavior or errors. For example, your function initialization may make a network call to fetch a parameter from AWS Parameter Store. If this task is not completed during initialization, the value may be null during an invocation. There can also be a delay between initialization and invoke which can trigger errors in time-sensitive operations. In particular, AWS service calls can rely on time-sensitive request signatures, resulting in service call failures if the call is not completed during the initialization phase. Completing tasks during initialization typically improves cold-start performance, and first invoke performance when using Provisioned Concurrency. For more information, see our blog post Using Node.js ES modules and top-level await in AWS Lambda.

Handler naming conventions

When you configure a function, the value of the Handler setting is the file name and the name of the exported handler method, separated by a dot. The default for functions created in the console and for examples in this guide is `index.handler`. This indicates the `handler` method that's exported from the `index.js` or `index.mjs` file.

If you create a function in the console using a different file name or function handler name, you must edit the default handler name.

To change the function handler name (console)

1. Open the Functions page of the Lambda console and choose your function.

2. Choose the **Code** tab.
3. Scroll down to the **Runtime settings** pane and choose **Edit**.
4. In **Handler**, enter the new name for your function handler.
5. Choose **Save**.

Defining and accessing the input event object

JSON is the most common and standard input format for Lambda functions. In this example, the function expects an input similar to the following:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

When working with Lambda functions in TypeScript, you can define the shape of the input event using a type or interface. In this example, we define the event structure using a type:

```
type OrderEvent = {
    order_id: string;
    amount: number;
    item: string;
}
```

After you define the type or interface, use it in your handler's signature to ensure type safety:

```
export const handler = async (event: OrderEvent): Promise<string> => {
```

During compilation, TypeScript validates that the event object contains the required fields with the correct types. For example, the TypeScript compiler reports an error if you try to use `event.order_id` as a number or `event.amount` as a string.

Valid handler patterns for TypeScript functions

We recommend that you use async/await to declare the function handler instead of using callbacks. Async/await is a concise and readable way to write asynchronous code, without the need

for nested callbacks or chaining promises. With async/await, you can write code that reads like synchronous code, while still being asynchronous and non-blocking.

The examples in this section use the `S3Event` type. However, you can use any other AWS event types in the @types/aws-lambda package, or define your own event type. To use types from @types/aws-lambda:

1. Add the @types/aws-lambda package as a development dependency:

```
npm install -D @types/aws-lambda
```

2. Import the types you need, such as `Context`, `S3Event`, or `Callback`.

async function handlers (recommended)

The `async` keyword marks a function as asynchronous, and the `await` keyword pauses the execution of the function until a `Promise` is resolved. The handler accepts the following arguments:

- `event`: Contains the input data passed to your function.
- `context`: Contains information about the invocation, function, and execution environment. For more information, see Using the Lambda context object to retrieve TypeScript function information.

Here are the valid signatures for the async/await pattern:

```
export const handler = async (event: S3Event): Promise<void> => { };
```

```
export const handler = async (event: S3Event, context: Context): Promise<void> => { };
```

> **Note**
>
> When processing arrays of items asynchronously, make sure to use await with `Promise.all` to ensure all operations complete. Methods like `forEach` don't wait for async callbacks to complete. For more information, see Array.prototype.forEach() in the Mozilla documentation.

Synchronous function handlers

Where your function does not perform any asynchronous tasks, you can use a synchronous function handler, using one of the following function signatures:

```
export const handler = (event: S3Event): void => { };
```

```
export const handler = (event: S3Event, context: Context): void => { };
```

Response streaming function handlers

Lambda supports response streaming with Node.js. Response streaming function handlers use the awslambda.streamifyResponse() decorator and take 3 parameters: event, responseStream, and context. The function signature is:

```
export const handler = awslambda.streamifyResponse(async (event: APIGatewayProxyEvent,
 responseStream: NodeJS.WritableStream, context: Context) => { });
```

For more information, see Response streaming for Lambda functions.

Callback-based function handlers

> **Note**
>
> Callback-based function handlers are only supported up to Node.js 22. Starting from Node.js 24, asynchronous tasks should be implemented using async function handlers.

Callback-based function handlers can use the event, context, and callback arguments. The callback argument expects an `Error` and a response, which must be JSON-serializable.

Here is the valid signature for the callback handler pattern:

```
export const handler = (event: S3Event, context: Context, callback: Callback<void>):
 void => { };
```

The function continues to execute until the event loop is empty or the function times out. The response isn't sent to the invoker until all event loop tasks are finished. If the function times out,

an error is returned instead. You can configure the runtime to send the response immediately by setting context.callbackWaitsForEmptyEventLoop to false.

Example TypeScript function with callback

The following example uses `APIGatewayProxyCallback`, which is a specialized callback type specific to API Gateway integrations. Most AWS event sources use the generic `Callback` type shown in the signatures above.

```
import { Context, APIGatewayProxyCallback, APIGatewayEvent } from 'aws-lambda';

export const lambdaHandler = (event: APIGatewayEvent, context: Context, callback:
 APIGatewayProxyCallback): void => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    callback(null, {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    });
};
```

Using the SDK for JavaScript v3 in your handler

Often, you'll use Lambda functions to interact with or make updates to other AWS resources. The simplest way to interface with these resources is to use the AWS SDK for JavaScript. All supported Lambda Node.js runtimes include the SDK for JavaScript version 3. However, we strongly recommend that you include the AWS SDK clients that you need in your deployment package. This maximizes backward compatibility during future Lambda runtime updates.

To add SDK dependencies to your function, use the `npm install` command for the specific SDK clients that you need. In the example code, we used the Amazon S3 client. Add this dependency by running the following command in the directory that contains your `package.json` file:

```
npm install @aws-sdk/client-s3
```

In the function code, import the client and commands that you need, as the example function demonstrates:

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

Then, initialize an Amazon S3 client:

```
const s3Client = new S3Client();
```

In this example, we initialized our Amazon S3 client outside of the main handler function to avoid having to initialize it every time we invoke our function. After you initialize your SDK client, you can then use it to make API calls for that AWS service. The example code calls the Amazon S3 PutObject API action as follows:

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: receiptContent
});
```

Accessing environment variables

In your handler code, you can reference any environment variables by using `process.env`. In this example, we reference the defined `RECEIPT_BUCKET` environment variable using the following lines of code:

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

Using global state

Lambda runs your static code during the initialization phase before invoking your function for the first time. Resources created during initialization stay in memory between invocations, so you can avoid having to create them every time you invoke your function.

In the example code, the S3 client initialization code is outside the handler. The runtime initializes the client before the function handles its first event, and the client remains available for reuse across all invocations.

Code best practices for TypeScript Lambda functions

Follow these guidelines when building Lambda functions:

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function.
- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries. For the Node.js and Python runtimes, these include the AWS SDKs. To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation.

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of function invocations and escalated costs. If you see an unintended volume of invocations, set the function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

Deploy transpiled TypeScript code in Lambda with .zip file archives

Before you can deploy TypeScript code to AWS Lambda, you need to transpile it into JavaScript. This page explains three ways to build and deploy TypeScript code to Lambda with .zip file archives:

- Using AWS Serverless Application Model (AWS SAM)
- Using the AWS Cloud Development Kit (AWS CDK)
- Using the AWS Command Line Interface (AWS CLI) and esbuild

AWS SAM and AWS CDK simplify building and deploying TypeScript functions. The AWS SAM template specification provides a simple and clean syntax to describe the Lambda functions, APIs, permissions, configurations, and events that make up your serverless application. The AWS CDK lets you build reliable, scalable, cost-effective applications in the cloud with the considerable expressive power of a programming language. The AWS CDK is intended for moderately to highly experienced AWS users. Both the AWS CDK and the AWS SAM use esbuild to transpile TypeScript code into JavaScript.

Using AWS SAM to deploy TypeScript code to Lambda

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application using the AWS SAM. This application implements a basic API backend. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function is invoked. The function returns a `hello world` message.

> **Note**
>
> AWS SAM uses esbuild to create Node.js Lambda functions from TypeScript code. esbuild support is currently in public preview. During public preview, esbuild support may be subject to backwards incompatible changes.

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- AWS SAM CLI version 1.75 or later
- Node.js

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --
runtime nodejs24.x
```

2. (Optional) The sample application includes configurations for commonly used tools, such as ESLlint for code linting and Jest for unit testing. To run lint and test commands:

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. Build the app.

```
cd sam-app
sam build
```

4. Deploy the app.

```
sam deploy --guided
```

5. Follow the on-screen prompts. To accept the default options provided in the interactive experience, respond with `Enter`.
6. The output shows the endpoint for the REST API. Open the endpoint in a browser to test the function. You should see this response:

```
{"message":"hello world"}
```

7. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Using the AWS CDK to deploy TypeScript code to Lambda

Follow the steps below to build and deploy a sample TypeScript application using the AWS CDK. This application implements a basic API backend. It consists of an API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function is invoked. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- AWS CDK version 2
- Node.js
- Either Docker or esbuild

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

   ```
   mkdir hello-world
   cd hello-world
   ```

2. Initialize the app.

   ```
   cdk init app --language typescript
   ```

3. Add the @types/aws-lambda package as a development dependency. This package contains the type definitions for Lambda.

   ```
   npm install -D @types/aws-lambda
   ```

4. Open the **lib** directory. You should see a file called **hello-world-stack.ts**. Create two new files in this directory: **hello-world.function.ts** and **hello-world.ts**.
5. Open **hello-world.function.ts** and add the following code to the file. This is the code for the Lambda function.

> **Note**
>
> The `import` statement imports the type definitions from @types/aws-lambda. It does not import the `aws-lambda` NPM package, which is an unrelated third-party tool. For more information, see aws-lambda in the DefinitelyTyped GitHub repository.

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. Open **hello-world.ts** and add the following code to the file. This contains the NodejsFunction construct, which creates the Lambda function, and the LambdaRestApi construct, which creates the REST API.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function');
    new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
  }
}
```

The `NodejsFunction` construct assumes the following by default:

- Your function handler is called `handler`.
- The .ts file that contains the function code (**hello-world.function.ts**) is in the same directory as the .ts file that contains the construct (**hello-world.ts**). The construct uses the construct's ID ("hello-world") and the name of the Lambda handler file ("function") to find the function code. For example, if your function code is in a file called **hello-world.my-function.ts**, the **hello-world.ts** file must reference the function code like this:

```
const helloFunction = new NodejsFunction(this, 'my-function');
```

You can change this behavior and configure other esbuild parameters. For more information, see Configuring esbuild in the AWS CDK API reference.

7. Open **hello-world-stack.ts**. This is the code that defines your AWS CDK stack. Replace the code with the following:

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

8. from the `hello-world` directory containing your `cdk.json` file, deploy your application.

```
cdk deploy
```

9. The AWS CDK builds and packages the Lambda function using esbuild, and then deploys the function to the Lambda runtime. The output shows the endpoint for the REST API. Open the endpoint in a browser to test the function. You should see this response:

```
{"message":"hello world"}
```

This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

Using the AWS CLI and esbuild to deploy TypeScript code to Lambda

The following example demonstrates how to transpile and deploy TypeScript code to Lambda using esbuild and the AWS CLI. esbuild produces one JavaScript file with all dependencies. This is the only file that you need to add to the .zip archive.

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Node.js
- An execution role for the Lambda function
- For Windows users, a zip file utility such as 7zip.

Deploy a sample function

1. On your local machine, create a project directory for your new function.
2. Create a new Node.js project with npm or a package manager of your choice.

```
npm init
```

3. Add the @types/aws-lambda and esbuild packages as development dependencies. The `@types/aws-lambda` package contains the type definitions for Lambda.

```
npm install -D @types/aws-lambda esbuild
```

4. Create a new file called **index.ts**. Add the following code to the new file. This is the code for the Lambda function. The function returns a `hello world` message. The function doesn't create any API Gateway resources.

> **Note**
>
> The `import` statement imports the type definitions from @types/aws-lambda. It does not import the `aws-lambda` NPM package, which is an unrelated third-party tool. For more information, see aws-lambda in the DefinitelyTyped GitHub repository.

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
      statusCode: 200,
      body: JSON.stringify({
          message: 'hello world',
      }),
   };
};
```

5. Add a build script to the **package.json** file. This configures esbuild to automatically create the .zip deployment package. For more information, see Build scripts in the esbuild documentation.

   Linux and MacOS

   ```
   "scripts": {
     "prebuild": "rm -rf dist",
     "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
   target=es2020 --outfile=dist/index.js",
     "postbuild": "cd dist && zip -r index.zip index.js*"
   },
   ```

   Windows

   In this example, the `"postbuild"` command uses the 7zip utility to create your .zip file. Use your own preferred Windows zip utility and modify the command as necessary.

   ```
   "scripts": {
     "prebuild": "del /q dist",
     "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
   target=es2020 --outfile=dist/index.js",
     "postbuild": "cd dist && 7z a -tzip index.zip index.js*"
   },
   ```

6. Build the package.

```
npm run build
```

7. Create a Lambda function using the .zip deployment package. Replace the highlighted text with the Amazon Resource Name (ARN) of your execution role.

```
aws lambda create-function --function-name hello-world --runtime "nodejs24.x" --
role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip"
 --handler index.handler
```

8. Run a test event to confirm that the function returns the following response. If you want to invoke this function using API Gateway, create and configure a REST API.

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

Deploy transpiled TypeScript code in Lambda with container images

You can deploy your TypeScript code to an AWS Lambda function as a Node.js container image. AWS provides base images for Node.js to help you build the container image. These base images are preloaded with a language runtime and other components that are required to run the image on Lambda. AWS provides a Dockerfile for each of the base images to help with building your container image.

If you use a community or private enterprise base image, you must add the Node.js runtime interface client (RIC) to the base image to make it compatible with Lambda.

Lambda provides a runtime interface emulator for local testing. The AWS base images for Node.js include the runtime interface emulator. If you use an alternative base image, such as an Alpine Linux or Debian image, you can build the emulator into your image or install it on your local machine.

Using a Node.js base image to build and package TypeScript function code

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.
- Node.js 22.x

Creating an image from a base image

To create an image from an AWS base image for Lambda

1. On your local machine, create a project directory for your new function.
2. Create a new Node.js project with `npm` or a package manager of your choice.

```
npm init
```

3. Add the @types/aws-lambda and esbuild packages as development dependencies. The `@types/aws-lambda` package contains the type definitions for Lambda.

```
npm install -D @types/aws-lambda esbuild
```

4. Add a build script to the `package.json` file.

```
  "scripts": {
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. Create a new file called `index.ts`. Add the following sample code to the new file. This is the code for the Lambda function. The function returns a `hello world` message.

> **Note**
>
> The `import` statement imports the type definitions from @types/aws-lambda. It does not import the `aws-lambda` NPM package, which is an unrelated third-party tool. For more information, see aws-lambda in the DefinitelyTyped GitHub repository.

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. Create a new Dockerfile with the following configuration:

- Set the `FROM` property to the URI of the base image.
- Set the `CMD` argument to specify the Lambda function handler.

The following example Dockerfile uses a multi-stage build. The first step transpiles the TypeScript code into JavaScript. The second step produces a container image that contains only JavaScript files and production dependencies.

Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:22 as builder
WORKDIR /usr/app
COPY package.json index.ts  ./
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:22
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

1. Start the Docker image with the **docker run** command. In this example, `docker-image` is the image name and `test` is the tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.

   Linux/macOS

   In Linux and macOS, run the following `curl` command:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   In PowerShell, run the following `Invoke-WebRequest` command:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Get the container ID.

```
docker ps
```

4. Use the docker kill command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Using the Lambda context object to retrieve TypeScript function information

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment.

To enable type checking for the context object, you must add the @types/aws-lambda package as a development dependency and import the `Context` type. For more information, see Type definitions for Lambda.

Context methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.

Context properties

- `functionName` – The name of the Lambda function.
- `functionVersion` – The version of the function.
- `invokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memoryLimitInMB` – The amount of memory that's allocated for the function.
- `awsRequestId` – The identifier of the invocation request.
- `logGroupName` – The log group for the function.
- `logStreamName` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
  - `cognitoIdentityId` – The authenticated Amazon Cognito identity.
  - `cognitoIdentityPoolId` – The Amazon Cognito identity pool that authorized the invocation.
- `clientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
  - `client.installation_id`

- `client.app_title`
- `client.app_version_name`
- `client.app_version_code`
- `client.app_package_name`
- `env.platform_version`
- `env.platform`
- `env.make`
- `env.model`
- `env.locale`
- `Custom` – Custom values that are set by the client application.

- `callbackWaitsForEmptyEventLoop` – By default (`true`), when using a callback-based function handler, Lambda waits for the event loop to be empty after the callback runs before ending the function invoke. Set to `false` to send the response and end the invoke immediately after the callback runs instead of waiting for the event loop to be empty. Outstanding events continue to run during the next invocation. Note that Lambda supports callback-based function handlers for Node.js 22 and earlier runtimes only.

Example index.ts file

The following example function logs context information and returns the location of the logs.

> **Note**
>
> Before using this code in a Lambda function, you must add the @types/aws-lambda package as a development dependency. This package contains the type definitions for Lambda. For more information, see Type definitions for Lambda.

```
import { Context } from 'aws-lambda';
export const lambdaHandler = async (event: string, context: Context): Promise<string>
 => {
  console.log('Remaining time: ', context.getRemainingTimeInMillis());
  console.log('Function name: ', context.functionName);
  return context.logStreamName;
};
```

Log and monitor TypeScript Lambda functions

AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function's code to the log stream. For more information about CloudWatch Logs, see Sending Lambda function logs to CloudWatch Logs.

To output logs from your function code, you can use methods on the console object. For more detailed logging, you can use any logging library that writes to `stdout` or `stderr`.

Sections

- Using logging tools and libraries
- Using Powertools for AWS Lambda (TypeScript) and AWS SAM for structured logging
- Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for structured logging
- Viewing logs in the Lambda console
- Viewing logs in the CloudWatch console

Using logging tools and libraries

Powertools for AWS Lambda (TypeScript) is a developer toolkit to implement Serverless best practices and increase developer velocity. The Logger utility provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization’s Logging RFC

Using Powertools for AWS Lambda (TypeScript) and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated Powertools for AWS Lambda (TypeScript) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 20 or later
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press `Enter`.

> **Note**
>
> For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the logs for the function, run sam logs. For more information, see Working with logs in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name sam-app
```

The log output looks like this:

```
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.552000
 START RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.594000
 2025-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
 INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
[{"Namespace":"sam-app","Dimensions":[["service"]],"Metrics":
[{"Name":"ColdStart","Unit":"Count"}]}]},"service":"helloWorld","ColdStart":1}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.595000
 2025-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"level":"INFO","message":"This is an INFO log - sending HTTP 200 - hello world
 response","service":"helloWorld","timestamp":"2025-08-31T09:33:10.594Z"}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.655000
 2025-08-31T09:33:10.655Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"_aws":{"Timestamp":1661938390655,"CloudWatchMetrics":[{"Namespace":"sam-
app","Dimensions":[["service"]],"Metrics":[]}]},"service":"helloWorld"}
```

```
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.754000 END
 RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.754000
 REPORT RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Duration: 201.55 ms Billed
 Duration: 202 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 252.42
 ms
XRAY TraceId: 1-630f2ad5-1de22b6d29a658a466e7ecf5 SegmentId: 567c116658fbf11a
 Sampled: true
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for structured logging

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated Powertools for AWS Lambda (TypeScript) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send

a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js 20 or later
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language typescript
```

3. Add the @types/aws-lambda package as a development dependency.

```
npm install -D @types/aws-lambda
```

4. Install the Powertools Logger utility.

```
npm install @aws-lambda-powertools/logger
```

5. Open the **lib** directory. You should see a file called **hello-world-stack.ts**. Create new two new files in this directory: **hello-world.function.ts** and **hello-world.ts**.
6. Open **hello-world.function.ts** and add the following code to the file. This is the code for the Lambda function.

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
```

```
import { Logger } from '@aws-lambda-powertools/logger';
const logger = new Logger();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  logger.info('This is an INFO log - sending HTTP 200 - hello world response');
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

7. Open **hello-world.ts** and add the following code to the file. This contains the NodejsFunction construct, which creates the Lambda function, configures environment variables for Powertools, and sets log retention to one week. It also includes the LambdaRestApi construct, which creates the REST API.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { RetentionDays } from 'aws-cdk-lib/aws-logs';
import { CfnOutput } from 'aws-cdk-lib';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        Powertools_SERVICE_NAME: 'helloWorld',
        LOG_LEVEL: 'INFO',
      },
      logRetention: RetentionDays.ONE_WEEK,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
```

```
}
```

8. Open **hello-world-stack.ts**. This is the code that defines your AWS CDK stack. Replace the code with the following:

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. Go back to the project directory.

```
cd hello-world
```

10. Deploy your application.

```
cdk deploy
```

11. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. Invoke the API endpoint:

```
curl <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

13. To get the logs for the function, run sam logs. For more information, see Working with logs in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name HelloWorldStack
```

The log output looks like this:

```
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.047000
 START RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.050000 {
"level": "INFO",
"message": "This is an INFO log - sending HTTP 200 - hello world response",
"service": "helloWorld",
"timestamp": "2025-08-31T14:48:37.048Z",
"xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.082000 END
 RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.082000
 REPORT RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Duration: 34.60 ms Billed
 Duration: 35 ms Memory Size: 128 MB Max Memory Used: 57 MB Init Duration: 173.48
 ms
```

14. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Viewing logs in the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Viewing logs in the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/*your-function-name***).

3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

Tracing TypeScript code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- AWS Distro for OpenTelemetry (ADOT) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- AWS X-Ray SDK for Node.js – An SDK for generating and sending trace data to X-Ray.
- Powertools for AWS Lambda (TypeScript) – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

> **Important**
>
> The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs.

Sections

- Using Powertools for AWS Lambda (TypeScript) and AWS SAM for tracing
- Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for tracing
- Interpreting an X-Ray trace

Using Powertools for AWS Lambda (TypeScript) and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated Powertools for AWS Lambda (TypeScript) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press `Enter`.

> **Note**
>
> For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the traces for the function, run sam traces.

```
sam traces
```

The trace output looks like this:

```
XRay Event [revision 1] at (2023-01-31T11:29:40.527000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.483s)
  - 0.425s - sam-app/Prod [HTTP: 200]
    - 0.422s - Lambda [HTTP: 200]
  - 0.406s - sam-app-HelloWorldFunction-Xyzv11a1bcde [HTTP: 200]
  - 0.172s - sam-app-HelloWorldFunction-Xyzv11a1bcde
    - 0.179s - Initialization
    - 0.112s - Invocation
      - 0.052s - ## app.lambdaHandler
        - 0.001s - ### MySubSegment
    - 0.059s - Overhead
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

Using Powertools for AWS Lambda (TypeScript) and the AWS CDK for tracing

Follow the steps below to download, build, and deploy a sample Hello World TypeScript application with integrated Powertools for AWS Lambda (TypeScript) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Node.js
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS Cloud Development Kit (AWS CDK) application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language typescript
```

3. Add the @types/aws-lambda package as a development dependency.

```
npm install -D @types/aws-lambda
```

4. Install the Powertools Tracer utility.

```
npm install @aws-lambda-powertools/tracer
```

5. Open the **lib** directory. You should see a file called **hello-world-stack.ts**. Create new two new files in this directory: **hello-world.function.ts** and **hello-world.ts**.

6. Open **hello-world.function.ts** and add the following code to the file. This is the code for the Lambda function.

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Tracer } from '@aws-lambda-powertools/tracer';
const tracer = new Tracer();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  // Get facade segment created by Lambda
  const segment = tracer.getSegment();

  // Create subsegment for the function and set it as active
  const handlerSegment = segment.addNewSubsegment(`## ${process.env._HANDLER}`);
  tracer.setSegment(handlerSegment);

  // Annotate the subsegment with the cold start and serviceName
  tracer.annotateColdStart();
  tracer.addServiceNameAnnotation();

  // Add annotation for the awsRequestId
  tracer.putAnnotation('awsRequestId', context.awsRequestId);
  // Create another subsegment and set it as active
  const subsegment = handlerSegment.addNewSubsegment('### MySubSegment');
  tracer.setSegment(subsegment);
  let response: APIGatewayProxyResult = {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
```

```
    }),
  };
  // Close subsegments (the Lambda one is closed automatically)
  subsegment.close(); // (### MySubSegment)
  handlerSegment.close(); // (## index.handler)

  // Set the facade segment as active again (the one created by Lambda)
  tracer.setSegment(segment);
  return response;
};
```

7. Open **hello-world.ts** and add the following code to the file. This contains the NodejsFunction construct, which creates the Lambda function, configures environment variables for Powertools, and sets log retention to one week. It also includes the LambdaRestApi construct, which creates the REST API.

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { CfnOutput } from 'aws-cdk-lib';
import { Tracing } from 'aws-cdk-lib/aws-lambda';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        POWERTOOLS_SERVICE_NAME: 'helloWorld',
      },
      tracing: Tracing.ACTIVE,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. Open **hello-world-stack.ts**. This is the code that defines your AWS CDK stack. Replace the code with the following:

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. Deploy your application.

```
cd ..
cdk deploy
```

10. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. Invoke the API endpoint:

```
curl <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

12. To get the traces for the function, run sam traces.

```
sam traces
```

The trace output looks like this:

```
XRay Event [revision 1] at (2023-01-31T11:50:06.997000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.449s)
  - 0.350s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde [HTTP: 200]
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
```

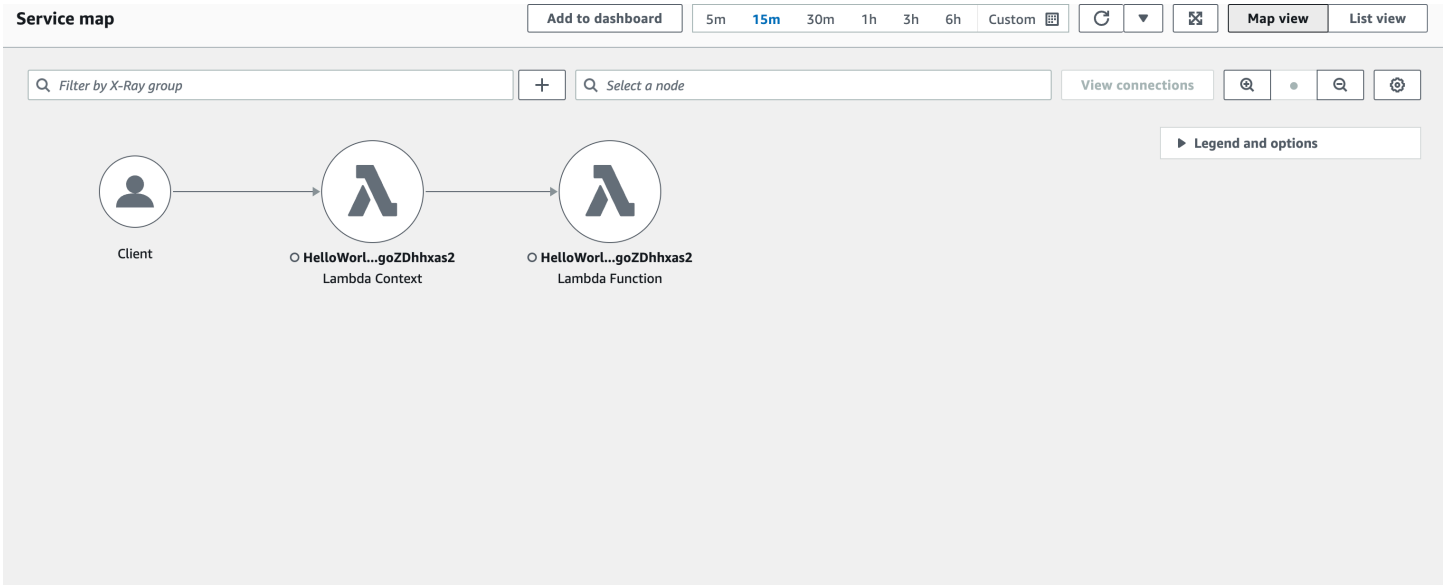
```
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Interpreting an X-Ray trace

After you've configured active tracing, you can observe specific requests through your application. The X-Ray trace map provides information about your application and all its components. The following example shows a trace from the sample application:



Building Lambda functions with Python

You can run Python code in AWS Lambda. Lambda provides runtimes for Python that run your code to process events. Your code runs in an environment that includes the SDK for Python (Boto3), with credentials from an AWS Identity and Access Management (IAM) role that you manage. To learn more about the SDK versions included with the Python runtimes, see the section called "Runtime-included SDK versions".

Lambda supports the following Python runtimes.

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Python 3.14	`python3.14`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
Python 3.13	`python3.13`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
Python 3.12	`python3.12`	Amazon Linux 2023	Oct 31, 2028	Nov 30, 2028	Jan 10, 2029
Python 3.11	`python3.11`	Amazon Linux 2	Jun 30, 2027	Jul 31, 2027	Aug 31, 2027
Python 3.10	`python3.10`	Amazon Linux 2	Oct 31, 2026	Nov 30, 2026	Jan 15, 2027

To create a Python function

1. Open the Lambda console.
2. Choose **Create function**.
3. Configure the following settings:
   - **Function name**: Enter a name for the function.
   - **Runtime**: Choose **Python 3.14**.

4. Choose **Create function**.

The console creates a Lambda function with a single source file named `lambda_function`. You can edit this file and add more files in the built-in code editor. In the **DEPLOY** section, choose **Deploy** to update your function's code. Then, to run your code, choose **Create test event** in the **TEST EVENTS** section.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any logs that your function outputs during invocation. If your function returns an error, Lambda formats the error and returns it to the invoker.

Topics

- Runtime-included SDK versions
- Disabled Python features
- Response format
- Graceful shutdown for extensions
- Define Lambda function handler in Python
- Working with .zip file archives for Python Lambda functions
- Deploy Python Lambda functions with container images
- Working with layers for Python Lambda functions
- Using the Lambda context object to retrieve Python function information
- Log and monitor Python Lambda functions
- AWS Lambda function testing in Python
- Instrumenting Python code in AWS Lambda

Runtime-included SDK versions

The version of the AWS SDK included in the Python runtime depends on the runtime version and your AWS Region. To find the version of the SDK included in the runtime you're using, create a Lambda function with the following code.

```
import boto3
import botocore
```

```
def lambda_handler(event, context):
   print(f'boto3 version: {boto3.__version__}')
   print(f'botocore version: {botocore.__version__}')
```

Disabled Python features

The following table lists Python features which are disabled in the Lambda managed runtimes and container base images for Python. These features must be enabled when the Python runtime executable is compiled and cannot be enabled by using an execution-time flag. To use these features in Lambda, you can deploy your own Python runtime build with these features enabled, using a container image or custom runtime.

Python feature	Affected Python versions	Status
Just-in-Time (JIT) compiler	Python 3.13 and later	The JIT compiler is experimental and is not recommended for production workloads. It is therefore disabled in the Lambda Python runtimes.
Free-threading	Python 3.13 and later	Free threading (option to disable the global interpreter lock) is disabled in Lambda Python builds due to the performance impact on single-threaded code.

Response format

In Python 3.12 and later Python runtimes, functions return Unicode characters as part of their JSON response. Earlier Python runtimes return escaped sequences for Unicode characters in responses. For example, in Python 3.11, if you return a Unicode string such as "こんにちは", it escapes the Unicode characters and returns "\u3053\u3093\u306b\u3061\u306f". The Python 3.12 runtime returns the original "こんにちは".

Using Unicode responses reduces the size of Lambda responses, making it easier to fit larger responses into the 6 MB maximum payload size for synchronous functions. In the previous example, the escaped version is 32 bytes—compared to 17 bytes with the Unicode string.

When you upgrade to Python 3.12 or later Python runtimes, you might need to adjust your code to account for the new response format. If the caller expects escaped Unicode, you must either add code to the returning function to escape the Unicode manually, or adjust the caller to handle the Unicode return.

Graceful shutdown for extensions

Python 3.12 and later Python runtimes offer improved graceful shutdown capabilities for functions with external extensions. When Lambda shuts down an execution environment, it sends a `SIGTERM` signal to the runtime and then a `SHUTDOWN` event to each registered external extension. You can catch the `SIGTERM` signal in your Lambda function and clean up resources such as database connections that were created by the function.

To learn more about the execution environment lifecycle, see Understanding the Lambda execution environment lifecycle. For examples of how to use graceful shutdown with extensions, see the AWS Samples GitHub repository.

Define Lambda function handler in Python

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

This page describes how to work with Lambda function handlers in Python, including naming conventions, valid handler signatures, and code best practices. This page also includes an example of a Python Lambda function that takes in information about an order, produces a text file receipt, and puts this file in an Amazon Simple Storage Service (Amazon S3) bucket.

Topics

- Example Python Lambda function code
- Handler naming conventions
- Using the Lambda event object
- Accessing and using the Lambda context object
- Valid handler signatures for Python handlers
- Returning a value
- Using the AWS SDK for Python (Boto3) in your handler
- Accessing environment variables
- Code best practices for Python Lambda functions

Example Python Lambda function code

The following example Python Lambda function code takes in information about an order, produces a text file receipt, and puts this file in an Amazon S3 bucket:

Example Python Lambda function

```
import json
import os
import logging
import boto3

# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')
```

```
# Initialize the logger
logger = logging.getLogger()
logger.setLevel("INFO")

def upload_receipt_to_s3(bucket_name, key, receipt_content):
    """Helper function to upload receipt to S3"""

    try:
        s3_client.put_object(
            Bucket=bucket_name,
            Key=key,
            Body=receipt_content
        )
    except Exception as e:
        logger.error(f"Failed to upload receipt to S3: {str(e)}")
        raise

def lambda_handler(event, context):
    """
    Main Lambda handler function
    Parameters:
        event: Dict containing the Lambda function event data
        context: Lambda runtime context
    Returns:
        Dict containing status message
    """
    try:
        # Parse the input event
        order_id = event['Order_id']
        amount = event['Amount']
        item = event['Item']

        # Access environment variables
        bucket_name = os.environ.get('RECEIPT_BUCKET')
        if not bucket_name:
            raise ValueError("Missing required environment variable RECEIPT_BUCKET")

        # Create the receipt content and key destination
        receipt_content = (
            f"OrderID: {order_id}\n"
            f"Amount: ${amount}\n"
            f"Item: {item}"
        )
        key = f"receipts/{order_id}.txt"
```

```
        # Upload the receipt to S3
        upload_receipt_to_s3(bucket_name, key, receipt_content)

        logger.info(f"Successfully processed order {order_id} and stored receipt in S3
bucket {bucket_name}")

        return {
            "statusCode": 200,
            "message": "Receipt processed successfully"
        }

    except Exception as e:
        logger.error(f"Error processing order: {str(e)}")
        raise
```

This file contains the following sections of code:

- `import` block: Use this block to include libraries that your Lambda function requires.
- Global initialization of SDK client and logger: Including initialization code outside of the handler takes advantage of execution environment re-use to improve the performance of your function. See the section called "Code best practices for Python Lambda functions" to learn more.
- `def upload_receipt_to_s3(bucket_name, key, receipt_content):` This is a helper function that's called by the main `lambda_handler` function.
- `def lambda_handler(event, context):` This is the **main handler function** for your code, which contains your main application logic. When Lambda invokes your function handler, the Lambda runtime passes two arguments to the function, the event object that contains data for your function to process and the context object that contains information about the function invocation.

Handler naming conventions

The function handler name defined at the time that you create a Lambda function is derived from:

- The name of the file in which the Lambda handler function is located.
- The name of the Python handler function.

In the example above, if the file is named `lambda_function.py`, the handler would be specified as `lambda_function.lambda_handler`. This is the default handler name given to functions you create using the Lambda console.

If you create a function in the console using a different file name or function handler name, you must edit the default handler name.

To change the function handler name (console)

1. Open the Functions page of the Lambda console and choose your function.
2. Choose the **Code** tab.
3. Scroll down to the **Runtime settings** pane and choose **Edit**.
4. In **Handler**, enter the new name for your function handler.
5. Choose **Save**.

Using the Lambda event object

When Lambda invokes your function, it passes an event object argument to the function handler. JSON objects are the most common event format for Lambda functions. In the code example in the previous section, the function expects an input in the following format:

```
{
    "Order_id": "12345",
    "Amount": 199.99,
    "Item": "Wireless Headphones"
}
```

If your function is invoked by another AWS service, the input event is also a JSON object. The exact format of the event object depends on the service that's invoking your function. To see the event format for a particular service, refer to the appropriate page in the *Integrating other services* chapter.

If the input event is in the form of a JSON object, the Lambda runtime converts the object to a Python dictionary. To assign values in the input JSON to variables in your code, use the standard Python dictionary methods as illustrated in the example code.

You can also pass data into your function as a JSON array, or as any of the other valid JSON data types. The following table defines how the Python runtime converts these JSON types.

JSON data type	Python data type
object	dictionary (`dict`)
array	list (`list`)
number	integer (`int`) or floating point number (`float`)
string	string (`str`)
Boolean	Boolean (`bool`)
null	NoneType (`NoneType`)

Accessing and using the Lambda context object

The Lambda context object contains information about the function invocation and execution environment. Lambda passes the context object to your function automatically when it's invoked. You can use the context object to output information about your function's invocation for monitoring purposes.

The context object is a Python class that's defined in the Lambda runtime interface client. To return the value of any of the context object properties, use the corresponding method on the context object. For example, the following code snippet assigns the value of the `aws_request_id` property (the identifier for the invocation request) to a variable named `request`.

```
request = context.aws_request_id
```

To learn more about using the Lambda context object, and to see a complete list of the available methods and properties, see the section called “Context”.

Valid handler signatures for Python handlers

When defining your handler function in Python, the function must take two arguments. The first of these arguments is the Lambda event object and the second one is the Lambda context object. By convention, these input arguments are usually named `event` and `context`, but you can give them any names you wish. If you declare your handler function with a single input argument,

Lambda will raise an error when it attempts to run your function. The most common way to declare a handler function in Python is as follows:

```
def lambda_handler(event, context):
```

You can also use Python type hints in your function declaration, as shown in the following example:

```
from typing import Dict, Any

def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, Any]:
```

To use specific AWS typing for events generated by other AWS services and for the context object, add the `aws-lambda-typing` package to your function's deployment package. You can install this library in your development environment by running **`pip install aws-lambda-typing`**. The following code snippet shows how to use AWS-specific type hints. In this example, the expected event is an Amazon S3 event.

```
from aws_lambda_typing.events import S3Event
from aws_lambda_typing.context import Context
from typing import Dict, Any

def lambda_handler(event: S3Event, context: Context) -> Dict[str, Any]:
```

You can't use the Python `async` function type for your handler function.

Returning a value

Optionally, a handler can return a value, which must be JSON serializable. Common return types include `dict`, `list`, `str`, `int`, `float`, and `bool`.

What happens to the returned value depends on the invocation type and the service that invoked the function. For example:

- If you use the `RequestResponse` invocation type to invoke a Lambda function synchronously, Lambda returns the result of the Python function call to the client invoking the Lambda function (in the HTTP response to the invocation request, serialized into JSON). For example, AWS Lambda console uses the `RequestResponse` invocation type, so when you invoke the function on the console, the console will display the returned value.

- If the handler returns objects that can't be serialized by `json.dumps`, the runtime returns an error.
- If the handler returns None, as Python functions without a `return` statement implicitly do, the runtime returns `null`.
- If you use the `Event` invocation type (an asynchronous invocation), the value is discarded.

In the example code, the handler returns the following Python dictionary:

```
{
  "statusCode": 200,
  "message": "Receipt processed successfully"
}
```

The Lambda runtime serializes this dictionary and returns it to the client that invoked the function as a JSON string.

> **Note**
>
> In Python 3.9 and later releases, Lambda includes the requestId of the invocation in the error response.

Using the AWS SDK for Python (Boto3) in your handler

Often, you'll use Lambda functions to interact with other AWS services and resources. The simplest way to interface with these resources is to use the AWS SDK for Python (Boto3). All supported Lambda Python runtimes include a version of the SDK for Python. However, we strongly recommend that you include the SDK in your function's deployment package if your code needs to use it. Including the SDK in your deployment package gives you full control over your dependencies and reduces the risk of version misalignment issues with other libraries. See the section called "Runtime dependencies in Python" and the section called "Backward compatibility" to learn more.

To use the SDK for Python in your Lambda function, add the following statement to the import block at the beginning of your function code:

```
import boto3
```

Use the `pip install` command to add the `boto3` library to your function's deployment package. For detailed instructions on how to add dependencies to a .zip deployment package, see the section called "Creating a .zip deployment package with dependencies". To learn more about adding dependencies to Lambda functions deployed as container images, see the section called "Creating an image from a base image" or the section called "Creating an image from an alternative base image".

When using `boto3` in your code, you don't need to provide any credentials to initialize a client. For example, in the example code, we use the following line of code to initialize an Amazon S3 client:

```
# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')
```

With Python, Lambda automatically creates environment variables with credentials. The `boto3` SDK checks your function's environment variables for these credentials during initialization.

Accessing environment variables

In your handler code, you can reference environment variables by using the `os.environ.get` method. In the example code, we reference the defined `RECEIPT_BUCKET` environment variable using the following line of code:

```
# Access environment variables
bucket_name = os.environ.get('RECEIPT_BUCKET')
```

Don't forget to include an `import os` statement in the import block at the beginning of your code.

Code best practices for Python Lambda functions

Adhere to the guidelines in the following list to use best coding practices when building your Lambda functions:

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function. For example, in Python, this may look like:

```
def lambda_handler(event, context):
    foo = event['foo']
    bar = event['bar']
```

```
    result = my_lambda_function(foo, bar)

def my_lambda_function(foo, bar):
    // MyLambdaFunction logic here
```

- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries. For the Node.js and Python runtimes, these include the AWS SDKs. To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation.

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of

function invocations and escalated costs. If you see an unintended volume of invocations, set the function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

Working with .zip file archives for Python Lambda functions

Your AWS Lambda function's code comprises a .py file containing your function's handler code, together with any additional packages and modules your code depends on. To deploy this function code to Lambda, you use a *deployment package*. This package may either be a .zip file archive or a container image. For more information about using container images with Python, see Deploy Python Lambda functions with container images.

To create your deployment package as .zip file archive, you can use your command-line tool's built-in .zip file archive utility, or any other .zip file utility such as 7zip. The examples shown in the following sections assume you're using a command-line `zip` tool in a Linux or MacOS environment. To use the same commands in Windows, you can install the Windows Subsystem for Linux to get a Windows-integrated version of Ubuntu and Bash.

Note that Lambda uses POSIX file permissions, so you may need to set permissions for the deployment package folder before you create the .zip file archive.

Topics

- Runtime dependencies in Python
- Creating a .zip deployment package with no dependencies
- Creating a .zip deployment package with dependencies
- Dependency search path and runtime-included libraries
- Using __pycache__ folders
- Creating .zip deployment packages with native libraries
- Creating and updating Python Lambda functions using .zip files

Runtime dependencies in Python

For Lambda functions that use the Python runtime, a dependency can be any Python package or module. When you deploy your function using a .zip archive, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. A layer is a separate .zip file that can contain additional code and other content. To learn more about using Lambda layers in Python, see the section called "Layers".

The Lambda Python runtimes include the AWS SDK for Python (Boto3) and its dependencies. Lambda provides the SDK in the runtime for deployment scenarios where you are unable to add your own dependencies. These scenarios include creating functions in the console using the

built-in code editor or using inline functions in AWS Serverless Application Model (AWS SAM) or CloudFormation templates.

Lambda periodically updates the libraries in the Python runtime to include the latest updates and security patches. If your function uses the version of the Boto3 SDK included in the runtime but your deployment package includes SDK dependencies, this can cause version misalignment issues. For example, your deployment package could include the SDK dependency urllib3. When Lambda updates the SDK in the runtime, compatibility issues between the new version of the runtime and the version of urllib3 in your deployment package can cause your function to fail.

> **Important**
>
> To maintain full control over your dependencies and to avoid possible version misalignment issues, we recommend you add all of your function's dependencies to your deployment package, even if versions of them are included in the Lambda runtime. This includes the Boto3 SDK.

To find out which version of the SDK for Python (Boto3) is included in the runtime you're using, see the section called "Runtime-included SDK versions".

Under the AWS shared responsibility model, you are responsible for the management of any dependencies in your functions' deployment packages. This includes applying updates and security patches. To update dependencies in your function's deployment package, first create a new .zip file and then upload it to Lambda. See Creating a .zip deployment package with dependencies and Creating and updating Python Lambda functions using .zip files for more information.

Creating a .zip deployment package with no dependencies

If your function code has no dependencies, your .zip file contains only the .py file with your function's handler code. Use your preferred zip utility to create a .zip file with your .py file at the root. If the .py file is not at the root of your .zip file, Lambda won't be able to run your code.

To learn how to deploy your .zip file to create a new Lambda function or update an existing one, see Creating and updating Python Lambda functions using .zip files.

Creating a .zip deployment package with dependencies

If your function code depends on additional packages or modules, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. The instructions

in this section show you how to include your dependencies in your .zip deployment package. For Lambda to run your code, the .py file containing your handler code and all of your function's dependencies must be installed at the root of the .zip file.

Suppose your function code is saved in a file named `lambda_function.py`. The following example CLI commands create a .zip file named `my_deployment_package.zip` containing your function code and its dependencies. You can either install your dependencies directly to a folder in your project directory or use a Python virtual environment.

To create the deployment package (project directory)

1. Navigate to the project directory containing your `lambda_function.py` source code file. In this example, the directory is named `my_function`.

   ```
   cd my_function
   ```

2. Create a new directory named package into which you will install your dependencies.

   ```
   mkdir package
   ```

   Note that for a .zip deployment package, Lambda expects your source code and its dependencies all to be at the root of the .zip file. However, installing dependencies directly in your project directory can introduce a large number of new files and folders and make navigating around your IDE difficult. You create a separate `package` directory here to keep your dependencies separate from your source code.

3. Install your dependencies in the `package` directory. The example below installs the Boto3 SDK from the Python Package Index using pip. If your function code uses Python packages you have created yourself, save them in the `package` directory.

   ```
   pip install --target ./package boto3
   ```

4. Create a .zip file with the installed libraries at the root.

   ```
   cd package
   zip -r ../my_deployment_package.zip .
   ```

   This generates a `my_deployment_package.zip` file in your project directory.

5. Add the lambda_function.py file to the root of the .zip file

```
cd ..
zip my_deployment_package.zip lambda_function.py
```

Your .zip file should have a flat directory structure, with your function's handler code and all your dependency folders installed at the root as follows.

```
my_deployment_package.zip
|- bin
|  |-jp.py
|- boto3
|  |-compat.py
|  |-data
|  |-docs
...
|- lambda_function.py
```

If the .py file containing your function's handler code is not at the root of your .zip file, Lambda will not be able to run your code.

To create the deployment package (virtual environment)

1. Create and activate a virtual environment in your project directory. In this example the project directory is named `my_function`.

```
~$ cd my_function
~/my_function$ python3.14 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. Install your required libraries using pip. The following example installs the Boto3 SDK

```
(my_virtual_env) ~/my_function$ pip install boto3
```

3. Use `pip show` to find the location in your virtual environment where pip has installed your dependencies.

```
(my_virtual_env) ~/my_function$ pip show <package_name>
```

The folder in which pip installs your libraries may be named `site-packages` or `dist-packages`. This folder may be located in either the `lib/python3.x` or `lib64/python3.x` directory (where python3.x represents the version of Python you are using).

4. Deactivate the virtual environment

```
(my_virtual_env) ~/my_function$ deactivate
```

5. Navigate into the directory containing the dependencies you installed with pip and create a .zip file in your project directory with the installed dependencies at the root. In this example, pip has installed your dependencies in the `my_virtual_env/lib/python3.14/site-packages` directory.

```
~/my_function$ cd my_virtual_env/lib/python3.14/site-packages
~/my_function/my_virtual_env/lib/python3.14/site-packages$ zip -r ../../../../
my_deployment_package.zip .
```

6. Navigate to the root of your project directory where the .py file containing your handler code is located and add that file to the root of your .zip package. In this example, your function code file is named `lambda_function.py`.

```
~/my_function/my_virtual_env/lib/python3.14/site-packages$ cd ../../../../
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

Dependency search path and runtime-included libraries

When you use an `import` statement in your code, the Python runtime searches the directories in its search path until it finds the module or package. By default, the first location the runtime searches is the directory into which your .zip deployment package is decompressed and mounted (`/var/task`). If you include a version of a runtime-included library in your deployment package, your version will take precedence over the version that's included in the runtime. Dependencies in your deployment package also have precedence over dependencies in layers.

When you add a dependency to a layer, Lambda extracts this to `/opt/python/lib/python3.x/site-packages` (where `python3.x` represents the version of the runtime you're using) or `/opt/python`. In the search path, these directories have precedence over the directories containing the runtime-included libraries and pip-installed libraries (`/var/runtime` and `/var/lang/`

`lib/python3.x/site-packages`). Libraries in function layers therefore have precedence over versions included in the runtime.

> **Note**
>
> In the Python 3.11 managed runtime and base image, the AWS SDK and its dependencies are installed in the `/var/lang/lib/python3.11/site-packages` directory.

You can see the full search path for your Lambda function by adding the following code snippet.

```
import sys

search_path = sys.path
print(search_path)
```

> **Note**
>
> Because dependencies in your deployment package or layers take precedence over runtime-included libraries, this can cause version misalignment problems if you include an SDK dependency such as urllib3 in your package without including the SDK as well. If you deploy your own version of a Boto3 dependency, you must also deploy Boto3 as a dependency in your deployment package. We recommend that you package all of your function's dependencies, even if versions of them are included in the runtime.

You can also add dependencies in a separate folder inside your .zip package. For example, you might add a version of the Boto3 SDK to a folder in your .zip package called `common`. When your .zip package is decompressed and mounted, this folder is placed inside the `/var/task` directory. To use a dependency from a folder in your .zip deployment package in your code, use an `import from` statement. For example, to use a version of Boto3 from a folder named `common` in your .zip package, use the following statement.

```
from common import boto3
```

Using __pycache__ folders

We recommend that you don't include `__pycache__` folders in your function's deployment package. Python bytecode that's compiled on a build machine with a different architecture or operating system might not be compatible with the Lambda execution environment.

Creating .zip deployment packages with native libraries

If your function uses only pure Python packages and modules, you can use the `pip install` command to install your dependencies on any local build machine and create your .zip file. Many popular Python libraries, including NumPy and Pandas, are not pure Python and contain code written in C or C++. When you add libraries containing C/C++ code to your deployment package, you must build your package correctly to ensure that it's compatible with the Lambda execution environment.

Most packages available on the Python Package Index (PyPI) are available as "wheels" (.whl files). A .whl file is a type of ZIP file which contains a built distribution with pre-compiled binaries for a particular operating system and instruction set architecture. To make your deployment package compatible with Lambda, you install the wheel for Linux operating systems and your function's instruction set architecture.

Some packages may only be available as source distributions. For these packages, you need to compile and build the C/C++ components yourself.

To see what distributions are available for your required package, do the following:

1. Search for the name of the package on the Python Package Index main page.
2. Choose the version of the package you want to use.
3. Choose **Download files**.

Working with built distributions (wheels)

To download a wheel that's compatible with Lambda, you use the pip `--platform` option.

If your Lambda function uses the **x86_64** instruction set architecture, run the following `pip install` command to install a compatible wheel in your `package` directory. Replace `--python 3.x` with the version of the Python runtime you are using.

```
pip install \
```

```
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

If your function uses the **arm64** instruction set architecture, run the following command. Replace `--python 3.x` with the version of the Python runtime you are using.

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

Working with source distributions

If your package is only available as a source distribution, you need to build the C/C++ libraries yourself. To make your package compatible with the Lambda execution environment, you need to build it in an environment that uses the same Amazon Linux operating system. You can do this by building your package in an Amazon Elastic Compute Cloud (Amazon EC2) Linux instance.

To learn how to launch and connect to an Amazon EC2 Linux instance, see Get started with Amazon EC2 in the *Amazon EC2 User Guide*.

Creating and updating Python Lambda functions using .zip files

After you have created your .zip deployment package, you can use it to create a new Lambda function or update an existing one. You can deploy your .zip package using the Lambda console, the AWS Command Line Interface, and the Lambda API. You can also create and update Lambda functions using AWS Serverless Application Model (AWS SAM) and CloudFormation.

The maximum size for a .zip deployment package for Lambda is 250 MB (unzipped). Note that this limit applies to the combined size of all the files you upload, including any Lambda layers.

The Lambda runtime needs permission to read the files in your deployment package. In Linux permissions octal notation, Lambda needs 644 permissions for non-executable files (rw-r--r--) and 755 permissions (rwxr-xr-x) for directories and executable files.

In Linux and MacOS, use the `chmod` command to change file permissions on files and directories in your deployment package. For example, to give a non-executable file the correct permissions, run the following command.

```
chmod 644 <filepath>
```

To change file permissions in Windows, see Set, View, Change, or Remove Permissions on an Object in the Microsoft Windows documentation.

> **Note**
>
> If you don't grant Lambda the permissions it needs to access directories in your deployment package, Lambda sets the permissions for those directories to 755 (rwxr-xr-x).

Creating and updating functions with .zip files using the console

To create a new function, you must first create the function in the console, then upload your .zip archive. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip file.

If your .zip file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the AWS Management Console, see Getting started with Amazon S3. To upload files using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> You cannot change the deployment package type (.zip or container image) for an existing function. For example, you cannot convert a container image function to use a .zip file archive. You must create a new function.

To create a new function (console)

1. Open the Functions page of the Lambda console and choose **Create Function**.
2. Choose **Author from scratch**.

3. Under **Basic information**, do the following:

    a. For **Function name**, enter the name for your function.

    b. For **Runtime**, select the runtime you want to use.

    c. (Optional) For **Architecture**, choose the instruction set architecture for your function. The default architecture is x86_64. Ensure that the .zip deployment package for your function is compatible with the instruction set architecture you select.

4. (Optional) Under **Permissions**, expand **Change default execution role**. You can create a new **Execution role** or use an existing one.

5. Choose **Create function**. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip file for.

2. Select the **Code** tab.

3. In the **Code source** pane, choose **Upload from**.

4. Choose **.zip file**.

5. To upload the .zip file, do the following:

    a. Select **Upload**, then select your .zip file in the file chooser.

    b. Choose **Open**.

    c. Choose **Save**.

To upload a .zip archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip file for.

2. Select the **Code** tab.

3. In the **Code source** pane, choose **Upload from**.

4. Choose **Amazon S3 location**.

5. Paste the Amazon S3 link URL of your .zip file and choose **Save**.

Updating .zip file functions using the console code editor

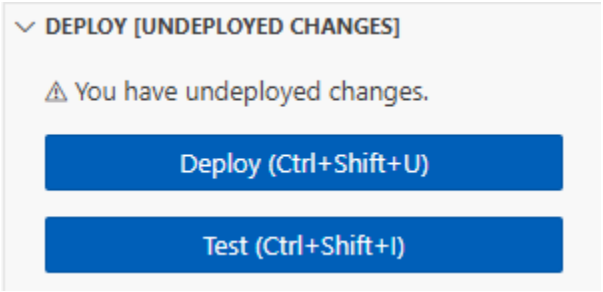
For some functions with .zip deployment packages, you can use the Lambda console's built-in code editor to update your function code directly. To use this feature, your function must meet the following criteria:

- Your function must use one of the interpreted language runtimes (Python, Node.js, or Ruby)
- Your function's deployment package must be smaller than 50 MB (unzipped).

Function code for functions with container image deployment packages cannot be edited directly in the console.

To update function code using the console code editor

1. Open the Functions page of the Lambda console and select your function.
2. Select the **Code** tab.
3. In the **Code source** pane, select your source code file and edit it in the integrated code editor.
4. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Creating and updating functions with .zip files using the AWS CLI

You can can use the AWS CLI to create a new function or to update an existing one using a .zip file. Use the create-function and update-function-code commands to deploy your .zip package. If your .zip file is smaller than 50MB, you can upload the .zip package from a file location on your local build machine. For larger files, you must upload your .zip package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> If you upload your .zip file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip file with the AWS CLI, you must specify the following:

- The name of your function (`--function-name`)
- Your function's runtime (`--runtime`)
- The Amazon Resource Name (ARN) of your function's execution role (`--role`)
- The name of the handler method in your function code (`--handler`)

You must also specify the location of your .zip file. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction \
--runtime python3.14 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--code` option as shown in the following example command. You only need to use the `S3ObjectVersion` parameter for versioned objects.

```
aws lambda create-function --function-name myFunction \
--runtime python3.14 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the `--function-name` parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda update-function-code --function-name myFunction \
```

```
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--s3-bucket` and `--s3-key` options as shown in the following example command. You only need to use the `--s3-object-version` parameter for versioned objects.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Creating and updating functions with .zip files using the Lambda API

To create and update functions using a .zip file archive, use the following API operations:

- CreateFunction
- UpdateFunctionCode

Creating and updating functions with .zip files using AWS SAM

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build, package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. To learn more about using AWS SAM to build and deploy Lambda functions, see Getting started with AWS SAM in the *AWS Serverless Application Model Developer Guide*.

You can also use AWS SAM to create a Lambda function using an existing .zip file archive. To create a Lambda function using AWS SAM, you can save your .zip file in an Amazon S3 bucket or in a local folder on your build machine. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

In your AWS SAM template, the `AWS::Serverless::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- `PackageType` - set to `Zip`
- `CodeUri` - set to the function code's Amazon S3 URI, path to local folder, or FunctionCode object

- `Runtime` - Set to your chosen runtime

With AWS SAM, if your .zip file is larger than 50MB, you don't need to upload it to an Amazon S3 bucket first. AWS SAM can upload .zip packages up to the maximum allowed size of 250MB (unzipped) from a location on your local build machine.

To learn more about deploying functions using .zip file in AWS SAM, see AWS::Serverless::Function in the *AWS SAM Developer Guide*.

Creating and updating functions with .zip files using CloudFormation

You can use CloudFormation to create a Lambda function using a .zip file archive. To create a Lambda function from a .zip file, you must first upload your file to an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide.*

For Node.js and Python runtimes, you can also provide inline source code in your CloudFormation template. CloudFormation then creates a .zip file containing your code when you build your function.

Using an existing .zip file

In your CloudFormation template, the `AWS::Lambda::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- `PackageType` - Set to `Zip`
- `Code` - Enter the Amazon S3 bucket name and the .zip file name in the `S3Bucket` and `S3Key` fields
- `Runtime` - Set to your chosen runtime

Creating a .zip file from inline code

You can declare simple functions written in Python or Node.js inline in an CloudFormation template. Because the code is embedded in YAML or JSON, you can't add any external dependenices to your deployment package. This means your function has to use the version of the AWS SDK that's included in the runtime. The requirements of the template, such as having to escape certain characters, also make it harder to use your IDE's syntax checking and code completion features. This means that your template might require additional testing. Because

of these limitations, declaring functions inline is best suited for very simple code that does not change frequently.

To create a .zip file from inline code for Node.js and Python runtimes, set the following properties in your template's `AWS::Lambda::Function` resource:

- `PackageType` - Set to `Zip`
- `Code` - Enter your function code in the `ZipFile` field
- `Runtime` - Set to your chosen runtime

The .zip file that CloudFormation generates cannot exceed 4MB. To learn more about deploying functions using .zip file in CloudFormation, see AWS::Lambda::Function in the *CloudFormation User Guide*.

Deploy Python Lambda functions with container images

There are three ways to build a container image for a Python Lambda function:

- Using an AWS base image for Python

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- Using an AWS OS-only base image

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the runtime interface client for Python in the image.

- Using a non-AWS base image

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the runtime interface client for Python in the image.

> **Tip**
>
> To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

This page explains how to build, test, and deploy container images for Lambda.

Topics

- AWS base images for Python
- Using an AWS base image for Python
- Using an alternative base image with the runtime interface client

AWS base images for Python

AWS provides the following base images for Python:

Tags	Runtime	Operating system	Dockerfile	Deprecation
3.14	Python 3.14	Amazon Linux 2023	Dockerfile for Python 3.14 on GitHub	Jun 30, 2029
3.13	Python 3.13	Amazon Linux 2023	Dockerfile for Python 3.13 on GitHub	Jun 30, 2029
3.12	Python 3.12	Amazon Linux 2023	Dockerfile for Python 3.12 on GitHub	Oct 31, 2028
3.11	Python 3.11	Amazon Linux 2	Dockerfile for Python 3.11 on GitHub	Jun 30, 2027
3.10	Python 3.10	Amazon Linux 2	Dockerfile for Python 3.10 on GitHub	Oct 31, 2026

Amazon ECR repository: gallery.ecr.aws/lambda/python

Python 3.12 and later base images are based on the Amazon Linux 2023 minimal container image. The Python 3.8-3.11 base images are based on the Amazon Linux 2 image. AL2023-based images provide several advantages over Amazon Linux 2, including a smaller deployment footprint and updated versions of libraries such as `glibc`.

AL2023-based images use `microdnf` (symlinked as `dnf`) as the package manager instead of `yum`, which is the default package manager in Amazon Linux 2. `microdnf` is a standalone implementation of `dnf`. For a list of packages that are included in AL2023-based images, refer to the **Minimal Container** columns in Comparing packages installed on Amazon Linux 2023 Container Images. For more information about the differences between AL2023 and Amazon Linux 2, see Introducing the Amazon Linux 2023 runtime for AWS Lambda on the AWS Compute Blog.

> **Note**
>
> To run AL2023-based images locally, including with AWS Serverless Application Model (AWS SAM), you must use Docker version 20.10.10 or later.

Dependency search path in the base images

When you use an `import` statement in your code, the Python runtime searches the directories in its search path until it finds the module or package. By default, the runtime searches the `{LAMBDA_TASK_ROOT}` directory first. If you include a version of a runtime-included library in your image, your version will take precedence over the version that's included in the runtime.

Other steps in the search path depend on which version of the Lambda base image for Python you're using:

- **Python 3.11 and later**: Runtime-included libraries and pip-installed libraries are installed in the `/var/lang/lib/python3.11/site-packages` directory. This directory has precedence over `/var/runtime` in the search path. You can override the SDK by using pip to install a newer version. You can use pip to verify that the runtime-included SDK and its dependencies are compatible with any packages that you install.
- **Python 3.8-3.10**: Runtime-included libraries are installed in the `/var/runtime` directory. Pip-installed libraries are installed in the `/var/lang/lib/python3.x/site-packages` directory. The `/var/runtime` directory has precedence over `/var/lang/lib/python3.x/site-packages` in the search path.

You can see the full search path for your Lambda function by adding the following code snippet.

```
import sys

search_path = sys.path
print(search_path)
```

Using an AWS base image for Python

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.
- Python

Creating an image from a base image

To create a container image from an AWS base image for Python

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
cd example
```

2. Create a new file called `lambda_function.py`. You can add the following sample function code to the file for testing, or use your own.

Example Python function

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. Create a new file called `requirements.txt`. If you're using the sample function code from the previous step, you can leave the file empty because there are no dependencies. Otherwise, list each required library. For example, here's what your `requirements.txt` should look like if your function uses the AWS SDK for Python (Boto3):

Example requirements.txt

```
boto3
```

4. Create a new Dockerfile with the following configuration:

- Set the `FROM` property to the URI of the base image.
- Use the `COPY` command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`, a Lambda-defined environment variable.
- Set the `CMD` argument to the Lambda function handler.

Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example Dockerfile

```
FROM public.ecr.aws/lambda/python:3.12

# Copy requirements.txt
COPY requirements.txt ${LAMBDA_TASK_ROOT}

# Install the specified packages
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.handler" ]
```

5. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

1. Start the Docker image with the **docker run** command. In this example, `docker-image` is the image name and `test` is the tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.

   Linux/macOS

   In Linux and macOS, run the following `curl` command:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   In PowerShell, run the following `Invoke-WebRequest` command:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Get the container ID.

```
docker ps
```

4. Use the docker kill command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Using an alternative base image with the runtime interface client

If you use an OS-only base image or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the Runtime API, which manages the interaction between Lambda and your function code.

Install the the runtime interface client for Python using the pip package manager:

```
pip install awslambdaric
```

You can also download the Python runtime interface client from GitHub.

The following example demonstrates how to build a container image for Python using a non-AWS base image. The example Dockerfile uses an official Python base image. The Dockerfile includes the runtime interface client for Python.

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)

- The Docker buildx plugin.
- Python

Creating an image from an alternative base image

To create a container image from a non-AWS base image

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
cd example
```

2. Create a new file called `lambda_function.py`. You can add the following sample function code to the file for testing, or use your own.

Example Python function

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. Create a new file called `requirements.txt`. If you're using the sample function code from the previous step, you can leave the file empty because there are no dependencies. Otherwise, list each required library. For example, here's what your `requirements.txt` should look like if your function uses the AWS SDK for Python (Boto3):

Example requirements.txt

```
boto3
```

4. Create a new Dockerfile. The following Dockerfile uses an official Python base image instead of an AWS base image. The Dockerfile includes the runtime interface client, which makes the image compatible with Lambda. The following example Dockerfile uses a multi-stage build.

   - Set the `FROM` property to the base image.
   - Set the `ENTRYPOINT` to the module that you want the Docker container to run when it starts. In this case, the module is the runtime interface client.
   - Set the `CMD` to the Lambda function handler.

Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM python:3.12 AS build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install \
    --target ${FUNCTION_DIR} \
        awslambdaric

# Use a slim version of the base Python image to reduce the final image size
FROM python:3.12-slim

# Include global arg in this stage of the build
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]
```

5. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test the image. You can build the emulator into your image or use the following procedure to install it on your local machine.

To install and run the runtime interface emulator on your local machine

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-
   runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
   download/aws-lambda-rie-arm64
   ```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

To install the arm64 emulator, replace the `$downloadLink` with the following:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Start the Docker image with the **docker run** command. Note the following:

- `docker-image` is the image name and `test` is the tag.
- `/usr/local/bin/python -m awslambdaric lambda_function.handler` is the ENTRYPOINT followed by the CMD from your Dockerfile.

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/python -m awslambdaric lambda_function.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/python -m awslambdaric lambda_function.handler
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

   Linux/macOS

   In Linux and macOS, run the following `curl` command:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   In PowerShell, run the following `Invoke-WebRequest` command:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. Get the container ID.

```
docker ps
```

5. Use the docker kill command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
```

```
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Example:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world:latest
   ```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

For an example of how to create a Python image from an Alpine base image, see Container image support for Lambda on the AWS Blog.

Working with layers for Python Lambda functions

Use Lambda layers to package code and dependencies that you want to reuse across multiple functions. Layers usually contain library dependencies, a custom runtime, or configuration files. Creating a layer involves three general steps:

1. Package your layer content. This means creating a .zip file archive that contains the dependencies you want to use in your functions.
2. Create the layer in Lambda.
3. Add the layer to your functions.

Topics

- Package your layer content
- Create the layer in Lambda
- Add the layer to your function
- Sample app

Package your layer content

To create a layer, bundle your packages into a .zip file archive that meets the following requirements:

- Build the layer using the same Python version that you plan to use for the Lambda function. For example, if you build your layer using Python 3.14, use the Python 3.14 runtime for your function.
- Your .zip file must include a `python` directory at the root level.
- The packages in your layer must be compatible with Linux. Lambda functions run on Amazon Linux.

You can create layers that contain either third-party Python libraries installed with `pip` (such as `requests` or `pandas`) or your own Python modules and packages.

Third-party dependencies

To create a layer using pip packages

1. Choose one of the following methods to install `pip` packages into the required top-level directory (`python/`):

   pip install

   For pure Python packages (like requests or boto3):

   ```
   pip install requests -t python/
   ```

   Some Python packages, such as NumPy and Pandas, include compiled C components. If you're building a layer with these packages on macOS or Windows, you might need to use this command to install a Linux-compatible wheel:

   ```
   pip install numpy --platform manylinux2014_x86_64 --only-binary=:all: -t python/
   ```

   For more information about working with Python packages that contain compiled components, see Creating .zip deployment packages with native libraries.

   requirements.txt

   Using a `requirements.txt` file helps you manage package versions and ensure consistent installations.

   Example requirements.txt

   ```
   requests==2.31.0
   boto3==1.37.34
   numpy==1.26.4
   ```

   If your `requirements.txt` file includes only pure Python packages (like requests or boto3):

   ```
   pip install -r requirements.txt -t python/
   ```

Some Python packages, such as NumPy and Pandas, include compiled C components. If you're building a layer with these packages on macOS or Windows, you might need to use this command to install a Linux-compatible wheel:

```
pip install -r requirements.txt --platform manylinux2014_x86_64 --only-
binary=:all: -t python/
```

For more information about working with Python packages that contain compiled components, see Creating .zip deployment packages with native libraries.

2. Zip the contents of the `python` directory.

Linux/macOS

```
zip -r layer.zip python/
```

PowerShell

```
Compress-Archive -Path .\python -DestinationPath .\layer.zip
```

The directory structure of your .zip file should look like this:

```
python/              # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

Note

If you use a Python virtual environment (venv) to install packages, your directory structure will be different (for example, `python/lib/python3.x/site-packages`). As long as your .zip file includes the `python` directory at the root level, Lambda can locate and import your packages.

Custom Python modules

To create a layer using your own code

1. Create the required top-level directory for your layer:

```
mkdir python
```

2. Create your Python modules in the `python` directory. The following example module validates orders by confirming that they contain the required information.

   Example custom module: validator.py

```
import json

def validate_order(order_data):
    """Validates an order and returns formatted data."""
    required_fields = ['product_id', 'quantity']

    # Check required fields
    missing_fields = [field for field in required_fields if field not in
 order_data]
    if missing_fields:
        raise ValueError(f"Missing required fields: {', '.join(missing_fields)}")

    # Validate quantity
    quantity = order_data['quantity']
    if not isinstance(quantity, int) or quantity < 1:
        raise ValueError("Quantity must be a positive integer")

    # Format and return the validated data
    return {
        'product_id': str(order_data['product_id']),
        'quantity': quantity,
        'shipping_priority': order_data.get('priority', 'standard')
    }

def format_response(status_code, body):
    """Formats the API response."""
    return {
        'statusCode': status_code,
        'body': json.dumps(body)
```

```
    }
```

3. Zip the contents of the `python` directory.

   Linux/macOS

   ```
   zip -r layer.zip python/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\python -DestinationPath .\layer.zip
   ```

   The directory structure of your .zip file should look like this:

   ```
   python/     # Required top-level directory
   ### validator.py
   ```

4. In your function, import and use the modules as you would with any Python package. Example:

   ```
   from validator import validate_order, format_response
   import json

   def lambda_handler(event, context):
       try:
           # Parse the order data from the event body
           order_data = json.loads(event.get('body', '{}'))

           # Validate and format the order
           validated_order = validate_order(order_data)

           return format_response(200, {
               'message': 'Order validated successfully',
               'order': validated_order
           })
       except ValueError as e:
           return format_response(400, {
               'error': str(e)
           })
       except Exception as e:
           return format_response(500, {
               'error': 'Internal server error'
   ```

```
        })
```

You can use the following test event to invoke the function:

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Expected response:

```
{
  "statusCode": 200,
  "body": "{\"message\": \"Order validated successfully\", \"order\": {\"product_id
\": \"ABC123\", \"quantity\": 2, \"shipping_priority\": \"express\"}}"
}
```

Create the layer in Lambda

You can publish your layer using either the AWS CLI or the Lambda console.

AWS CLI

Run the publish-layer-version AWS CLI command to create the Lambda layer:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes python3.14
```

The compatible runtimes parameter is optional. When specified, Lambda uses this parameter to filter layers in the Lambda console.

Console

To create a layer (console)

1. Open the Layers page of the Lambda console.
2. Choose **Create layer**.
3. Choose **Upload a .zip file**, and then upload the .zip archive that you created earlier.

4. (Optional) For **Compatible runtimes**, choose the Python runtime that corresponds to the Python version you used to build your layer.
5. Choose **Create**.

Add the layer to your function

AWS CLI

To attach the layer to your function, run the update-function-configuration AWS CLI command. For the `--layers` parameter, use the layer ARN. The ARN must specify the version (for example, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). For more information, see Layers and layer versions.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

Console

To add a layer to a function

1. Open the Functions page of the Lambda console.
2. Choose the function.
3. Scroll down to the **Layers** section, and then choose **Add a layer**.
4. Under **Choose a layer**, select **Custom layers**, and then choose your layer.

> **Note**
>
> If you didn't add a compatible runtime when you created the layer, your layer won't be listed here. You can specify the layer ARN instead.

5. Choose **Add**.

Sample app

For more examples of how to use Lambda layers, see the layer-python sample application in the AWS Lambda Developer Guide GitHub repository. This application includes two layers that contain Python libraries. After creating the layers, you can deploy and invoke the corresponding functions to confirm that the layers work as expected.

Using the Lambda context object to retrieve Python function information

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment. For more information on how the context object is passed to the function handler, see Define Lambda function handler in Python.

Context methods

- `get_remaining_time_in_millis` – Returns the number of milliseconds left before the execution times out.

Context properties

- `function_name` – The name of the Lambda function.
- `function_version` – The version of the function.
- `invoked_function_arn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_mb` – The amount of memory that's allocated for the function.
- `aws_request_id` – The identifier of the invocation request.
- `log_group_name` – The log group for the function.
- `log_stream_name` – The log stream for the function instance.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
  - `cognito_identity_id` – The authenticated Amazon Cognito identity.
  - `cognito_identity_pool_id` – The Amazon Cognito identity pool that authorized the invocation.
- `client_context` – (mobile apps) Client context that's provided to Lambda by the client application.
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`

- `client.app_package_name`
- `custom` – A `dict` of custom values set by the mobile client application.
- `env` – A `dict` of environment information provided by the AWS SDK.

Powertools for Lambda (Python) provides an interface definition for the Lambda context object. You can use the interface definition for type hints, or to further inspect the structure of the Lambda context object. For the interface definition, see lambda_context.py in the *powertools-lambda-python* repository on GitHub.

The following example shows a handler function that logs context information.

Example handler.py

```
import time

def lambda_handler(event, context):
    print("Lambda function ARN:", context.invoked_function_arn)
    print("CloudWatch log stream name:", context.log_stream_name)
    print("CloudWatch log group name:",  context.log_group_name)
    print("Lambda Request ID:", context.aws_request_id)
    print("Lambda function memory limits in MB:", context.memory_limit_in_mb)
    # We have added a 1 second delay so you can see the time remaining in
 get_remaining_time_in_millis.
    time.sleep(1)
    print("Lambda time remaining in MS:", context.get_remaining_time_in_millis())
```

In addition to the options listed above, you can also use the AWS X-Ray SDK for Instrumenting Python code in AWS Lambda to identify critical code paths, trace their performance and capture the data for analysis.

Log and monitor Python Lambda functions

AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function's code to the log stream. For more information about CloudWatch Logs, see Sending Lambda function logs to CloudWatch Logs.

To output logs from your function code, you can use the built-in `logging` module. For more detailed entries, you can use any logging library that writes to `stdout` or `stderr`.

Printing to the log

To send basic output to the logs, you can use a `print` method in your function. The following example logs the values of the CloudWatch Logs log group and stream, and the event object.

Note that if your function outputs logs using Python `print` statements, Lambda can only send log outputs to CloudWatch Logs in plain text format. To capture logs in structured JSON, you need to use a supported logging library. See the section called "Using Lambda advanced logging controls with Python" for more information.

Example lambda_function.py

```
import os
def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    print(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    print('## EVENT')
    print(event)
```

Example log output

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
/aws/lambda/my-function
2025/08/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c
## EVENT
{'key': 'value'}
```

```
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 147 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

The Python runtime logs the START, END, and REPORT lines for each invocation. The REPORT line includes the following data:

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function. When invocations share an execution environment, Lambda reports the maximum memory used across all invocations. This behavior might result in a higher than expected reported value.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using a logging library

For more detailed logs, use the logging module in the standard library, or any third party logging library that writes to `stdout` or `stderr`.

For supported Python runtimes, you can choose whether logs created using the standard `logging` module are captured in plain text or JSON. To learn more, see the section called “Using Lambda advanced logging controls with Python”.

Currently, the default log format for all Python runtimes is plain text. The following example shows how log outputs created using the standard `logging` module are captured in plain text in CloudWatch Logs.

```
import os
import logging
logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
    logger.info(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    logger.info(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    logger.info('## EVENT')
    logger.info(event)
```

The output from `logger` includes the log level, timestamp, and request ID.

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    /aws/
lambda/my-function
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    2025/01/31/
[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d
[INFO]  2025-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT
[INFO]  2025-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 117 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

Note

When your function's log format is set to plain text, the default log-level setting for Python runtimes is WARN. This means that Lambda only sends log outputs of level WARN and lower to CloudWatch Logs. To change the default log level, use the Python `logging` `setLevel()` method as shown in this example code. If you set your function's log format to JSON, we recommend that you configure your function's log level using Lambda Advanced Logging Controls and not by setting the log level in code. To learn more, see the section called “Using log-level filtering with Python”

Using Lambda advanced logging controls with Python

To give you more control over how your functions' logs are captured, processed, and consumed, you can configure the following logging options for supported Lambda Python runtimes:

- **Log format** - select between plain text and structured JSON format for your function's logs
- **Log level** - for logs in JSON format, choose the detail level of the logs Lambda sends to Amazon CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

For more information about these logging options, and instructions on how to configure your function to use them, see the section called "Configuring advanced logging controls for Lambda functions".

To learn more about using the log format and log level options with your Python Lambda functions, see the guidance in the following sections.

Using structured JSON logs with Python

If you select JSON for your function's log format, Lambda will send logs output by the Python standard logging library to CloudWatch as structured JSON. Each JSON log object contains at least four key value pairs with the following keys:

- `"timestamp"` - the time the log message was generated
- `"level"` - the log level assigned to the message
- `"message"` - the contents of the log message
- `"requestId"` - the unique request ID for the function invocation

The Python `logging` library can also add extra key value pairs such as `"logger"` to this JSON object.

The examples in the following sections show how log outputs generated using the Python `logging` library are captured in CloudWatch Logs when you configure your function's log format as JSON.

Note that if you use the `print` method to produce basic log outputs as described in the section called "Printing to the log", Lambda will capture these outputs as plain text, even if you configure your function's logging format as JSON.

Standard JSON log outputs using Python logging library

The following example code snippet and log output show how standard log outputs generated using the Python `logging` library are captured in CloudWatch Logs when your function's log format is set to JSON.

Example Python logging code

```
import logging
logger = logging.getLogger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON log record

```
{
    "timestamp":"2025-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Logging extra parameters in JSON

When your function's log format is set to JSON, you can also log additional parameters with the standard Python `logging` library by using the `extra` keyword to pass a Python dictionary to the log output.

Example Python logging code

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example JSON log record

```
{
  "timestamp": "2025-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [
    3
  ]
}
```

Logging exceptions in JSON

The following code snippet shows how Python exceptions are captured in your function's log output when you configure the log format as JSON. Note that log outputs generated using `logging.exception` are assigned the log level ERROR.

Example Python logging code

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON log record

```
{
  "timestamp": "2025-11-02T16:18:57Z",
  "level": "ERROR",
  "message": "msg",
  "logger": "root",
  "stackTrace": [
    "  File \"/var/task/lambda_function.py\", line 15, in lambda_handler\n    raise
 Exception(\"exception\")\n"
  ],
```

```
  "errorType": "Exception",
  "errorMessage": "exception",
  "requestId": "3f9d155c-0f09-46b7-bdf1-e91dab220855",
  "location": "/var/task/lambda_function.py:lambda_handler:17"
}
```

JSON structured logs with other logging tools

If your code already uses another logging library, such as Powertools for AWS Lambda, to produce JSON structured logs, you don't need to make any changes. AWS Lambda doesn't double-encode any logs that are already JSON encoded. Even if you configure your function to use the JSON log format, your logging outputs appear in CloudWatch in the JSON structure you define.

The following example shows how log outputs generated using the Powertools for AWS Lambda package are captured in CloudWatch Logs. The format of this log output is the same whether your function's logging configuration is set to JSON or TEXT. For more information about using Powertools for AWS Lambda, see the section called "Using Powertools for AWS Lambda (Python) and AWS SAM for structured logging" and the section called "Using Powertools for AWS Lambda (Python) and AWS CDK for structured logging"

Example Python logging code snippet (using Powertools for AWS Lambda)

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON log record (using Powertools for AWS Lambda)

```
{
    "level": "INFO",
    "location": "lambda_handler:7",
    "message": "Inside the handler function",
    "timestamp": "2025-10-31 22:38:21,010+0000",
    "service": "service_undefined",
    "xray_trace_id": "1-654181dc-65c15d6b0fecbdd1531ecb30"
}
```

Using log-level filtering with Python

By configuring log-level filtering, you can choose to send only logs of a certain logging level or lower to CloudWatch Logs. To learn how to configure log-level filtering for your function, see the section called "Log-level filtering".

For AWS Lambda to filter your application logs according to their log level, your function must use JSON formatted logs. You can achieve this in two ways:

- Create log outputs using the standard Python `logging` library and configure your function to use JSON log formatting. AWS Lambda then filters your log outputs using the "level" key value pair in the JSON object described in the section called "Using structured JSON logs with Python". To learn how to configure your function's log format, see the section called "Configuring advanced logging controls for Lambda functions".
- Use another logging library or method to create JSON structured logs in your code that include a "level" key value pair defining the level of the log output. For example, you can use Powertools for AWS Lambda to generate JSON structured log outputs from your code.

  You can also use a print statement to output a JSON object containing a log level identifier. The following print statement produces a JSON formatted output where the log level is set to INFO. AWS Lambda will send the JSON object to CloudWatch Logs if your function's logging level is set to INFO, DEBUG, or TRACE.

  ```
  print('{"msg":"My log message", "level":"info"}')
  ```

For Lambda to filter your function's logs, you must also include a `"timestamp"` key value pair in your JSON log output. The time must be specified in valid RFC 3339 timestamp format. If you don't supply a valid timestamp, Lambda will assign the log the level INFO and add a timestamp for you.

Viewing logs in Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Viewing logs in CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/*your-function-name***).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

Viewing logs with AWS CLI

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
```

```
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more

information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Using other logging tools and libraries

Powertools for AWS Lambda (Python) is a developer toolkit to implement Serverless best practices and increase developer velocity. The Logger utility provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON

- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC

Using Powertools for AWS Lambda (Python) and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated Powertools for Python modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.9
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Python template.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press `Enter`.

> **Note**
>
> For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter `y`.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the logs for the function, run sam logs. For more information, see Working with logs in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name sam-app
```

The log output looks like this:

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2025-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.114000 {
  "level": "INFO",
```

```
  "location": "hello:23",
  "message": "Hello world API - HTTP 200",
  "timestamp": "2025-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
```

```
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 756 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Using Powertools for AWS Lambda (Python) and AWS CDK for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated Powertools for AWS Lambda (Python) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.9
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world
```

```
cd hello-world
```

2. Initialize the app.

```
cdk init app --language python
```

3. Install the Python dependencies.

```
pip install -r requirements.txt
```

4. Create a directory **lambda_function** under the root folder.

```
mkdir lambda_function
cd lambda_function
```

5. Create a file **app.py** and add the following code to the file. This is the code for the Lambda function.

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/
metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/
logger/
```

```
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. Open the **hello_world** directory. You should see a file called **hello_world_stack.py**.

```
cd ..
cd hello_world
```

7. Open **hello_world_stack.py** and add the following code to the file. This contains the Lambda Constructor, which creates the Lambda function, configures environment variables for Powertools and sets log retention to one week, and the ApiGatewayv1 Constructor, which creates the REST API.

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # Using AWS Lambda Powertools via Lambda Layer
```

```
            # This imports the Powertools layer which provides observability
 features for Lambda functions
            # For available versions, see: https://docs.aws.amazon.com/powertools/
python/latest/#lambda-layer
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. Deploy your application.

```
cd ..
cdk deploy
```

9. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Invoke the API endpoint:

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

11. To get the logs for the function, run sam logs. For more information, see Working with logs in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name HelloWorldStack
```

The log output looks like this:

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2025-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
    "message": "Hello world API - HTTP 200",
  "timestamp": "2025-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
```

```
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
```

```
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 756 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

12. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

AWS Lambda function testing in Python

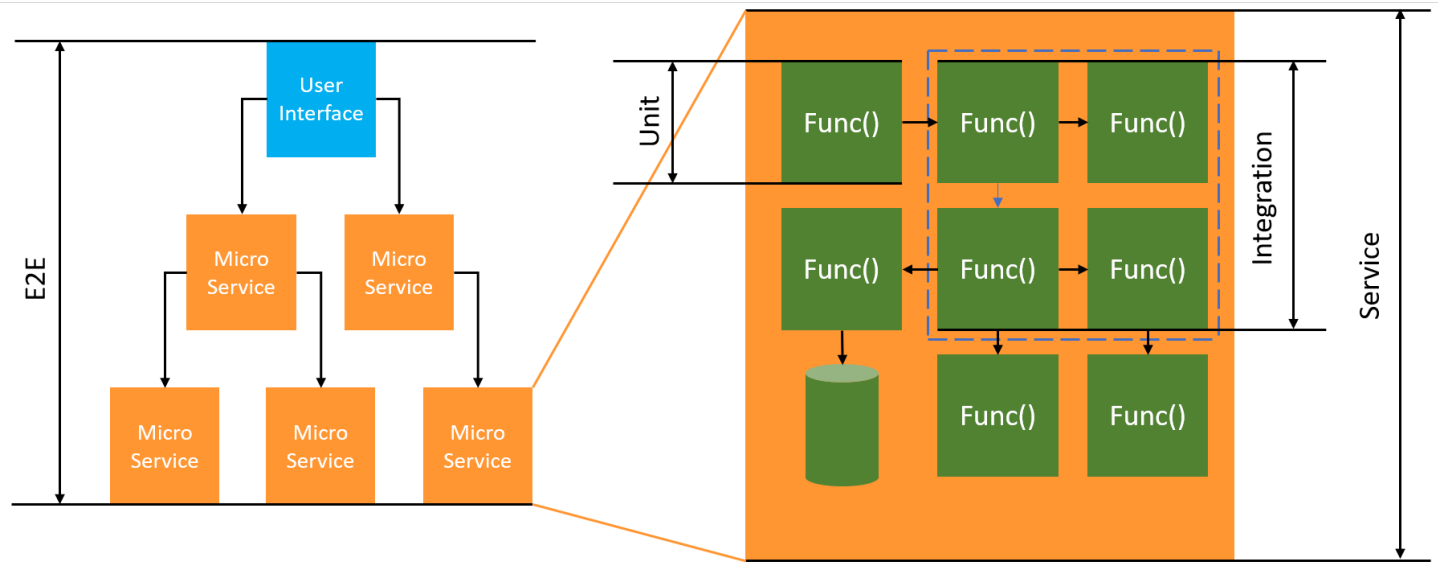
> **Note**
>
> See the Testing functions chapter for a complete introduction to techniques and best practices for testing serverless solutions.

Testing serverless functions uses traditional test types and techniques, but you must also consider testing serverless applications as a whole. Cloud-based tests will provide the **most accurate** measure of quality of both your functions and serverless applications.

A serverless application architecture includes managed services that provide critical application functionality through API calls. For this reason, your development cycle should include automated tests that verify functionality when your function and services interact.

If you do not create cloud-based tests, you could encounter issues due to differences between your local environment and the deployed environment. Your continuous integration process should run tests against a suite of resources provisioned in the cloud before promoting your code to the next deployment environment, such as QA, Staging, or Production.

Continue reading this short guide to learn about testing strategies for serverless applications, or visit the Serverless Test Samples repository to dive in with practical examples, specific to your chosen language and runtime.



For serverless testing, you will still write *unit*, *integration* and *end-to-end* tests.

- **Unit tests** - Tests that run against an isolated block of code. For example, verifying the business logic to calculate the delivery charge given a particular item and destination.
- **Integration tests** - Tests involving two or more components or services that interact, typically in a cloud environment. For example, verifying a function processes events from a queue.
- **End-to-end tests** - Tests that verify behavior across an entire application. For example, ensuring infrastructure is set up correctly and that events flow between services as expected to record a customer's order.

Testing your serverless applications

You will generally use a mix of approaches to test your serverless application code, including testing in the cloud, testing with mocks, and occasionally testing with emulators.

Testing in the cloud

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. You run tests against code deployed in the cloud and interacting with cloud-based services. This approach provides the **most accurate** measure of quality of your code.

A convenient way to debug your Lambda function in the cloud is through the console with a test event. A *test event* is a JSON input to your function. If your function does not require input, the event can be an empty JSON document (`{}`). The console provides sample events for a variety of service integrations. After creating an event in the console, you can share it with your team to make testing easier and consistent.

> **Note**
>
> Testing a function in the console is a quick way to get started, but automating your test cycles ensures application quality and development speed.

Testing tools

Tools and techniques exist to accelerate development feedback loops. For example, AWS SAM Accelerate and AWS CDK watch mode both decrease the time required to update cloud environments.

Moto is a Python library for mocking AWS services and resources, so that you can test your functions with little or no modification using decorators to intercept and simulate responses.

The validation feature of the Powertools for AWS Lambda (Python) provides decorators so you can validate input events and output responses from your Python functions.

For more information, read the blog post Unit Testing Lambda with Python and Mock AWS Services.

To reduce the latency involved with cloud deployment iterations, see AWS Serverless Application Model (AWS SAM) Accelerate, AWS Cloud Development Kit (AWS CDK) watch mode. These tools monitor your infrastructure and code for changes. They react to these changes by creating and deploying incremental updates automatically into your cloud environment.

Examples that use these tools are available in the Python Test Samples code repository.

Instrumenting Python code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- AWS Distro for OpenTelemetry (ADOT) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- AWS X-Ray SDK for Python – An SDK for generating and sending trace data to X-Ray.
- Powertools for AWS Lambda (Python) – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

> **Important**
>
> The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs.

Sections

- Using Powertools for AWS Lambda (Python) and AWS SAM for tracing
- Using Powertools for AWS Lambda (Python) and the AWS CDK for tracing
- Using ADOT to instrument your Python functions
- Using the X-Ray SDK to instrument your Python functions
- Activating tracing with the Lambda console
- Activating tracing with the Lambda API
- Activating tracing with CloudFormation

- Interpreting an X-Ray trace
- Storing runtime dependencies in a layer (X-Ray SDK)

Using Powertools for AWS Lambda (Python) and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated Powertools for AWS Lambda (Python) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.11
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Python template.

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.11 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press `Enter`.

> **Note**
>
> For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the traces for the function, run sam traces.

```
sam traces
```

The trace output looks like this:

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
```

```
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
    - total requests: 0
    - ok count(2XX): 0
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

Using Powertools for AWS Lambda (Python) and the AWS CDK for tracing

Follow the steps below to download, build, and deploy a sample Hello World Python application with integrated Powertools for AWS Lambda (Python) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics

using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Python 3.11
- AWS CLI version 2
- AWS CDK version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language python
```

3. Install the Python dependencies.

```
pip install -r requirements.txt
```

4. Create a directory **lambda_function** under the root folder.

```
mkdir lambda_function
cd lambda_function
```

5. Create a file **app.py** and add the following code to the file. This is the code for the Lambda function.

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
```

```
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. Open the **hello_world** directory. You should see a file called **hello_world_stack.py**.

```
cd ..
cd hello_world
```

7. Open **hello_world_stack.py** and add the following code to the file. This contains the Lambda Constructor, which creates the Lambda function, configures environment variables for Powertools and sets log retention to one week, and the ApiGatewayv1 Constructor, which creates the REST API.

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_11,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )
```

```
        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. Deploy your application.

```
cd ..
cdk deploy
```

9. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

11. To get the traces for the function, run sam traces.

```
sam traces
```

The traces output looks like this:

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
```

```
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0.924
   Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
  - Edges: []
    Summary_statistics:
      - total requests: 1
      - ok count(2XX): 1
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0.016
   Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
    Summary_statistics:
      - total requests: 0
      - ok count(2XX): 0
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

12. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Using ADOT to instrument your Python functions

ADOT provides fully managed Lambda layers that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see Custom configuration for the ADOT Collector on Lambda in the ADOT documentation.

For Python runtimes, you can add the **AWS managed Lambda layer for ADOT Python** to automatically instrument your functions. This layer works for both arm64 and x86_64 architectures. For detailed instructions on how to add this layer, see AWS Distro for OpenTelemetry Lambda Support for Python in the ADOT documentation.

Using the X-Ray SDK to instrument your Python functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Python. To get the SDK, add the `aws-xray-sdk` package to your application's dependencies.

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

In your function code, you can instrument AWS SDK clients by patching the `boto3` library with the `aws_xray_sdk.core` module.

Example function – Tracing an AWS SDK client

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES\r' + jsonpickle.encode(dict(**os.environ)))
  ...
```

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Under **Additional monitoring tools**, choose **Edit**.
5. Under **CloudWatch Application Signals and AWS X-Ray**, choose **Enable** for **Lambda service traces**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an CloudFormation template, use the `TracingConfig` property.

Example function-inline.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example template.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpreting an X-Ray trace

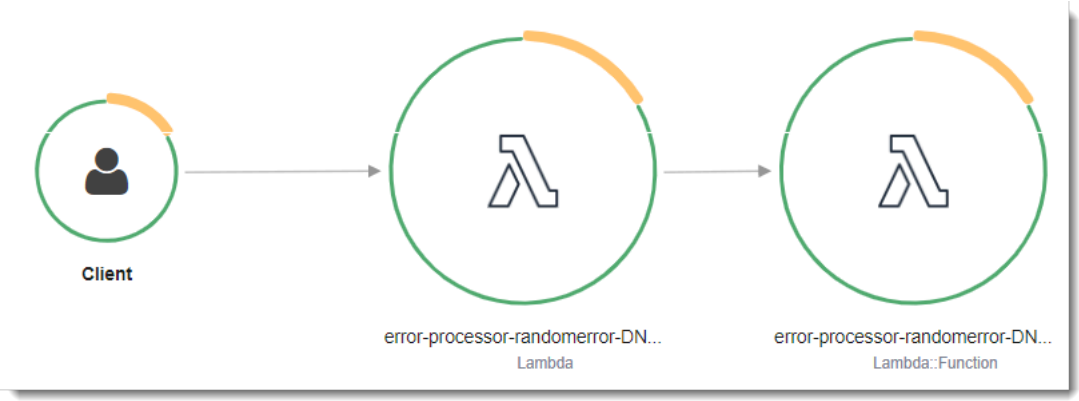
Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

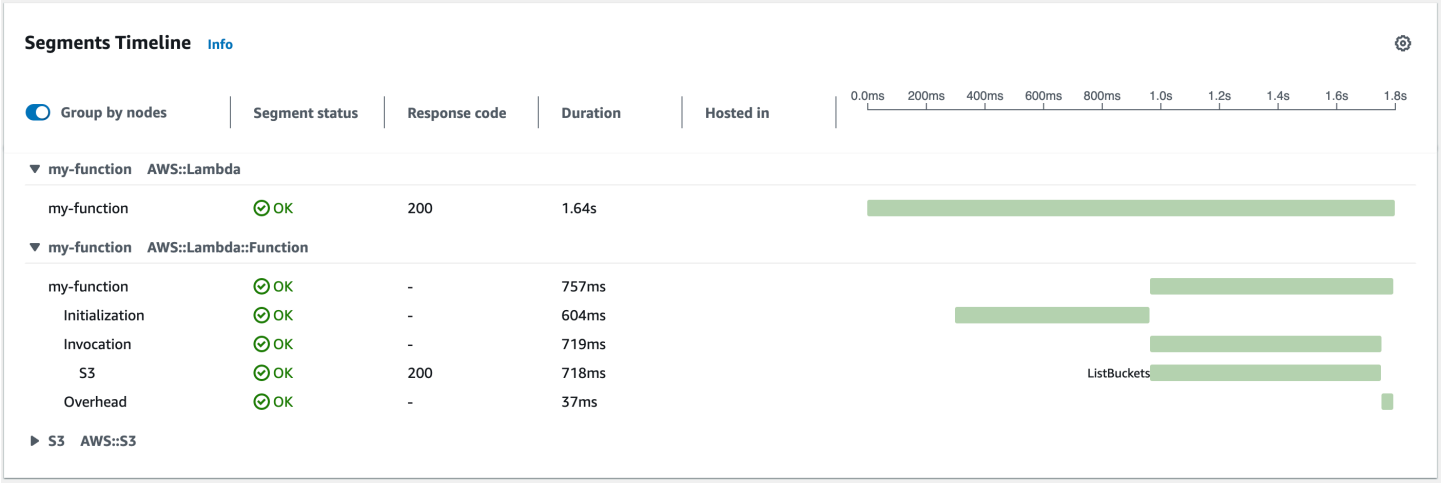


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

In X-Ray, a *trace* records information about a request that is processed by one or more *services*. Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes:



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.



This example expands the `AWS::Lambda::Function` segment to show its three subsegments.

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
> The example trace shown here illustrates the old-style function segment. The differences between the old- and new-style segments are described in the following paragraphs.
> These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

The old-style function segment contains the following subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

The new-style function segment doesn't contain an `Invocation` subsegment. Instead, customer subsegments are attached directly to the function segment. For more information about the structure of the old- and new-style function segments, see the section called “Understanding X-Ray traces”.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the AWS X-Ray SDK for Python in the *AWS X-Ray Developer Guide*.

> **Pricing**
>
> You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a Lambda layer.

The following example shows an `AWS::Serverless::LayerVersion` resource that stores the AWS X-Ray SDK for Python.

Example template.yml – Dependencies layer

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-python-lib
      Description: Dependencies for the blank-python sample app.
      ContentUri: package/.
      CompatibleRuntimes:
        - python3.11
```

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the blank-python sample application.

Building Lambda functions with Ruby

You can run Ruby code in AWS Lambda. Lambda provides runtimes for Ruby that run your code to process events. Your code runs in an environment that includes the AWS SDK for Ruby, with credentials from an AWS Identity and Access Management (IAM) role that you manage. To learn more about the SDK versions included with the Ruby runtimes, see the section called "Runtime-included SDK versions".

Lambda supports the following Ruby runtimes.

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Ruby 3.4	`ruby3.4`	Amazon Linux 2023	Mar 31, 2028	Apr 30, 2028	May 31, 2028
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	Mar 31, 2027	Apr 30, 2027	May 31, 2027
Ruby 3.2	`ruby3.2`	Amazon Linux 2	Mar 31, 2026	Aug 31, 2026	Sep 30, 2026

To create a Ruby function

1. Open the Lambda console.
2. Choose **Create function**.
3. Configure the following settings:
   - **Function name**: Enter a name for the function.
   - **Runtime**: Choose **Ruby 3.4**.
4. Choose **Create function**.

The console creates a Lambda function with a single source file named `lambda_function.rb`. You can edit this file and add more files in the built-in code editor. In the **DEPLOY** section, choose

Deploy to update your function's code. Then, to run your code, choose **Create test event** in the **TEST EVENTS** section.

The `lambda_function.rb` file exports a function named `lambda_handler` that takes an event object and a context object. This is the handler function that Lambda calls when the function is invoked. The Ruby function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is `lambda_function.lambda_handler`.

When you save your function code, the Lambda console creates a .zip file archive deployment package. When you develop your function code outside of the console (using an IDE) you need to create a deployment package to upload your code to the Lambda function.

The function runtime passes a context object to the handler, in addition to the invocation event. The context object contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any logs that your function outputs during invocation. If your function returns an error, Lambda formats the error and returns it to the invoker.

Topics

- Runtime-included SDK versions
- Enabling Yet Another Ruby JIT (YJIT)
- Define Lambda function handler in Ruby
- Deploy Ruby Lambda functions with .zip file archives
- Deploy Ruby Lambda functions with container images
- Working with layers for Ruby Lambda functions
- Using the Lambda context object to retrieve Ruby function information
- Log and monitor Ruby Lambda functions
- Instrumenting Ruby code in AWS Lambda

Runtime-included SDK versions

The version of the AWS SDK included in the Ruby runtime depends on the runtime version and your AWS Region. The AWS SDK for Ruby is designed to be modular and is separated by AWS

service. To find the version number of a particular service gem included in the runtime you're using, create a Lambda function with code in the following format. Replace `aws-sdk-s3` and `Aws::S3`with the name of the service gems your code uses.

```
require 'aws-sdk-s3'

def lambda_handler(event:, context:)
  puts "Service gem version: #{Aws::S3::GEM_VERSION}"
  puts "Core version: #{Aws::CORE_GEM_VERSION}"
end
```

Enabling Yet Another Ruby JIT (YJIT)

The Ruby runtimes support YJIT, a lightweight, minimalistic Ruby JIT compiler. YJIT provides significantly higher performance, but also uses more memory than the Ruby interpreter. YJIT is recommended for Ruby on Rails workloads.

YJIT is not enabled by default. To enable YJIT for a Ruby function, set the `RUBY_YJIT_ENABLE` environment variable to 1. To confirm that YJIT is enabled, print the result of the `RubyVM::YJIT.enabled?` method.

Example— Confirm that YJIT is enabled

```
puts(RubyVM::YJIT.enabled?())
# => true
```

Define Lambda function handler in Ruby

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

Topics

- Ruby handler basics
- Code best practices for Ruby Lambda functions

Ruby handler basics

In the following example, the file `function.rb` defines a handler method named `handler`. The handler function takes two objects as input and returns a JSON document.

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

In your function configuration, the `handler` setting tells Lambda where to find the handler. For the preceding example, the correct value for this setting is **`function.handler`**. It includes two names separated by a dot: the name of the file and the name of the handler method.

You can also define your handler method in a class. The following example defines a handler method named `process` on a class named `Handler` in a module named `LambdaFunctions`.

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
      "Hello!"
    end
  end
```

```
end
```

In this case, the handler setting is **`source.LambdaFunctions::Handler.process`**.

The two objects that the handler accepts are the invocation event and context. The event is a Ruby object that contains the payload that's provided by the invoker. If the payload is a JSON document, the event object is a Ruby hash. Otherwise, it's a string. The context object has methods and properties that provide information about the invocation, the function, and the execution environment.

The function handler is executed every time your Lambda function is invoked. Static code outside of the handler is executed once per instance of the function. If your handler uses resources like SDK clients and database connections, you can create them outside of the handler method to reuse them for multiple invocations.

Each instance of your function can process multiple invocation events, but it only processes one event at a time. The number of instances processing an event at any given time is your function's *concurrency*. For more information about the Lambda execution environment, see Understanding the Lambda execution environment lifecycle.

Code best practices for Ruby Lambda functions

Adhere to the guidelines in the following list to use best coding practices when building your Lambda functions:

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function. For example, in Ruby, this may look like:

```
def lambda_handler(event:, context:)
    foo = event['foo']
    bar = event['bar']

    result = my_lambda_function(foo:, bar:)

end

def my_lambda_function(foo:, bar:)
    // MyLambdaFunction logic here

end
```

- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries. For the Ruby runtime, these include the AWS SDK. To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation. For functions authored in Ruby, avoid uploading the entire AWS SDK library as part of your deployment package. Instead, selectively depend on the gems which pick up components of the SDK you need (e.g. the DynamoDB or Amazon S3 SDK gems).

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of function invocations and escalated costs. If you see an unintended volume of invocations, set the function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

Deploy Ruby Lambda functions with .zip file archives

Your AWS Lambda function's code comprises a .rb file containing your function's handler code, together with any additional dependencies (gems) your code depends on. To deploy this function code to Lambda, you use a *deployment package*. This package may either be a .zip file archive or a container image. For more information about using container images with Ruby, see Deploy Ruby Lambda functions with container images.

To create your deployment package as .zip file archive, you can use your command-line tool's built-in .zip file archive utility, or any other .zip file utility such as 7zip. The examples shown in the following sections assume you're using a command-line `zip` tool in a Linux or MacOS environment. To use the same commands in Windows, you can install the Windows Subsystem for Linux to get a Windows-integrated version of Ubuntu and Bash.

Note that Lambda uses POSIX file permissions, so you may need to set permissions for the deployment package folder before you create the .zip file archive.

The example commands in the following sections use the Bundler utility to add dependencies to your deployment package. To install bundler, run the following command.

```
gem install bundler
```

Sections

- Dependencies in Ruby
- Creating a .zip deployment package with no dependencies
- Creating a .zip deployment package with dependencies
- Creating a Ruby layer for your dependencies
- Creating .zip deployment packages with native libraries
- Creating and updating Ruby Lambda functions using .zip files

Dependencies in Ruby

For Lambda functions that use the Ruby runtime, a dependency can be any Ruby gem. When you deploy your function using a .zip archive, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. A layer is a separate .zip file that can contain additional code and other content. To learn more about using Lambda layers, see *Lambda layers*.

The Ruby runtime includes the AWS SDK for Ruby. If your function uses the SDK, you don't need to bundle it with your code. However, to maintain full control of your dependencies, or to use a specific version of the SDK, you can add it to your function's deployment package. You can either include the SDK in your .zip file, or add it using a Lambda layer. Dependencies in your .zip file or in Lambda layers take precedence over versions included in the runtime. To find out which version of the SDK for Ruby is included in your runtime version, see the section called “Runtime-included SDK versions”.

Under the AWS shared responsibility model, you are responsible for the management of any dependencies in your functions' deployment packages. This includes applying updates and security patches. To update dependencies in your function's deployment package, first create a new .zip file and then upload it to Lambda. See Creating a .zip deployment package with dependencies and Creating and updating Ruby Lambda functions using .zip files for more information.

Creating a .zip deployment package with no dependencies

If your function code has no dependencies, your .zip file contains only the .rb file with your function’s handler code. Use your preferred zip utility to create a .zip file with your .rb file at the root. If the .rb file is not at the root of your .zip file, Lambda won’t be able to run your code.

To learn how to deploy your .zip file to create a new Lambda function or update an existing one, see Creating and updating Ruby Lambda functions using .zip files.

Creating a .zip deployment package with dependencies

If your function code depends on additional Ruby gems, you can either add these dependencies to your .zip file with your function code or use a Lambda layer. The instructions in this section show you how to include dependencies in your .zip deployment package. For instructions on how to include your dependencies in a layer, see the section called “Creating a Ruby layer for your dependencies”.

Suppose your function code is saved in a file named `lambda_function.rb` in your project directory. The following example CLI commands create a .zip file named `my_deployment_package.zip` containing your function code and its dependencies.

To create the deployment package

1. In your project directory, create a `Gemfile` to specify your dependencies in.

```
bundle init
```

2. Using your preferred text editor, edit the `Gemfile` to specify your function's dependencies. For example, to use the TZInfo gem, edit your `Gemfile` to look like the following.

```
source "https://rubygems.org"
gem "tzinfo"
```

3. Run the following command to install the gems specified in your `Gemfile` in your project directory. This command sets `vendor/bundle` as the default path for gem installations.

```
bundle config set --local path 'vendor/bundle' && bundle install
```

You should see output similar to the following.

```
Fetching gem metadata from https://rubygems.org/..........
Resolving dependencies...
Using bundler 2.4.13
Fetching tzinfo 2.0.6
Installing tzinfo 2.0.6
...
```

> **Note**
>
> To install gems globally again later, run the following command.
>
> ```
> bundle config set --local system 'true'
> ```

4. Create a .zip file archive containing the `lambda_function.rb` file with your function's handler code and the dependencies you installed in the previous step.

```
zip -r my_deployment_package.zip lambda_function.rb vendor
```

You should see output similar to the following.

```
adding: lambda_function.rb (deflated 37%)
  adding: vendor/ (stored 0%)
  adding: vendor/bundle/ (stored 0%)
```

```
  adding: vendor/bundle/ruby/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/build_info/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
...
```

Creating a Ruby layer for your dependencies

To learn how to package your Ruby dependencies into a Lambda layer, see the section called "Layers".

Creating .zip deployment packages with native libraries

Many common Ruby gems such as `nokogiri`, `nio4r`, and `mysql` contain native extensions written in C. When you add libraries containing C code to your deployment package, you must build your package correctly to ensure that it's compatible with the Lambda execution environment.

For production applications, we recommend building and deploying your code using the AWS Serverless Application Model (AWS SAM). In AWS SAM use the `sam build --use-container` option to build your function inside a Lambda-like Docker container. To learn more about using AWS SAM to deploy your function code, see Building applications in the *AWS SAM Developer Guide*.

To create a .zip deployment package containing gems with native extensions without using AWS SAM, you can alternatively use a container to bundle your dependencies in an environment that is the same as the Lambda Ruby runtime environment. To complete these steps, you must have Docker installed on your build machine. To learn more about installing Docker, see Install Docker Engine.

To create a .zip deployment package in a Docker container

1. Create a folder on your local build machine to save your container in. Inside that folder, create a file named `dockerfile` and paste the following code into it.

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   CMD "/bin/bash"
   ```

2. Inside the folder you created your `dockerfile` in, run the following command to create the Docker container.

```
docker build -t awsruby32 .
```

3. Navigate to the project directory containing the `.rb` file with your function's handler code and the `Gemfile` specifying your function's dependencies. From inside that directory, run the following command to start the Lambda Ruby container.

   Linux/MacOS

   ```
   docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
   ```

   > **Note**
   >
   > In MacOS, you might see a warning informing you that the requested image's platform does not match the detected host platform. Ignore this warning.

   Windows PowerShell

   ```
   docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
   ```

   When your container starts, you should see a bash prompt.

   ```
   bash-4.2#
   ```

4. Configure the bundle utility to install the gems specified in your `Gemfile` in a local `vendor/bundle` directory and install your dependencies.

   ```
   bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
   ```

5. Create the .zip deployment package with your function code and its dependencies. In this example, the file containing your function's handler code is named `lambda_function.rb`.

   ```
   bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
   ```

6. Exit the container and return to your local project directory.

   ```
   bash-4.2# exit
   ```

You can now use the .zip file deployment package to create or update your Lambda function. See Creating and updating Ruby Lambda functions using .zip files

Creating and updating Ruby Lambda functions using .zip files

After you have created your .zip deployment package, you can use it to create a new Lambda function or update an existing one. You can deploy your .zip package using the Lambda console, the AWS Command Line Interface, and the Lambda API. You can also create and update Lambda functions using AWS Serverless Application Model (AWS SAM) and CloudFormation.

The maximum size for a .zip deployment package for Lambda is 250 MB (unzipped). Note that this limit applies to the combined size of all the files you upload, including any Lambda layers.

The Lambda runtime needs permission to read the files in your deployment package. In Linux permissions octal notation, Lambda needs 644 permissions for non-executable files (rw-r--r--) and 755 permissions (rwxr-xr-x) for directories and executable files.

In Linux and MacOS, use the `chmod` command to change file permissions on files and directories in your deployment package. For example, to give a non-executable file the correct permissions, run the following command.

```
chmod 644 <filepath>
```

To change file permissions in Windows, see Set, View, Change, or Remove Permissions on an Object in the Microsoft Windows documentation.

> **Note**
>
> If you don't grant Lambda the permissions it needs to access directories in your deployment package, Lambda sets the permissions for those directories to 755 (rwxr-xr-x).

Creating and updating functions with .zip files using the console

To create a new function, you must first create the function in the console, then upload your .zip archive. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip file.

If your .zip file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the AWS Management Console, see Getting started with Amazon S3. To upload files using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> You cannot change the deployment package type (.zip or container image) for an existing function. For example, you cannot convert a container image function to use a .zip file archive. You must create a new function.

To create a new function (console)

1. Open the Functions page of the Lambda console and choose **Create Function**.
2. Choose **Author from scratch**.
3. Under **Basic information**, do the following:
   a. For **Function name**, enter the name for your function.
   b. For **Runtime**, select the runtime you want to use.
   c. (Optional) For **Architecture**, choose the instruction set architecture for your function. The default architecture is x86_64. Ensure that the .zip deployment package for your function is compatible with the instruction set architecture you select.
4. (Optional) Under **Permissions**, expand **Change default execution role**. You can create a new **Execution role** or use an existing one.
5. Choose **Create function**. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **.zip file**.

5. To upload the .zip file, do the following:

    a. Select **Upload**, then select your .zip file in the file chooser.

    b. Choose **Open**.

    c. Choose **Save**.

To upload a .zip archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **Amazon S3 location**.
5. Paste the Amazon S3 link URL of your .zip file and choose **Save**.

Updating .zip file functions using the console code editor

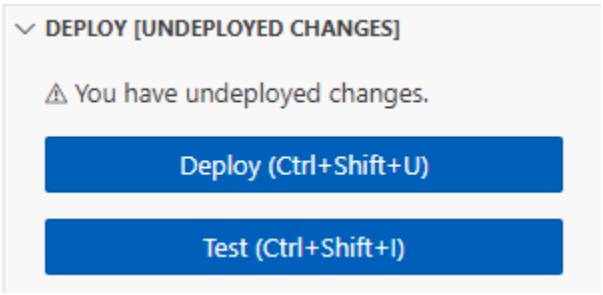
For some functions with .zip deployment packages, you can use the Lambda console's built-in code editor to update your function code directly. To use this feature, your function must meet the following criteria:

- Your function must use one of the interpreted language runtimes (Python, Node.js, or Ruby)
- Your function's deployment package must be smaller than 50 MB (unzipped).

Function code for functions with container image deployment packages cannot be edited directly in the console.

To update function code using the console code editor

1. Open the Functions page of the Lambda console and select your function.
2. Select the **Code** tab.
3. In the **Code source** pane, select your source code file and edit it in the integrated code editor.
4. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Creating and updating functions with .zip files using the AWS CLI

You can can use the AWS CLI to create a new function or to update an existing one using a .zip file. Use the create-function and update-function-code commands to deploy your .zip package. If your .zip file is smaller than 50MB, you can upload the .zip package from a file location on your local build machine. For larger files, you must upload your .zip package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> If you upload your .zip file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip file with the AWS CLI, you must specify the following:

- The name of your function (`--function-name`)
- Your function's runtime (`--runtime`)
- The Amazon Resource Name (ARN) of your function's execution role (`--role`)
- The name of the handler method in your function code (`--handler`)

You must also specify the location of your .zip file. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--code` option as shown in the following example command. You only need to use the `S3ObjectVersion` parameter for versioned objects.

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the `--function-name` parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--s3-bucket` and `--s3-key` options as shown in the following example command. You only need to use the `--s3-object-version` parameter for versioned objects.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Creating and updating functions with .zip files using the Lambda API

To create and update functions using a .zip file archive, use the following API operations:

- CreateFunction
- UpdateFunctionCode

Creating and updating functions with .zip files using AWS SAM

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build,

package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. To learn more about using AWS SAM to build and deploy Lambda functions, see Getting started with AWS SAM in the *AWS Serverless Application Model Developer Guide*.

You can also use AWS SAM to create a Lambda function using an existing .zip file archive. To create a Lambda function using AWS SAM, you can save your .zip file in an Amazon S3 bucket or in a local folder on your build machine. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

In your AWS SAM template, the `AWS::Serverless::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- `PackageType` - set to `Zip`
- `CodeUri` - set to the function code's Amazon S3 URI, path to local folder, or FunctionCode object
- `Runtime` - Set to your chosen runtime

With AWS SAM, if your .zip file is larger than 50MB, you don't need to upload it to an Amazon S3 bucket first. AWS SAM can upload .zip packages up to the maximum allowed size of 250MB (unzipped) from a location on your local build machine.

To learn more about deploying functions using .zip file in AWS SAM, see AWS::Serverless::Function in the *AWS SAM Developer Guide*.

Creating and updating functions with .zip files using CloudFormation

You can use CloudFormation to create a Lambda function using a .zip file archive. To create a Lambda function from a .zip file, you must first upload your file to an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide.*

In your CloudFormation template, the `AWS::Lambda::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- `PackageType` - Set to `Zip`
- `Code` - Enter the Amazon S3 bucket name and the .zip file name in the `S3Bucket` and `S3Key` fields
- `Runtime` - Set to your chosen runtime

The .zip file that CloudFormation generates cannot exceed 4MB. To learn more about deploying functions using .zip file in CloudFormation, see AWS::Lambda::Function in the *CloudFormation User Guide*.

Deploy Ruby Lambda functions with container images

There are three ways to build a container image for a Ruby Lambda function:

- Using an AWS base image for Ruby

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- Using an AWS OS-only base image

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the runtime interface client for Ruby in the image.

- Using a non-AWS base image

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the runtime interface client for Ruby in the image.

> **Tip**
>
> To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

This page explains how to build, test, and deploy container images for Lambda.

Topics

- AWS base images for Ruby
- Using an AWS base image for Ruby
- Using an alternative base image with the runtime interface client

AWS base images for Ruby

AWS provides the following base images for Ruby:

Tags	Runtime	Operating system	Dockerfile	Deprecation
3.4	Ruby 3.4	Amazon Linux 2023	Dockerfile for Ruby 3.4 on GitHub	Mar 31, 2028
3.3	Ruby 3.3	Amazon Linux 2023	Dockerfile for Ruby 3.3 on GitHub	Mar 31, 2027
3.2	Ruby 3.2	Amazon Linux 2	Dockerfile for Ruby 3.2 on GitHub	Mar 31, 2026

Amazon ECR repository: gallery.ecr.aws/lambda/ruby

Using an AWS base image for Ruby

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.
- Ruby

Creating an image from a base image

To create a container image for Ruby

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
cd example
```

2. Create a new file called `Gemfile`. This is where you list your application's required RubyGems packages. The AWS SDK for Ruby is available from RubyGems. You should choose specific AWS service gems to install. For example, to use the Ruby gem for Lambda, your Gemfile should look like this:

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

   Alternatively, the aws-sdk gem contains every available AWS service gem. This gem is very large. We recommend that you use it only if you depend on many AWS services.

3. Install the dependencies specified in the Gemfile using bundle install.

```
bundle install
```

4. Create a new file called `lambda_function.rb`. You can add the following sample function code to the file for testing, or use your own.

   Example Ruby function

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. Create a new Dockerfile. The following is an example Dockerfile that uses an AWS base image. This Dockerfiles uses the following configuration:

   - Set the `FROM` property to the URI of the base image.
   - Use the `COPY` command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`, a Lambda-defined environment variable.
   - Set the `CMD` argument to the Lambda function handler.

   Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-

privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example Dockerfile

```
FROM public.ecr.aws/lambda/ruby:3.4

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

1. Start the Docker image with the **docker run** command. In this example, `docker-image` is the image name and `test` is the tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.

   Linux/macOS

   In Linux and macOS, run the following `curl` command:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   In PowerShell, run the following `Invoke-WebRequest` command:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Get the container ID.

```
docker ps
```

4. Use the docker kill command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Using an alternative base image with the runtime interface client

If you use an OS-only base image or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the Runtime API, which manages the interaction between Lambda and your function code.

Install the Lambda runtime interface client for Ruby using the RubyGems.org package manager:

```
gem install aws_lambda_ric
```

You can also download the Ruby runtime interface client from GitHub.

The following example demonstrates how to build a container image for Ruby using a non-AWS base image. The example Dockerfile uses an official Ruby base image. The Dockerfile includes the runtime interface client.

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)

- The Docker buildx plugin.
- Ruby

Creating an image from an alternative base image

To create a container image for Ruby using an alternative base image

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
cd example
```

2. Create a new file called `Gemfile`. This is where you list your application's required RubyGems packages. The AWS SDK for Ruby is available from RubyGems. You should choose specific AWS service gems to install. For example, to use the Ruby gem for Lambda, your Gemfile should look like this:

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

   Alternatively, the aws-sdk gem contains every available AWS service gem. This gem is very large. We recommend that you use it only if you depend on many AWS services.

3. Install the dependencies specified in the Gemfile using bundle install.

```
bundle install
```

4. Create a new file called `lambda_function.rb`. You can add the following sample function code to the file for testing, or use your own.

   Example Ruby function

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. Create a new Dockerfile. The following Dockerfile uses a Ruby base image instead of an AWS base image. The Dockerfile includes the runtime interface client for Ruby, which makes the image compatible with Lambda. Alternatively, you can add the runtime interface client to your application's Gemfile.

   - Set the FROM property to the Ruby base image.
   - Create a directory for the function code and an environment variable that points to that directory. In this example, the directory is `/var/task`, which mirrors the Lambda execution environment. However, you can choose any directory for the function code because the Dockerfile doesn't use an AWS base image.
   - Set the ENTRYPOINT to the module that you want the Docker container to run when it starts. In this case, the module is the runtime interface client.
   - Set the CMD argument to the Lambda function handler.

   Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

   Example Dockerfile

```
FROM ruby:2.7

# Install the runtime interface client for Ruby
RUN gem install aws_lambda_ric

# Add the runtime interface client to the PATH
ENV PATH="/usr/local/bundle/bin:${PATH}"

# Create a directory for the Lambda function
ENV LAMBDA_TASK_ROOT=/var/task
RUN mkdir -p ${LAMBDA_TASK_ROOT}
WORKDIR ${LAMBDA_TASK_ROOT}

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
```

```
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "aws_lambda_ric" ]

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test the image. You can build the emulator into your image or use the following procedure to install it on your local machine.

To install and run the runtime interface emulator on your local machine

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.

Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-
runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

To install the arm64 emulator, replace the `$downloadLink` with the following:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Start the Docker image with the **docker run** command. Note the following:

   - `docker-image` is the image name and `test` is the tag.
   - `aws_lambda_ric lambda_function.LambdaFunction::Handler.process` is the `ENTRYPOINT` followed by the `CMD` from your Dockerfile.

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

Linux/macOS

In Linux and macOS, run the following `curl` command:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

In PowerShell, run the following `Invoke-WebRequest` command:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Get the container ID.

```
docker ps
```

5. Use the docker kill command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

- `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
- Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Working with layers for Ruby Lambda functions

Use Lambda layers to package code and dependencies that you want to reuse across multiple functions. Layers usually contain library dependencies, a custom runtime, or configuration files. Creating a layer involves three general steps:

1. Package your layer content. This means creating a .zip file archive that contains the dependencies you want to use in your functions.
2. Create the layer in Lambda.
3. Add the layer to your functions.

Topics

- Package your layer content
- Create the layer in Lambda
- Using gems from layers in a function
- Add the layer to your function
- Sample app

Package your layer content

To create a layer, bundle your packages into a .zip file archive that meets the following requirements:

- Create the layer using the same Ruby version that you plan to use for the Lambda function. For example, if you create your layer for Ruby 3.4, use the Ruby 3.4 runtime for your function.
- Your layer's .zip file must use one of these directory structures:
  - `ruby/gems/`*`x.x.x`* (where *`x.x.x`* is your Ruby version, for example `3.4.0`)
  - `ruby/lib`

  For more information, see Layer paths for each Lambda runtime.
- The packages in your layer must be compatible with Linux. Lambda functions run on Amazon Linux.

You can create layers that contain either third-party Ruby gems or your own Ruby modules and classes. Many popular Ruby gems contain native extensions (C code) that must be compiled for the Lambda Linux environment.

Pure Ruby gems

Pure Ruby gems contain only Ruby code and don't require compilation. These gems are simpler to package and work across different platforms.

To create a layer using pure Ruby gems

1. Create a `Gemfile` to specify the pure Ruby gems you want to include in your layer:

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'tzinfo'
   ```

2. Install the gems to `vendor/bundle` directory using Bundler:

   ```
   bundle config set --local path vendor/bundle
   bundle install
   ```

3. Copy the installed gems to the directory structure that Lambda requires `ruby/gems/3.4.0`):

   ```
   mkdir -p ruby/gems/3.4.0
   cp -r vendor/bundle/ruby/3.4.0*/* ruby/gems/3.4.0/
   ```

4. Zip the layer content:

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

   The directory structure of your .zip file should look like this:

```
ruby/
### gems/
    ### 3.4.0/
        ### gems/
        #   ### concurrent-ruby-1.3.5/
        #   ### tzinfo-2.0.6/
        ### specifications/
        ### cache/
        ### build_info/
        ### (other bundler directories)
```

> **Note**
>
> You must require each gem individually in your function code. You can't use `bundler/setup` or `Bundler.require`. For more information, see Using gems from layers in a function.

Gems with native extensions

Many popular Ruby gems contain native extensions (C code) that must be compiled for the target platform. Popular gems with native extensions include nokogiri, pg, mysql2, sqlite3, and ffi. These gems must be built in a Linux environment that is compatible with the Lambda runtime.

To create a layer using gems with native extensions

1. Create a `Gemfile`.

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'nokogiri'
   gem 'httparty'
   ```

2. Use Docker to build the gems in a Linux environment that is compatible with Lambda. Specify an AWS base image in your Dockerfile:

Example Dockerfile for Ruby 3.4

```
FROM public.ecr.aws/lambda/ruby:3.4

# Copy Gemfile
COPY Gemfile ./

# Install system dependencies for native extensions
RUN dnf update -y && \
    dnf install -y gcc gcc-c++ make

# Configure bundler and install gems
RUN bundle config set --local path vendor/bundle && \
    bundle install

# Create the layer structure
RUN mkdir -p ruby/gems/3.4.0 && \
    cp -r vendor/bundle/ruby/3.4.0*/* ruby/gems/3.4.0/

# Create the layer zip file
RUN zip -r layer.zip ruby/
```

3. Build the image and extract the layer:

```
docker build -t ruby-layer-builder .
docker run --rm -v $(pwd):/output --entrypoint cp ruby-layer-builder layer.zip /
output/
```

This builds the gems in the correct Linux environment and copies the `layer.zip` file to your local directory. The directory structure of your .zip file should look like this:

```
ruby/
### gems/
    ### 3.4.0/
        ### gems/
        #   ### bigdecimal-3.2.2/
        #   ### csv-3.3.5/
        #   ### httparty-0.23.1/
        #   ### mini_mime-1.1.5/
        #   ### multi_xml-0.7.2/
        #   ### nokogiri-1.18.8-x86_64-linux-gnu/
```

```
#   ### racc-1.8.1/
### build_info/
### cache/
### specifications/
### (other bundler directories)
```

> **Note**
>
> You must require each gem individually in your function code. You can't use `bundler/setup` or `Bundler.require`. For more information, see Using gems from layers in a function.

Custom Ruby modules

To create a layer using your own code

1. Create the required directory structure for your layer:

```
mkdir -p ruby/lib
```

2. Create your Ruby modules in the `ruby/lib` directory. The following example module validates orders by confirming that they contain the required information.

   Example ruby/lib/order_validator.rb

```
require 'json'

module OrderValidator
  class ValidationError < StandardError; end

  def self.validate_order(order_data)
    # Validates an order and returns formatted data
    required_fields = %w[product_id quantity]

    # Check required fields
    missing_fields = required_fields.reject { |field| order_data.key?(field) }
    unless missing_fields.empty?
      raise ValidationError, "Missing required fields: #{missing_fields.join(',
 ')}"
    end
```

```
    # Validate quantity
    quantity = order_data['quantity']
    unless quantity.is_a?(Integer) && quantity > 0
      raise ValidationError, 'Quantity must be a positive integer'
    end

    # Format and return the validated data
    {
      'product_id' => order_data['product_id'].to_s,
      'quantity' => quantity,
      'shipping_priority' => order_data.fetch('priority', 'standard')
    }
  end

  def self.format_response(status_code, body)
    # Formats the API response
    {
      statusCode: status_code,
      body: JSON.generate(body)
    }
  end
end
```

3. Zip the layer content:

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

   The directory structure of your .zip file should look like this:

   ```
   ruby/
   ### lib/
       ### order_validator.rb
   ```

4. In your function, require and use the modules. You must require each gem individually in your function code. You can't use `bundler/setup` or `Bundler.require`. For more information, see Using gems from layers in a function. Example:

```
require 'json'
require 'order_validator'

def lambda_handler(event:, context:)
  begin
    # Parse the order data from the event body
    order_data = JSON.parse(event['body'] || '{}')

    # Validate and format the order
    validated_order = OrderValidator.validate_order(order_data)

    OrderValidator.format_response(200, {
      message: 'Order validated successfully',
      order: validated_order
    })
  rescue OrderValidator::ValidationError => e
    OrderValidator.format_response(400, {
      error: e.message
    })
  rescue => e
    OrderValidator.format_response(500, {
      error: 'Internal server error'
    })
  end
end
```

You can use the following test event to invoke the function:

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

Expected response:

```
{
  "statusCode": 200,
```

```
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"product_id\":
\"ABC123\",\"quantity\":2,\"shipping_priority\":\"express\"}}"
}
```

Create the layer in Lambda

You can publish your layer using either the AWS CLI or the Lambda console.

AWS CLI

Run the publish-layer-version AWS CLI command to create the Lambda layer:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes ruby3.4
```

The compatible runtimes parameter is optional. When specified, Lambda uses this parameter to filter layers in the Lambda console.

Console

To create a layer (console)

1. Open the Layers page of the Lambda console.
2. Choose **Create layer**.
3. Choose **Upload a .zip file**, and then upload the .zip archive that you created earlier.
4. (Optional) For **Compatible runtimes**, choose the Ruby runtime that corresponds to the Ruby version you used to build your layer.
5. Choose **Create**.

Using gems from layers in a function

In your function code, you must explicitly require each gem that you want to use. Bundler commands such as `bundler/setup` and `Bundler.require` are not supported. Here's how to properly use gems from a layer in a Lambda function:

```
# Correct: Use explicit requires for each gem
```

```
require 'nokogiri'
require 'httparty'

def lambda_handler(event:, context:)
  # Use the gems directly
  doc = Nokogiri::HTML(event['html'])
  response = HTTParty.get(event['url'])
  # ... rest of your function
end

# Incorrect: These Bundler commands will not work
# require 'bundler/setup'
# Bundler.require
```

Add the layer to your function

AWS CLI

To attach the layer to your function, run the update-function-configuration AWS CLI command. For the `--layers` parameter, use the layer ARN. The ARN must specify the version (for example, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). For more information, see Layers and layer versions.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

Console

To add a layer to a function

1. Open the Functions page of the Lambda console.
2. Choose the function.
3. Scroll down to the **Layers** section, and then choose **Add a layer**.
4. Under **Choose a layer**, select **Custom layers**, and then choose your layer.

> **Note**
>
> If you didn't add a compatible runtime when you created the layer, your layer won't be listed here. You can specify the layer ARN instead.

5. Choose **Add**.

Sample app

For more examples of how to use Lambda layers, see the layer-ruby sample application in the AWS Lambda Developer Guide GitHub repository. This application includes a layer that contains the tzinfo library. After creating the layer, you can deploy and invoke the corresponding function to confirm that the layer works as expected.

Using the Lambda context object to retrieve Ruby function information

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context methods

- `get_remaining_time_in_millis` – Returns the number of milliseconds left before the execution times out.

Context properties

- `function_name` – The name of the Lambda function.
- `function_version` – The version of the function.
- `invoked_function_arn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `memory_limit_in_mb` – The amount of memory that's allocated for the function.
- `aws_request_id` – The identifier of the invocation request.
- `log_group_name` – The log group for the function.
- `log_stream_name` – The log stream for the function instance.
- `deadline_ms`– The date that the execution times out, in Unix time milliseconds.
- `identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `client_context`– (mobile apps) Client context that's provided to Lambda by the client application.

Log and monitor Ruby Lambda functions

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see Sending Lambda function logs to CloudWatch Logs.

This page describes how to produce log output from your Lambda function's code, and access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- Creating a function that returns logs
- Viewing logs in the Lambda console
- Viewing logs in the CloudWatch console
- Viewing logs using the AWS Command Line Interface (AWS CLI)
- Deleting logs
- Working with the Ruby logger library

Creating a function that returns logs

To output logs from your function code, you can use `puts` statements, or any logging library that writes to `stdout` or `stderr`. The following example logs the values of environment variables and the event object.

Example lambda_function.rb

```
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
    puts event.to_a
end
```

Example log format

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 147 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

The Ruby runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function. When invocations share an execution environment, Lambda reports the maximum memory used across all invocations. This behavior might result in a higher than expected reported value.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

For more detailed logs, use the the section called “Working with the Ruby logger library”.

Viewing logs in the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Viewing logs in the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/*your-function-name***).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

Viewing logs using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
```

```
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Working with the Ruby logger library

The Ruby logger library returns streamlined logs that are easily read. Use the logger utility to output detailed information, messages, and errors codes related to your function.

```
# lambda_function.rb
```

```
require 'logger'

def handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info('## ENVIRONMENT VARIABLES')
  logger.info(ENV.to_a)
  logger.info('## EVENT')
  logger.info(event)
  event.to_a
end
```

The output from `logger` includes the log level, timestamp, and request ID.

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES

[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125
    environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}

END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 117 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

Instrumenting Ruby code in AWS Lambda

Lambda integrates with AWS X-Ray to enable you to trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, from the frontend API to storage and database on the backend. By simply adding the X-Ray SDK library to your build configuration, you can record errors and latency for any call that your function makes to an AWS service.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.



To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Under **Additional monitoring tools**, choose **Edit**.
5. Under **CloudWatch Application Signals and AWS X-Ray**, choose **Enable** for **Lambda service traces**.

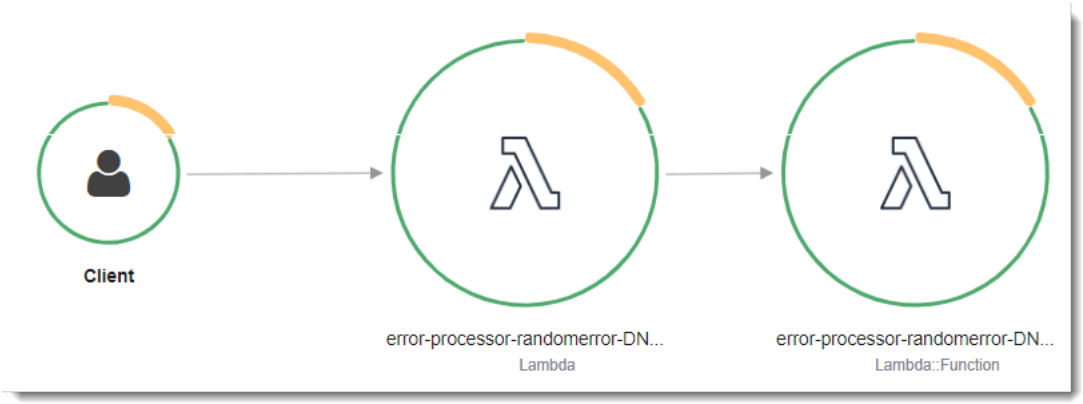
6. Choose **Save**.

> **Pricing**
>
> You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

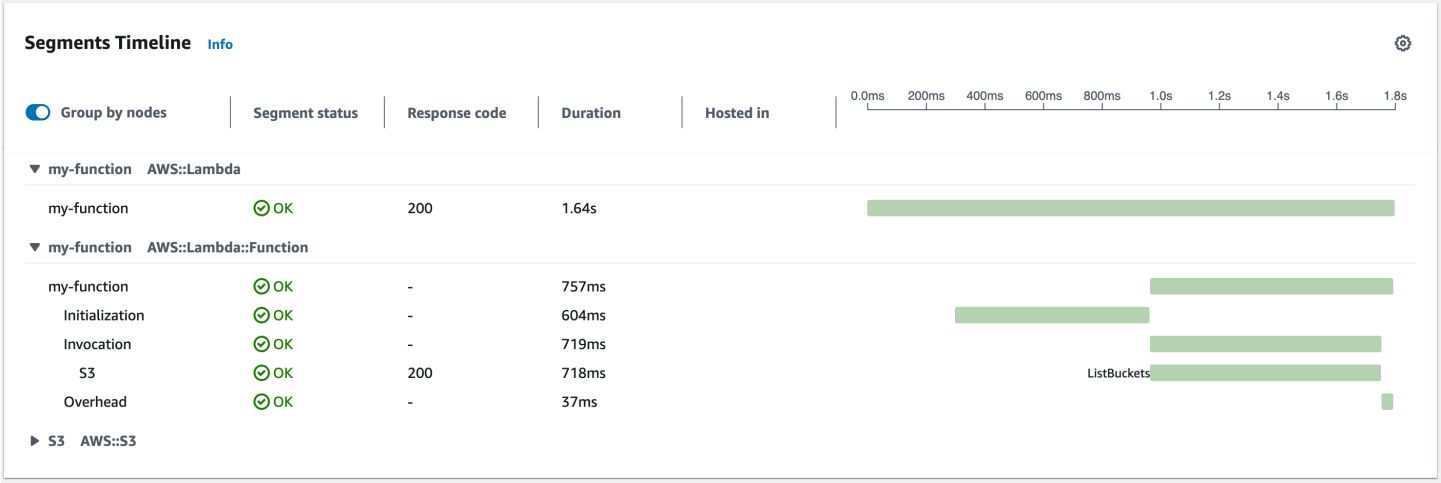
Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

In X-Ray, a *trace* records information about a request that is processed by one or more *services*. Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes:



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.



This example expands the `AWS::Lambda::Function` segment to show its three subsegments.

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
> The example trace shown here illustrates the old-style function segment. The differences between the old- and new-style segments are described in the following paragraphs.
> These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

The old-style function segment contains the following subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

The new-style function segment doesn't contain an `Invocation` subsegment. Instead, customer subsegments are attached directly to the function segment. For more information about the structure of the old- and new-style function segments, see the section called “Understanding X-Ray traces”.

You can instrument your handler code to record metadata and trace downstream calls. To record detail about calls that your handler makes to other resources and services, use the X-Ray SDK for Ruby. To get the SDK, add the `aws-xray-sdk` package to your application's dependencies.

Example blank-ruby/function/Gemfile

```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

To instrument AWS SDK clients, require the `aws-xray-sdk/lambda` module after creating a client in initialization code.

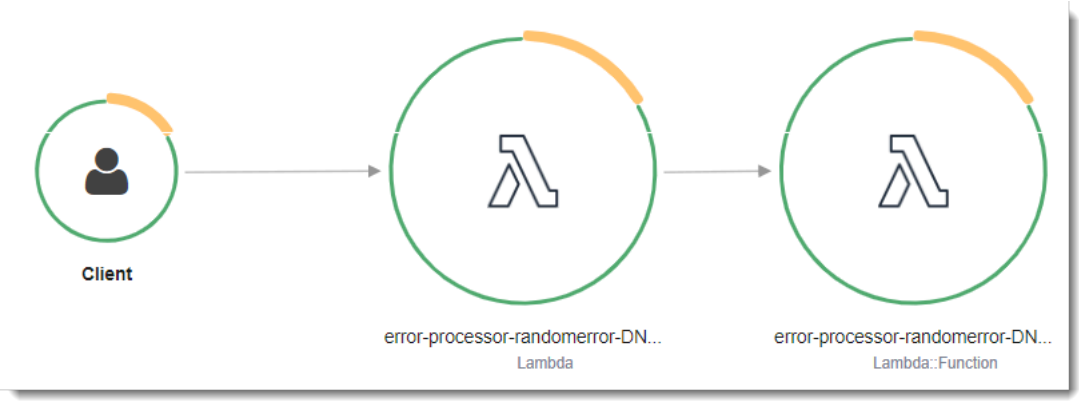
Example blank-ruby/function/lambda_function.rb – Tracing an AWS SDK client

```
# lambda_function.rb
require 'logger'
require 'json'
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

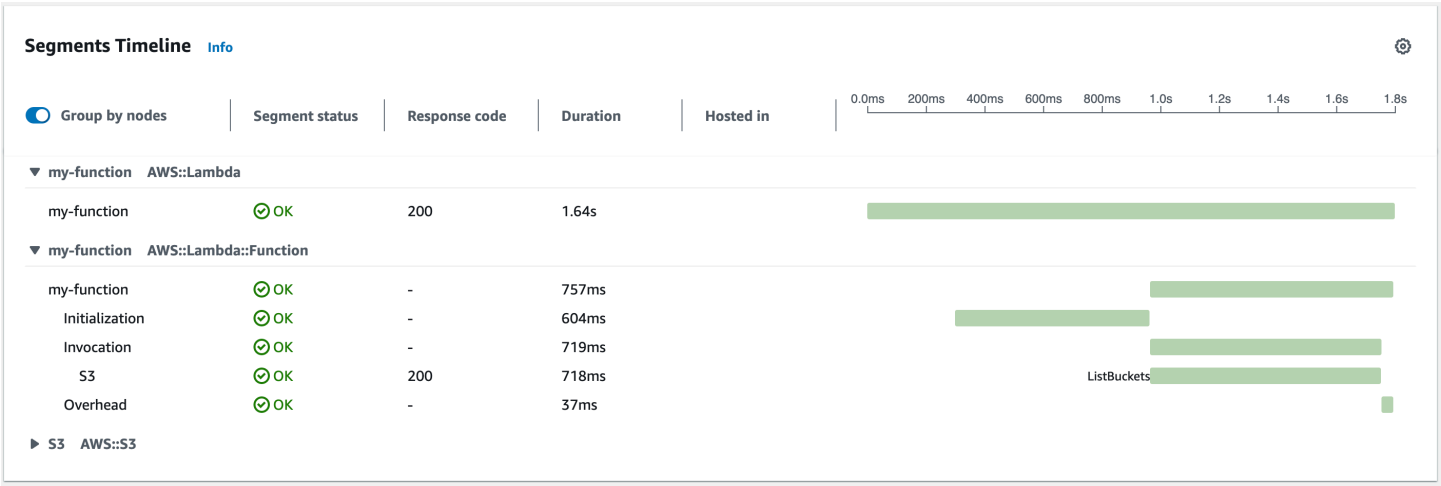
require 'aws-xray-sdk/lambda'

def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...
```

In X-Ray, a *trace* records information about a request that is processed by one or more *services*. Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes:



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.



This example expands the `AWS::Lambda::Function` segment to show its three subsegments.

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
> The example trace shown here illustrates the old-style function segment. The differences between the old- and new-style segments are described in the following paragraphs.

> These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

The old-style function segment contains the following subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

The new-style function segment doesn't contain an `Invocation` subsegment. Instead, customer subsegments are attached directly to the function segment. For more information about the structure of the old- and new-style function segments, see the section called "Understanding X-Ray traces".

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see The X-Ray SDK for Ruby in the AWS X-Ray Developer Guide.

Sections

- Enabling active tracing with the Lambda API
- Enabling active tracing with CloudFormation
- Storing runtime dependencies in a layer

Enabling active tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Enabling active tracing with CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an CloudFormation template, use the `TracingConfig` property.

Example function-inline.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example template.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Storing runtime dependencies in a layer

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a Lambda layer.

The following example shows an `AWS::Serverless::LayerVersion` resource that stores X-Ray SDK for Ruby.

Example template.yml – Dependencies layer

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-ruby-lib
      Description: Dependencies for the blank-ruby sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - ruby2.5
```

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build changes to generate the layer archive prior to deployment. For a working example, see the blank-ruby sample application.

Building Lambda functions with Java

You can run Java code in AWS Lambda. Lambda provides runtimes for Java that run your code to process events. Your code runs in an Amazon Linux environment that includes AWS credentials from an AWS Identity and Access Management (IAM) role that you manage.

Lambda supports the following Java runtimes.

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
Java 25	`java25`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
Java 21	`java21`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
Java 17	`java17`	Amazon Linux 2	Jun 30, 2027	Jul 31, 2027	Aug 31, 2027
Java 11	`java11`	Amazon Linux 2	Jun 30, 2027	Jul 31, 2027	Aug 31, 2027
Java 8	`java8.al2`	Amazon Linux 2	Jun 30, 2027	Jul 31, 2027	Aug 31, 2027

AWS provides the following libraries for Java functions. These libraries are available through Maven Central Repository.

- com.amazonaws:aws-lambda-java-core (required) – Defines handler method interfaces and the context object that the runtime passes to the handler. If you define your own input types, this is the only library that you need.
- com.amazonaws:aws-lambda-java-events – Input types for events from services that invoke Lambda functions.
- com.amazonaws:aws-lambda-java-log4j2 – An appender library for Apache Log4j 2 that you can use to add the request ID for the current invocation to your function logs.

- AWS SDK for Java 2.0 – The official AWS SDK for the Java programming language.

Add these libraries to your build definition as follows:

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

Important

Don't use private components of the JDK API, such as private fields, methods, or classes. Non-public API components can change or be removed in any update, causing your application to break.

To create a Java function

1. Open the Lambda console.
2. Choose **Create function**.
3. Configure the following settings:
   - **Function name**: Enter a name for the function.
   - **Runtime**: Choose **Java 25**.
4. Choose **Create function**.

The console creates a Lambda function with a handler class named `Hello`. Since Java is a compiled language, you can't view or edit the source code in the Lambda console, but you can modify its configuration, invoke it, and configure triggers.

> **Note**
>
> To get started with application development in your local environment, deploy one of the sample applications available in this guide's GitHub repository.

The `Hello` class has a function named `handleRequest` that takes an event object and a context object. This is the handler function that Lambda calls when the function is invoked. The Java function runtime gets invocation events from Lambda and passes them to the handler. In the function configuration, the handler value is `example.Hello::handleRequest`.

To update the function's code, you create a deployment package, which is a .zip file archive that contains your function code. As your function development progresses, you will want to store your function code in source control, add libraries, and automate deployments. Start by creating a deployment package and updating your code at the command line.

The function runtime passes a context object to the handler, in addition to the invocation event. The context object contains additional information about the invocation, the function, and the execution environment. More information is available from environment variables.

Your Lambda function comes with a CloudWatch Logs log group. The function runtime sends details about each invocation to CloudWatch Logs. It relays any logs that your function outputs during invocation. If your function returns an error, Lambda formats the error and returns it to the invoker.

Topics

- Define Lambda function handler in Java
- Deploy Java Lambda functions with .zip or JAR file archives
- Deploy Java Lambda functions with container images
- Working with layers for Java Lambda functions
- Customize serialization for Lambda Java functions
- Customize Java runtime startup behavior for Lambda functions
- Using the Lambda context object to retrieve Java function information
- Log and monitor Java Lambda functions
- Instrumenting Java code in AWS Lambda
- Java sample applications for AWS Lambda

Define Lambda function handler in Java

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

This page describes how to work with Lambda function handlers in Java, including options for project setup, naming conventions, and best practices. This page also includes an example of a Java Lambda function that takes in information about an order, produces a text file receipt, and puts this file in an Amazon Simple Storage Service (Amazon S3) bucket. For information about how to deploy your function after writing it, see the section called “Deploy .zip file archives” or the section called “Deploy container images”.

Sections

- Setting up your Java handler project
- Example Java Lambda function code
- Valid class definitions for Java handlers
- Handler naming conventions
- Defining and accessing the input event object
- Accessing and using the Lambda context object
- Using the AWS SDK for Java v2 in your handler
- Accessing environment variables
- Using global state
- Code best practices for Java Lambda functions

Setting up your Java handler project

When working with Lambda functions in Java, the process involves writing your code, compiling it, and deploying the compiled artifacts to Lambda. You can initialize a Java Lambda project in various ways. For instance, you can use tools like the Maven Archetype for Lambda functions, the AWS SAM CLI sam init command, or even a standard Java project setup in your preferred IDE, such as IntelliJ IDEA or Visual Studio Code. Alternatively, you can create the required file structure manually.

A typical Java Lambda function project follows this general structure:

```
/project-root
```

```
    # src
        # main
            # java
                # example
                    # OrderHandler.java (contains main handler)
                    # <other_supporting_classes>
     # build.gradle OR pom.xml
```

You can use either Maven or Gradle to build your project and manage dependencies.

The main handler logic for your function resides in a Java file under the `src/main/java/example` directory. In the example on this page, we name this file `OrderHandler.java`. Apart from this file, you can include additional Java classes as needed. When deploying your function to Lambda, make sure you specify the Java class that contains the main handler method that Lambda should invoke during an invocation.

Example Java Lambda function code

The following example Java 21 Lambda function code takes in information about an order, produces a text file receipt, and puts this file in an Amazon S3 bucket.

Example `OrderHandler.java` Lambda function

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;

import java.nio.charset.StandardCharsets;

/**
 * Lambda handler for processing orders and storing receipts in S3.
 */
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {

    private static final S3Client S3_CLIENT = S3Client.builder().build();

    /**
     * Record to model the input event.
```

```
     */
    public record Order(String orderId, double amount, String item) {}

    @Override
    public String handleRequest(Order event, Context context) {
        try {
            // Access environment variables
            String bucketName = System.getenv("RECEIPT_BUCKET");
            if (bucketName == null || bucketName.isEmpty()) {
                throw new IllegalArgumentException("RECEIPT_BUCKET environment variable
is not set");
            }

            // Create the receipt content and key destination
            String receiptContent = String.format("OrderID: %s\nAmount: $%.2f\nItem:
%s",
                    event.orderId(), event.amount(), event.item());
            String key = "receipts/" + event.orderId() + ".txt";

            // Upload the receipt to S3
            uploadReceiptToS3(bucketName, key, receiptContent);

            context.getLogger().log("Successfully processed order " + event.orderId() +
                    " and stored receipt in S3 bucket " + bucketName);
            return "Success";

        } catch (Exception e) {
            context.getLogger().log("Failed to process order: " + e.getMessage());
            throw new RuntimeException(e);
        }
    }

    private void uploadReceiptToS3(String bucketName, String key, String
receiptContent) {
        try {
            PutObjectRequest putObjectRequest = PutObjectRequest.builder()
                    .bucket(bucketName)
                    .key(key)
                    .build();

            // Convert the receipt content to bytes and upload to S3
            S3_CLIENT.putObject(putObjectRequest,
RequestBody.fromBytes(receiptContent.getBytes(StandardCharsets.UTF_8)));
        } catch (S3Exception e) {
```

```
            throw new RuntimeException("Failed to upload receipt to S3: " +
 e.awsErrorDetails().errorMessage(), e);
        }
    }
}
```

This `OrderHandler.java` file contains the following sections of code:

- `package example`: In Java, this can be anything, but it must match the directory structure of your project. Here, we use `package example` because the directory structure is `src/main/java/example`.
- `import` statements: Use these to import Java classes that your Lambda function requires.
- `public class OrderHandler ...`: This defines your Java class, and must be a valid class definition.
- `private static final S3Client S3_CLIENT ...`: This initializes an S3 client outside of any of the class's methods. This causes Lambda to run this code during the initialization phase.
- `public record Order ...`: Define the shape of the expected input event in this custom Java record.
- `public String handleRequest(Order event, Context context)`: This is the **main handler method**, which contains your main application logic.
- `private void uploadReceiptToS3(...) {}`: This is a helper method that's referenced by the main `handleRequest` handler method.

Sample build.gradle and pom.xml file

The following `build.gradle` or `pom.xml` file accompanies this function.

build.gradle

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
```

```
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.3'
    implementation 'software.amazon.awssdk:s3:2.28.29'
    implementation 'org.slf4j:slf4j-nop:2.0.16'
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtimeClasspath
    }
}

java {
    sourceCompatibility = JavaVersion.VERSION_21
    targetCompatibility = JavaVersion.VERSION_21
}

build.dependsOn buildZip
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>example-java</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>example-java-function</name>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.3</version>
        </dependency>
```

```
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>s3</artifactId>
            <version>2.28.29</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-nop</artifactId>
            <version>2.0.16</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.5.2</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <configuration>
                    <createDependencyReducedPom>false</createDependencyReducedPom>
                    <filters>
                        <filter>
                            <artifact>*:*</artifact>
                            <excludes>
                                <exclude>META-INF/*</exclude>
                                <exclude>META-INF/versions/**</exclude>
                            </excludes>
                        </filter>
                    </filters>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
```

```
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.13.0</version>
                <configuration>
                    <release>21</release>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

For this function to work properly, its execution role must allow the `s3:PutObject` action. Also, ensure that you define the `RECEIPT_BUCKET` environment variable. After a successful invocation, the Amazon S3 bucket should contain a receipt file.

> **Note**
>
> This function may require additional configuration settings to run successfully without timing out. We recommend configuring 256 MB of memory, and a 10 second timeout. The first invocation may take extra time due to a cold start. Subsequent invocations should run much faster due to reuse of the execution environment.

Valid class definitions for Java handlers

To define your class, the aws-lambda-java-core library defines two interfaces for handler methods. Use the provided interfaces to simplify handler configuration and validate the method signature at compile time.

- com.amazonaws.services.lambda.runtime.RequestHandler
- com.amazonaws.services.lambda.runtime.RequestStreamHandler

The `RequestHandler` interface is a generic type that takes two parameters: the input type and the output type. Both types must be objects. In this example, our `OrderHandler` class implements `RequestHandler<OrderHandler.Order, String>`. The input type is the `Order` record we define within the class, and the output type is `String`.

```
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {
```

```
    ...
}
```

When you use this interface, the Java runtime deserializes the event into the object with the input type, and serializes the output into text. Use this interface when the built-in serialization works with your input and output types.

To use your own serialization, you can implement the `RequestStreamHandler` interface. With this interface, Lambda passes your handler an input stream and output stream. The handler reads bytes from the input stream, writes to the output stream, and returns void. For an example of this using the Java 21 runtime, see HandlerStream.java.

If you're working only with basic and generic types (i.e. `String`, `Integer`, `List`, or `Map`) in your Java function , you don't need to implement an interface. For example, if your function takes in a `Map<String, String>` input and returns a `String`, your class definition and handler signature may look like the following:

```
public class ExampleHandler {
    public String handleRequest(Map<String, String> input, Context context) {
        ...
    }
}
```

In addition, when you don't implement an interface, the context object is optional. For example, your class definition and handler signature may look like the following:

```
public class NoContextHandler {
   public String handleRequest(Map<String, String> input) {
        ...
   }
}
```

Handler naming conventions

For Lambda functions in Java, if you are implementing either the `RequestHandler` or `RequestStreamHandler` interface, your main handler method must be named `handleRequest`. Also, include the `@Override` tag above your `handleRequest` method. When you deploy your function to Lambda, specify the main handler in your function's configuration in the following format:

- *<package>*.*<Class>* – For example, `example.OrderHandler`.

For Lambda functions in Java that don't implement the `RequestHandler` or `RequestStreamHandler` interface, you can use any name for the handler. When you deploy your function to Lambda, specify the main handler in your function's configuration in the following format:

- *<package>*.*<Class>*::*<handler_method_name>* – For example, `example.Handler::mainHandler`.

Defining and accessing the input event object

JSON is the most common and standard input format for Lambda functions. In this example, the function expects an input similar to the following:

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

When working with Lambda functions in Java 17 or newer, you can define the shape of the expected input event as a Java record. In this example, we define a record within the `OrderHandler` class to represent an `Order` object:

```
public record Order(String orderId, double amount, String item) {}
```

This record matches the expected input shape. After you define your record, you can write a handler signature that takes in a JSON input that conforms to the record definition. The Java runtime automatically deserializes this JSON into a Java object. You can then access the fields of the object. For example, `event.orderId` retrieves the value of `orderId` from the original input.

> **Note**
>
> Java records are a feature of Java 17 runtimes and newer only. In all Java runtimes, you can use a class to represent event data. In such cases, you can use a library like jackson to deserialize JSON inputs.

Other input event types

There are many possible input events for Lambda functions in Java:

- `Integer`, `Long`, `Double`, etc. – The event is a number with no additional formatting—for example, `3.5`. The Java runtime converts the value into an object of the specified type.
- `String` – The event is a JSON string, including quotes—for example, `"My string"`. The runtime converts the value into a `String` object without quotes.
- `List<Integer>`, `List<String>`, `List<Object>`, etc. – The event is a JSON array. The runtime deserializes it into an object of the specified type or interface.
- `InputStream` – The event is any JSON type. The runtime passes a byte stream of the document to the handler without modification. You deserialize the input and write output to an output stream.
- Library type – For events sent by other AWS services, use the types in the aws-lambda-java-events library. For example, if your Lambda function is invoked by Amazon Simple Queue Service (SQS), use the `SQSEvent` object as the input.

Accessing and using the Lambda context object

The Lambda context object contains information about the invocation, function, and execution environment. In this example, the context object is of type `com.amazonaws.services.lambda.runtime.Context`, and is the second argument of the main handler function.

```
public String handleRequest(Order event, Context context) {
    ...
}
```

If your class implements either the RequestHandler or RequestStreamHandler interface, the context object is a required argument. Otherwise, the context object is optional. For more information about valid accepted handler signatures, see the section called "Valid class definitions for Java handlers".

If you make calls to other services using the AWS SDK, the context object is required in a few key areas. For example, to produce function logs for Amazon CloudWatch, you can use the `context.getLogger()` method to get a `LambdaLogger` object for logging. In this example, we can use the logger to log an error message if processing fails for any reason:

```
context.getLogger().log("Failed to process order: " + e.getMessage());
```

Outside of logging, you can also use the context object for function monitoring. For more information about the context object, see the section called “Context”.

Using the AWS SDK for Java v2 in your handler

Often, you'll use Lambda functions to interact with or make updates to other AWS resources. The simplest way to interface with these resources is to use the AWS SDK for Java v2.

> **Note**
>
> The AWS SDK for Java (v1) is in maintenance mode, and will reach end-of-support on December 31, 2025. We recommend that you use only the AWS SDK for Java v2 going forward.

To add SDK dependencies to your function, add them in your `build.gradle` for Gradle or `pom.xml` file for Maven. We recommend only adding the libraries that you need for your function. In the example code earlier, we used the `software.amazon.awssdk.services.s3` library. In Gradle, you can add this dependency by adding the following line in the dependencies section of your `build.gradle`:

```
implementation 'software.amazon.awssdk:s3:2.28.29'
```

In Maven, add the following lines in the `<dependencies>` section of your `pom.xml`:

```
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>s3</artifactId>
        <version>2.28.29</version>
    </dependency>
```

> **Note**
>
> This may not be the most recent version of the SDK. Choose the appropriate version of the SDK for your application.

Then, import the dependencies directly in your Java class:

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;
```

The example code then initializes an Amazon S3 client as follows:

```
private static final S3Client S3_CLIENT = S3Client.builder().build();
```

In this example, we initialized our Amazon S3 client outside of the main handler function to avoid having to initialize it every time we invoke our function. After you initialize your SDK client, you can then use it to interact with other AWS services. The example code calls the Amazon S3 `PutObject` API as follows:

```
PutObjectRequest putObjectRequest = PutObjectRequest.builder()
    .bucket(bucketName)
    .key(key)
    .build();

// Convert the receipt content to bytes and upload to S3
S3_CLIENT.putObject(putObjectRequest,
 RequestBody.fromBytes(receiptContent.getBytes(StandardCharsets.UTF_8)));
```

Accessing environment variables

In your handler code, you can reference any environment variables by using the `System.getenv()` method. In this example, we reference the defined `RECEIPT_BUCKET` environment variable using the following line of code:

```
String bucketName = System.getenv("RECEIPT_BUCKET");
if (bucketName == null || bucketName.isEmpty()) {
    throw new IllegalArgumentException("RECEIPT_BUCKET environment variable is not
 set");
}
```

Using global state

Lambda runs your static code and the class constructor during the initialization phase before invoking your function for the first time. Resources created during initialization stay in memory between invocations, so you can avoid having to create them every time you invoke your function.

In the example code, the S3 client initialization code is outside the main handler method. The runtime initializes the client before the function handles its first event, and the client remains available for reuse across all invocations.

Code best practices for Java Lambda functions

Adhere to the guidelines in the following list to use best coding practices when building your Lambda functions:

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function.
- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries. To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup. For example, prefer simpler Java dependency injection (IoC) frameworks like Dagger or Guice, over more complex ones like Spring Framework.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation. For functions authored in Java, avoid uploading the entire AWS SDK library as part of your deployment package. Instead, selectively depend on the modules which pick up components of the SDK you need (e.g. DynamoDB, Amazon S3 SDK modules and Lambda core libraries).

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of function invocations and escalated costs. If you see an unintended volume of invocations, set the function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

- **Avoid using the Java DNS cache.** Lambda functions already cache DNS responses. If you use another DNS cache, then you might experience connection timeouts.

  The `java.util.logging.Logger` class can indirectly enable the JVM DNS cache. To override the default settings, set networkaddress.cache.ttl to 0 before initializing `logger`. Example:

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
```

```
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

- **Reduce the time it takes Lambda to unpack deployment packages** authored in Java by putting your dependency `.jar` files in a separate /lib directory. This is faster than putting all your function's code in a single jar with a large number of `.class` files. See Deploy Java Lambda functions with .zip or JAR file archives for instructions.

Deploy Java Lambda functions with .zip or JAR file archives

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

This page describes how to create your deployment package as a .zip file or Jar file, and then use the deployment package to deploy your function code to AWS Lambda using the AWS Command Line Interface (AWS CLI).

Important

Java 25 introduced support for Ahead-of-Time (AOT) caches. We strongly recommend not using AOT caches when deploying your functions as .zip or JAR file archives, since the caches may cause unexpected behavior when Lambda updates the managed runtime. For further information, see Ahead-of-Time (AOT) and CDS caches.

Sections

- Prerequisites
- Tools and libraries
- Building a deployment package with Gradle
- Using layers for dependencies
- Building a deployment package with Maven
- Uploading a deployment package with the Lambda console
- Uploading a deployment package with the AWS CLI
- Uploading a deployment package with AWS SAM

Prerequisites

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

Tools and libraries

AWS provides the following libraries for Java functions. These libraries are available through Maven Central Repository.

- com.amazonaws:aws-lambda-java-core (required) – Defines handler method interfaces and the context object that the runtime passes to the handler. If you define your own input types, this is the only library that you need.
- com.amazonaws:aws-lambda-java-events – Input types for events from services that invoke Lambda functions.
- com.amazonaws:aws-lambda-java-log4j2 – An appender library for Apache Log4j 2 that you can use to add the request ID for the current invocation to your function logs.
- AWS SDK for Java 2.0 – The official AWS SDK for the Java programming language.

Add these libraries to your build definition as follows:

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
```

```
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

To create a deployment package, compile your function code and dependencies into a single .zip file or Java Archive (JAR) file. For Gradle, use the `Zip` build type. For Apache Maven, use the Maven Shade plugin. To upload your deployment package, use the Lambda console, the Lambda API, or AWS Serverless Application Model (AWS SAM).

> **Note**
>
> To keep your deployment package size small, package your function's dependencies in layers. Layers enable you to manage your dependencies independently, can be used by multiple functions, and can be shared with other accounts. For more information, see *Lambda layers*.

Building a deployment package with Gradle

To create a deployment package with your function's code and dependencies in Gradle, use the `Zip` build type. Here's an example from a complete sample build.gradle file:

Example build.gradle – Build task

```
task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

This build configuration produces a deployment package in the `build/distributions` directory. Within the `into('lib')` statement, the `jar` task assembles a jar archive containing your main classes into a folder named `lib`. Additionally, the `configurations.runtimeClassPath` task copies dependency libraries from the build's classpath into the same `lib` folder.

Example build.gradle – Dependencies

```
dependencies {
```

```
    ...
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    implementation 'org.apache.logging.log4j:log4j-api:2.17.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.17.1'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.17.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    ...
}
```

Lambda loads JAR files in Unicode alphabetical order. If multiple JAR files in the `lib` directory contain the same class, the first one is used. You can use the following shell script to identify duplicate classes:

Example test-zip.sh

```
mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort |
 uniq -c | sort
```

Using layers for dependencies

You can package your function's dependencies in layers to keep your deployment package small and manage dependencies independently. For more information, see the section called “Layers”.

Building a deployment package with Maven

To build a deployment package with Maven, use the Maven Shade plugin. The plugin creates a JAR file that contains the compiled function code and all of its dependencies.

Example pom.xml – Plugin configuration

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
```

```
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

To build the deployment package, use the `mvn package` command.

```
[INFO] Scanning for projects...
[INFO] -----------------------< com.example:java-maven >-----------------------
[INFO] Building java-maven-function 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ java-maven ---
[INFO] Building jar: target/java-maven-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-shade-plugin:3.2.2:shade (default) @ java-maven ---
[INFO] Including com.amazonaws:aws-lambda-java-core:jar:1.2.2 in the shaded jar.
[INFO] Including com.amazonaws:aws-lambda-java-events:jar:3.11.1 in the shaded jar.
[INFO] Including joda-time:joda-time:jar:2.6 in the shaded jar.
[INFO] Including com.google.code.gson:gson:jar:2.8.6 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing target/java-maven-1.0-SNAPSHOT.jar with target/java-maven-1.0-
SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.321 s
[INFO] Finished at: 2020-03-03T09:07:19Z
[INFO] ------------------------------------------------------------------------
```

This command generates a JAR file in the `target` directory.

> **Note**
>
> If you're working with a multi-release JAR (MRJAR), you must include the MRJAR (i.e. the shaded JAR produced by the Maven Shade plugin) in the `lib` directory and zip it before uploading your deployment package to Lambda. Otherwise, Lambda may not properly unpack your JAR file, causing your `MANIFEST.MF` file to be ignored.

If you use the appender library (`aws-lambda-java-log4j2`), you must also configure a transformer for the Maven Shade plugin. The transformer library combines versions of a cache file that appear in both the appender library and in Log4j.

Example pom.xml – Plugin configuration with Log4j 2 appender

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCacheFile
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
        <dependencies>
          <dependency>
            <groupId>com.github.edwgiz</groupId>
            <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
            <version>2.13.0</version>
          </dependency>
        </dependencies>
      </plugin>
```

Uploading a deployment package with the Lambda console

To create a new function, you must first create the function in the console, then upload your .zip or JAR file. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip or JAR file.

If your deployment package file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip or JAR files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the AWS Management Console, see Getting started with Amazon S3. To upload files using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> You cannot change the deployment package type (.zip or container image) for an existing function. For example, you cannot convert a container image function to use a .zip file archive. You must create a new function.

To create a new function (console)

1. Open the Functions page of the Lambda console and choose **Create Function**.
2. Choose **Author from scratch**.
3. Under **Basic information**, do the following:
   a. For **Function name**, enter the name for your function.
   b. For **Runtime**, select the runtime you want to use.
   c. (Optional) For **Architecture**, choose the instruction set architecture for your function. The default architecture is x86_64. Ensure that the .zip deployment package for your function is compatible with the instruction set architecture you select.
4. (Optional) Under **Permissions**, expand **Change default execution role**. You can create a new **Execution role** or use an existing one.
5. Choose **Create function**. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip or JAR archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip or JAR file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **.zip or .jar file**.

5. To upload the .zip or JAR file, do the following:

   a. Select **Upload**, then select your .zip or JAR file in the file chooser.

   b. Choose **Open**.

   c. Choose **Save**.

To upload a .zip or JAR archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip or JAR file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **Amazon S3 location**.
5. Paste the Amazon S3 link URL of your .zip file and choose **Save**.

Uploading a deployment package with the AWS CLI

You can can use the AWS CLI to create a new function or to update an existing one using a .zip or JAR file. Use the create-function and update-function-code commands to deploy your .zip or JAR package. If your file is smaller than 50MB, you can upload the package from a file location on your local build machine. For larger files, you must upload your .zip or JAR package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> If you upload your .zip or JAR file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip or JAR file with the AWS CLI, you must specify the following:

- The name of your function (`--function-name`)
- Your function's runtime (`--runtime`)
- The Amazon Resource Name (ARN) of your function's execution role (`--role`)
- The name of the handler method in your function code (`--handler`)

You must also specify the location of your .zip or JAR file. If your .zip or JAR file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction \
--runtime java25 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--code` option as shown in the following example command. You only need to use the `S3ObjectVersion` parameter for versioned objects.

```
aws lambda create-function --function-name myFunction \
--runtime java25 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the `--function-name` parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--s3-bucket` and `--s3-key` options as shown in the following example command. You only need to use the `--s3-object-version` parameter for versioned objects.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Uploading a deployment package with AWS SAM

You can use AWS SAM to automate deployments of your function code, configuration, and dependencies. AWS SAM is an extension of CloudFormation that provides a simplified syntax

for defining serverless applications. The following example template defines a function with a deployment package in the `build/distributions` directory that Gradle uses:

Example template.yml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/java-basic.zip
      Handler: example.Handler
      Runtime: java25
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSLambdaVPCAccessExecutionRole
      Tracing: Active
```

To create the function, use the `package` and `deploy` commands. These commands are customizations to the AWS CLI. They wrap other commands to upload the deployment package to Amazon S3, rewrite the template with the object URI, and update the function's code.

The following example script runs a Gradle build and uploads the deployment package that it creates. It creates an CloudFormation stack the first time you run it. If the stack already exists, the script updates it.

Example deploy.sh

```
#!/bin/bash
set -eo pipefail
aws cloudformation package --template-file template.yml --s3-bucket MY_BUCKET --output-
template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name java-basic --
capabilities CAPABILITY_NAMED_IAM
```

For a complete working example, see the following sample applications:

Sample Lambda applications in Java

- example-java – A Java function that demonstrates how you can use Lambda to process orders. This function illustrates how to define and deserialize a custom input event object, use the AWS SDK, and output logging.
- java-basic – A collection of minimal Java functions with unit tests and variable logging configuration.
- java-events – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the aws-lambda-java-events library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- s3-java – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- layer-java – A Java function that illustrates how to use a Lambda layer to package dependencies separate from your core function code.

Deploy Java Lambda functions with container images

There are three ways to build a container image for a Java Lambda function:

- Using an AWS base image for Java

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.

- Using an AWS OS-only base image

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the runtime interface client for Java in the image.

- Using a non-AWS base image

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the runtime interface client for Java in the image.

> **Tip**
>
> To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

This page explains how to build, test, and deploy container images for Lambda.

Topics

- AWS base images for Java
- Using an AWS base image for Java
- Using an alternative base image with the runtime interface client

AWS base images for Java

AWS provides the following base images for Java:

Tags	Runtime	Operating system	Dockerfile	Deprecation
25	Java 25	Amazon Linux 2023	Dockerfile for Java 25 on GitHub	Jun 30, 2029
21	Java 21	Amazon Linux 2023	Dockerfile for Java 21 on GitHub	Jun 30, 2029
17	Java 17	Amazon Linux 2	Dockerfile for Java 17 on GitHub	Jun 30, 2027
11	Java 11	Amazon Linux 2	Dockerfile for Java 11 on GitHub	Jun 30, 2027
8.al2	Java 8	Amazon Linux 2	Dockerfile for Java 8 on GitHub	Jun 30, 2027

Amazon ECR repository: gallery.ecr.aws/lambda/java

The Java 21 and later base images are based on the Amazon Linux 2023 minimal container image. Earlier base images use Amazon Linux 2. AL2023 provides several advantages over Amazon Linux 2, including a smaller deployment footprint and updated versions of libraries such as `glibc`.

AL2023-based images use `microdnf` (symlinked as `dnf`) as the package manager instead of `yum`, which is the default package manager in Amazon Linux 2. `microdnf` is a standalone implementation of `dnf`. For a list of packages that are included in AL2023-based images, refer to the **Minimal Container** columns in Comparing packages installed on Amazon Linux 2023 Container Images. For more information about the differences between AL2023 and Amazon Linux 2, see Introducing the Amazon Linux 2023 runtime for AWS Lambda on the AWS Compute Blog.

> **Note**
>
> To run AL2023-based images locally, including with AWS Serverless Application Model (AWS SAM), you must use Docker version 20.10.10 or later.

Using an AWS base image for Java

Prerequisites

To complete the steps in this section, you must have the following:

- Java (for example, Amazon Corretto)
- Apache Maven or Gradle
- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.

Creating an image from a base image

Maven

1. Run the following command to create a Maven project using the archetype for Lambda. The following parameters are required:

   - **service** – The AWS service client to use in the Lambda function. For a list of available sources, see aws-sdk-java-v2/services on GitHub.
   - **region** – The AWS Region where you want to create the Lambda function.
   - **groupId** – The full package namespace of your application.
   - **artifactId** – Your project name. This becomes the name of the directory for your project.

   In Linux and macOS, run this command:

```
mvn -B archetype:generate \
   -DarchetypeGroupId=software.amazon.awssdk \
   -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
   -DgroupId=com.example.myapp \
```

```
   -DartifactId=myapp
```

In PowerShell, run this command:

```
mvn -B archetype:generate `
   "-DarchetypeGroupId=software.amazon.awssdk" `
   "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
 `
   "-DgroupId=com.example.myapp" `
   "-DartifactId=myapp"
```

The Maven archetype for Lambda is preconfigured to compile with Java SE 8 and includes a dependency to the AWS SDK for Java. If you create your project with a different archetype or by using another method, you must configure the Java compiler for Maven and declare the SDK as a dependency.

2. Open the *myapp*/src/main/java/com/example/*myapp* directory, and find the App.java file. This is the code for the Lambda function. You can use the provided sample code for testing, or replace it with your own.
3. Navigate back to the project's root directory, and then create a new Dockerfile with the following configuration:

   - Set the FROM property to the URI of the base image.
   - Set the CMD argument to the Lambda function handler.

   Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the root user when no USER instruction is provided.

   Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Maven layout
COPY target/classes ${LAMBDA_TASK_ROOT}
COPY target/dependency/* ${LAMBDA_TASK_ROOT}/lib/
```

```
# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.myapp.App::handleRequest" ]
```

4. Compile the project and collect the runtime dependencies.

```
mvn compile dependency:copy-dependencies -DincludeScope=runtime
```

5. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-
image:test .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

Gradle

1. Create a directory for the project, and then switch to that directory.

```
mkdir example
cd example
```

2. Run the following command to have Gradle generate a new Java application project in the `example` directory in your environment. For **Select build script DSL**, choose **2: Groovy**.

```
gradle init --type java-application
```

3. Open the */example*/app/src/main/java/*example* directory, and find the `App.java` file. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

Example App.java

```
package com.example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class App implements RequestHandler<Object, String> {
    public String handleRequest(Object input, Context context) {
        return "Hello world!";
    }
}
```

4. Open the `build.gradle` file. If you're using the sample function code from the previous step, replace the contents of `build.gradle` with the following. If you're using your own function code, modify your `build.gradle` file as needed.

Example build.gradle (Groovy DSL)

```
plugins {
  id 'java'
}
group 'com.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
  mavenCentral()
}
dependencies {
  implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
}
jar {
  manifest {
      attributes 'Main-Class': 'com.example.App'
  }
}
```

5. The `gradle init` command from step 2 also generated a dummy test case in the `app/test` directory. For the purposes of this tutorial, skip running tests by deleting the `/test` directory.
6. Build the project.

```
gradle build
```

7. In the project's root directory (`/example`), create a Dockerfile with the following configuration:

   - Set the `FROM` property to the URI of the base image.
   - Use the COPY command to copy the function code and runtime dependencies to `{LAMBDA_TASK_ROOT}`, a Lambda-defined environment variable.
   - Set the `CMD` argument to the Lambda function handler.

   Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

   Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Gradle layout
COPY app/build/classes/java/main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.App::handleRequest" ]
```

8. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-
image:test .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using

the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

1. Start the Docker image with the **docker run** command. In this example, `docker-image` is the image name and `test` is the tag.

   ```
   docker run --platform linux/amd64 -p 9000:8080 docker-image:test
   ```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

   > **Note**
   >
   > If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

2. From a new terminal window, post an event to the local endpoint.

   Linux/macOS

   In Linux and macOS, run the following `curl` command:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   In PowerShell, run the following `Invoke-WebRequest` command:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

3. Get the container ID.

```
docker ps
```

4. Use the docker kill command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
```

```
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Using an alternative base image with the runtime interface client

If you use an OS-only base image or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the Runtime API, which manages the interaction between Lambda and your function code.

Install the runtime interface client for Java in your Dockerfile, or as a dependency in your project. For example, to install the runtime interface client using the Maven package manager, add the following to your `pom.xml` file:

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

For package details, see AWS Lambda Java Runtime Interface Client in the Maven Central Repository. You can also review the runtime interface client source code in the AWS Lambda Java Support Libraries GitHub repository.

The following example demonstrates how to build a container image for Java using an Amazon Corretto image. Amazon Corretto is a no-cost, multiplatform, production-ready distribution of the Open Java Development Kit (OpenJDK). The Maven project includes the runtime interface client as a dependency.

Prerequisites

To complete the steps in this section, you must have the following:

- Java (for example, Amazon Corretto)
- Apache Maven
- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.

Creating an image from an alternative base image

1. Create a Maven project. The following parameters are required:

   - **groupId** – The full package namespace of your application.
   - **artifactId** – Your project name. This becomes the name of the directory for your project.

   Linux/macOS

   ```
   mvn -B archetype:generate \
      -DarchetypeArtifactId=maven-archetype-quickstart \
      -DgroupId=example \
      -DartifactId=myapp \
      -DinteractiveMode=false
   ```

   PowerShell

   ```
   mvn -B archetype:generate `
      -DarchetypeArtifactId=maven-archetype-quickstart `
   ```

```
    -DgroupId=example `
    -DartifactId=myapp `
    -DinteractiveMode=false
```

2. Open the project directory.

```
cd myapp
```

3. Open the `pom.xml` file and replace the contents with the following. This file includes the aws-lambda-java-runtime-interface-client as a dependency. Alternatively, you can install the runtime interface client in the Dockerfile. However, the simplest approach is to include the library as a dependency.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>example</groupId>
  <artifactId>hello-lambda</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello-lambda</name>
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
      <version>2.3.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.1.2</version>
        <executions>
          <execution>
```

```
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

4. Open the *myapp*/src/main/java/com/example/*myapp* directory, and find the App.java file. This is the code for the Lambda function. Replace the code with the following.

   Example function handler

   ```
   package example;

   public class App {
       public static String sayHello() {
           return "Hello world!";
       }
   }
   ```

5. The mvn -B archetype:generate command from step 1 also generated a dummy test case in the src/test directory. For the purposes of this tutorial, skip over running tests by deleting this entire generated /test directory.
6. Navigate back to the project's root directory, and then create a new Dockerfile. The following example Dockerfile uses an Amazon Corretto image. Amazon Corretto is a no-cost, multiplatform, production-ready distribution of the OpenJDK.

   - Set the FROM property to the URI of the base image.
   - Set the ENTRYPOINT to the module that you want the Docker container to run when it starts. In this case, the module is the runtime interface client.
   - Set the CMD argument to the Lambda function handler.

   Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-

privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example Dockerfile

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as base

# Configure the build environment
FROM base as build
RUN yum install -y maven
WORKDIR /src

# Cache and copy dependencies
ADD pom.xml .
RUN mvn dependency:go-offline dependency:copy-dependencies

# Compile the function
ADD . .
RUN mvn package

# Copy the function artifact and dependencies onto a clean base
FROM base
WORKDIR /function

COPY --from=build /src/target/dependency/*.jar ./
COPY --from=build /src/target/*.jar ./

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/bin/java", "-cp", "./*",
 "com.amazonaws.services.lambda.runtime.api.client.AWSLambda" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "example.App::sayHello" ]
```

7. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test the image. You can build the emulator into your image or use the following procedure to install it on your local machine.

To install and run the runtime interface emulator on your local machine

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }
   ```

```
$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

To install the arm64 emulator, replace the `$downloadLink` with the following:

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. Start the Docker image with the **docker run** command. Note the following:

   - `docker-image` is the image name and `test` is the tag.
   - `/usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello` is the `ENTRYPOINT` followed by the `CMD` from your Dockerfile.

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /usr/bin/java -cp './*'
    com.amazonaws.services.lambda.runtime.api.client.AWSLambda
    example.App::sayHello
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /usr/bin/java -cp './*'
    com.amazonaws.services.lambda.runtime.api.client.AWSLambda
    example.App::sayHello
   ```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

   Linux/macOS

   In Linux and macOS, run the following `curl` command:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   In PowerShell, run the following `Invoke-WebRequest` command:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. Get the container ID.

   ```
   docker ps
   ```

5. Use the docker kill command to stop the container. In this command, replace 3766c4ab331c with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.

   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
```

```
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Example:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
   world:latest
   ```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
```

```
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Working with layers for Java Lambda functions

Use Lambda layers to package code and dependencies that you want to reuse across multiple functions. Layers usually contain library dependencies, a custom runtime, or configuration files. Creating a layer involves three general steps:

1. Package your layer content. This means creating a .zip file archive that contains the dependencies you want to use in your functions.
2. Create the layer in Lambda.
3. Add the layer to your functions.

Topics

- Package your layer content
- Create the layer in Lambda
- Add the layer to your function

Package your layer content

To create a layer, bundle your packages into a .zip file archive that meets the following requirements:

- Ensure that the Java version that Maven or Gradle refers to is the same as the Java version of the function that you intend to deploy. For example, for a Java 25 function, the `mvn -v` command should list Java 25 in the output.
- Your dependencies must be stored in the `java/lib` directory, at the root of the .zip file. For more information, see Layer paths for each Lambda runtime.
- The packages in your layer must be compatible with Linux. Lambda functions run on Amazon Linux.

You can create layers that contain either third-party Java libraries or your own Java modules and packages. The following procedure uses Maven. You can also use Gradle to package your layer content.

To create a layer using Maven dependencies

1. Create an Apache Maven project with a `pom.xml` file that defines your dependencies.

   The following example includes Jackson Databind for JSON processing. The `<build>` section uses the maven-dependency-plugin to create separate JAR files for each dependency instead of bundling them into a single uber-jar. If you want to create an uber-jar, use the maven-shade-plugin.

   Example pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.17.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.13.0</version>
            <configuration>
                <source>21</source>
                <target>21</target>
                <release>21</release>
            </configuration>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>3.6.1</version>
            <executions>
                <execution>
                    <id>copy-dependencies</id>
                    <phase>package</phase>
                    <goals>
                        <goal>copy-dependencies</goal>
                    </goals>
```

```
                    <configuration>
                        <outputDirectory>${project.build.directory}/lib</
outputDirectory>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

2. Build the project. This command creates all dependency JAR files in the `target/lib/` directory.

```
mvn clean package
```

3. Create the required directory structure for your layer:

```
mkdir -p java/lib
```

4. Copy the dependency JAR files to the `java/lib` directory:

```
cp target/lib/*.jar java/lib/
```

5. Zip the layer content:

   Linux/macOS

   ```
   zip -r layer.zip java/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\java -DestinationPath .\layer.zip
   ```

   The directory structure of your .zip file should look like this:

```
java/
### lib/
    ### jackson-databind-2.17.0.jar
    ### jackson-core-2.17.0.jar
    ### jackson-annotations-2.17.0.jar
```

> **Note**
>
> Make sure your .zip file includes the `java` directory at the root level with `lib` inside it. This structure ensures that Lambda can locate and import your libraries. Each dependency is kept as a separate JAR file rather than bundled into an uber-jar.

Create the layer in Lambda

You can publish your layer using either the AWS CLI or the Lambda console.

AWS CLI

Run the publish-layer-version AWS CLI command to create the Lambda layer:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes java25
```

The compatible runtimes parameter is optional. When specified, Lambda uses this parameter to filter layers in the Lambda console.

Console

To create a layer (console)

1. Open the Layers page of the Lambda console.
2. Choose **Create layer**.
3. Choose **Upload a .zip file**, and then upload the .zip archive that you created earlier.
4. (Optional) For **Compatible runtimes**, choose the Java runtime that corresponds to the Java version you used to build your layer.
5. Choose **Create**.

Add the layer to your function

AWS CLI

To attach the layer to your function, run the update-function-configuration AWS CLI command. For the `--layers` parameter, use the layer ARN. The ARN must specify the version (for

example, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`). For more information, see Layers and layer versions.

```
aws lambda update-function-configuration --function-name my-function --cli-binary-format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1"
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

Console

To add a layer to a function

1. Open the Functions page of the Lambda console.
2. Choose the function.
3. Scroll down to the **Layers** section, and then choose **Add a layer**.
4. Under **Choose a layer**, select **Custom layers**, and then choose your layer.

> **Note**
>
> If you didn't add a compatible runtime when you created the layer, your layer won't be listed here. You can specify the layer ARN instead.

5. Choose **Add**.

Customize serialization for Lambda Java functions

The Lambda Java managed runtimes support custom serialization for JSON events. Custom serialization can simplify your code and potentially improve performance.

Topics

- When to use custom serialization
- Implementing custom serialization
- Testing custom serialization

When to use custom serialization

When your Lambda function is invoked, the input event data needs to be deserialized into a Java object, and the output from your function needs to be serialized back into a format that can be returned as the function's response. The Lambda Java managed runtimes provide default serialization and deserialization capabilities that work well for handling event payloads from various AWS services, such as Amazon API Gateway and Amazon Simple Queue Service (Amazon SQS). To work with these service integration events in your function, add the aws-java-lambda-events dependency to your project. This AWS library contains Java objects representing these service integration events.

You can also use your own objects to represent the event JSON that you pass to your Lambda function. The managed runtime attempts to serialize the JSON to a new instance of your object with its default behavior. If the default serializer doesn't have the desired behavior for your use case, use custom serialization.

For example, assume that your function handler expects a `Vehicle` class as input, with the following structure:

```
public class Vehicle {
    private String vehicleType;
    private long vehicleId;
}
```

However, the JSON event payload looks like this:

```
{
```

```
    "vehicle-type": "car",
    "vehicleID": 123
}
```

In this scenario, the default serialization in the managed runtime expects the JSON property names to match the camel case Java class property names (`vehicleType`, `vehicleId`). Because the property names in the JSON event aren't in camel case (`vehicle-type`, `vehicleID`), you must use custom serialization.

Implementing custom serialization

Use a Service Provider Interface to load a serializer of your choice instead of the managed runtime's default serialization logic. You can serialize your JSON event payloads directly into Java objects, using the standard `RequestHandler` interface.

To use custom serialization in your Lambda Java function

1. Add the aws-lambda-java-core library as a dependency. This library includes the CustomPojoSerializer interface, along with other interface definitions for working with Java in Lambda.
2. Create a file named `com.amazonaws.services.lambda.runtime.CustomPojoSerializer` in the `src/main/resources/META-INF/services/` directory of your project.
3. In this file, specify the fully qualified name of your custom serializer implementation, which must implement the `CustomPojoSerializer` interface. Example:

   ```
   com.mycompany.vehicles.CustomLambdaSerializer
   ```
4. Implement the `CustomPojoSerializer` interface to provide your custom serialization logic.
5. Use the standard `RequestHandler` interface in your Lambda function. The managed runtime will use your custom serializer.

For more examples of how to implement custom serialization using popular libraries such as fastJson, Gson, Moshi, and jackson-jr, see the custom-serialization sample in the AWS GitHub repository.

Testing custom serialization

Test your function to make sure that your serialization and deserialization logic is working as expected. You can use the AWS Serverless Application Model Command Line Interface (AWS SAM CLI) to emulate the invocation of your Lambda payload. This can help you quickly test and iterate on your function as you introduce a custom serializer.

1. Create a file with the JSON event payload that you would like to invoke your function with then call the AWS SAM CLI.
2. Run the sam local invoke command to invoke your function locally. Example:

```
sam local invoke -e src/test/resources/event.json
```

For more information, see Locally invoke Lambda functions with AWS SAM.

Customize Java runtime startup behavior for Lambda functions

This page describes settings specific to Java functions in AWS Lambda. You can use these settings to customize Java runtime startup behavior. This can reduce overall function latency and improve overall function performance, without having to modify any code.

Sections

- Understanding the JAVA_TOOL_OPTIONS environment variable
- Log4j patch for Log4Shell
- Ahead-of-Time (AOT) and CDS caches

Understanding the `JAVA_TOOL_OPTIONS` environment variable

In Java, Lambda supports the `JAVA_TOOL_OPTIONS` environment variable to set additional command-line variables in Lambda. You can use this environment variable in various ways, such as to customize tiered-compilation settings. The next example demonstrates how to use the `JAVA_TOOL_OPTIONS` environment variable for this use case.

Example: Customizing tiered compilation settings

Tiered compilation is a feature of the Java virtual machine (JVM). You can use specific tiered compilation settings to make best use of the JVM's just-in-time (JIT) compilers. Typically, the C1 compiler is optimized for fast start-up time. The C2 compiler is optimized for best overall performance, but it also uses more memory and takes a longer time to achieve it. There are 5 different levels of tiered compilation. At Level 0, the JVM interprets Java byte code. At Level 4, the JVM uses the C2 compiler to analyze profiling data collected during application startup. Over time, it monitors code usage to identify the best optimizations.

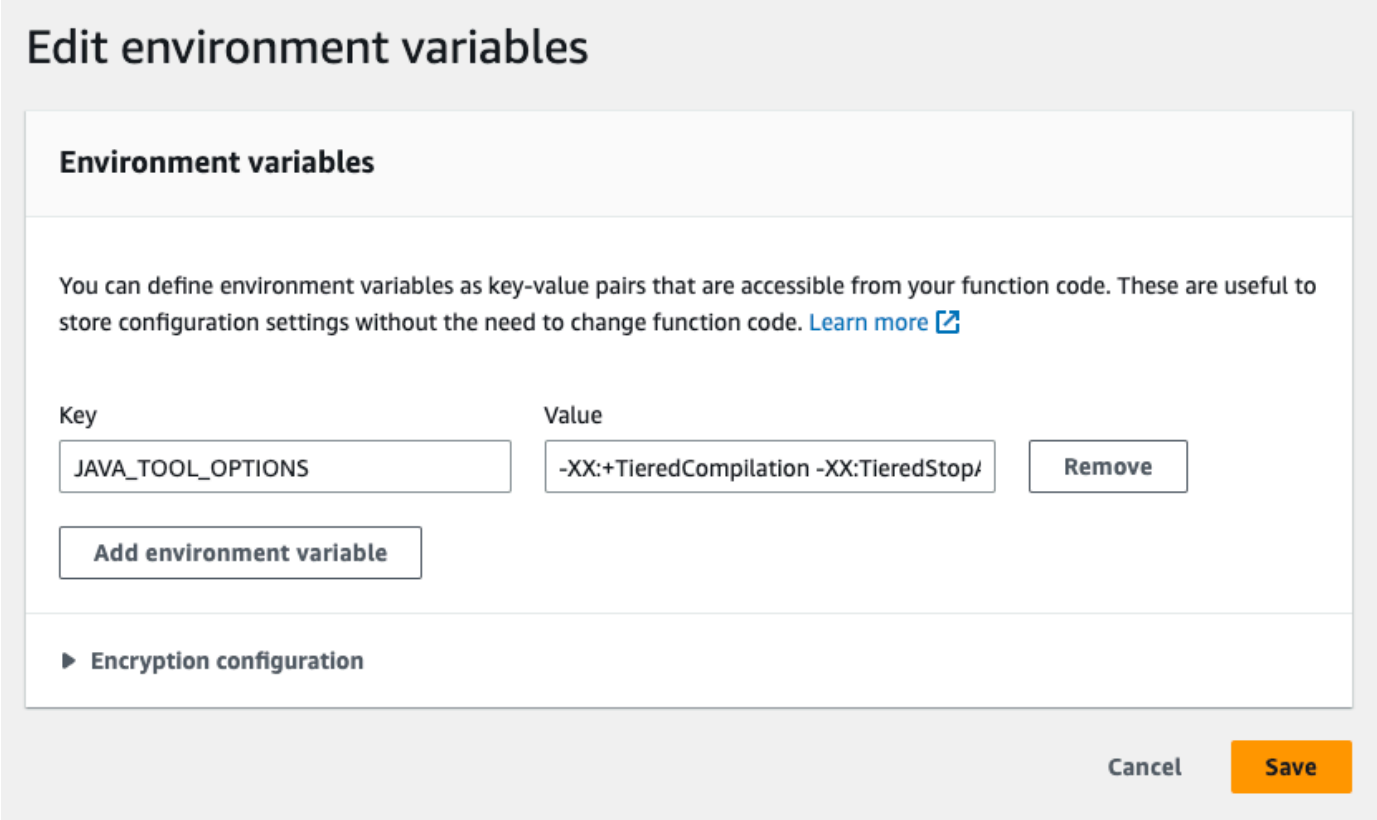
Customizing the tiered compilation level can help you tune your Java function performance. For small functions that execute quickly, setting the tiered compilation to level 1 can help improve cold start performance by having the JVM use the C1 compiler. This setting quickly produces optimized native code but it doesn't generate any profiling data and never uses the C2 compiler. For larger, computationally-intensive functions, setting tiered compilation to level 4 maximizes overall performance at the expense of additional memory consumption and additional optimization work during the first invokes after each Lambda execution environment is provisioned.

For Java 11 runtimes and below, Lambda uses the default JVM tiered compilation settings. For Java 17 and Java 21, Lambda configures the JVM to stop tiered compilation at level 1 by

default. From Java 25, Lambda still stops tiered compilation at level 1 by default, except when using SnapStart or Provisioned concurrency, in which case the default JVM settings are used. This improves performance for SnapStart and Provisioned concurrency without incurring a cold start penalty since tiered compilation is performed outside of the invoke path in these cases. To maximize this benefit, you can use priming - executing code paths during function initialization to trigger JIT before taking the SnapStart snapshot or when Provisioned Concurrency execution environments are pre-provisioned. For further information, see the blog post Optimizing cold start performance of AWS Lambda using advanced priming strategies with SnapStart.

To customize tiered compilation settings (console)

1. Open the Functions page in the Lambda console.
2. Choose a Java function that you want to customize tiered compilation for.
3. Choose the **Configuration** tab, then choose **Environment variables** in the left menu.
4. Choose **Edit**.
5. Choose **Add environment variable**.
6. For the key, enter `JAVA_TOOL_OPTIONS`. For the value, enter `-XX:+TieredCompilation -XX:TieredStopAtLevel=1`.



7. Choose **Save**.

> **Note**
>
> You can also use Lambda SnapStart to mitigate cold start issues. SnapStart uses cached snapshots of your execution environment to significantly improve start-up performance. For more information about SnapStart features, limitations, and supported regions, see Improving startup performance with Lambda SnapStart.

Example: Customizing GC behavior using JAVA_TOOL_OPTIONS

Java 11 runtimes use the Serial garbage collector (GC) for garbage collection. By default, Java 17 runtimes also use the Serial GC. However, with Java 17 you can also use the `JAVA_TOOL_OPTIONS` environment variable to change the default GC. You can choose between the Parallel GC and Shenandoah GC.

For example, if your workload uses more memory and multiple CPUs, consider using the Parallel GC for better performance. You can do this by appending the following to the value of your `JAVA_TOOL_OPTIONS` environment variable:

```
-XX:+UseParallelGC
```

If your workload has many short-lived objects, you may benefit from lower memory consumption by enabling the generational mode of the Shenandoah garbage collector introduced in Java 25. To do this, append the following to the value of your `JAVA_TOOL_OPTIONS` environment variable:

```
-XX:+UseShenandoahGC -XX:ShenandoahGCMode=generational
```

Log4j patch for Log4Shell

Lambda runtimes for Java 8, 11, 17, and 21 include a patch to mitigate the Log4Shell vulnerability (CVE-2021-44228) in Log4j, a popular Java logging framework. This patch incurs a cold start performance overhead. If you are using a patched version of Log4j (version 2.17.0 or later), you can disable this patch to improve cold start performance. To disable the patch, set the `AWS_LAMBDA_DISABLE_CVE_2021_44228_PROTECTION` environment variable to `true`.

Starting from Java 25, Lambda runtimes no longer include the Log4Shell patch. You must verify you are using Log4j version 2.17.0 or later.

Ahead-of-Time (AOT) and CDS caches

Starting with Java 25, the Lambda runtime includes an Ahead-of-Time (AOT) cache for the Java runtime interface client (RIC), a runtime component which actively polls for events from the Lambda Runtime API. This improves cold start performance.

AOT caches are specific to a JVM build. When Lambda updates the managed runtime, it also updates the AOT cache for the RIC. However, if you deploy your own AOT caches, these may be invalidated or result in unexpected behavior following a runtime update. We therefore strongly recommend not using AOT caches when using managed runtimes. To use AOT caches, you should deploy your functions using container images.

AOT caches cannot be used with Class Data Sharing (CDS) caches. If you deploy CDS caches in your Lambda function, then Lambda disables the AOT cache.

Using the Lambda context object to retrieve Java function information

When Lambda runs your function, it passes a context object to the handler. This object provides methods and properties that provide information about the invocation, function, and execution environment.

Context methods

- `getRemainingTimeInMillis()` – Returns the number of milliseconds left before the execution times out.
- `getFunctionName()` – Returns the name of the Lambda function.
- `getFunctionVersion()` – Returns the version of the function.
- `getInvokedFunctionArn()` – Returns the Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `getMemoryLimitInMB()` – Returns the amount of memory that's allocated for the function.
- `getAwsRequestId()` – Returns the identifier of the invocation request.
- `getLogGroupName()` – Returns the log group for the function.
- `getLogStreamName()` – Returns the log stream for the function instance.
- `getIdentity()` – (mobile apps) Returns information about the Amazon Cognito identity that authorized the request.
- `getClientContext()` – (mobile apps) Returns the client context that's provided to Lambda by the client application.
- `getLogger()` – Returns the logger object for the function.

The following example shows a function that uses the context object to access the Lambda logger.

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
```

```
import java.util.Map;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, Void>{

  @Override
  public Void handleRequest(Map<String,String> event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass());
    return null;
  }
}
```

The function logs the class type of the incoming event before returning `null`.

Example log output

```
EVENT TYPE: class java.util.LinkedHashMap
```

The interface for the context object is available in the aws-lambda-java-core library. You can implement this interface to create a context class for testing. The following example shows a context class that returns dummy values for most properties and a working test logger.

Example src/test/java/example/TestContext.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.CognitoIdentity;
import com.amazonaws.services.lambda.runtime.ClientContext;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class TestContext implements Context{

  public TestContext() {}
  public String getAwsRequestId(){
    return new String("495b12a8-xmpl-4eca-8168-160484189f99");
  }
  public String getLogGroupName(){
    return new String("/aws/lambda/my-function");
  }
```

```
  public String getLogStreamName(){
    return new String("2020/02/26/[$LATEST]704f8dxmpla04097b9134246b8438f1a");
  }
  public String getFunctionName(){
    return new String("my-function");
  }
  public String getFunctionVersion(){
    return new String("$LATEST");
  }
  public String getInvokedFunctionArn(){
    return new String("arn:aws:lambda:us-east-2:123456789012:function:my-function");
  }
  public CognitoIdentity getIdentity(){
    return null;
  }
  public ClientContext getClientContext(){
    return null;
  }
  public int getRemainingTimeInMillis(){
    return 300000;
  }
  public int getMemoryLimitInMB(){
    return 512;
  }
  public LambdaLogger getLogger(){
    return new TestLogger();
  }

}
```

For more information on logging, see Log and monitor Java Lambda functions.

Context in sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of the context object. Each sample application includes scripts for easy deployment and cleanup, an AWS Serverless Application Model (AWS SAM) template, and supporting resources.

Sample Lambda applications in Java

- example-java – A Java function that demonstrates how you can use Lambda to process orders. This function illustrates how to define and deserialize a custom input event object, use the AWS SDK, and output logging.

- java-basic – A collection of minimal Java functions with unit tests and variable logging configuration.
- java-events – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the aws-lambda-java-events library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- s3-java – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- layer-java – A Java function that illustrates how to use a Lambda layer to package dependencies separate from your core function code.

Log and monitor Java Lambda functions

AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function's code to the log stream. For more information about CloudWatch Logs, see Sending Lambda function logs to CloudWatch Logs.

To output logs from your function code, you can use methods on java.lang.System, or any logging module that writes to stdout or stderr.

Sections

- Creating a function that returns logs
- Using Lambda advanced logging controls with Java
- Implementing advanced logging with Log4j2 and SLF4J
- Using other logging tools and libraries
- Using Powertools for AWS Lambda (Java) and AWS SAM for structured logging
- Viewing logs in the Lambda console
- Viewing logs in the CloudWatch console
- Viewing logs using the AWS Command Line Interface (AWS CLI)
- Deleting logs
- Sample logging code

Creating a function that returns logs

To output logs from your function code, you can use methods on java.lang.System, or any logging module that writes to `stdout` or `stderr`. The aws-lambda-java-core library provides a logger class named `LambdaLogger` that you can access from the context object. The logger class supports multiline logs.

The following example uses the `LambdaLogger` logger provided by the context object.

Example Handler.java

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Object, String>{
```

```
  Gson gson = new GsonBuilder().setPrettyPrinting().create();
  @Override
  public String handleRequest(Object event, Context context)
  {
    LambdaLogger logger = context.getLogger();
    String response = new String("SUCCESS");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    return response;
  }
}
```

Example log format

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
EVENT:
{
  "records": [
    {
      "messageId": "19dd0b57-xmpl-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
       ...
    }
  ]
}
```

```
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed
 Duration: 724 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

The Java runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details:

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function. When invocations share an execution environment, Lambda reports the maximum memory used across all invocations. This behavior might result in a higher than expected reported value.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using Lambda advanced logging controls with Java

To give you more control over how your functions' logs are captured, processed, and consumed, you can configure the following logging options for supported Java runtimes:

- **Log format** - select between plain text and structured JSON format for your function's logs
- **Log level** - for logs in JSON format, choose the detail level of the logs Lambda sends to CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

For more information about these logging options, and instructions on how to configure your function to use them, see the section called "Configuring advanced logging controls for Lambda functions".

To use the log format and log level options with your Java Lambda functions, see the guidance in the following sections.

Using structured JSON log format with Java

If you select JSON for your function's log format, Lambda will send logs output using the `LambdaLogger` class as structured JSON. Each JSON log object contains at least four key value pairs with the following keys:

- `"timestamp"` - the time the log message was generated
- `"level"` - the log level assigned to the message
- `"message"` - the contents of the log message
- `"AWSrequestId"` - the unique request ID for the function invocation

Depending on the logging method you use, log outputs from your function captured in JSON format can also contain additional key value pairs.

To assign a level to logs you create using the `LambdaLogger` logger, you need to provide a `LogLevel` argument in your logging command as shown in the following example.

Example Java logging code

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

This log output by this example code would be captured in CloudWatch Logs as follows:

Example JSON log record

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

If you don't assign a level to your log output, Lambda will automatically assign it the level INFO.

If your code already uses another logging library to produce JSON structured logs, you don’t need to make any changes. Lambda doesn’t double-encode any logs that are already JSON encoded.

Even if you configure your function to use the JSON log format, your logging outputs appear in CloudWatch in the JSON structure you define.

Using log-level filtering with Java

For AWS Lambda to filter your application logs according to their log level, your function must use JSON formatted logs. You can achieve this in two ways:

- Create log outputs using the standard `LambdaLogger` and configure your function to use JSON log formatting. Lambda then filters your log outputs using the "level" key value pair in the JSON object described in the section called "Using structured JSON log format with Java". To learn how to configure your function's log format, see the section called "Configuring advanced logging controls for Lambda functions".
- Use another logging library or method to create JSON structured logs in your code that include a "level" key value pair defining the level of the log output. You can use any logging library that can write JSON logs to `stdout` or `stderr`. For example, you can use Powertools for AWS Lambda or the Log4j2 package to generate JSON structured log outputs from your code. See the section called "Using Powertools for AWS Lambda (Java) and AWS SAM for structured logging" and the section called "Implementing advanced logging with Log4j2 and SLF4J" to learn more.

When you configure your function to use log-level filtering, you must select from the following options for the level of logs you want Lambda to send to CloudWatch Logs:

Log level	Standard usage
TRACE (most detail)	The most fine-grained information used to trace the path of your code's execution
DEBUG	Detailed information for system debugging
INFO	Messages that record the normal operation of your function
WARN	Messages about potential errors that may lead to unexpected behavior if unaddressed
ERROR	Messages about problems that prevent the code from performing as expected

Log level	Standard usage
FATAL (least detail)	Messages about serious errors that cause the application to stop functioning

For Lambda to filter your function's logs, you must also include a `"timestamp"` key value pair in your JSON log output. The time must be specified in valid RFC 3339 timestamp format. If you don't supply a valid timestamp, Lambda will assign the log the level INFO and add a timestamp for you.

Lambda sends logs of the selected level and lower to CloudWatch. For example, if you configure a log level of WARN, Lambda will send logs corresponding to the WARN, ERROR, and FATAL levels.

Implementing advanced logging with Log4j2 and SLF4J

Note

AWS Lambda does not include Log4j2 in its managed runtimes or base container images. These are therefore not affected by the issues described in CVE-2021-44228, CVE-2021-45046, and CVE-2021-45105.
For cases where a customer function includes an impacted Log4j2 version, we have applied a change to the Lambda Java managed runtimes and base container images that helps to mitigate the issues in CVE-2021-44228, CVE-2021-45046, and CVE-2021-45105. As a result of this change, customers using Log4J2 may see an additional log entry, similar to `"Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)"`. Any log strings that reference the jndi mapper in the Log4J2 output will be replaced with `"Patched JndiLookup::lookup()"`.
Independent of this change, we strongly encourage all customers whose functions include Log4j2 to update to the latest version. Specifically, customers using the aws-lambda-java-log4j2 library in their functions should update to version 1.5.0 (or later), and redeploy their functions. This version updates the underlying Log4j2 utility dependencies to version 2.17.0 (or later). The updated aws-lambda-java-log4j2 binary is available at the Maven repository and its source code is available in Github.
Lastly, take note that any libraries related to **aws-lambda-java-log4j (v1.0.0 or 1.0.1)** should **not** be used under **any** circumstance. These libraries are related to version 1.x of log4j which went end of life in 2015. The libraries are not supported, not maintained, not patched, and have known security vulnerabilities.

To customize log output, support logging during unit tests, and log AWS SDK calls, use Apache Log4j2 with SLF4J. Log4j is a logging library for Java programs that enables you to configure log levels and use appender libraries. SLF4J is a facade library that lets you change which library you use without changing your function code.

To add the request ID to your function's logs, use the appender in the aws-lambda-java-log4j2 library.

Example src/main/resources/log4j2.xml – Appender configuration

```
<Configuration>
  <Appenders>
    <Lambda name="Lambda" format="${env:AWS_LAMBDA_LOG_FORMAT:-TEXT}">
       <LambdaTextFormat>
         <PatternLayout>
             <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n </
pattern>
         </PatternLayout>
       </LambdaTextFormat>
       <LambdaJSONFormat>
         <JsonTemplateLayout eventTemplateUri="classpath:LambdaLayout.json" />
       </LambdaJSONFormat>
     </Lambda>
   </Appenders>
   <Loggers>
     <Root level="${env:AWS_LAMBDA_LOG_LEVEL:-INFO}">
       <AppenderRef ref="Lambda"/>
     </Root>
     <Logger name="software.amazon.awssdk" level="WARN" />
     <Logger name="software.amazon.awssdk.request" level="DEBUG" />
   </Loggers>
 </Configuration>
```

You can decide how your Log4j2 logs are configured for either plain text or JSON outputs by specifying a layout under the `<LambdaTextFormat>` and `<LambdaJSONFormat>` tags.

In this example, in text mode, each line is prepended with the date, time, request ID, log level, and class name. In JSON mode, the `<JsonTemplateLayout>` is used with a configuration that ships together with the `aws-lambda-java-log4j2` library.

SLF4J is a facade library for logging in Java code. In your function code, you use the SLF4J logger factory to retrieve a logger with methods for log levels like `info()` and `warn()`. In your build

configuration, you include the logging library and SLF4J adapter in the classpath. By changing the libraries in the build configuration, you can change the logger type without changing your function code. SLF4J is required to capture logs from the SDK for Java.

In the following example code, the handler class uses SLF4J to retrieve a logger.

Example src/main/java/example/HandlerS3.java – Logging with SLF4J

```
package example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;

import static org.apache.logging.log4j.CloseableThreadContext.put;


public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);

    @Override
    public String handleRequest(S3Event event, Context context) {
        for(var record : event.getRecords()) {
            try (var loggingCtx = put("awsRegion", record.getAwsRegion())) {
                loggingCtx.put("eventName", record.getEventName());
                loggingCtx.put("bucket", record.getS3().getBucket().getName());
                loggingCtx.put("key", record.getS3().getObject().getKey());

                logger.info("Handling s3 event");
            }
        }

        return "Ok";
    }
}
```

This code produces log outputs like the following:

Example log format

```
{
    "timestamp": "2023-11-15T16:56:00.815Z",
    "level": "INFO",
    "message": "Handling s3 event",
    "logger": "example.HandlerS3",
    "AWSRequestId": "0bced576-3936-4e5a-9dcd-db9477b77f97",
    "awsRegion": "eu-south-1",
    "bucket": "java-logging-test-input-bucket",
    "eventName": "ObjectCreated:Put",
    "key": "test-folder/"
}
```

The build configuration takes runtime dependencies on the Lambda appender and SLF4J adapter, and implementation dependencies on Log4j2.

Example build.gradle – Logging dependencies

```
dependencies {
    ...
    'com.amazonaws:aws-lambda-java-log4j2:[1.6.0,)',
    'com.amazonaws:aws-lambda-java-events:[3.11.3,)',
    'org.apache.logging.log4j:log4j-layout-template-json:[2.17.1,)',
    'org.apache.logging.log4j:log4j-slf4j2-impl:[2.19.0,)',
    ...
}
```

When you run your code locally for tests, the context object with the Lambda logger is not available, and there's no request ID for the Lambda appender to use. For example test configurations, see the sample applications in the next section.

Using other logging tools and libraries

Powertools for AWS Lambda (Java) is a developer toolkit to implement Serverless best practices and increase developer velocity. The Logging utility provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON

- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time
- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC

Using Powertools for AWS Lambda (Java) and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated Powertools for AWS Lambda (Java) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Java template.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press `Enter`.

> **Note**
>
> For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter `y`.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the logs for the function, run sam logs. For more information, see Working with logs in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name sam-app
```

The log output looks like this:

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.095000
 INIT_START Runtime Version: java:11.v15    Runtime Version ARN: arn:aws:lambda:eu-
central-1::runtime:0a25e3e7a1cc9ce404bc435eeb2ad358d8fa64338e618d0c224fe509403583ca
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000
 Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
 Transforming org/apache/logging/log4j/core/lookup/JndiLookup
 (lambdainternal.CustomerClassLoader@1a6c5a9e)
```

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000
 START RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
  "_aws": {
    "Timestamp": 1675416276051,
    "CloudWatchMetrics": [
      {
        "Namespace": "sam-app-powerools-java",
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ],
        "Dimensions": [
          [
            "Service",
            "FunctionName"
          ]
        ]
      }
    ]
  },
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
  "functionVersion": "$LATEST",
  "ColdStart": 1.0,
  "Service": "service_undefined",
  "logStreamId": "2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.974000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.AWSXRayRecorder <init>
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.config.DaemonConfiguration <init>
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000
 INFO: Environment variable AWS_XRAY_DAEMON_ADDRESS is set. Emitting to daemon on
 address XXXX.XXXX.XXXX.XXXX:2000.
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.331000
 09:24:37.294 [main] INFO  helloworld.App - {"version":null,"resource":"/
hello","path":"/hello/","httpMethod":"GET","headers":{"Accept":"*/
*","CloudFront-Forwarded-Proto":"https","CloudFront-Is-Desktop-
```

```
Viewer":"true","CloudFront-Is-Mobile-Viewer":"false","CloudFront-Is-
SmartTV-Viewer":"false","CloudFront-Is-Tablet-Viewer":"false","CloudFront-
Viewer-ASN":"16509","CloudFront-Viewer-Country":"IE","Host":"XXXX.execute-
api.eu-central-1.amazonaws.com","User-Agent":"curl/7.86.0","Via":"2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)","X-Amz-
Cf-Id":"t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q==","X-
Amzn-Trace-Id":"Root=1-63dcd2d1-25f90b9d1c753a783547f4dd","X-Forwarded-
For":"XX.XXX.XXX.XX, XX.XXX.XXX.XX","X-Forwarded-Port":"443","X-
Forwarded-Proto":"https"},"multiValueHeaders":{"Accept":["*/
*"],"CloudFront-Forwarded-Proto":["https"],"CloudFront-Is-Desktop-Viewer":
["true"],"CloudFront-Is-Mobile-Viewer":["false"],"CloudFront-Is-SmartTV-
Viewer":["false"],"CloudFront-Is-Tablet-Viewer":["false"],"CloudFront-Viewer-
ASN":["16509"],"CloudFront-Viewer-Country":["IE"],"Host":["XXXX.execute-
api.eu-central-1.amazonaws.com"],"User-Agent":["curl/7.86.0"],"Via":["2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)"],"X-Amz-
Cf-Id":["t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q=="],"X-
Amzn-Trace-Id":["Root=1-63dcd2d1-25f90b9d1c753a783547f4dd"],"X-Forwarded-
For":["XXX, XXX"],"X-Forwarded-Port":["443"],"X-Forwarded-Proto":
["https"]},"queryStringParameters":null,"multiValueQueryStringParameters":null,"pathParamet
{"accountId":"XXX","stage":"Prod","resourceId":"at73a1","requestId":"ba09ecd2-
acf3-40f6-89af-fad32df67597","operationName":null,"identity":
{"cognitoIdentityPoolId":null,"accountId":null,"cognitoIdentityId":null,"caller":null,"apiK
hello","httpMethod":"GET","apiId":"XXX","path":"/Prod/
hello/","authorizer":null},"body":null,"isBase64Encoded":false}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.351000
 09:24:37.351 [main] INFO  helloworld.App - Retrieving https://
checkip.amazonaws.com
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.313000 {
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "xray_trace_id": "1-63dcd2d1-25f90b9d1c753a783547f4dd",
  "functionVersion": "$LATEST",
  "Service": "service_undefined",
  "logStreamId": "2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 END
 RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000
 REPORT RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765    Duration: 4118.98 ms
 Billed Duration: 5275 ms    Memory Size: 512 MB    Max Memory Used: 152 MB    Init
 Duration: 1155.47 ms
```

```
XRAY TraceId: 1-63dcd2d1-25f90b9d1c753a783547f4dd    SegmentId: 3a028fee19b895cb
 Sampled: true
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Viewing logs in the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Viewing logs in the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/*your-function-name***).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

Viewing logs using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
```

```
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Sample logging code

The GitHub repository for this guide includes sample applications that demonstrate the use of various logging configurations. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Sample Lambda applications in Java

- example-java – A Java function that demonstrates how you can use Lambda to process orders. This function illustrates how to define and deserialize a custom input event object, use the AWS SDK, and output logging.

- java-basic – A collection of minimal Java functions with unit tests and variable logging configuration.
- java-events – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the aws-lambda-java-events library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- s3-java – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- layer-java – A Java function that illustrates how to use a Lambda layer to package dependencies separate from your core function code.

The `java-basic` sample application shows a minimal logging configuration that supports logging tests. The handler code uses the `LambdaLogger` logger provided by the context object. For tests, the application uses a custom `TestLogger` class that implements the `LambdaLogger` interface with a Log4j2 logger. It uses SLF4J as a facade for compatibility with the AWS SDK. Logging libraries are excluded from build output to keep the deployment package small.

Instrumenting Java code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- AWS Distro for OpenTelemetry (ADOT) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- AWS X-Ray SDK for Java – An SDK for generating and sending trace data to X-Ray.
- Powertools for AWS Lambda (Java) – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

> **Important**
>
> The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs.

Sections

- Using Powertools for AWS Lambda (Java) and AWS SAM for tracing
- Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing
- Using ADOT to instrument your Java functions
- Using the X-Ray SDK to instrument your Java functions
- Activating tracing with the Lambda console
- Activating tracing with the Lambda API
- Activating tracing with CloudFormation

- Interpreting an X-Ray trace
- Storing runtime dependencies in a layer (X-Ray SDK)
- X-Ray tracing in sample applications (X-Ray SDK)

Using Powertools for AWS Lambda (Java) and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated Powertools for AWS Lambda (Java) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11 or later
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World Java template.

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press `Enter`.

   > **Note**
   >
   > For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter `y`.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

   If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the traces for the function, run sam traces.

```
sam traces
```

   The trace output looks like this:

```
New XRay Service Graph
  Start time: 2025-02-03 14:31:48+01:00
  End time: 2025-02-03 14:31:48+01:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-y9Iu1FLJJBGD -
 Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 5.587
  Reference Id: 1 - client - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: [0]
   Summary_statistics:
```

```
    - total requests: 0
    - ok count(2XX): 0
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 3] at (2025-02-03T14:31:48.500000) with id
 (1-63dd0cc4-3c869dec72a586875da39777) and duration (5.603s)
 - 5.587s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD [HTTP: 200]
 - 4.053s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD
   - 1.181s - Initialization
   - 4.037s - Invocation
     - 1.981s - ## handleRequest
       - 1.840s - ## getPageContents
   - 0.000s - Overhead
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Using Powertools for AWS Lambda (Java) and the AWS CDK for tracing

Follow the steps below to download, build, and deploy a sample Hello World Java application with integrated Powertools for AWS Lambda (Java) modules using the AWS CDK. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- Java 11 or later
- AWS CLI version 2
- AWS CDK version 2

- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS CDK application

1. Create a project directory for your new application.

```
mkdir hello-world
cd hello-world
```

2. Initialize the app.

```
cdk init app --language java
```

3. Create a maven project with the following command:

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. Open `pom.xml` in the `hello-world\app\Function` directory and replace the existing code with the following code that includes dependencies and maven plugins for Powertools.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>helloworld</groupId>
  <artifactId>Function</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Function</name>
  <url>http://maven.apache.org</url>
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <log4j.version>2.17.2</log4j.version>
</properties>
    <dependencies>
        <dependency>
```

```
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-tracing</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-metrics</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-logging</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.2</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-events</artifactId>
            <version>3.11.1</version>
        </dependency>
  </dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>aspectj-maven-plugin</artifactId>
            <version>1.14.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
                <complianceLevel>${maven.compiler.target}</complianceLevel>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
```

```
                        <artifactId>powertools-tracing</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-metrics</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-logging</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

 implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCache
                                </transformer>
                            </transformers>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>

                        </configuration>
                    </execution>
                </executions>
```

```
                <dependencies>
                    <dependency>
                        <groupId>com.github.edwgiz</groupId>
                        <artifactId>maven-shade-plugin.log4j2-cachefile-
transformer</artifactId>
                        <version>2.15</version>
                    </dependency>
                </dependencies>
        </plugin>
    </plugins>
</build>
</project>
```

5. Create the `hello-world\app\src\main\resource` directory and create `log4j.xml` for the log configuration.

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. Open `log4j.xml` and add the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="JsonAppender" target="SYSTEM_OUT">
            <JsonTemplateLayout
 eventTemplateUri="classpath:LambdaJsonLayout.json" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="JsonLogger" level="INFO" additivity="false">
            <AppenderRef ref="JsonAppender"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="JsonAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

7. Open `App.java` from the `hello-world\app\Function\src\main\java\helloworld` directory and replace the existing code with the following code. This is the code for the Lambda function.

```
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.metrics.Metrics;
import software.amazon.lambda.powertools.tracing.CaptureMode;
import software.amazon.lambda.powertools.tracing.Tracing;

import static software.amazon.lambda.powertools.tracing.CaptureMode.*;

/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent,
 APIGatewayProxyResponseEvent> {
    Logger log = LogManager.getLogger(App.class);


    @Logging(logEvent = true)
    @Tracing(captureMode = DISABLED)
    @Metrics(captureColdStart = true)
    public APIGatewayProxyResponseEvent handleRequest(final
 APIGatewayProxyRequestEvent input, final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");

        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
                .withHeaders(headers);
        try {
```

```
            final String pageContents = this.getPageContents("https://
checkip.amazonaws.com");
            String output = String.format("{ \"message\": \"hello world\",
 \"location\": \"%s\" }", pageContents);

            return response
                    .withStatusCode(200)
                    .withBody(output);
        } catch (IOException e) {
            return response
                    .withBody("{}")
                    .withStatusCode(500);
        }
    }
    @Tracing(namespace = "getPageContents")
    private String getPageContents(String address) throws IOException {
        log.info("Retrieving {}", address);
        URL url = new URL(address);
        try (BufferedReader br = new BufferedReader(new
 InputStreamReader(url.openStream()))) {
            return br.lines().collect(Collectors.joining(System.lineSeparator()));
        }
    }
}
```

8. Open `HelloWorldStack.java` from the `hello-world\src\main\java\com\myorg` directory and replace the existing code with the following code. This code uses Lambda Constructorand the ApiGatewayv2 Constructor to create a REST API and a Lambda function.

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.apigatewayv2.alpha.*;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegration;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegrationProps
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.FunctionProps;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Tracing;
import software.amazon.awscdk.services.logs.RetentionDays;
```

```
import software.amazon.awscdk.services.s3.assets.AssetOptions;
import software.constructs.Construct;

import java.util.Arrays;
import java.util.List;

import static java.util.Collections.singletonList;
import static software.amazon.awscdk.BundlingOutput.ARCHIVED;

public class HelloWorldStack extends Stack {
    public HelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloWorldStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        List<String> functionPackagingInstructions = Arrays.asList(
                "/bin/sh",
                "-c",
                "cd Function " +
                        "&& mvn clean install " +
                        "&& cp /asset-input/Function/target/function.jar /asset-
output/"
        );
        BundlingOptions.Builder builderOptions = BundlingOptions.builder()
                .command(functionPackagingInstructions)
                .image(Runtime.JAVA_11.getBundlingImage())
                .volumes(singletonList(
                        // Mount local .m2 repo to avoid download all the
 dependencies again inside the container
                        DockerVolume.builder()
                                .hostPath(System.getProperty("user.home") +
 "/.m2/")
                                .containerPath("/root/.m2/")
                                .build()
                ))
                .user("root")
                .outputType(ARCHIVED);

        Function function = new Function(this, "Function", FunctionProps.builder()
                .runtime(Runtime.JAVA_11)
                .code(Code.fromAsset("app", AssetOptions.builder()
```

```
                        .bundling(builderOptions
                                .command(functionPackagingInstructions)
                                .build())
                        .build()))
                .handler("helloworld.App::handleRequest")
                .memorySize(1024)
                .tracing(Tracing.ACTIVE)
                .timeout(Duration.seconds(10))
                .logRetention(RetentionDays.ONE_WEEK)
                .build());

        HttpApi httpApi = new HttpApi(this, "sample-api", HttpApiProps.builder()
                .apiName("sample-api")
                .build());

        httpApi.addRoutes(AddRoutesOptions.builder()
                .path("/")
                .methods(singletonList(HttpMethod.GET))
                .integration(new HttpLambdaIntegration("function", function,
 HttpLambdaIntegrationProps.builder()
                        .payloadFormatVersion(PayloadFormatVersion.VERSION_2_0)
                        .build()))
                .build());

        new CfnOutput(this, "HttpApi", CfnOutputProps.builder()
                .description("Url for Http Api")
                .value(httpApi.getApiEndpoint())
                .build());
    }
}
```

9. Open `pom.xml` from the `hello-world` directory and replace the existing code with the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myorg</groupId>
    <artifactId>hello-world</artifactId>
```

```
    <version>0.1</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cdk.version>2.70.0</cdk.version>
        <constructs.version>[10.0.0,11.0.0)</constructs.version>
        <junit.version>5.7.1</junit.version>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <mainClass>com.myorg.HelloWorldApp</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <!-- AWS Cloud Development Kit -->
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>aws-cdk-lib</artifactId>
            <version>${cdk.version}</version>
        </dependency>
        <dependency>
            <groupId>software.constructs</groupId>
            <artifactId>constructs</artifactId>
            <version>${constructs.version}</version>
        </dependency>
```

```
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter</artifactId>
          <version>${junit.version}</version>
          <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-integrations-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
    </dependencies>
</project>
```

10. Make sure you're in the `hello-world` directory and deploy your application.

```
cdk deploy
```

11. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

    If successful, you'll see this response:

```
{"message":"hello world"}
```

13. To get the traces for the function, run sam traces.

```
sam traces
```

    The trace output looks like this:

```
New XRay Service Graph
  Start time: 2025-02-03 14:59:50+00:00
  End time: 2025-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2025-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

14. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
cdk destroy
```

Using ADOT to instrument your Java functions

ADOT provides fully managed Lambda layers that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see Custom configuration for the ADOT Collector on Lambda in the ADOT documentation.

For Java runtimes, you can choose between two layers to consume:

- **AWS managed Lambda layer for ADOT Java (Auto-instrumentation Agent)** – This layer automatically transforms your function code at startup to collect tracing data. For detailed instructions on how to consume this layer together with the ADOT Java agent, see AWS Distro for OpenTelemetry Lambda Support for Java (Auto-instrumentation Agent) in the ADOT documentation.
- **AWS managed Lambda layer for ADOT Java** – This layer also provides built-in instrumentation for Lambda functions, but it requires a few manual code changes to initialize the OTel SDK. For detailed instructions on how to consume this layer, see AWS Distro for OpenTelemetry Lambda Support for Java in the ADOT documentation.

Using the X-Ray SDK to instrument your Java functions

To record data about calls that your function makes to other resources and services in your application, you can add the X-Ray SDK for Java to your build configuration. The following example shows a Gradle build configuration that includes the libraries that activate automatic instrumentation of AWS SDK for Java 2.x clients.

Example build.gradle – Tracing dependencies

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.16.1')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    ...
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    ...
}
```

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Under **Additional monitoring tools**, choose **Edit**.
5. Under **CloudWatch Application Signals and AWS X-Ray**, choose **Enable** for **Lambda service traces**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an CloudFormation template, use the `TracingConfig` property.

Example function-inline.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example template.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpreting an X-Ray trace

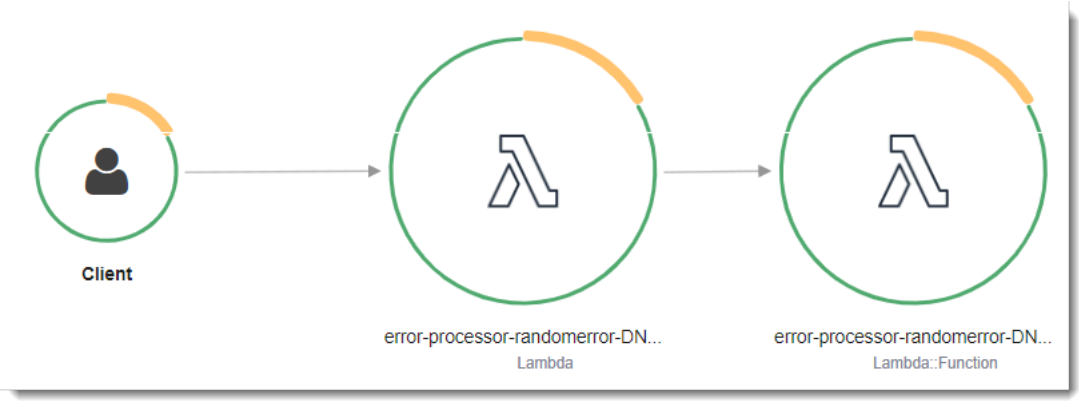
Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

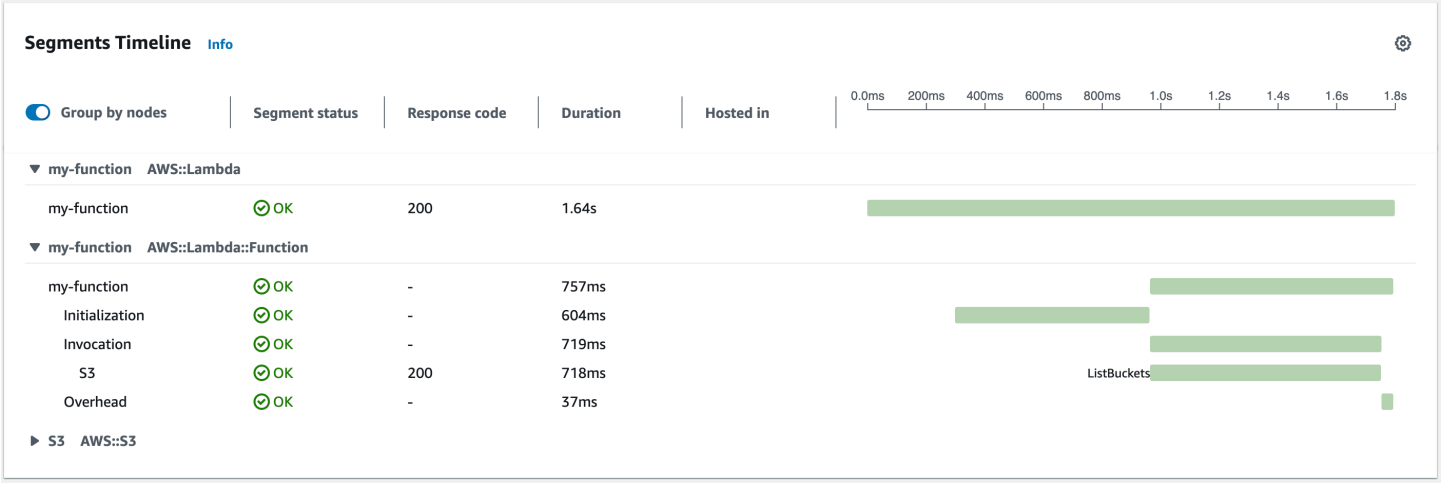


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

In X-Ray, a *trace* records information about a request that is processed by one or more *services*. Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes:



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.



This example expands the `AWS::Lambda::Function` segment to show its three subsegments.

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
> The example trace shown here illustrates the old-style function segment. The differences between the old- and new-style segments are described in the following paragraphs.
> These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

The old-style function segment contains the following subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

The new-style function segment doesn't contain an `Invocation` subsegment. Instead, customer subsegments are attached directly to the function segment. For more information about the structure of the old- and new-style function segments, see the section called “Understanding X-Ray traces”.

Note

Lambda SnapStart functions also include a `Restore` subsegment. The `Restore` subsegment shows the time it takes for Lambda to restore a snapshot, load the runtime, and run any after-restore runtime hooks. The process of restoring snapshots can include time spent on activities outside the MicroVM. This time is reported in the `Restore` subsegment. You aren't charged for the time spent outside the microVM to restore a snapshot.

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see AWS X-Ray SDK for Java in the *AWS X-Ray Developer Guide*.

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

Storing runtime dependencies in a layer (X-Ray SDK)

If you use the X-Ray SDK to instrument AWS SDK clients your function code, your deployment package can become quite large. To avoid uploading runtime dependencies every time you update your function code, package the X-Ray SDK in a Lambda layer.

The following example shows an `AWS::Serverless::LayerVersion` resource that stores the AWS SDK for Java and X-Ray SDK for Java.

Example template.yml – Dependencies layer

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/blank-java.zip
      Tracing: Active
      Layers:
```

```
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-java-lib
      Description: Dependencies for the blank-java sample app.
      ContentUri: build/blank-java-lib.zip
      CompatibleRuntimes:
        - java25
```

With this configuration, you update the library layer only if you change your runtime dependencies. Since the function deployment package contains only your code, this can help reduce upload times.

Creating a layer for dependencies requires build configuration changes to generate the layer archive prior to deployment. For a working example, see the java-basic sample application on GitHub.

X-Ray tracing in sample applications (X-Ray SDK)

The GitHub repository for this guide includes sample applications that demonstrate the use of X-Ray tracing. Each sample application includes scripts for easy deployment and cleanup, an AWS SAM template, and supporting resources.

Sample Lambda applications in Java

- example-java – A Java function that demonstrates how you can use Lambda to process orders. This function illustrates how to define and deserialize a custom input event object, use the AWS SDK, and output logging.
- java-basic – A collection of minimal Java functions with unit tests and variable logging configuration.
- java-events – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the aws-lambda-java-events library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- s3-java – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- layer-java – A Java function that illustrates how to use a Lambda layer to package dependencies separate from your core function code.

All of the sample applications have active tracing enabled for Lambda functions. For example, the `s3-java` application shows automatic instrumentation of AWS SDK for Java 2.x clients, segment management for tests, custom subsegments, and the use of Lambda layers to store runtime dependencies.

Java sample applications for AWS Lambda

The GitHub repository for this guide provides sample applications that demonstrate the use of Java in AWS Lambda. Each sample application includes scripts for easy deployment and cleanup, an CloudFormation template, and supporting resources.

Sample Lambda applications in Java

- example-java – A Java function that demonstrates how you can use Lambda to process orders. This function illustrates how to define and deserialize a custom input event object, use the AWS SDK, and output logging.
- java-basic – A collection of minimal Java functions with unit tests and variable logging configuration.
- java-events – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the aws-lambda-java-events library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- s3-java – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- layer-java – A Java function that illustrates how to use a Lambda layer to package dependencies separate from your core function code.

Running popular Java frameworks on Lambda

- spring-cloud-function-samples – An example from Spring that shows how to use the Spring Cloud Function framework to create AWS Lambda functions.
- Serverless Spring Boot Application Demo – An example that shows how to set up a typical Spring Boot application in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime.
- Serverless Micronaut Application Demo – An example that shows how to use Micronaut in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime. Learn more in the Micronaut/Lambda guides.
- Serverless Quarkus Application Demo – An example that shows how to use Quarkus in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime. Learn more in the Quarkus/Lambda guide and Quarkus/SnapStart guide.

If you're new to Lambda functions in Java, start with the `java-basic` examples. To get started with Lambda event sources, see the `java-events` examples. Both of these example sets show the use of Lambda's Java libraries, environment variables, the AWS SDK, and the AWS X-Ray SDK. These examples require minimal setup and you can deploy them from the command line in less than a minute.

Building Lambda functions with Go

Go is implemented differently than other managed runtimes. Because Go compiles natively to an executable binary, it doesn't require a dedicated language runtime. Use an OS-only runtime (the `provided` runtime family) to deploy Go functions to Lambda.

Topics

- Go runtime support
- Tools and libraries
- Define Lambda function handlers in Go
- Using the Lambda context object to retrieve Go function information
- Deploy Go Lambda functions with .zip file archives
- Deploy Go Lambda functions with container images
- Working with layers for Go Lambda functions
- Log and monitor Go Lambda functions
- Instrumenting Go code in AWS Lambda

Go runtime support

The Go 1.x managed runtime for Lambda is deprecated. If you have functions that use the Go 1.x runtime, you must migrate your functions to `provided.al2023` or `provided.al2`. The `provided.al2023` and `provided.al2` runtimes offer several advantages over `go1.x`, including support for the arm64 architecture (AWS Graviton2 processors), smaller binaries, and slightly faster invoke times.

No code changes are required for this migration. The only required changes relate to how you build your deployment package and which runtime you use to create your function. For more information, see Migrating AWS Lambda functions from the Go1.x runtime to the custom runtime on Amazon Linux 2 on the *AWS Compute Blog*.

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
OS-only Runtime	`provided.al2023`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029
OS-only Runtime	`provided.al2`	Amazon Linux 2	Jul 31, 2026	Aug 31, 2026	Sep 30, 2026

Tools and libraries

Lambda provides the following tools and libraries for the Go runtime:

- AWS SDK for Go v2: The official AWS SDK for the Go programming language.
- github.com/aws/aws-lambda-go/lambda: The implementation of the Lambda programming model for Go. This package is used by AWS Lambda to invoke your handler.
- github.com/aws/aws-lambda-go/lambdacontext: Helpers for accessing context information from the context object.
- github.com/aws/aws-lambda-go/events: This library provides type definitions for common event source integrations.
- github.com/aws/aws-lambda-go/cmd/build-lambda-zip: This tool can be used to create a .zip file archive on Windows.

For more information, see aws-lambda-go on GitHub.

Lambda provides the following sample applications for the Go runtime:

Sample Lambda applications in Go

- go-al2 – A hello world function that returns the public IP address. This app uses the `provided.al2` custom runtime.
- blank-go – A Go function that shows the use of Lambda's Go libraries, logging, environment variables, and the AWS SDK. This app uses the `go1.x` runtime.

Define Lambda function handlers in Go

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

This page describes how to work with Lambda function handlers in Go, including project setup, naming conventions, and best practices. This page also includes an example of a Go Lambda function that takes in information about an order, produces a text file receipt, and puts this file in an Amazon Simple Storage Service (Amazon S3) bucket. For information about how to deploy your function after writing it, see the section called "Deploy .zip file archives" or the section called "Deploy container images".

Topics

- Setting up your Go handler project
- Example Go Lambda function code
- Handler naming conventions
- Defining and accessing the input event object
- Accessing and using the Lambda context object
- Valid handler signatures for Go handlers
- Using the AWS SDK for Go v2 in your handler
- Accessing environment variables
- Using global state
- Code best practices for Go Lambda functions

Setting up your Go handler project

A Lambda function written in Go is authored as a Go executable. You can initialize a Go Lambda function project the same way you initialize any other Go project using the following `go mod init` command:

```
go mod init example-go
```

Here, `example-go` is the module name. You can replace this with anything. This command initializes your project and generates the `go.mod` file that lists your project's dependencies.

Use the `go get` command to add any external dependencies to your project. For example, for all Lambda functions in Go, you must include the github.com/aws/aws-lambda-go/lambda package, which implements the Lambda programming model for Go. Include this package with the following `go get` command:

```
go get github.com/aws/aws-lambda-go
```

Your function code should live in a Go file. In the following example, we name this file `main.go`. In this file, you implement your core function logic in a handler method, as well as a `main()` function that calls this handler.

Example Go Lambda function code

The following example Go Lambda function code takes in information about an order, produces a text file receipt, and puts this file in an Amazon S3 bucket.

Example `main.go` Lambda function

```
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "log"
 "os"
 "strings"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

type Order struct {
 OrderID string  `json:"order_id"`
 Amount  float64 `json:"amount"`
 Item    string  `json:"item"`
}

var (
 s3Client *s3.Client
)
```

```
func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Fatalf("unable to load SDK config, %v", err)
 }

 s3Client = s3.NewFromConfig(cfg)
}

func uploadReceiptToS3(ctx context.Context, bucketName, key, receiptContent string)
 error {
 _, err := s3Client.PutObject(ctx, &s3.PutObjectInput{
  Bucket: &bucketName,
  Key:    &key,
  Body:   strings.NewReader(receiptContent),
 })
 if err != nil {
  log.Printf("Failed to upload receipt to S3: %v", err)
  return err
 }
 return nil
}

func handleRequest(ctx context.Context, event json.RawMessage) error {
 // Parse the input event
 var order Order
 if err := json.Unmarshal(event, &order); err != nil {
  log.Printf("Failed to unmarshal event: %v", err)
  return err
 }

 // Access environment variables
 bucketName := os.Getenv("RECEIPT_BUCKET")
 if bucketName == "" {
  log.Printf("RECEIPT_BUCKET environment variable is not set")
  return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
 }

 // Create the receipt content and key destination
 receiptContent := fmt.Sprintf("OrderID: %s\nAmount: $%.2f\nItem: %s",
  order.OrderID, order.Amount, order.Item)
 key := "receipts/" + order.OrderID + ".txt"
```

```
 // Upload the receipt to S3 using the helper method
 if err := uploadReceiptToS3(ctx, bucketName, key, receiptContent); err != nil {
  return err
 }

 log.Printf("Successfully processed order %s and stored receipt in S3 bucket %s",
 order.OrderID, bucketName)
 return nil
}

func main() {
 lambda.Start(handleRequest)
}
```

This `main.go` file contains the following sections of code:

- `package main`: In Go, the package containing your `func main()` function must always be named `main`.
- `import` block: Use this block to include libraries that your Lambda function requires.
- `type Order struct {}` block: Define the shape of the expected input event in this Go struct.
- `var ()` block: Use this block to define any global variables that you'll use in your Lambda function.
- `func init() {}`: Include any code you want Lambda to run during the during the initialization phase in this `init()` method.
- `func uploadReceiptToS3(...) {}`: This is a helper method that's referenced by the main `handleRequest` handler method.
- `func handleRequest(ctx context.Context, event json.RawMessage) error {}`: This is the **main handler method**, which contains your main application logic.
- `func main() {}`: This is a required entry point for your Lambda handler. The argument to the `lambda.Start()` method is your main handler method.

For this function to work properly, its execution role must allow the `s3:PutObject` action. Also, ensure that you define the `RECEIPT_BUCKET` environment variable. After a successful invocation, the Amazon S3 bucket should contain a receipt file.

Handler naming conventions

For Lambda functions in Go, you can use any name for the handler. In this example, the handler method name is `handleRequest`. To reference the handler value in your code, you can use the _HANDLER environment variable.

For Go functions deployed using a .zip deployment package, the executable file that contains your function code must be named `bootstrap`. In addition, the `bootstrap` file must be at the root of the .zip file. For Go functions deployed using a container image, you can use any name for the executable file.

Defining and accessing the input event object

JSON is the most common and standard input format for Lambda functions. In this example, the function expects an input similar to the following:

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

When working with Lambda functions in Go, you can define the shape of the expected input event as a Go struct. In this example, we define a struct to represent an `Order`:

```
type Order struct {
    OrderID string  `json:"order_id"`
    Amount  float64 `json:"amount"`
    Item    string  `json:"item"`
}
```

This struct matches the expected input shape. After you define your struct, you can write a handler signature that takes in a generic JSON type compatible with the encoding/json standard library. You can then deserialize it into your struct using the func Unmarshal function. This is illustrated in the first few lines of the handler:

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    // Parse the input event
    var order Order
    if err := json.Unmarshal(event, &order); err != nil {
```

```
        log.Printf("Failed to unmarshal event: %v", err)
        return err
    ...
}
```

After this deserialization, you can access the fields of the `order` variable. For example, `order.OrderID` retrieves the value of `"order_id"` from the original input.

> **Note**
>
> The `encoding/json` package can access only exported fields. To be exported, field names in the event struct must be capitalized.

Accessing and using the Lambda context object

The Lambda context object contains information about the invocation, function, and execution environment. In this example, we declared this variable as `ctx` in the handler signature:

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    ...
}
```

The `ctx context.Context` input is an optional argument in your function handler. For more information about accepted handler signatures, see the section called "Valid handler signatures for Go handlers".

If you make calls to other services using the AWS SDK, the context object is required in a few key areas. For example, to properly initialize your SDK clients, you can load the correct AWS SDK configuration using the context object as follows:

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
```

SDK calls themselves may require the context object as an input. For example, the `s3Client.PutObject` call accepts the context object as its first argument:

```
// Upload the receipt to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
```

```
    })
```

Outside of AWS SDK requests, you can also use the context object for function monitoring. For more information about the context object, see the section called “Context”.

Valid handler signatures for Go handlers

You have several options when building a Lambda function handler in Go, but you must adhere to the following rules:

- The handler must be a function.
- The handler may take between 0 and 2 arguments. If there are two arguments, the first argument must implement `context.Context`.
- The handler may return between 0 and 2 arguments. If there is a single return value, it must implement `error`. If there are two return values, the second value must implement `error`.

The following lists valid handler signatures. `TIn` and `TOut` represent types compatible with the *encoding/json* standard library. For more information, see func Unmarshal to learn how these types are deserialized.

- `func ()`
- `func () error`
- `func () (TOut, error)`
- `func (TIn) error`
- `func (TIn) (TOut, error)`
- `func (context.Context) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) error`
- `func (context.Context, TIn) (TOut, error)`

Using the AWS SDK for Go v2 in your handler

Often, you'll use Lambda functions to interact with or make updates to other AWS resources. The simplest way to interface with these resources is to use the AWS SDK for Go v2.

> **Note**
>
> The AWS SDK for Go (v1) is in maintenance mode, and will reach end-of-support on July 31, 2025. We recommend that you use only the AWS SDK for Go v2 going forward.

To add SDK dependencies to your function, use the `go get` command for the specific SDK clients that you need. In the example code earlier, we used the `config` library and the `s3` library. Add these dependencies by running the following commands in the directory that contains your `go.mod` and `main.go` files:

```
go get github.com/aws/aws-sdk-go-v2/config
go get github.com/aws/aws-sdk-go-v2/service/s3
```

Then, import the dependencies accordingly in your function's import block:

```
import (
    ...
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
)
```

When using the SDK in your handler, configure your clients with the right settings. The simplest way to do this is to use the default credential provider chain. This example illustrates one way to load this configuration:

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    if err != nil {
        log.Printf("Failed to load AWS SDK config: %v", err)
        return err
    }
```

After loading this configuration into the `cfg` variable, you can pass this variable into client instantiations. The example code instantiates an Amazon S3 client as follows:

```
// Create an S3 client
    s3Client := s3.NewFromConfig(cfg)
```

In this example, we initialized our Amazon S3 client in the `init()` function to avoid having to initialize it every time we invoke our function. The problem is that in the `init()` function, Lambda doesn't have access to the context object. As a workaround, you can pass in a placeholder like `context.TODO()` during the initialization phase. Later, when you make a call using the client, pass in the full context object. This workaround is also described in the section called "Using the context in AWS SDK client initializations and calls".

After you configure and initialize your SDK client, you can then use it to interact with other AWS services. The example code calls the Amazon S3 `PutObject` API as follows:

```
_, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
    Bucket: &bucketName,
    Key:    &key,
    Body:   strings.NewReader(receiptContent),
})
```

Accessing environment variables

In your handler code, you can reference any environment variables by using the `os.Getenv()` method. In this example, we reference the defined `RECEIPT_BUCKET` environment variable using the following line of code:

```
// Access environment variables
    bucketName := os.Getenv("RECEIPT_BUCKET")
    if bucketName == "" {
        log.Printf("RECEIPT_BUCKET environment variable is not set")
        return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
    }
```

Using global state

To avoid creating new resources every time you invoke your function, you can declare and modify global variables outside of your Lambda function's handler code. You define these global variables in a `var` block or statement. In addition, your handler may declare an `init()` function that is executed during the initialization phase. The `init` method behaves the same in AWS Lambda as it does in standard Go programs.

Code best practices for Go Lambda functions

Adhere to the guidelines in the following list to use best coding practices when building your Lambda functions:

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation.

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of function invocations and escalated costs. If you see an unintended volume of invocations, set the function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

Using the Lambda context object to retrieve Go function information

In Lambda, the context object provides methods and properties with information about the invocation, function, and execution environment. When Lambda runs your function, it passes a context object to the handler. To use the context object in your handler, you can optionally declare it as an input parameter to your handler. The context object is necessary if you want to do the following in your handler:

- You need access to any of the global variables, methods, or properties offered by the context object. These methods and properties are useful for tasks like determining the entity that invoked your function or measuring the invocation time of your function, as illustrated in the section called "Accessing invoke context information".
- You need to use the AWS SDK for Go to make calls to other services. The context object is an important input parameter to most of these calls. For more information, see the section called "Using the context in AWS SDK client initializations and calls".

Topics

- Supported variables, methods, and properties in the context object
- Accessing invoke context information
- Using the context in AWS SDK client initializations and calls

Supported variables, methods, and properties in the context object

The Lambda context library provides the following global variables, methods, and properties.

Global variables

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The version of the function.
- `MemoryLimitInMB` – The amount of memory that's allocated for the function.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.

Context methods

- `Deadline` – Returns the date that the execution times out, in Unix time milliseconds.

Context properties

- `InvokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `AwsRequestID` – The identifier of the invocation request.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `ClientContext` – (mobile apps) Client context that's provided to Lambda by the client application.

Accessing invoke context information

Lambda functions have access to metadata about their environment and the invocation request. This can be accessed at Package context. Should your handler include `context.Context` as a parameter, Lambda will insert information about your function into the context's `Value` property. Note that you need to import the `lambdacontext` library to access the contents of the `context.Context` object.

```
package main

import (
        "context"
        "log"
        "github.com/aws/aws-lambda-go/lambda"
        "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
        lc, _ := lambdacontext.FromContext(ctx)
        log.Print(lc.Identity.CognitoIdentityPoolID)
}

func main() {
        lambda.Start(CognitoHandler)
```

```
}
```

In the example above, `lc` is the variable used to consume the information that the context object captured and `log.Print(lc.Identity.CognitoIdentityPoolID)` prints that information, in this case, the CognitoIdentityPoolID.

The following example introduces how to use the context object to monitor how long your Lambda function takes to complete. This allows you to analyze performance expectations and adjust your function code accordingly, if needed.

```
package main

import (
        "context"
        "log"
        "time"
        "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {

        deadline, _ := ctx.Deadline()
        deadline = deadline.Add(-100 * time.Millisecond)
        timeoutChannel := time.After(time.Until(deadline))

        for {

                select {

                case <- timeoutChannel:
                        return "Finished before timing out.", nil

                default:
                        log.Print("hello!")
                        time.Sleep(50 * time.Millisecond)
                }
        }
}

func main() {
        lambda.Start(LongRunningHandler)
}
```

Using the context in AWS SDK client initializations and calls

If your handler needs to use the AWS SDK for Go to make calls to other services, include the context object as an input to your handler. In AWS, it's a best practice to pass in the context object in most AWS SDK calls. For example, the Amazon S3 `PutObject` call accepts the context object (`ctx`) as its first argument:

```
// Upload an object to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

To initialize your SDK clients properly, you can also use the context object to load the correct configuration before passing that configuration object to the client:

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    ...
    s3Client = s3.NewFromConfig(cfg)
```

If you want to initialize your SDK clients outside of your main handler (i.e. during the initialization phase), you can pass in a placeholder context object:

```
func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 ...
 s3Client = s3.NewFromConfig(cfg)
}
```

If you initialize your clients this way, ensure that you pass in the correct context object in SDK calls from your main handler.

Deploy Go Lambda functions with .zip file archives

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

This page describes how to create a .zip file as your deployment package for the Go runtime, and then use the .zip file to deploy your function code to AWS Lambda using the AWS Management Console, AWS Command Line Interface (AWS CLI), and AWS Serverless Application Model (AWS SAM).

Note that Lambda uses POSIX file permissions, so you may need to set permissions for the deployment package folder before you create the .zip file archive.

Sections

- Creating a .zip file on macOS and Linux
- Creating a .zip file on Windows
- Creating and updating Go Lambda functions using .zip files

Creating a .zip file on macOS and Linux

The following steps show how to compile your executable using the `go build` command and create a .zip file deployment package for Lambda. Before compiling your code, make sure you have installed the lambda package from GitHub. This module provides an implementation of the runtime interface, which manages the interaction between Lambda and your function code. To download this library, run the following command.

```
go get github.com/aws/aws-lambda-go/lambda
```

If your function uses the AWS SDK for Go, download the standard set of SDK modules, along with any AWS service API clients required by your application. To learn how to install the SDK for Go, see Getting Started with the AWS SDK for Go V2.

Using the provided runtime family

Go is implemented differently than other managed runtimes. Because Go compiles natively to an executable binary, it doesn't require a dedicated language runtime. Use an OS-only runtime (the `provided` runtime family) to deploy Go functions to Lambda.

To create a .zip deployment package (macOS/Linux)

1. In the project directory that contains your application's `main.go` file, compile your executable. Note the following:

   - The executable must be named `bootstrap`. For more information, see Handler naming conventions.
   - Set your target instruction set architecture. OS-only runtimes support both arm64 and x86_64.
   - You can use the optional `lambda.norpc` tag to exclude the Remote Procedure Call (RPC) component of the lambda library. The RPC component is only required if you are using the deprecated Go 1.x runtime. Excluding the RPC reduces the size of the deployment package.

   For the arm64 architecture:

   ```
   GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
   ```

   For the x86_64 architecture:

   ```
   GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
   ```

2. (Optional) You may need to compile packages with `CGO_ENABLED=0` set on Linux:

   ```
   GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
    main.go
   ```

   This command creates a stable binary package for standard C library (`libc`) versions, which may be different on Lambda and other devices.

3. Create a deployment package by packaging the executable in a .zip file.

   ```
   zip myFunction.zip bootstrap
   ```

   > **Note**
   >
   > The `bootstrap` file must be at the root of the .zip file.

4. Create the function. Note the following:

- The binary must be named `bootstrap`, but the handler name can be anything. For more information, see Handler naming conventions.
- The `--architectures` option is only required if you're using arm64. The default value is x86_64.
- For `--role`, specify the Amazon Resource Name (ARN) of the execution role.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

Creating a .zip file on Windows

The following steps show how to download the build-lambda-zip tool for Windows from GitHub, compile your executable, and create a .zip deployment package.

> **Note**
>
> If you have not already done so, you must install git and then add the `git` executable to your Windows %PATH% environment variable.

Before compiling your code, make sure you have installed the lambda library from GitHub. To download this library, run the following command.

```
go get github.com/aws/aws-lambda-go/lambda
```

If your function uses the AWS SDK for Go, download the standard set of SDK modules, along with any AWS service API clients required by your application. To learn how to install the SDK for Go, see Getting Started with the AWS SDK for Go V2.

Using the provided runtime family

Go is implemented differently than other managed runtimes. Because Go compiles natively to an executable binary, it doesn't require a dedicated language runtime. Use an OS-only runtime (the `provided` runtime family) to deploy Go functions to Lambda.

To create a .zip deployment package (Windows)

1. Download the **build-lambda-zip** tool from GitHub.

   ```
   go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
   ```

2. Use the tool from your GOPATH to create a .zip file. If you have a default installation of Go, the tool is typically in `%USERPROFILE%\Go\bin`. Otherwise, navigate to where you installed the Go runtime and do one of the following:

   cmd.exe

   In cmd.exe, run one of the following, depending on your target instruction set architecture. OS-only runtimes support both arm64 and x86_64.

   You can use the optional `lambda.norpc` tag to exclude the Remote Procedure Call (RPC) component of the lambda library. The RPC component is only required if you are using the deprecated Go 1.x runtime. Excluding the RPC reduces the size of the deployment package.

   Example— For the x86_64 architecture

   ```
   set GOOS=linux
   set GOARCH=amd64
   set CGO_ENABLED=0
   go build -tags lambda.norpc -o bootstrap main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
   ```

   Example— For the arm64 architecture

   ```
   set GOOS=linux
   set GOARCH=arm64
   set CGO_ENABLED=0
   go build -tags lambda.norpc -o bootstrap main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
   ```

PowerShell

In PowerShell, run one of the following, depending on your target instruction set architecture. OS-only runtimes support both arm64 and x86_64.

You can use the optional `lambda.norpc` tag to exclude the Remote Procedure Call (RPC) component of the lambda library. The RPC component is only required if you are using the deprecated Go 1.x runtime. Excluding the RPC reduces the size of the deployment package.

For the x86_64 architecture:

```
$env:GOOS = "linux"
$env:GOARCH = "amd64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

For the arm64 architecture:

```
$env:GOOS = "linux"
$env:GOARCH = "arm64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

3. Create the function. Note the following:

- The binary must be named `bootstrap`, but the handler name can be anything. For more information, see Handler naming conventions.
- The `--architectures` option is only required if you're using arm64. The default value is x86_64.
- For `--role`, specify the Amazon Resource Name (ARN) of the execution role.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

Creating and updating Go Lambda functions using .zip files

After you have created your .zip deployment package, you can use it to create a new Lambda function or update an existing one. You can deploy your .zip package using the Lambda console, the AWS Command Line Interface, and the Lambda API. You can also create and update Lambda functions using AWS Serverless Application Model (AWS SAM) and CloudFormation.

The maximum size for a .zip deployment package for Lambda is 250 MB (unzipped). Note that this limit applies to the combined size of all the files you upload, including any Lambda layers.

The Lambda runtime needs permission to read the files in your deployment package. In Linux permissions octal notation, Lambda needs 644 permissions for non-executable files (rw-r--r--) and 755 permissions (rwxr-xr-x) for directories and executable files.

In Linux and MacOS, use the `chmod` command to change file permissions on files and directories in your deployment package. For example, to give a non-executable file the correct permissions, run the following command.

```
chmod 644 <filepath>
```

To change file permissions in Windows, see Set, View, Change, or Remove Permissions on an Object in the Microsoft Windows documentation.

> **Note**
>
> If you don't grant Lambda the permissions it needs to access directories in your deployment package, Lambda sets the permissions for those directories to 755 (rwxr-xr-x).

Creating and updating functions with .zip files using the console

To create a new function, you must first create the function in the console, then upload your .zip archive. To update an existing function, open the page for your function, then follow the same procedure to add your updated .zip file.

If your .zip file is less than 50MB, you can create or update a function by uploading the file directly from your local machine. For .zip files greater than 50MB, you must upload your package to an Amazon S3 bucket first. For instructions on how to upload a file to an Amazon S3 bucket using the

AWS Management Console, see Getting started with Amazon S3. To upload files using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> You cannot convert an existing container image function to use a .zip archive. You must create a new function.

To create a new function (console)

1. Open the Functions page of the Lambda console and choose **Create Function**.
2. Choose **Author from scratch**.
3. Under **Basic information**, do the following:
   a. For **Function name**, enter the name for your function.
   b. For **Runtime**, choose `provided.al2023`.
4. (Optional) Under **Permissions**, expand **Change default execution role**. You can create a new **Execution role** or use an existing one.
5. Choose **Create function**. Lambda creates a basic 'Hello world' function using your chosen runtime.

To upload a .zip archive from your local machine (console)

1. In the Functions page of the Lambda console, choose the function you want to upload the .zip file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **.zip file**.
5. To upload the .zip file, do the following:
   a. Select **Upload**, then select your .zip file in the file chooser.
   b. Choose **Open**.
   c. Choose **Save**.

To upload a .zip archive from an Amazon S3 bucket (console)

1. In the Functions page of the Lambda console, choose the function you want to upload a new .zip file for.
2. Select the **Code** tab.
3. In the **Code source** pane, choose **Upload from**.
4. Choose **Amazon S3 location**.
5. Paste the Amazon S3 link URL of your .zip file and choose **Save**.

Creating and updating functions with .zip files using the AWS CLI

You can can use the AWS CLI to create a new function or to update an existing one using a .zip file. Use the create-function and update-function-code commands to deploy your .zip package. If your .zip file is smaller than 50MB, you can upload the .zip package from a file location on your local build machine. For larger files, you must upload your .zip package from an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

> **Note**
>
> If you upload your .zip file from an Amazon S3 bucket using the AWS CLI, the bucket must be located in the same AWS Region as your function.

To create a new function using a .zip file with the AWS CLI, you must specify the following:

- The name of your function (`--function-name`)
- Your function's runtime (`--runtime`)
- The Amazon Resource Name (ARN) of your function's execution role (`--role`)
- The name of the handler method in your function code (`--handler`)

You must also specify the location of your .zip file. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda create-function --function-name myFunction \
```

```
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--code` option as shown in the following example command. You only need to use the `S3ObjectVersion` parameter for versioned objects.

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

To update an existing function using the CLI, you specify the the name of your function using the `--function-name` parameter. You must also specify the location of the .zip file you want to use to update your function code. If your .zip file is located in a folder on your local build machine, use the `--zip-file` option to specify the file path, as shown in the following example command.

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

To specify the location of .zip file in an Amazon S3 bucket, use the `--s3-bucket` and `--s3-key` options as shown in the following example command. You only need to use the `--s3-object-version` parameter for versioned objects.

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

Creating and updating functions with .zip files using the Lambda API

To create and update functions using a .zip file archive, use the following API operations:

- CreateFunction
- UpdateFunctionCode

Creating and updating functions with .zip files using AWS SAM

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build, package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. To learn more about using AWS SAM to build and deploy Lambda functions, see Getting started with AWS SAM in the *AWS Serverless Application Model Developer Guide*.

You can also use AWS SAM to create a Lambda function using an existing .zip file archive. To create a Lambda function using AWS SAM, you can save your .zip file in an Amazon S3 bucket or in a local folder on your build machine. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide*.

In your AWS SAM template, the `AWS::Serverless::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- `PackageType` - set to `Zip`
- `CodeUri` - set to the function code's Amazon S3 URI, path to local folder, or FunctionCode object
- `Runtime` - Set to your chosen runtime

With AWS SAM, if your .zip file is larger than 50MB, you don't need to upload it to an Amazon S3 bucket first. AWS SAM can upload .zip packages up to the maximum allowed size of 250MB (unzipped) from a location on your local build machine.

To learn more about deploying functions using .zip file in AWS SAM, see AWS::Serverless::Function in the *AWS SAM Developer Guide*.

Example: Using AWS SAM to build a Go function with provided.al2023

1. Create an AWS SAM template with the following properties:

   - **BuildMethod**: Specifies the compiler for your application. Use `go1.x`.
   - **Runtime**: Use `provided.al2023`.
   - **CodeUri**: Enter the path to your code.

- **Architectures**: Use `[arm64]` for the arm64 architecture. For the x86_64 instruction set architecture, use `[amd64]` or remove the `Architectures` property.

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Metadata:
      BuildMethod: go1.x
    Properties:
      CodeUri: hello-world/ # folder where your main program resides
      Handler: bootstrap
      Runtime: provided.al2023
      Architectures: [arm64]
```

2. Use the sam build command to compile the executable.

```
sam build
```

3. Use the sam deploy command to deploy the function to Lambda.

```
sam deploy --guided
```

Creating and updating functions with .zip files using CloudFormation

You can use CloudFormation to create a Lambda function using a .zip file archive. To create a Lambda function from a .zip file, you must first upload your file to an Amazon S3 bucket. For instructions on how to upload a file to an Amazon S3 bucket using the AWS CLI, see Move objects in the *AWS CLI User Guide.*

In your CloudFormation template, the `AWS::Lambda::Function` resource specifies your Lambda function. In this resource, set the following properties to create a function using a .zip file archive:

- `PackageType` - Set to `Zip`
- `Code` - Enter the Amazon S3 bucket name and the .zip file name in the `S3Bucket` and `S3Key` fields

- `Runtime` - Set to your chosen runtime

The .zip file that CloudFormation generates cannot exceed 4MB. To learn more about deploying functions using .zip file in CloudFormation, see AWS::Lambda::Function in the *CloudFormation User Guide*.

Deploy Go Lambda functions with container images

There are two ways to build a container image for a Go Lambda function:

- Using an AWS OS-only base image

  Go is implemented differently than other managed runtimes. Because Go compiles natively to an executable binary, it doesn't require a dedicated language runtime. Use an OS-only base image to build Go images for Lambda. To make the image compatible with Lambda, you must include the `aws-lambda-go/lambda` package in the image.
- Using a non-AWS base image

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the `aws-lambda-go/lambda` package in the image.

> **Tip**
>
> To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

This page explains how to build, test, and deploy container images for Lambda.

AWS base images for deploying Go functions

Go is implemented differently than other managed runtimes. Because Go compiles natively to an executable binary, it doesn't require a dedicated language runtime. Use an OS-only base image to deploy Go functions to Lambda.

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
OS-only Runtime	`provided.al2023`	Amazon Linux 2023	Jun 30, 2029	Jul 31, 2029	Aug 31, 2029

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
OS-only Runtime	`provided.al2`	Amazon Linux 2	Jul 31, 2026	Aug 31, 2026	Sep 30, 2026

Amazon Elastic Container Registry Public Gallery: gallery.ecr.aws/lambda/provided

Go runtime interface client

The `aws-lambda-go/lambda` package includes an implementation of the runtime interface. For examples of how to use `aws-lambda-go/lambda` in your image, see Using an AWS OS-only base image or Using a non-AWS base image.

Using an AWS OS-only base image

Go is implemented differently than other managed runtimes. Because Go compiles natively to an executable binary, it doesn't require a dedicated language runtime. Use an OS-only base image to build container images for Go functions.

Tags	Runtime	Operating system	Dockerfile	Deprecation
al2023	OS-only Runtime	Amazon Linux 2023	Dockerfile for OS-only Runtime on GitHub	Jun 30, 2029
al2	OS-only Runtime	Amazon Linux 2	Dockerfile for OS-only Runtime on GitHub	Jul 31, 2026

For more information about these base images, see provided in the Amazon ECR public gallery.

You must include the aws-lambda-go/lambda package with your Go handler. This package implements the programming model for Go, including the runtime interface.

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.
- Go

Creating an image from the provided.al2023 base image

To build and deploy a Go function with the `provided.al2023` base image

1. Create a directory for the project, and then switch to that directory.

```
mkdir hello
cd hello
```

2. Initialize a new Go module.

```
go mod init example.com/hello-world
```

3. Add the **lambda** library as a dependency of your new module.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Create a file named `main.go` and then open it in a text editor. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
```

```
}

func main() {
 lambda.Start(handler)
}
```

5. Use a text editor to create a Dockerfile in your project directory.

- The following example Dockerfile uses a multi-stage build. This allows you to use a different base image in each step. You can use one image, such as a Go base image, to compile your code and build the executable binary. You can then use a different image, such as `provided.al2023`, in the final `FROM` statement to define the image that you deploy to Lambda. The build process is separated from the final deployment image, so the final image only contains the files needed to run the application.
- You can use the optional `lambda.norpc` tag to exclude the Remote Procedure Call (RPC) component of the lambda library. The RPC component is only required if you are using the deprecated Go 1.x runtime. Excluding the RPC reduces the size of the deployment package.
- Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example— Multi-stage build Dockerfile

> **Note**
>
> Make sure that the version of Go that you specify in your Dockerfile (for example, `golang:1.20`) is the same version of Go that you used to create your application.

```
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build with optional lambda.norpc tag
COPY main.go .
RUN go build -tags lambda.norpc -o main main.go
# Copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2023
```

```
COPY --from=build /helloworld/main ./main
ENTRYPOINT [ "./main" ]
```

6. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test your image. The runtime interface emulator is included in the `provided.al2023` base image.

To run the runtime interface emulator on your local machine

1. Start the Docker image with the **docker run** command. Note the following:

   - `docker-image` is the image name and `test` is the tag.
   - `./main` is the ENTRYPOINT from your Dockerfile.

```
docker run -d -p 9000:8080 \
--entrypoint /usr/local/bin/aws-lambda-rie \
docker-image:test ./main
```

   This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

2. From a new terminal window, post an event to the following endpoint using a **curl** command:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. Some functions might require a JSON payload. Example:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

3. Get the container ID.

```
docker ps
```

4. Use the docker kill command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.
   - Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
   - Replace `111122223333` with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.
4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace <ECRrepositoryUri> with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

Example:

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.

7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Using a non-AWS base image

You can build a container image for Go from a non-AWS base image. The example Dockerfile in the following steps uses an Alpine base image.

You must include the aws-lambda-go/lambda package with your Go handler. This package implements the programming model for Go, including the runtime interface.

Prerequisites

To complete the steps in this section, you must have the following:

- AWS CLI version 2
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.
- Go

Creating an image from an alternative base image

To build and deploy a Go function with an Alpine base image

1. Create a directory for the project, and then switch to that directory.

```
mkdir hello
cd hello
```

2. Initialize a new Go module.

```
go mod init example.com/hello-world
```

3. Add the **lambda** library as a dependency of your new module.

```
go get github.com/aws/aws-lambda-go/lambda
```

4. Create a file named `main.go` and then open it in a text editor. This is the code for the Lambda function. You can use the following sample code for testing, or replace it with your own.

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. Use a text editor to create a Dockerfile in your project directory. The following example Dockerfile uses an Alpine base image. Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example Dockerfile

> **Note**
>
> Make sure that the version of Go that you specify in your Dockerfile (for example, `golang:1.20`) is the same version of Go that you used to create your application.

```
FROM golang:1.20.2-alpine3.16 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build
COPY main.go .
RUN go build -o main main.go
# Copy artifacts to a clean image
FROM alpine:3.16
COPY --from=build /helloworld/main /main
ENTRYPOINT [ "/main" ]
```

6. Build the Docker image with the docker build command. The following example names the image `docker-image` and gives it the `test` tag. To make your image compatible with Lambda, you must use the `--provenance=false` option.

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> The command specifies the `--platform linux/amd64` option to ensure that your container is compatible with the Lambda execution environment regardless of the architecture of your build machine. If you intend to create a Lambda function using the ARM64 instruction set architecture, be sure to change the command to use the `--platform linux/arm64` option instead.

(Optional) Test the image locally

Use the runtime interface emulator to locally test the image. You can build the emulator into your image or use the following procedure to install it on your local machine.

To install and run the runtime interface emulator on your local machine

1. From your project directory, run the following command to download the runtime interface emulator (x86-64 architecture) from GitHub and install it on your local machine.

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   To install the arm64 emulator, replace the GitHub repository URL in the previous command with the following:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   To install the arm64 emulator, replace the `$downloadLink` with the following:

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. Start the Docker image with the **docker run** command. Note the following:

- `docker-image` is the image name and `test` is the tag.
- `/main` is the ENTRYPOINT from your Dockerfile.

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /main
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /main
```

This command runs the image as a container and creates a local endpoint at `localhost:9000/2015-03-31/functions/function/invocations`.

> **Note**
>
> If you built the Docker image for the ARM64 instruction set architecture, be sure to use the `--platform linux/arm64` option instead of `--platform linux/amd64`.

3. Post an event to the local endpoint.

Linux/macOS

In Linux and macOS, run the following `curl` command:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

In PowerShell, run the following `Invoke-WebRequest` command:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

This command invokes the function with an empty event and returns a response. If you're using your own function code rather than the sample function code, you might want to invoke the function with a JSON payload. Example:

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. Get the container ID.

```
docker ps
```

5. Use the docker kill command to stop the container. In this command, replace `3766c4ab331c` with the container ID from the previous step.

```
docker kill 3766c4ab331c
```

Deploying the image

To upload the image to Amazon ECR and create the Lambda function

1. Run the get-login-password command to authenticate the Docker CLI to your Amazon ECR registry.

- Set the `--region` value to the AWS Region where you want to create the Amazon ECR repository.
- Replace 111122223333 with your AWS account ID.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> The Amazon ECR repository must be in the same AWS Region as the Lambda function.

If successful, you see a response like this:

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. Copy the `repositoryUri` from the output in the previous step.

4. Run the docker tag command to tag your local image into your Amazon ECR repository as the latest version. In this command:

   - `docker-image:test` is the name and tag of your Docker image. This is the image name and tag that you specified in the `docker build` command.
   - Replace `<ECRrepositoryUri>` with the `repositoryUri` that you copied. Make sure to include `:latest` at the end of the URI.

   ```
   docker tag docker-image:test <ECRrepositoryUri>:latest
   ```

   Example:

   ```
   docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

5. Run the docker push command to deploy your local image to the Amazon ECR repository. Make sure to include `:latest` at the end of the repository URI.

   ```
   docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
   ```

6. Create an execution role for the function, if you don't already have one. You need the Amazon Resource Name (ARN) of the role in the next step.
7. Create the Lambda function. For `ImageUri`, specify the repository URI from earlier. Make sure to include `:latest` at the end of the URI.

   ```
   aws lambda create-function \
     --function-name hello-world \
     --package-type Image \
     --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
     --role arn:aws:iam::111122223333:role/lambda-ex
   ```

   Note

   You can create a function using an image in a different AWS account, as long as the image is in the same Region as the Lambda function. For more information, see Amazon ECR cross-account permissions.

8. Invoke the function.

```
aws lambda invoke --function-name hello-world response.json
```

You should see a response like this:

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. To see the output of the function, check the `response.json` file.

To update the function code, you must build the image again, upload the new image to the Amazon ECR repository, and then use the update-function-code command to deploy the image to the Lambda function.

Lambda resolves the image tag to a specific image digest. This means that if you point the image tag that was used to deploy the function to a new image in Amazon ECR, Lambda doesn't automatically update the function to use the new image.

To deploy the new image to the same Lambda function, you must use the update-function-code command, even if the image tag in Amazon ECR remains the same. In the following example, the `--publish` option creates a new version of the function using the updated container image.

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

Working with layers for Go Lambda functions

We don't recommend using layers to manage dependencies for Lambda functions written in Go. This is because Lambda functions in Go compile into a single executable, which you provide to Lambda when you deploy your function. This executable contains your compiled function code, along with all of its dependencies. Using layers not only complicates this process, but also leads to increased cold start times because your functions need to manually load extra assemblies into memory during the init phase.

To use external dependencies with your Go handlers, include them directly in your deployment package. By doing so, you simplify the deployment process and also take advantage of built-in Go compiler optimizations. For an example of how to import and use a dependency like the AWS SDK for Go in your function, see the section called “Handler”.

Log and monitor Go Lambda functions

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see Sending Lambda function logs to CloudWatch Logs.

This page describes how to produce log output from your Lambda function's code, and access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- Creating a function that returns logs
- Viewing logs in the Lambda console
- Viewing logs in the CloudWatch console
- Viewing logs using the AWS Command Line Interface (AWS CLI)
- Deleting logs

Creating a function that returns logs

To output logs from your function code, you can use methods on the fmt package, or any logging library that writes to `stdout` or `stderr`. The following example uses the log package.

Example main.go – Logging

```
func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  // event
  eventJson, _ := json.MarshalIndent(event, "", "  ")
  log.Printf("EVENT: %s", eventJson)
  // environment variables
  log.Printf("REGION: %s", os.Getenv("AWS_REGION"))
  log.Println("ALL ENV VARS:")
  for _, element := range os.Environ() {
    log.Println(element)
  }
```

Example log format

```
START RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Version: $LATEST
```

```
2020/03/27 03:40:05 EVENT: {
  "Records": [
    {
      "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
      "md5OfBody": "7b27xmplb47ff90a553787216d55d91d",
      "md5OfMessageAttributes": "",
      "attributes": {
        "ApproximateFirstReceiveTimestamp": "1523232000001",
        "ApproximateReceiveCount": "1",
        "SenderId": "123456789012",
        "SentTimestamp": "1523232000000"
      },
      ...
2020/03/27 03:40:05 AWS_LAMBDA_LOG_STREAM_NAME=2020/03/27/
[$LATEST]569cxmplc3c34c7489e6a97ad08b4419
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_NAME=blank-go-function-9DV3XMPL6XBC
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_MEMORY_SIZE=128
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_VERSION=$LATEST
2020/03/27 03:40:05 AWS_EXECUTION_ENV=AWS_Lambda_go1.x
END RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71
REPORT RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Duration: 38.66 ms Billed
 Duration: 243 ms Memory Size: 128 MB Max Memory Used: 54 MB Init Duration: 203.69 ms
XRAY TraceId: 1-5e7d7595-212fxmpl9ee07c4884191322 SegmentId: 42ffxmpl0645f474 Sampled:
 true
```

The Go runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function. When invocations share an execution environment, Lambda reports the maximum memory used across all invocations. This behavior might result in a higher than expected reported value.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.

- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Viewing logs in the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Viewing logs in the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/*your-function-name***).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

Viewing logs using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
```

```
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Instrumenting Go code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of two SDK libraries:

- AWS Distro for OpenTelemetry (ADOT) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- AWS X-Ray SDK for Go – An SDK for generating and sending trace data to X-Ray.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

> **Important**
>
> The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs.

Sections

- Using ADOT to instrument your Go functions
- Using the X-Ray SDK to instrument your Go functions
- Activating tracing with the Lambda console
- Activating tracing with the Lambda API
- Activating tracing with CloudFormation
- Interpreting an X-Ray trace

Using ADOT to instrument your Go functions

ADOT provides fully managed Lambda layers that package everything you need to collect telemetry data using the OTel SDK. By consuming this layer, you can instrument your Lambda functions without having to modify any function code. You can also configure your layer to do custom initialization of OTel. For more information, see Custom configuration for the ADOT Collector on Lambda in the ADOT documentation.

For Go runtimes, you can add the **AWS managed Lambda layer for ADOT Go** to automatically instrument your functions. For detailed instructions on how to add this layer, see AWS Distro for OpenTelemetry Lambda Support for Go in the ADOT documentation.

Using the X-Ray SDK to instrument your Go functions

To record details about calls that your Lambda function makes to other resources in your application, you can also use the AWS X-Ray SDK for Go. To get the SDK, download the SDK from its GitHub repository with `go get`:

```
go get github.com/aws/aws-xray-sdk-go
```

To instrument AWS SDK clients, pass the client to the `xray.AWS()` method. You can then trace calls by using the `WithContext` version of the method.

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

After you add the correct dependencies and make the necessary code changes, activate tracing in your function's configuration via the Lambda console or the API.

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.

3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Under **Additional monitoring tools**, choose **Edit**.
5. Under **CloudWatch Application Signals and AWS X-Ray**, choose **Enable** for **Lambda service traces**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an CloudFormation template, use the `TracingConfig` property.

Example function-inline.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
```

```
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example template.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpreting an X-Ray trace

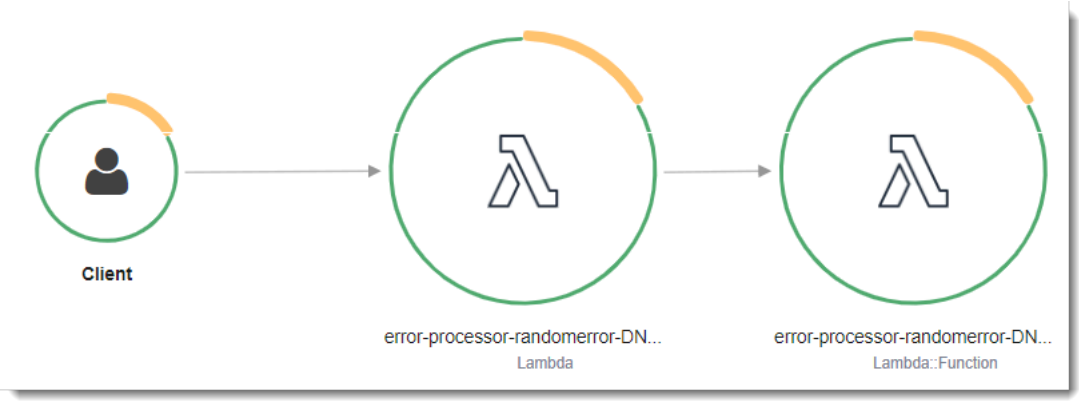
Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

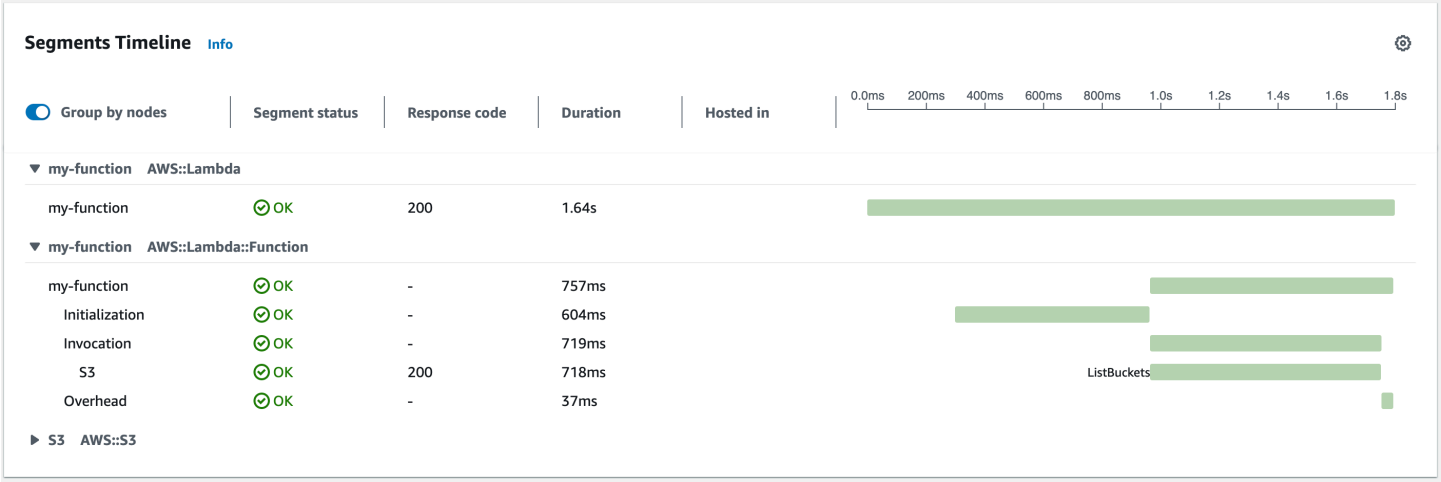


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

In X-Ray, a *trace* records information about a request that is processed by one or more *services*. Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes:



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.



This example expands the `AWS::Lambda::Function` segment to show its three subsegments.

Note

AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
The example trace shown here illustrates the old-style function segment. The differences between the old- and new-style segments are described in the following paragraphs.
These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

The old-style function segment contains the following subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

The new-style function segment doesn't contain an `Invocation` subsegment. Instead, customer subsegments are attached directly to the function segment. For more information about the structure of the old- and new-style function segments, see the section called "Understanding X-Ray traces".

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the AWS X-Ray SDK for Go in the *AWS X-Ray Developer Guide*.

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

Building Lambda functions with C#

You can run your .NET application in Lambda using the managed .NET 8 runtime, a custom runtime, or a container image. After your application code is compiled, you can deploy it to Lambda either as a .zip file or a container image. Lambda provides the following runtimes for .NET languages:

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
.NET 10	`dotnet10`	Amazon Linux 2023	Nov 14, 2028	Dec 14, 2028	Jan 15, 2029
.NET 9 (container only)	`dotnet9`	Amazon Linux 2023	Nov 10, 2026	Not scheduled	Not scheduled
.NET 8	`dotnet8`	Amazon Linux 2023	Nov 10, 2026	Dec 10, 2026	Jan 11, 2027

Setting up your .NET development environment

To develop and build your Lambda functions, you can use any of the commonly available .NET integrated development environments (IDEs), including Microsoft Visual Studio, Visual Studio Code, and JetBrains Rider. To simplify your development experience, AWS provides a set of .NET project templates, as well as the `Amazon.Lambda.Tools` command line interface (CLI).

Run the following .NET CLI commands to install these project templates and command line tools.

Installing the .NET project templates

To install the project templates, run the following command:

```
dotnet new install Amazon.Lambda.Templates
```

Installing and updating the CLI tools

Run the following commands to install, update, and uninstall the `Amazon.Lambda.Tools` CLI.

To install the command line tools:

```
dotnet tool install -g Amazon.Lambda.Tools
```

To update the command line tools:

```
dotnet tool update -g Amazon.Lambda.Tools
```

To uninstall the command line tools:

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

Define Lambda function handler in C#

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

This page describes how to work with Lambda function handlers in C# to work with the .NET managed runtime, including options for project setup, naming conventions, and best practices. This page also includes an example of a C# Lambda function that takes in information about an order, produces a text file receipt, and puts this file in an Amazon Simple Storage Service (S3) bucket. For information about how to deploy your function after writing it, see the section called “Deployment package” or the section called “Deploy container images”.

Topics

- Setting up your C# handler project
- Example C# Lambda function code
- Class library handlers
- Executable assembly handlers
- Valid handler signatures for C# functions
- Handler naming conventions
- Serialization in C# Lambda functions
- File-based functions
- Accessing and using the Lambda context object
- Using the SDK for .NET v3 in your handler
- Accessing environment variables
- Using global state
- Simplify function code with the Lambda Annotations framework
- Code best practices for C# Lambda functions

Setting up your C# handler project

When working with Lambda functions in C#, the process involves writing your code, then deploying your code to Lambda. There are two different execution models for deploying Lambda functions in .NET: the class library approach and the executable assembly approach.

In the class library approach, you package your function code as a .NET assembly (`.dll`) and deploy it to Lambda with the .NET managed runtime (`dotnet8`). For the handler name, Lambda expects a string in the format `AssemblyName::Namespace.Classname::Methodname`. During the function's initialization phase, your function's class is initialized, and any code in the constructor is run.

In the executable assembly approach, you use the top-level statements feature that was first introduced in C# 9. This approach generates an executable assembly which Lambda runs whenever it receives an invoke command for your function. In this approach, you also use the .NET managed runtime (`dotnet8`). For the handler name, you provide Lambda with the name of the executable assembly to run.

The main example on this page illustrates the class library approach. You can initialize your C# Lambda project in various ways, but the easiest way is to use the .NET CLI with the `Amazon.Lambda.Tools` CLI. Set up the `Amazon.Lambda.Tools` CLI by following the steps in the section called "Development environment". Then, initialize your project with the following command:

```
dotnet new lambda.EmptyFunction --name ExampleCS
```

This command generates the following file structure:

```
/project-root
    # src
        # ExampleCS
            # Function.cs (contains main handler)
            # Readme.md
            # aws-lambda-tools-defaults.json
            # ExampleCS.csproj
    # test
         # ExampleCS.Tests
            # FunctionTest.cs (contains main handler)
            # ExampleCS.Tests.csproj
```

In this file structure, the main handler logic for your function resides in the `Function.cs` file.

Example C# Lambda function code

The following example C# Lambda function code takes in information about an order, produces a text file receipt, and puts this file in an Amazon S3 bucket.

Example `Function.cs` Lambda function

```
using System;
using System.Text;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using Amazon.S3.Model;

// Assembly attribute to enable Lambda function logging
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace ExampleLambda;

public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}

public class OrderHandler
{
    private static readonly AmazonS3Client s3Client = new();

    public async Task<string> HandleRequest(Order order, ILambdaContext context)
    {
        try
        {
            string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
            if (string.IsNullOrWhiteSpace(bucketName))
            {
                throw new ArgumentException("RECEIPT_BUCKET environment variable is not
 set");
            }

            string receiptContent = $"OrderID: {order.OrderId}\nAmount:
 ${order.Amount:F2}\nItem: {order.Item}";
            string key = $"receipts/{order.OrderId}.txt";

            await UploadReceiptToS3(bucketName, key, receiptContent);
```

```
            context.Logger.LogInformation($"Successfully processed order
 {order.OrderId} and stored receipt in S3 bucket {bucketName}");
            return "Success";
        }
        catch (Exception ex)
        {
            context.Logger.LogError($"Failed to process order: {ex.Message}");
            throw;
        }
    }

    private async Task UploadReceiptToS3(string bucketName, string key, string
 receiptContent)
    {
        try
        {
            var putRequest = new PutObjectRequest
            {
                BucketName = bucketName,
                Key = key,
                ContentBody = receiptContent,
                ContentType = "text/plain"
            };

            await s3Client.PutObjectAsync(putRequest);
        }
        catch (AmazonS3Exception ex)
        {
            throw new Exception($"Failed to upload receipt to S3: {ex.Message}", ex);
        }
    }
}
```

This `Function.cs` file contains the following sections of code:

- `using` statements: Use these to import C# classes that your Lambda function requires.
- `[assembly: LambdaSerializer(...)]`: `LambdaSerializer` is an assembly attribute that tells Lambda to automatically convert JSON event payloads into C# objects before passing them to your function.
- `namespace ExampleLambda`: This defines the namespace. In C#, the namespace name doesn't have to match the filename.

- `public class Order {...}`: This defines the shape of the expected input event.
- `public class OrderHandler {...}`: This defines your C# class. Within it, you'll define the main handler method and any other helper methods.
- `private static readonly AmazonS3Client s3Client = new();`: This initializes an Amazon S3 client with the default credential provider chain, outside of the main handler method. This causes Lambda to run this code during the initialization phase.
- `public async ... HandleRequest (Order order, ILambdaContext context)`: This is the **main handler method**, which contains your main application logic.
- `private async Task UploadReceiptToS3(...) {}`: This is a helper method that's referenced by the main `handleRequest` handler method.

Because this function requires an Amazon S3 SDK client, you must add it to your project's dependencies. You can do so by navigating to `src/ExampleCS` and running the following command:

```
dotnet add package AWSSDK.S3
```

Add metadata information to aws-lambda-tools-defaults.json

By default, the generated `aws-lambda-tools-defaults.json` file doesn't contain `profile` or `region` information for your function. In addition, update the `function-handler` string to the correct value (`ExampleCS::ExampleLambda.OrderHandler::HandleRequest`). You can manually make this update and add the necessary metadata to use a specific credentials profile and region for your function. For example, your `aws-lambda-tools-defaults.json` file should look similar to this:

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual Studio
 and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "default",
  "region": "us-east-1",
```

```
  "configuration": "Release",
  "function-architecture": "x86_64",
  "function-runtime": "dotnet8",
  "function-memory-size": 512,
  "function-timeout": 30,
  "function-handler": "ExampleCS::ExampleLambda.OrderHandler::HandleRequest"
}
```

For this function to work properly, its execution role must allow the `s3:PutObject` action. Also, ensure that you define the `RECEIPT_BUCKET` environment variable. After a successful invocation, the Amazon S3 bucket should contain a receipt file.

Class library handlers

The main example code on this page illustrates a class library handler. Class library handlers have the following structure:

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace NAMESPACE;

...

public class CLASSNAME {
    public async Task<string> METHODNAME (...) {
    ...
    }
}
```

When you create a Lambda function, you need to provide Lambda with information about your function's handler in the form of a string in the Handler field. This tells Lambda which method in your code to run when your function is invoked. In C#, for class library handlers, the format of the handler string is `ASSEMBLY::TYPE::METHOD`, where:

- `ASSEMBLY` is the name of the .NET assembly file for your application. If you're using the `Amazon.Lambda.Tools` CLI to build your application and you don't set the assembly name using the `AssemblyName` property in the `.csproj` file, then `ASSEMBLY` is simply the name of your `.csproj` file.
- `TYPE` is the full name of the handler type, which is `NAMESPACE.CLASSNAME`.

- `METHOD` is the name of the main handler method in your code, which is `METHODNAME`.

For the main example code on this page, if the assembly is named `ExampleCS`, then the full handler string is `ExampleCS::ExampleLambda.OrderHandler::HandleRequest`.

Executable assembly handlers

You can also define Lambda functions in C# as an executable assembly. Executable assembly handlers utilize C#'s top-level statements feature, in which the compiler generates the `Main()` method and puts your function code within it. When using executable assemblies, the Lambda runtime must be bootstrapped. To do this, use the `LambdaBootstrapBuilder.Create` method in your code. The inputs to this method are the main handler function as well as the Lambda serializer to use. The following shows an example of an executable assembly handler in C#:

```
namespace GetProductHandler;

IDatabaseRepository repo = new DatabaseRepository();

await LambdaBootstrapBuilder.Create<APIGatewayProxyRequest>(Handler, new
 DefaultLambdaJsonSerializer())
    .Build()
    .RunAsync();

async Task<APIGatewayProxyResponse> Handler(APIGatewayProxyRequest apigProxyEvent,
 ILambdaContext context)
{
    var id = apigProxyEvent.PathParameters["id"];
    var databaseRecord = await this.repo.GetById(id);

    return new APIGatewayProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = JsonSerializer.Serialize(databaseRecord)
    };
};
```

In the Handler field for executable assembly handlers, the handler string that tells Lambda how to run your code is the name of the assembly. In this example, that's `GetProductHandler`.

Valid handler signatures for C# functions

In C#, valid Lambda handler signatures take between 0 and 2 arguments. Typically, your handler signature has two arguments, as shown in the main example:

```
public async Task<string> HandleRequest(Order order, ILambdaContext context)
```

When providing two arguments, the first argument must be the event input, and the second argument must be the Lambda context object. Both arguments are optional. For example, the following are also valid Lambda handler signatures in C#:

- `public async Task<string> HandleRequest()`
- `public async Task<string> HandleRequest(Order order)`
- `public async Task<string> HandleRequest(ILambdaContext context)`

Apart from the base syntax of the handler signature, there are some additional restrictions:

- You cannot use the `unsafe` keyword in the handler signature. However, you can use the `unsafe` context inside the handler method and its dependencies. For more information, see unsafe (C# reference) on the Microsoft documentation website.
- The handler may not use the `params` keyword, or use `ArgIterator` as an input or return parameter. These keywords support a variable number of parameters. The maximum number of arguments your handler can accept is two.
- The handler may not be a generic method. In other words, it can't use generic type parameters such as `<T>`.
- Lambda doesn't support async handlers with `async void` in the signature.

Handler naming conventions

Lambda handlers in C# don't have strict naming restrictions. However, you must ensure that you provide the correct handler string to Lambda when you deploy your function. The right handler string depends on if you're deploying a class library handler or an executable assembly handler.

Although you can use any name for your handler, function names in C# are generally in PascalCase. Also, although the file name doesn't need to match the class name or handler name, it's generally

a best practice to use a filename like `OrderHandler.cs` if your class name is `OrderHandler`. For example, you can modify the file name in this example from `Function.cs` to `OrderHandler.cs`.

Serialization in C# Lambda functions

JSON is the most common and standard input format for Lambda functions. In this example, the function expects an input similar to the following:

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

In C#, you can define the shape of the expected input event in a class. In this example, we define the `Order` class to model this input:

```
public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}
```

If your Lambda function uses input or output types other than a `Stream` object, you must add a serialization library to your application. This lets you convert the JSON input into an instance of the class that you defined. There are two methods of serialization for C# functions in Lambda: reflection-based serialization and source-generated serialization.

Reflection-based serialization

AWS provides pre-built libraries that you can quickly add to your application. These libraries implement serialization using reflection. Use one of the following packages to implement reflection-based serialization:

- `Amazon.Lambda.Serialization.SystemTextJson` – In the backend, this package uses `System.Text.Json` to perform serialization tasks.
- `Amazon.Lambda.Serialization.Json` – In the backend, this package uses `Newtonsoft.Json` to perform serialization tasks.

You can also create your own serialization library by implementing the `ILambdaSerializer` interface, which is available as part of the `Amazon.Lambda.Core` library. This interface defines two methods:

- `T Deserialize<T>(Stream requestStream);`

  You implement this method to deserialize the request payload from the `Invoke` API into the object that is passed to your Lambda function handler.
- `T Serialize<T>(T response, Stream responseStream);`

  You implement this method to serialize the result returned from your Lambda function handler into the response payload that the `Invoke` API operation returns.

The main example on this page uses reflection-based serialization. Reflection-based serialization works out of the box with AWS Lambda and requires no additional setup, making it a good choice for simplicity. However, it does require more function memory usage. You may also see higher function latencies due to runtime reflection.

Source-generated serialization

With source-generated serialization, serialization code is generated at compile time. This removes the need for reflection and can improve the performance of your function. To use source-generated serialization in your function, you must do the following:

- Create a new partial class that inherits from `JsonSerializerContext`, adding `JsonSerializable` attributes for all types that require serialization or deserialization.
- Configure the `LambdaSerializer` to use a `SourceGeneratorLambdaJsonSerializer<T>`.
- Update any manual serialization and deserialization in your application code to use the newly created class.

The following example shows how you can modify the main example on this page, which uses reflection-based serialization, to use source-generated serialization instead.

```
using System.Text.Json;
using System.Text.Json.Serialization;

...
```

```
public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}

[JsonSerializable(typeof(Order))]
public partial class OrderJsonContext : JsonSerializerContext {}

public class OrderHandler
{

    ...

    public async Task<string> HandleRequest(string input, ILambdaContext context)
    {

    var order = JsonSerializer.Deserialize(input, OrderJsonContext.Default.Order);

    ...

    }

}
```

Source-generated serialization requires more setup than reflection-based serialization. However, functions using source-generated tend to use less memory and have better performance due to compile-time code generation. To help eliminate function cold starts, consider switching to source-generated serialization.

> **Note**
>
> If you want to use native ahead-of-time compilation (AOT) with Lambda, you must use source-generated serialization.

File-based functions

Introduced in .NET 10, file-based apps enable you to build .NET applications from a single `.cs` file, without a `.csproj` file or directory structure. Lambda supports file-based functions, starting with .NET 10. They offer a streamlined, lightweight way to build Lambda functions in C#.

The fastest way to get started creating a C# file-based Lambda function is to use the `Amazon.Lambda.Templates` package. To install the package, run the following command:

```
dotnet new install Amazon.Lambda.Templates
```

Next, create a C# file-based Lambda example function:

```
dotnet new lambda.FileBased -n MyLambdaFunction
```

File-based functions use executable assembly handlers. You must therefore include the `Amazon.Lambda.RuntimeSupport` NuGet package and use the `LambdaBootstrapBuilder.Create` method to register the .NET handler function for the event type and start the .NET Lambda runtime client.

File-based functions use .NET Native AOT by default, which requires source-generated serialization. You can disable Native AOT by specifying `#:property PublishAot=false` in your source file. For more information on using Native AOT in Lambda, see the section called “Native AOT compilation”.

Accessing and using the Lambda context object

The Lambda context object contains information about the invocation, function, and execution environment. In this example, the context object is of type `Amazon.Lambda.Core.ILambdaContext`, and is the second argument of the main handler function.

```
public async Task<string> HandleRequest(Order order, ILambdaContext context) {
    ...
}
```

The context object is an optional input. For more information about valid accepted handler signatures, see the section called “Valid handler signatures for C# functions”.

The context object is useful for producing function logs to Amazon CloudWatch. You can use the `context.getLogger()` method to get a `LambdaLogger` object for logging. In this example, we can use the logger to log an error message if processing fails for any reason:

```
context.Logger.LogError($"Failed to process order: {ex.Message}");
```

Outside of logging, you can also use the context object for function monitoring. For more information about the context object, see the section called “Context”.

Using the SDK for .NET v3 in your handler

Often, you'll use Lambda functions to interact with or make updates to other AWS resources. The simplest way to interface with these resources is to use the SDK for .NET v3.

> **Note**
>
> The SDK for .NET (v2) is deprecated. We recommend that you use only the SDK for .NET v3.

You can add SDK dependencies to your project using the following `Amazon.Lambda.Tools` command:

```
dotnet add package <package_name>
```

For example, in the main example on this page, we need to use the Amazon S3 API to upload a receipt to S3. We can import the Amazon S3 SDK client with the following command:

```
dotnet add package AWSSDK.S3
```

This command adds the dependency to your project. You should also see a line similar to the following in your project's `.csproj` file:

```
<PackageReference Include="AWSSDK.S3" Version="3.7.2.18" />
```

Then, import the dependencies directly in your C# code:

```
using Amazon.S3;
```

```
using Amazon.S3.Model;
```

The example code then initializes an Amazon S3 client (using the default credential provider chain) as follows:

```
private static readonly AmazonS3Client s3Client = new();
```

In this example, we initialized our Amazon S3 client outside of the main handler function to avoid having to initialize it every time we invoke our function. After you initialize your SDK client, you can then use it to interact with other AWS services. The example code calls the Amazon S3 `PutObject` API as follows:

```
var putRequest = new PutObjectRequest
{
    BucketName = bucketName,
    Key = key,
    ContentBody = receiptContent,
    ContentType = "text/plain"
};

await s3Client.PutObjectAsync(putRequest);
```

Accessing environment variables

In your handler code, you can reference any environment variables by using the `System.Environment.GetEnvironmentVariable` method. In this example, we reference the defined `RECEIPT_BUCKET` environment variable using the following lines of code:

```
string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
if (string.IsNullOrWhiteSpace(bucketName))
{
    throw new ArgumentException("RECEIPT_BUCKET environment variable is not set");
}
```

Using global state

Lambda runs your static code and the class constructor during the initialization phase before invoking your function for the first time. Resources created during initialization stay in memory between invocations, so you can avoid having to create them every time you invoke your function.

In the example code, the S3 client initialization code is outside the main handler method. The runtime initializes the client before the function handles its first event, which can lead to longer processing times. Subsequent events are much faster because Lambda doesn't need to initialize the client again.

Simplify function code with the Lambda Annotations framework

Lambda Annotations is a framework for .NET 8 which simplifies writing Lambda functions using C#. The Annotations framework uses source generators to generate code that translates from the Lambda programming model to the simplified code. With the Annotations framework, you can replace much of the code in a Lambda function written using the regular programming model. Code written using the framework uses simpler expressions that allow you to focus on your business logic. See Amazon.Lambda.Annotations in the nuget documentation for examples.

For an example of a full application utilizing Lambda Annotations, see the PhotoAssetManager example in the `awsdocs/aws-doc-sdk-examples` GitHub repository. The main `Function.cs` file in the `PamApiAnnotations` directory uses Lambda Annotations. For comparison, the `PamApi` directory has equivalent files written using the regular Lambda programming model.

Dependency injection with Lambda Annotations framework

You can also use the Lambda Annotations framework to add dependency injection to your Lambda functions using syntax you are familiar with. When you add a `[LambdaStartup]` attribute to a `Startup.cs` file, the Lambda Annotations framework will generate the required code at compile time.

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Your Lambda function can inject services using either constructor injection or by injecting into individual methods using the `[FromServices]` attribute.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(IDatabaseRepository repo)
    {
        this._repo = repo;
    }

    [LambdaFunction]
    [HttpApi(LambdaHttpMethod.Get, "/product/{id}")]
    public async Task<Product> FunctionHandler([FromServices] IDatabaseRepository
 repository, string id)
    {
        return await this._repo.GetById(id);
    }
}
```

Code best practices for C# Lambda functions

Adhere to the guidelines in the following list to use best coding practices when building your Lambda functions:

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function.
- **Control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains a number of libraries. To enable the latest set of features and security updates, Lambda will periodically update these libraries. These updates may introduce subtle changes to the behavior of your Lambda function. To have full control of the dependencies your function uses, package all of your dependencies with your deployment package.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked

ahead of invocation. For functions authored in .NET, avoid uploading the entire AWS SDK library as part of your deployment package. Instead, selectively depend on the modules which pick up components of the SDK you need (e.g. DynamoDB, Amazon S3 SDK modules and Lambda core libraries).

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of function invocations and escalated costs. If you see an unintended volume of invocations, set the function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

Build and deploy C# Lambda functions with .zip file archives

A .NET deployment package (.zip file archive) contains your function's compiled assembly along with all of its assembly dependencies. The package also contains a *proj*`.deps.json` file. This signals to the .NET runtime all of your function's dependencies and a *proj*`.runtimeconfig.json` file, which is used to configure the runtime.

To deploy individual Lambda functions, you can use the `Amazon.Lambda.Tools` .NET Lambda Global CLI. Using the `dotnet lambda deploy-function` command automatically creates a .zip deployment package and deploys it to Lambda. However, we recommend that you use frameworks like the AWS Serverless Application Model (AWS SAM) or the AWS Cloud Development Kit (AWS CDK) to deploy your .NET applications to AWS.

Serverless applications usually comprise a combination of Lambda functions and other managed AWS services working together to perform a particular business task. AWS SAM and AWS CDK simplify building and deploying Lambda functions with other AWS services at scale. The AWS SAM template specification provides a simple and clean syntax to describe Lambda functions, APIs, permissions, configurations, and other AWS resources that make up your serverless application. With the AWS CDK you define cloud infrastructure as code to help you build reliable, scalable, cost-effective applications in the cloud using modern programming languages and frameworks like .NET. Both the AWS CDK and the AWS SAM use the .NET Lambda Global CLI to package your functions.

While it's possible to use Lambda layers with functions in C# by using the .NET Core CLI, we recommend against it. Functions in C# that use layers manually load the shared assemblies into memory during the Init phase, which can increase cold start times. Instead, include all shared code at compile time to avoid the performance impact of loading assemblies at runtime.

You can find instructions for building and deploying .NET Lambda functions using the AWS SAM, the AWS CDK, and the .NET Lambda Global CLI in the following sections.

Topics

- Using the .NET Lambda Global CLI
- Deploy C# Lambda functions using AWS SAM
- Deploy C# Lambda functions using AWS CDK
- Deploy ASP.NET applications

Using the .NET Lambda Global CLI

The .NET CLI and the .NET Lambda Global Tools extension (`Amazon.Lambda.Tools`) offer a cross-platform way to create .NET-based Lambda applications, package them, and deploy them to Lambda. In this section, you learn how to create new Lambda .NET projects using the .NET CLI and Amazon Lambda templates, and to package and deploy them using `Amazon.Lambda.Tools`

Topics

- Prerequisites
- Creating .NET projects using the .NET CLI
- Deploying .NET projects using the .NET CLI
- Using Lambda layers with the .NET CLI

Prerequisites

.NET 8 SDK

If you haven't already done so, install the .NET 8 SDK and Runtime.

AWS Amazon.Lambda.Templates .NET project templates

To generate your Lambda function code, use the Amazon.Lambda.Templates NuGet package. To install this template package, run the following command:

```
dotnet new install Amazon.Lambda.Templates
```

AWS Amazon.Lambda.Tools .NET Global CLI tools

To create your Lambda functions, you use the Amazon.Lambda.Tools .NET Global Tools extension. To install Amazon.Lambda.Tools, run the following command:

```
dotnet tool install -g Amazon.Lambda.Tools
```

For more information about the Amazon.Lambda.Tools .NET CLI extension, see the AWS Extensions for .NET CLI repository on GitHub.

Creating .NET projects using the .NET CLI

In the .NET CLI, you use the `dotnet new` command to create .NET projects from the command line. Lambda offers additional templates using the `Amazon.Lambda.Templates` NuGet package.

After installing this package, run the following command to see a list of the available templates.

```
dotnet new list
```

To examine details about a template, use the `help` option. For example, to see details about the `lambda.EmptyFunction` template, run the following command.

```
dotnet new lambda.EmptyFunction --help
```

To create a basic template for a .NET Lambda function, use the `lambda.EmptyFunction` template. This creates a simple function that takes a string as input and converts it to upper case using the `ToUpper` method. This template supports the following options:

- `--name` – The name of the function.
- `--region` – The AWS Region to create the function in.
- `--profile` – The name of a profile in your AWS SDK for .NET credentials file. To learn more about credential profiles in .NET, see Configure AWS credentials in the *AWS SDK for .NET Developer Guide*.

In this example, we create a new empty function named `myDotnetFunction` using the default profile and AWS Region settings:

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

This command creates the following files and directories in your project directory.

```
### myDotnetFunction
    ### src
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
```

```
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

Under the `src/myDotnetFunction` directory, examine the following files:

- **aws-lambda-tools-defaults.json**: This is where you specify the command line options when deploying your Lambda function. For example:

```
  "profile" : "default",
  "region" : "us-east-2",
  "configuration" : "Release",
  "function-architecture": "x86_64",
  "function-runtime":"dotnet8",
  "function-memory-size" : 256,
  "function-timeout" : 30,
  "function-handler" : "myDotnetFunction::myDotnetFunction.Function::FunctionHandler"
```

- **Function.cs**: Your Lambda handler function code. It's a C# template that includes the default `Amazon.Lambda.Core` library and a default `LambdaSerializer` attribute. For more information on serialization requirements and options, see Serialization in C# Lambda functions. It also includes a sample function that you can edit to apply your Lambda function code.

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into
 a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializ

namespace myDotnetFunction;

public class Function
{

    /// <summary>
    /// A simple function that takes a string and does a ToUpper
    /// </summary#
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
```

```
    public string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}
```

- **myDotnetFunction.csproj**: An MSBuild file that lists the files and assemblies that comprise your application.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
    <!-- This property makes the build directory similar to a publish directory and
 helps the AWS .NET Lambda Mock Test Tool find project dependencies. -->
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
    <!-- Generate ready to run images during publishing to improve cold start time.
 -->
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
 Version="2.4.0" />
  </ItemGroup>
</Project>
```

- **Readme**: Use this file to document your Lambda function.

Under the `myfunction/test` directory, examine the following files:

- **myDotnetFunction.Tests.csproj**: As noted previously, this is an MSBuild file that lists the files and assemblies that comprise your test project. Note also that it includes the `Amazon.Lambda.Core` library, so you can seamlessly integrate any Lambda templates required to test your function.

```
<Project Sdk="Microsoft.NET.Sdk">
   ...
```

```
    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
   ...
```

- **FunctionTest.cs**: The same C# code template file that it is included in the `src` directory. Edit this file to mirror your function's production code and test it before uploading your Lambda function to a production environment.

```
using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;

using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
        [Fact]
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

Deploying .NET projects using the .NET CLI

To build your deployment package and deploy it to Lambda, you use the `Amazon.Lambda.Tools` CLI tools. To deploy your function from the files you created in the previous steps, first navigate into the folder containing your function's `.csproj` file.

```
cd myDotnetFunction/src/myDotnetFunction
```

To deploy your code to Lambda as a .zip deployment package, run the following command. Choose your own function name.

```
dotnet lambda deploy-function myDotnetFunction
```

During the deployment, the wizard asks you to select a the section called "Execution role (permissions for functions to access other resources)". For this example, select the `lambda_basic_role`.

After you have deployed your function, you can test it in the cloud using the `dotnet lambda invoke-function` command. For the example code in the `lambda.EmptyFunction` template, you can test your function by passing in a string using the `--payload` option.

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

If your function has been successfully deployed, you should see output similar to the following.

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
Amazon Lambda Tools for .NET Core applications (5.8.0)
Project Home: https://github.com/aws/aws-extensions-for-dotnet-cli, https://github.com/
aws/aws-lambda-dotnet

Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

Using Lambda layers with the .NET CLI

> **Note**
>
> While it's possible to use layers with functions in .NET, we recommend against it. Functions in .NET that use layers manually load the shared assemblies into memory during the `Init` phase, which can increase cold start times. Instead, include all shared code at compile time to take advantage of the built-in optimizations of the .NET compiler.

The .NET CLI supports commands to help you publish layers and deploy C# functions that consume layers. To publish a layer to a specified Amazon S3 bucket, run the following command in the same directory as your `.csproj` file:

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

Then, when you deploy your function using the .NET CLI, specify the layer ARN the consume in the following command:

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

For a complete example of a Hello World function, see the blank-csharp-with-layer sample.

Deploy C# Lambda functions using AWS SAM

The AWS Serverless Application Model (AWS SAM) is a toolkit that helps streamline the process of building and running serverless applications on AWS. You define the resources for your application in a YAML or JSON template and use the AWS SAM command line interface (AWS SAM CLI) to build, package, and deploy your applications. When you build a Lambda function from an AWS SAM template, AWS SAM automatically creates a .zip deployment package or container image with your function code and any dependencies you specify. AWS SAM then deploys your function using an CloudFormation stack. To learn more about using AWS SAM to build and deploy Lambda functions, see Getting started with AWS SAM in the *AWS Serverless Application Model Developer Guide*.

The following steps show you how to download, build, and deploy a sample .NET Hello World application using AWS SAM. This sample application uses a Lambda function and an Amazon API Gateway endpoint to implement a basic API backend. When you send an HTTP GET request to your API Gateway endpoint, API Gateway invokes your Lambda function. The function returns a "hello world" message, along with the IP address of the Lambda function instance that processes your request.

When you build and deploy your application using AWS SAM, behind the scenes the AWS SAM CLI uses the `dotnet lambda package` command to package the individual Lambda function code bundles.

Prerequisites

.NET 8 SDK

Install the .NET 8 SDK and Runtime.

AWS SAM CLI version 1.39 or later

To learn how to install the latest version of the AWS SAM CLI, see Installing the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello world .NET template using the following command.

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

This command creates the following files and directories in your project directory.

```
### sam-app
    ### README.md
    ### events
    #   ### event.json
    ### omnisharp.json
    ### samconfig.toml
    ### src
    #   ### HelloWorld
    #       ### Function.cs
    #       ### HelloWorld.csproj
    #       ### aws-lambda-tools-defaults.json
    ### template.yaml
    ### test
        ### HelloWorld.Test
            ### FunctionTest.cs
            ### HelloWorld.Tests.csproj
```

2. Navigate into the directory containing the `template.yaml file`. This file is a tempate that defines the AWS resources for your application, including your Lambda function and an API Gateway API.

```
cd sam-app
```

3. To build the source of your application, run the following command.

```
sam build
```

4. To deploy your application to AWS, run the following command.

```
sam deploy --guided
```

This command packages and deploys your application with the following series of prompts. To accept the default options, press Enter.

> **Note**
>
> For **HelloWorldFunction may not have authorization defined, is this okay?**, be sure to enter `y`.

- **Stack Name**: The name of the stack to deploy to CloudFormation. This name must be unique to your AWS account and AWS Region.
- **AWS Region**: The AWS Region you want to deploy your app to.
- **Confirm changes before deploy**: Select yes to manually review any change sets before AWS SAM deploys application changes. If you select no, the AWS SAM CLI automatically deploys application changes.
- **Allow SAM CLI IAM role creation**: Many AWS SAM templates, including the Hello world one in this example, create AWS Identity and Access Management (IAM) roles to give your Lambda functions permission to access other AWS services. Select Yes to provide permission to deploy a CloudFormation stack that creates or modifies IAM roles.
- **Disable rollback**: By default, if AWS SAM encounters an error during creation or deployment of your stack, it rolls the stack back to the previous version. Select No to accept this default.
- **HelloWorldFunction may not have authorization defined, is this okay**: Enter `y`.
- **Save arguments to samconfig.toml**: Select yes to save your configuration choices. In the future, you can re-run `sam deploy` without parameters to deploy changes to your application.

5. When the deployment of your application is complete, the CLI returns the Amazon Resource Name (ARN) of the Hello World Lambda function and the IAM role created for it. It also

displays the endpoint of your API Gateway API. To test your application, open the endpoint in a browser. You should see a response similar to the following.

```
{"message":"hello world","location":"34.244.135.203"}
```

6. To delete your resources, run the following command. Note that the API endpoint you created is a public endpoint accessible over the internet. We recommend that you delete this endpoint after testing.

```
sam delete
```

Next steps

To learn more about using AWS SAM to build and deploy Lambda functions using .NET, see the following resources:

- The *AWS Serverless Application Model (AWS SAM) Developer Guide*
- Building Serverless .NET Applications with AWS Lambda and the SAM CLI

Deploy C# Lambda functions using AWS CDK

The AWS Cloud Development Kit (AWS CDK) is an open-source software development framework for defining cloud infrastructure as code with modern programming languages and frameworks like .NET. AWS CDK projects are executed to generate CloudFormation templates which are then used to deploy your code.

To build and deploy an example Hello world .NET application using the AWS CDK, follow the instructions in the following sections. The sample application implements a basic API backend consisting of an API Gateway endpoint and a Lambda function. API Gateway invokes the Lambda function when you send an HTTP GET request to the endpoint. The function returns a Hello world message, along with the IP address of the Lambda instance that processes your request.

Prerequisites

.NET 8 SDK

Install the .NET 8 SDK and Runtime.

AWS CDK version 2

To learn how to install the latest version of the AWS CDK see Getting started with the AWS CDK in the *AWS Cloud Development Kit (AWS CDK) v2 Developer Guide*.

Deploy a sample AWS CDK application

1. Create a project directory for the sample application and navigate into it.

```
mkdir hello-world
cd hello-world
```

2. Initialize a new AWS CDK application by running the following command.

```
cdk init app --language csharp
```

The command creates the following files and directories in your project directory

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. Open the `src` directory and create a new Lambda function using the .NET CLI. This is the function you will deploy using the AWS CDK. In this example, you create a Hello world function named `HelloWorldLambda`using the `lambda.EmptyFunction` template.

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

After this step, your directory structure inside your project directory should look like the following.

```
### README.md
### cdk.json
```

```
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
    ### HelloWorldLambda
        ### src
        #   ### HelloWorldLambda
        #       ### Function.cs
        #       ### HelloWorldLambda.csproj
        #       ### Readme.md
        #       ### aws-lambda-tools-defaults.json
        ### test
            ### HelloWorldLambda.Tests
                ### FunctionTest.cs
                ### HelloWorldLambda.Tests.csproj
```

4. Open the `HelloWorldStack.cs` file from the `src/HelloWorld` directory. Replace the contents of the file with the following code.

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.Logs;
using Constructs;

namespace CdkTest
{
    public class HelloWorldStack : Stack
    {
        internal HelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var buildOption = new BundlingOptions()
            {
                Image = Runtime.DOTNET_8.BundlingImage,
                User = "root",
                OutputType = BundlingOutput.ARCHIVED,
                Command = new string[]{
            "/bin/sh",
                "-c",
                " dotnet tool install -g Amazon.Lambda.Tools"+
```

```
                " && dotnet build"+
                " && dotnet lambda package --output-package /asset-output/
function.zip"
                }
            };

             var helloWorldLambdaFunction = new Function(this,
 "HelloWorldFunction", new FunctionProps
            {
                Runtime = Runtime.DOTNET_8,
                MemorySize = 1024,
                LogRetention = RetentionDays.ONE_DAY,
                Handler =
 "HelloWorldLambda::HelloWorldLambda.Function::FunctionHandler",
                Code = Code.FromAsset("./src/HelloWorldLambda/src/
HelloWorldLambda", new Amazon.CDK.AWS.S3.Assets.AssetOptions
                {
                    Bundling = buildOption
                }),
            });
        }
    }
}
```

This is the code to compile and bundle the application code, as well as the definition of the Lambda function itself. the `BundlingOptions` object allows a zip file to be created, along with a set of commands that are used to generate the contents of the zip file. In this instance, the `dotnet lambda package` command is used to compile and generate the zip file.

5. To deploy your application, run the following command.

```
cdk deploy
```

6. Invoke your deployed Lambda function using the .NET Lambda CLI.

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. After you've finished testing, you can delete the resources you created, unless you want to retain them. Run the following command to delete your resources.

```
cdk destroy
```

Next steps

To learn more about using AWS CDK to build and deploy Lambda functions using .NET, see the following resources:

- Working with the AWS CDK in C#
- Build, package, and publish .NET C# Lambda functions with the AWS CDK

Deploy ASP.NET applications

As well as hosting event-driven functions, you can also use .NET with Lambda to host lightweight ASP.NET applications. You can build and deploy ASP.NET applications using the `Amazon.Lambda.AspNetCoreServer` NuGet package. In this section, you learn how to deploy an ASP.NET web API to Lambda using the .NET Lambda CLI tooling.

Topics

- Prerequisites
- Deploying an ASP.NET Web API to Lambda
- Deploying ASP.NET minimal APIs to Lambda

Prerequisites

.NET 8 SDK

Install the .NET 8 SDK and ASP.NET Core Runtime.

Amazon.Lambda.Tools

To create your Lambda functions, you use the Amazon.Lambda.Tools .NET Global Tools extension. To install Amazon.Lambda.Tools, run the following command:

```
dotnet tool install -g Amazon.Lambda.Tools
```

For more information about the Amazon.Lambda.Tools .NET CLI extension, see the AWS Extensions for .NET CLI repository on GitHub.

Amazon.Lambda.Templates

To generate your Lambda function code, use the Amazon.Lambda.Templates NuGet package. To install this template package, run the following command:

```
dotnet new --install Amazon.Lambda.Templates
```

Deploying an ASP.NET Web API to Lambda

To deploy a web API using ASP.NET, you can use the .NET Lambda templates to create a new web API project. Use the following command to initialize a new ASP.NET web API project. In the example command, we name the project `AspNetOnLambda`.

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

This command creates the following files and directories in your project directory.

```
.
### AspNetOnLambda
    ### src
    #   ### AspNetOnLambda
    #       ### AspNetOnLambda.csproj
    #       ### Controllers
    #       #   ### ValuesController.cs
    #       ### LambdaEntryPoint.cs
    #       ### LocalEntryPoint.cs
    #       ### Readme.md
    #       ### Startup.cs
    #       ### appsettings.Development.json
    #       ### appsettings.json
    #       ### aws-lambda-tools-defaults.json
    #       ### serverless.template
    ### test
        ### AspNetOnLambda.Tests
            ### AspNetOnLambda.Tests.csproj
            ### SampleRequests
            #   ### ValuesController-Get.json
            ### ValuesControllerTests.cs
            ### appsettings.json
```

When Lambda invokes your function, the entry point it uses is the `LambdaEntryPoint.cs` file. The file created by the .NET Lambda template contains the following code.

```
namespace AspNetOnLambda;
```

```
public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
    {
    }
}
```

The entry point used by Lambda must inherit from one of the three base classes in the `Amazon.Lambda.AspNetCoreServer` package. These three base classes are:

- `APIGatewayProxyFunction`
- `APIGatewayHttpApiV2ProxyFunction`
- `ApplicationLoadBalancerFunction`

The default class used when you create your `LambdaEntryPoint.cs` file using the provided .NET Lambda template is `APIGatewayProxyFunction`. The base class you use in your function depends on which API layer sits in front of your Lambda function.

Each of the three base classes contains a public method named `FunctionHandlerAsync`. The name of this method will form part of the handler string Lambda uses to invoke your function. The `FunctionHandlerAsync` method transforms the inbound event payload into the correct ASP.NET format and the ASP.NET response back to a Lambda response payload. For the example `AspNetOnLambda` project shown, the handler string would be as follows.

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

To deploy the API to Lambda, run the following commands to navigate into the directory containing your source code file and deploy your function using CloudFormation.

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> When you deploy an API using the **`dotnet lambda deploy-serverless`** command, CloudFormation gives your Lambda function a name based on the stack name you specify during the deployment. To give your Lambda function a custom name, edit the `serverless.template` file to add a `FunctionName` property to the `AWS::Serverless::Function` resource. See Name type in the *CloudFormation User Guide* to learn more.

Deploying ASP.NET minimal APIs to Lambda

To deploy an ASP.NET minimal API to Lambda, you can use the .NET Lambda templates to create a new minimal API project. Use the following command to initialize a new minimal API project. In this example, we name the project `MinimalApiOnLambda`.

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

The command creates the following files and directories in your project directory.

```
### MinimalApiOnLambda
    ### src
        ### MinimalApiOnLambda
            ### Controllers
            #   ### CalculatorController.cs
            ### MinimalApiOnLambda.csproj
            ### Program.cs
            ### Readme.md
            ### appsettings.Development.json
            ### appsettings.json
            ### aws-lambda-tools-defaults.json
            ### serverless.template
```

The `Program.cs` file contains the following code.

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();
```

```
// Add AWS Lambda support. When application is run in Lambda Kestrel is swapped out as
 the web server with Amazon.Lambda.AspNetCoreServer. This
// package will act as the webserver translating request and responses between the
 Lambda event source and ASP.NET Core.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.RestApi);

var app = builder.Build();


app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.MapGet("/", () => "Welcome to running ASP.NET Core Minimal API on AWS Lambda");

app.Run();
```

To configure your minimal API to run on Lambda, you may need to edit this code so that requests and responses between Lambda and ASP.NET Core are properly translated. By default, the function is configured for a REST API event source. For an HTTP API or application load balancer, replace `(LambdaEventSource.RestApi)` with one of the following options:

- `(LambdaEventSource.HttpAPi)`
- `(LambdaEventSource.ApplicationLoadBalancer)`

To deploy your minimal API to Lambda, run the following commands to navigate into the directory containing your source code file and deploy your function using CloudFormation.

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

Working with layers for .NET Lambda functions

We don't recommend using layers to manage dependencies for Lambda functions written in .NET. This is because .NET is a compiled language, and your functions still have to manually load any shared assemblies into memory during the Init phase, which can increase cold start times. Using layers not only complicates the deployment process, but also prevents you from taking advantage of built-in compiler optimizations.

To use external dependencies with your .NET handlers, include them directly in your deployment package at compile time. By doing so, you simplify the deployment process and also take advantage of built-in .NET compiler optimizations. For an example of how to import and use dependencies like NuGet packages in your function, see the section called “Handler”.

Deploy .NET Lambda functions with container images

There are three ways to build a container image for a .NET Lambda function:

- Using an AWS base image for .NET

  The AWS base images are preloaded with a language runtime, a runtime interface client to manage the interaction between Lambda and your function code, and a runtime interface emulator for local testing.
- Using an AWS OS-only base image

  AWS OS-only base images contain an Amazon Linux distribution and the runtime interface emulator. These images are commonly used to create container images for compiled languages, such as Go and Rust, and for a language or language version that Lambda doesn't provide a base image for, such as Node.js 19. You can also use OS-only base images to implement a custom runtime. To make the image compatible with Lambda, you must include the the runtime interface client for .NET in the image.
- Using a non-AWS base image

  You can use an alternative base image from another container registry, such as Alpine Linux or Debian. You can also use a custom image created by your organization. To make the image compatible with Lambda, you must include the the runtime interface client for .NET in the image.

> **Tip**
>
> To reduce the time it takes for Lambda container functions to become active, see Use multi-stage builds in the Docker documentation. To build efficient container images, follow the Best practices for writing Dockerfiles.

This page explains how to build, test, and deploy container images for Lambda.

Topics

- AWS base images for .NET
- Using an AWS base image for .NET
- Using an alternative base image with the runtime interface client

AWS base images for .NET

AWS provides the following base images for .NET:

Tags	Runtime	Operating system	Dockerfile	Deprecation
10	.NET 10	Amazon Linux 2023	Dockerfile for .NET 10 on GitHub	Nov 14, 2028
9	.NET 9	Amazon Linux 2023	Dockerfile for .NET 9 on GitHub	Nov 10, 2026
8	.NET 8	Amazon Linux 2023	Dockerfile for .NET 8 on GitHub	Nov 10, 2026

Amazon ECR repository: gallery.ecr.aws/lambda/dotnet

Using an AWS base image for .NET

Prerequisites

To complete the steps in this section, you must have the following:

- .NET SDK – The following steps use the .NET 8 base image. Make sure that your .NET version matches the version of the base image that you specify in your Dockerfile.
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.

Creating and deploying an image using a base image

In the following steps, you use Amazon.Lambda.Templates and Amazon.Lambda.Tools to create a .NET project. Then, you build a Docker image, upload the image to Amazon ECR, and deploy it to a Lambda function.

1. Install the Amazon.Lambda.Templates NuGet package.

```
dotnet new install Amazon.Lambda.Templates
```

2. Create a .NET project using the `lambda.image.EmptyFunction` template.

```
dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
```

The project files are stored in the *MyFunction*/src/*MyFunction* directory:

- **aws-lambda-tools-defaults.json**: Specifies the command line options for deploying your Lambda function.
- **Function.cs**: Your Lambda handler function code. This is a C# template that includes the default `Amazon.Lambda.Core` library and a default `LambdaSerializer` attribute. For more information about serialization requirements and options, see Serialization in C# Lambda functions. You can use the provided code for testing, or replace it with your own.
- **MyFunction.csproj**: A .NET project file, which lists the files and assemblies that comprise your application.
- **Dockerfile**: You can use the provided Dockerfile for testing, or replace it with your own. If you use your own, make sure to:
  - Set the `FROM` property to the URI of the base image. The base image and the `TargetFramework` in the `MyFunction.csproj` file must both use the same .NET version. For example, to use .NET 9:
    - Dockerfile: `FROM` *public.ecr.aws/lambda/dotnet:9*
    - MyFunction.csproj: `<TargetFramework>`*net9.0*`</TargetFramework>`
  - Set the `CMD` argument to the Lambda function handler. This should match the `image-command` in `aws-lambda-tools-defaults.json`.

3. Install the Amazon.Lambda.Tools .NET Global Tool.

```
dotnet tool install -g Amazon.Lambda.Tools
```

If Amazon.Lambda.Tools is already installed, make sure that you have the latest version.

```
dotnet tool update -g Amazon.Lambda.Tools
```

4. Change the directory to *MyFunction*/src/*MyFunction*, if you're not there already.

```
cd src/MyFunction
```

5. Use Amazon.Lambda.Tools to build the Docker image, push it to a new Amazon ECR repository, and deploy the Lambda function.

   For `--function-role`, specify the role name—not the Amazon Resource Name (ARN)—of the execution role for the function. For example, `lambda-role`.

   ```
   dotnet lambda deploy-function MyFunction --function-role lambda-role
   ```

   For more information about the Amazon.Lambda.Tools .NET Global Tool, see the AWS Extensions for .NET CLI repository on GitHub.

6. Invoke the function.

   ```
   dotnet lambda invoke-function MyFunction --payload "Testing the function"
   ```

   If everything is successful, you see a response similar to the following:

   ```
   Payload:
   {"Lower":"testing the function","Upper":"TESTING THE FUNCTION"}

   Log Tail:
   INIT_REPORT Init Duration: 9999.81 ms   Phase: init     Status: timeout
   START RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed Version: $LATEST
   END RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed
   REPORT RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed  Duration: 3173.06 ms
    Billed Duration: 3174 ms        Memory Size: 512 MB     Max Memory Used: 24 MB
   ```

7. Delete the Lambda function.

   ```
   dotnet lambda delete-function MyFunction
   ```

Using an alternative base image with the runtime interface client

If you use an OS-only base image or an alternative base image, you must include the runtime interface client in your image. The runtime interface client extends the Runtime API, which manages the interaction between Lambda and your function code.

The following example demonstrates how to build a container image for .NET using a non-AWS base image, and how to add the Amazon.Lambda.RuntimeSupport package, which is the Lambda runtime interface client for .NET. The example Dockerfile uses the Microsoft .NET 8 base image.

Prerequisites

To complete the steps in this section, you must have the following:

- .NET SDK – The following steps use a .NET 9 base image. Make sure that your .NET version matches the version of the base image that you specify in your Dockerfile.
- Docker (minimum version 25.0.0)
- The Docker buildx plugin.

Creating and deploying an image using an alternative base image

1. Install the Amazon.Lambda.Templates NuGet package.

   ```
   dotnet new install Amazon.Lambda.Templates
   ```

2. Create a .NET project using the `lambda.CustomRuntimeFunction` template. This template includes the Amazon.Lambda.RuntimeSupport package.

   ```
   dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
   ```

3. Navigate to the *`MyFunction`*`/src/`*`MyFunction`* directory. This is where the project files are stored. Examine the following files:

   - **aws-lambda-tools-defaults.json** – This file is where you specify the command line options when deploying your Lambda function.
   - **Function.cs** – The code contains a class with a `Main` method that initializes the `Amazon.Lambda.RuntimeSupport` library as the bootstrap. The `Main` method is the entry point for the function's process. The `Main` method wraps the function handler in a wrapper that the bootstrap can work with. For more information, see Using Amazon.Lambda.RuntimeSupport as a class library in the GitHub repository.
   - **MyFunction.csproj** – A .NET project file, which lists the files and assemblies that comprise your application.
   - **Readme.md** – This file contains more information about the sample Lambda function.

4. Open the `aws-lambda-tools-defaults.json` file and Add the following lines:

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

- **package-type**: Defines the deployment package as a container image.
- **docker-host-build-output-dir**: Sets the output directory for the build process.

Example aws-lambda-tools-defaults.json

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual
 Studio and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "",
  "region": "us-east-1",
  "configuration": "Release",
  "function-runtime": "provided.al2023",
  "function-memory-size": 256,
  "function-timeout": 30,
  "function-handler": "bootstrap",
  "msbuild-parameters": "--self-contained true",
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
}
```

5. Create a Dockerfile in the *MyFunction*`/src/`*MyFunction* directory. The following example Dockerfile uses a Microsoft .NET base image instead of an AWS base image.

- Set the `FROM` property to the base image identifier. The base image and the `TargetFramework` in the `MyFunction.csproj` file must both use the same .NET version.
- Use the `COPY` command to copy the function into the `/var/task` directory.

- Set the ENTRYPOINT to the module that you want the Docker container to run when it starts. In this case, the module is the bootstrap, which initializes the `Amazon.Lambda.RuntimeSupport` library.

Note that the example Dockerfile does not include a USER instruction. When you deploy a container image to Lambda, Lambda automatically defines a default Linux user with least-privileged permissions. This is different from standard Docker behavior which defaults to the `root` user when no USER instruction is provided.

Example Dockerfile

```
# You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
FROM mcr.microsoft.com/dotnet/runtime:9.0

# Set the image's internal work directory
WORKDIR /var/task

# Copy function code to Lambda-defined environment variable
COPY "bin/Release/net9.0/linux-x64"  .

# Set the entrypoint to the bootstrap
ENTRYPOINT ["/usr/bin/dotnet", "exec", "/var/task/bootstrap.dll"]
```

6. Install the Amazon.Lambda.Tools .NET Global Tools extension.

```
dotnet tool install -g Amazon.Lambda.Tools
```

If Amazon.Lambda.Tools is already installed, make sure that you have the latest version.

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. Use Amazon.Lambda.Tools to build the Docker image, push it to a new Amazon ECR repository, and deploy the Lambda function.

For `--function-role`, specify the role name—not the Amazon Resource Name (ARN)—of the execution role for the function. For example, `lambda-role`.

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

For more information about the Amazon.Lambda.Tools .NET CLI extension, see the AWS Extensions for .NET CLI repository on GitHub.

8. Invoke the function.

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

If everything is successful, you see the following:

```
Payload:
"TESTING THE FUNCTION"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

9. Delete the Lambda function.

```
dotnet lambda delete-function MyFunction
```

Compile .NET Lambda function code to a native runtime format

.NET 8 supports native ahead-of-time (AOT) compilation. With native AOT, you can compile your Lambda function code to a native runtime format, which removes the need to compile .NET code at runtime. Native AOT compilation can reduce the cold start time for Lambda functions that you write in .NET. For more information, see Introducing the .NET 8 runtime for AWS Lambda on the AWS Compute Blog.

Sections

- Lambda runtime
- Prerequisites
- Getting started
- Serialization
- Trimming
- Troubleshooting

Lambda runtime

To deploy a Lambda function build with native AOT compilation, use the managed .NET 8 Lambda runtime. This runtime supports the use of both x86_64 and arm64 architectures.

When you deploy a .NET Lambda function without using AOT, your application is first compiled into Intermediate Language (IL) code. At runtime, the just-in-time (JIT) compiler in the Lambda runtime takes the IL code and compiles it into machine code as needed. With a Lambda function that is compiled ahead of time with native AOT, you compile your code into machine code when you deploy your function, so you're not dependent on the .NET runtime or SDK in the Lambda runtime to compile your code before it runs.

One limitation of AOT is that your application code must be compiled in an environment with the same Amazon Linux 2023 (AL2023) operating system that the .NET 8 runtime uses. The .NET Lambda CLI provides functionality to compile your application in a Docker container using an AL2023 image.

To avoid potential issues with cross-architecture compatibility, we strongly recommend that you compile your code in an environment with the same processor architecture that you configure for your function. To learn more about the limitations of cross-architecture compilation, see Cross-compilation in the Microsoft .NET documentation.

Prerequisites

Docker

To use native AOT, your function code must be compiled in an environment with the same AL2023 operating system as the .NET 8 runtime. The .NET CLI commands in the following sections use Docker to develop and build Lambda functions in an AL2023 environment.

.NET 8 SDK

Native AOT compilation is a feature of .NET 8. You must install the .NET 8 SDK on your build machine, not only the runtime.

Amazon.Lambda.Tools

To create your Lambda functions, you use the Amazon.Lambda.Tools .NET Global Tools extension. To install Amazon.Lambda.Tools, run the following command:

```
dotnet tool install -g Amazon.Lambda.Tools
```

For more information about the Amazon.Lambda.Tools .NET CLI extension, see the AWS Extensions for .NET CLI repository on GitHub.

Amazon.Lambda.Templates

To generate your Lambda function code, use the Amazon.Lambda.Templates NuGet package. To install this template package, run the following command:

```
dotnet new install Amazon.Lambda.Templates
```

Getting started

Both the .NET Global CLI and the AWS Serverless Application Model (AWS SAM) provide getting started templates for building applications using native AOT. To build your first native AOT Lambda function, carry out the steps in the following instructions.

To initialize and deploy a native AOT compiled Lambda function

1. Initialize a new project using the native AOT template and then navigate into the directory containing the created `.cs` and `.csproj` files. In this example, we name our function `NativeAotSample`.

```
dotnet new lambda.NativeAOT -n NativeAotSample
cd ./NativeAotSample/src/NativeAotSample
```

The `Function.cs` file created by the native AOT template contains the following function code.

```
using Amazon.Lambda.Core;
using Amazon.Lambda.RuntimeSupport;
using Amazon.Lambda.Serialization.SystemTextJson;
using System.Text.Json.Serialization;

namespace NativeAotSample;

public class Function
{
    /// <summary>
    /// The main entry point for the Lambda function. The main function is called
 once during the Lambda init phase. It
    /// initializes the .NET Lambda runtime client passing in the function handler
 to invoke for each Lambda event and
    /// the JSON serializer to use for converting Lambda JSON format to the .NET
 types.
    /// </summary>
    private static async Task Main()
    {
        Func<string, ILambdaContext, string> handler = FunctionHandler;
        await LambdaBootstrapBuilder.Create(handler, new
 SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>())
            .Build()
            .RunAsync();
    }

    /// <summary>
    /// A simple function that takes a string and does a ToUpper.
    ///
    /// To use this handler to respond to an AWS event, reference the appropriate
 package from
    /// https://github.com/aws/aws-lambda-dotnet#events
    /// and change the string input parameter to the desired event type. When the
 event type
    /// is changed, the handler type registered in the main method needs to be
 updated and the LambdaFunctionJsonSerializerContext
```

```
    /// defined below will need the JsonSerializable updated. If the return type
 and event type are different then the
    /// LambdaFunctionJsonSerializerContext must have two JsonSerializable
 attributes, one for each type.
    ///
    // When using Native AOT extra testing with the deployed Lambda functions is
 required to ensure
    // the libraries used in the Lambda function work correctly with Native AOT. If
 a runtime
    // error occurs about missing types or methods the most likely solution will be
 to remove references to trim-unsafe
    // code or configure trimming options. This sample defaults to partial TrimMode
 because currently the AWS
    // SDK for .NET does not support trimming. This will result in a larger
 executable size, and still does not
    // guarantee runtime trimming errors won't be hit.
    /// </summary>
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public static string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}

/// <summary>
/// This class is used to register the input event and return type for the
 FunctionHandler method with the System.Text.Json source generator.
/// There must be a JsonSerializable attribute for each type used as the input and
 return type or a runtime error will occur
/// from the JSON serializer unable to find the serialization information for
 unknown types.
/// </summary>
[JsonSerializable(typeof(string))]
public partial class LambdaFunctionJsonSerializerContext : JsonSerializerContext
{
    // By using this partial class derived from JsonSerializerContext, we can
 generate reflection free JSON Serializer code at compile time
    // which can deserialize our class and properties. However, we must attribute
 this class to tell it what types to generate serialization code for.
    // See https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-
text-json-source-generation
```

Native AOT compiles your application into a single, native binary. The entrypoint of that binary is the `static Main` method. Within `static Main`, the Lambda runtime is bootstrapped and the `FunctionHandler` method set up. As part of the runtime bootstrap, a source generated serializer is configured using `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()`

2. To deploy your application to Lambda, ensure that Docker is running in your local environment and run the following command.

```
dotnet lambda deploy-function
```

   Behind the scenes, the .NET global CLI downloads an AL2023 Docker image and compiles your application code inside a running container. The compiled binary is output back to your local filesystem before being deployed to Lambda.

3. Test your function by running the following command. Replace `<FUNCTION_NAME>` with the name you chose for your function in the deployment wizard.

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

   The response from the CLI includes performance details for the cold start (initialization duration) and total run time for your function invocation.

4. To delete the AWS resources you created by following the preceding steps, run the following command. Replace `<FUNCTION_NAME>` with the name you chose for your function in the deployment wizard. By deleting AWS resources that you're no longer using, you prevent unnecessary charges being billed to your AWS account.

```
dotnet lambda delete-function <FUNCTION_NAME>
```

Serialization

To deploy functions to Lambda using native AOT, your function code must use source generated serialization. Instead of using run-time reflection to gather the metadata needed to access object properties for serialization, source generators generate C# source files that are compiled when you build your application. To configure your source generated serializer correctly, ensure that you include any input and output objects your function uses, as well as any custom types. For

example, a Lambda function that receives events from an API Gateway REST API and returns a custom `Product` type would include a serializer defined as follows.

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

Trimming

Native AOT trims your application code as part of the compilation to ensure that the binary is as small as possible. .NET 8 for Lambda provides improved trimming support compared to previous versions of .NET. Support has been added to the Lambda runtime libraries, AWS .NET SDK, .NET Lambda Annotations, and .NET 8 itself.

These improvements offer the potential to eliminate build-time trimming warnings, but .NET will never be completely trim safe. This means that parts of libraries that your function relies on may be trimmed out as part of the compilation step. You can manage this by defining `TrimmerRootAssemblies` as part of your `.csproj` file as shown in the following example.

```
<ItemGroup>
    <TrimmerRootAssembly Include="AWSSDK.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Handlers.AwsSdk" />
    <TrimmerRootAssembly Include="Amazon.Lambda.APIGatewayEvents" />
    <TrimmerRootAssembly Include="bootstrap" />
    <TrimmerRootAssembly Include="Shared" />
</ItemGroup>
```

Note that when you receive a trim warning, adding the class that generates the warning to `TrimmerRootAssembly` might not resolve the issue. A trim warning indicates that the class is trying to access some other class that can't be determined until runtime. To avoid runtime errors, add this second class to `TrimmerRootAssembly`.

To learn more about managing trim warnings, see Introduction to trim warnings in the Microsoft .NET documentation.

Troubleshooting

Error: Cross-OS native compilation is not supported.

Your version of the Amazon.Lambda.Tools .NET Core global tool is out of date. Update to the latest version and try again.

Docker: image operating system "linux" cannot be used on this platform.

Docker on your system is configured to use Windows containers. Swap to Linux containers to run the native AOT build environment.

For more information about common errors, see the AWS NativeAOT for .NET repository on GitHub.

Using the Lambda context object to retrieve C# function information

When Lambda runs your function, it passes a context object to the handler. This object provides properties with information about the invocation, function, and execution environment.

Context properties

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The version of the function.
- `InvokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `MemoryLimitInMB` – The amount of memory that's allocated for the function.
- `AwsRequestId` – The identifier of the invocation request.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.
- `RemainingTime` (`TimeSpan`) – The number of milliseconds left before the execution times out.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `ClientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
- `Logger` The logger object for the function.

You can use information in the `ILambdaContext` object to output information about your function's invocation for monitoring purposes. The following code provides an example of how to add context information to a structured logging framework. In this example, the function adds `AwsRequestId` to the log outputs. The function also uses the `RemainingTime` property to cancel an inflight task if the Lambda function timeout is about to be reached.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
```

```
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        Logger.AppendKey("AwsRequestId", context.AwsRequestId);

        var id = request.PathParameters["id"];

        using var cts = new CancellationTokenSource();

        try
        {
            cts.CancelAfter(context.RemainingTime.Add(TimeSpan.FromSeconds(-1)));

            var databaseRecord = await this._repo.GetById(id, cts.Token);

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.OK,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
        catch (Exception ex)
        {
            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.InternalServerError,
                Body = JsonSerializer.Serialize(new { error = ex.Message })
            };
        }
        finally
        {
            cts.Cancel();
        }
    }
}
```

Log and monitor C# Lambda functions

AWS Lambda automatically monitors Lambda functions and sends log entries to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation and other output from your function's code to the log stream. For more information about CloudWatch Logs, see Sending Lambda function logs to CloudWatch Logs.

Sections

- Creating a function that returns logs
- Using Lambda advanced logging controls with .NET
- Additional logging tools and libraries
- Using Powertools for AWS Lambda (.NET) and AWS SAM for structured logging
- Viewing logs in the Lambda console
- Viewing logs in the CloudWatch console
- Viewing logs using the AWS Command Line Interface (AWS CLI)
- Deleting logs

Creating a function that returns logs

To output logs from your function code, you can use the ILambdaLogger on the context object, the methods on the Console class, or any logging library that writes to `stdout` or `stderr`.

The .NET runtime logs the `START`, `END`, and `REPORT` lines for each invocation. The report line provides the following details.

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.
- **Max Memory Used** – The amount of memory used by the function. When invocations share an execution environment, Lambda reports the maximum memory used across all invocations. This behavior might result in a higher than expected reported value.

- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Using Lambda advanced logging controls with .NET

To give you more control over how your functions' logs are captured, processed, and consumed, you can configure the following logging options for supported .NET runtimes:

- **Log format** - select between plain text and structured JSON format for your function's logs
- **Log level** - for logs in JSON format, choose the detail level of the logs Lambda sends to CloudWatch, such as ERROR, DEBUG, or INFO
- **Log group** - choose the CloudWatch log group your function sends logs to

For more information about these logging options, and instructions on how to configure your function to use them, see the section called "Configuring advanced logging controls for Lambda functions".

To use the log format and log level options with your .NET Lambda functions, see the guidance in the following sections.

Using structured JSON log format with .NET

If you select JSON for your function's log format, Lambda will send logs output using ILambdaLogger as structured JSON. Each JSON log object contains at least five key value pairs with the following keys:

- `"timestamp"` - the time the log message was generated
- `"level"` - the log level assigned to the message
- `"requestId"` - the unique request ID for the function invocation
- `"traceId"` - the `_X_AMZN_TRACE_ID` environment variable
- `"message"` - the contents of the log message

The `ILambdaLogger` instance can add additional key value pairs, for example when logging exceptions. You can also supply your own additional parameters as described in the section the section called “Customer-provided log parameters”.

> **Note**
>
> If your code already uses another logging library to produce JSON-formatted logs, ensure that your function's log format is set to plain text. Setting the log format to JSON will result in your log outputs being double-encoded.

The following example logging command shows how to write a log message with the level `INFO`.

Example.NET logging code

```
context.Logger.LogInformation("Fetching cart from database");
```

You can also use a generic log method that takes the log level as an argument as shown in the following example.

```
context.Logger.Log(LogLevel.Information, "Fetching cart from database");
```

The log output by these example code snippets would be captured in CloudWatch Logs as follows:

Example JSON log record

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Fetching cart from database"
}
```

> **Note**
>
> If you configure your function's log format to to use plain text rather than JSON, then the log level captured in the message follows the Microsoft convention of using a four-character label. For example, a log level of `Debug` is represented in the message as `dbug`.

When you configure your function to use JSON formatted logs, the log level captured in the log uses the full label as shown in the example JSON log record.

If you don't assign a level to your log output, Lambda will automatically assign it the level INFO.

Logging exceptions in JSON

When using structured JSON logging with `ILambdaLogger`, you can log exceptions in your code as shown in the following example.

Example usage of exception logging

```
try
{
    connection.ExecuteQuery(query);
}
catch(Exception e)
{
    context.Logger.LogWarning(e, "Error executing query");
}
```

The log format output by this code is shown in the following example JSON. Note that the `message` property in the JSON is populated using the message argument provided in the `LogWarning` call, while the `errorMessage` property comes from the `Message` property of the exception itself.

Example JSON log record

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Warning",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Error executing query",
    "errorType": "System.Data.SqlClient.SqlException",
    "errorMessage": "Connection closed",
    "stackTrace": ["<call exception.StackTrace>"]
}
```

If your function's logging format is set to JSON, Lambda also outputs JSON-formatted log messages when your code throws an uncaught exception. The following example code snippet and log message show how uncaught exceptions are logged.

Example exception code

```
throw new ApplicationException("Invalid data");
```

Example JSON log record

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Error",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Invalid data",
    "errorType": "System.ApplicationException",
    "errorMessage": "Invalid data",
    "stackTrace": ["<call exception.StackTrace>"]
}
```

Customer-provided log parameters

With JSON-formatted log messages, you can supply additional log parameters and include these in the log `message`. The following code snippet example shows a command to add two user-supplied parameters labeled `retryAttempt` and `uri`. In the example, the value of these parameters come from the `retryAttempt` and `uriDestination` arguments passed into the logging command.

Example JSON logging command with additional parameters

```
context.Logger.LogInformation("Starting retry {retryAttempt} to make GET request to
 {uri}", retryAttempt, uriDestination);
```

The log message output by this command is shown in the following example JSON.

Example JSON log record

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
```

```
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Starting retry 1 to make GET request to http://example.com/",
    "retryAttempt": 1,
    "uri": "http://example.com/"
}
```

Tip

You can also use positional properties instead of names when specifying additional parameters. For example, the logging command in the previous example could also be written as follows:

```
context.Logger.LogInformation("Starting retry {0} to make GET request to {1}",
 retryAttempt, uriDestination);
```

Note that when you supply additional logging parameters, Lambda captures them as top-level properties in the JSON log record. This approach differs from some popular .NET logging libraries such as `Serilog`, which captures additional parameters in a separate child object.

If the argument you supply for an additional parameter is a complex object, by default Lambda uses the `ToString()` method to supply the value. To indicate that an argument should be JSON serialized, use the `@` prefix as shown in the following code snippet. In this example, `User` is an object with `FirstName` and `LastName` properties.

Example JSON logging command with JSON serialized object

```
context.Logger.LogInformation("User {@user} logged in", User);
```

The log message output by this command is shown in the following example JSON.

Example JSON log record

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "User {@user} logged in",
```

```
    "user":
    {
        "FirstName": "John",
        "LastName": "Doe"
    }
}
```

If the argument for an additional parameter is an array or implements `IList` or `IDictionary`, then Lambda adds the argument to the JSON log message as an array as shown in the following example JSON log record. In this example, `{users}` takes an `IList` argument containing instances of the `User` property with the same format as the previous example. Lambda converts this `IList` into an array, with each value being created using the `ToString` method.

Example JSON log record with an `IList` argument

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "{users} have joined the group",
    "users":
    [
        "Rosalez, Alejandro",
        "Stiles, John"
    ]
}
```

You can also JSON serialize the list by using the `@` prefix in your logging command. In the following example JSON log record, the `users` property is JSON serialized.

Example JSON log record with a JSON serialized `IList` argument

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "{@users} have joined the group",
    "users":
    [
        {
```

```
            "FirstName": "Alejandro",
            "LastName": "Rosalez"
        },
        {
            "FirstName": "John",
            "LastName": "Stiles"
        }
    ]
}
```

Using log-level filtering with .NET

By configuring log-level filtering, you can choose to send only logs of a certain detail level or lower to CloudWatch Logs. To learn how to configure log-level filtering for your function, see the section called “Log-level filtering”.

For AWS Lambda to filter your log messages by log level, you can either use JSON formatted logs or use the .NET `Console` methods to output log messages. To create JSON formatted logs, configure your function's log type to JSON and use the `ILambdaLogger` instance.

With JSON-formatted logs, Lambda filters your log outputs using the “level” key value pair in the JSON object described in the section called “Using structured JSON log format with .NET”.

If you use the .NET `Console` methods to write messages to CloudWatch Logs, Lambda applies log levels to your messages as follows:

- **Console.WriteLine method** - Lambda applies a log-level of `INFO`
- **Console.Error method** - Lambda applies a log-level of `ERROR`

When you configure your function to use log-level filtering, you must select from the following options for the level of logs you want Lambda to send to CloudWatch Logs. Note the mapping of the log levels used by Lambda with the standard Microsoft levels used by the .NET `ILambdaLogger`.

Lambda log level	Equivalent Microsoft level	Standard usage
TRACE (most detail)	Trace	The most fine-grained information used to trace the path of your code's execution

Lambda log level	Equivalent Microsoft level	Standard usage
DEBUG	Debug	Detailed information for system debugging
INFO	Information	Messages that record the normal operation of your function
WARN	Warning	Messages about potential errors that may lead to unexpected behavior if unaddressed
ERROR	Error	Messages about problems that prevent the code from performing as expected
FATAL (least detail)	Critical	Messages about serious errors that cause the application to stop functioning

Lambda sends logs of the selected detail level and lower to CloudWatch. For example, if you configure a log level of WARN, Lambda will send logs corresponding to the WARN, ERROR, and FATAL levels.

Additional logging tools and libraries

Powertools for AWS Lambda (.NET) is a developer toolkit to implement Serverless best practices and increase developer velocity. The Logging utility provides a Lambda optimized logger which includes additional information about function context across all your functions with output structured as JSON. Use this utility to do the following:

- Capture key fields from the Lambda context, cold start and structures logging output as JSON
- Log Lambda invocation events when instructed (disabled by default)
- Print all the logs only for a percentage of invocations via log sampling (disabled by default)
- Append additional keys to structured log at any point in time

- Use a custom log formatter (Bring Your Own Formatter) to output logs in a structure compatible with your organization's Logging RFC

Using Powertools for AWS Lambda (.NET) and AWS SAM for structured logging

Follow the steps below to download, build, and deploy a sample Hello World C# application with integrated Powertools for AWS Lambda (.NET) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a `hello world` message.

Prerequisites

To complete the steps in this section, you must have the following:

- .NET 8
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press `Enter`.

   > **Note**
   >
   > For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter `y`.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

   If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the logs for the function, run sam logs. For more information, see Working with logs in the *AWS Serverless Application Model Developer Guide*.

```
sam logs --stack-name sam-app
```

   The log output looks like this:

```
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8
 2025-09-20T14:15:27.988000 INIT_START Runtime Version:
 dotnet:6.v13        Runtime Version ARN: arn:aws:lambda:ap-
southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17dae3997e79b17d939a288aa1ec
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:28.229000
 START RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:29.259000
 2025-09-20T14:15:29.201Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528962,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ColdStart","Unit":"Count"}],"Dimensions":
[["FunctionName"],["Service"]]}]},"FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
```

```
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.479000
 2025-09-20T14:15:30.479Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
 {"ColdStart":true,"XrayTraceId":"1-63f3807f-5dbcb9910c96f50742707542","CorrelationId":"d3d
a549-4d67b2fdc015","FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionVersion":"$LATEST","FunctionMemorySize":256,"FunctionArn":"arn:aws:l
southeast-2:123456789012:function:sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionRequestId":"bed25b38-d012-42e7-ba28-
f272535fb80e","Timestamp":"2025-09-20T14:15:30.4602970Z","Level":"Information","Service":"P
 world API - HTTP 200"}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.599000
 2025-09-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions":
[["Service"]]}]},"Service":"PowertoolsHelloWorld","ApiRequestCount":1}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.680000 END
 RequestId: bed25b38-d012-42e7-ba28-f272535fb80e
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.680000
 REPORT RequestId: bed25b38-d012-42e7-ba28-f272535fb80e  Duration: 2450.99 ms
  Billed Duration: 2692 ms Memory Size: 256 MB     Max Memory Used: 74 MB  Init
 Duration: 240.05 ms
XRAY TraceId: 1-63f3807f-5dbcb9910c96f50742707542       SegmentId: 16b362cd5f52cba0
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

Managing log retention

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which CloudWatch automatically deletes the logs. To set up log retention, add the following to your AWS SAM template:

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
```

```
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

Viewing logs in the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Viewing logs in the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/*your-function-name***).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

Viewing logs using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
```

```
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Instrumenting C# code in AWS Lambda

Lambda integrates with AWS X-Ray to help you trace, debug, and optimize Lambda applications. You can use X-Ray to trace a request as it traverses resources in your application, which may include Lambda functions and other AWS services.

To send tracing data to X-Ray, you can use one of three SDK libraries:

- AWS Distro for OpenTelemetry (ADOT) – A secure, production-ready, AWS-supported distribution of the OpenTelemetry (OTel) SDK.
- AWS X-Ray SDK for .NET – An SDK for generating and sending trace data to X-Ray.
- Powertools for AWS Lambda (.NET) – A developer toolkit to implement Serverless best practices and increase developer velocity.

Each of the SDKs offer ways to send your telemetry data to the X-Ray service. You can then use X-Ray to view, filter, and gain insights into your application's performance metrics to identify issues and opportunities for optimization.

> **Important**
>
> The X-Ray and Powertools for AWS Lambda SDKs are part of a tightly integrated instrumentation solution offered by AWS. The ADOT Lambda Layers are part of an industry-wide standard for tracing instrumentation that collect more data in general, but may not be suited for all use cases. You can implement end-to-end tracing in X-Ray using either solution. To learn more about choosing between them, see Choosing between the AWS Distro for Open Telemetry and X-Ray SDKs.

Sections

- Using Powertools for AWS Lambda (.NET) and AWS SAM for tracing
- Using the X-Ray SDK to instrument your .NET functions
- Activating tracing with the Lambda console
- Activating tracing with the Lambda API
- Activating tracing with CloudFormation
- Interpreting an X-Ray trace

Using Powertools for AWS Lambda (.NET) and AWS SAM for tracing

Follow the steps below to download, build, and deploy a sample Hello World C# application with integrated Powertools for AWS Lambda (.NET) modules using the AWS SAM. This application implements a basic API backend and uses Powertools for emitting logs, metrics, and traces. It consists of an Amazon API Gateway endpoint and a Lambda function. When you send a GET request to the API Gateway endpoint, the Lambda function invokes, sends logs and metrics using Embedded Metric Format to CloudWatch, and sends traces to AWS X-Ray. The function returns a hello world message.

Prerequisites

To complete the steps in this section, you must have the following:

- .NET 8
- AWS CLI version 2
- AWS SAM CLI version 1.75 or later. If you have an older version of the AWS SAM CLI, see Upgrading the AWS SAM CLI.

Deploy a sample AWS SAM application

1. Initialize the application using the Hello World TypeScript template.

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. Build the app.

```
cd sam-app && sam build
```

3. Deploy the app.

```
sam deploy --guided
```

4. Follow the on-screen prompts. To accept the default options provided in the interactive experience, press `Enter`.

> **Note**
>
> For **HelloWorldFunction may not have authorization defined, Is this okay?**, make sure to enter y.

5. Get the URL of the deployed application:

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. Invoke the API endpoint:

```
curl <URL_FROM_PREVIOUS_STEP>
```

If successful, you'll see this response:

```
{"message":"hello world"}
```

7. To get the traces for the function, run sam traces.

```
sam traces
```

The trace output looks like this:

```
New XRay Service Graph
  Start time: 2023-02-20 23:05:16+08:00
  End time: 2023-02-20 23:05:16+08:00
  Reference Id: 0 - AWS::Lambda - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges:
 [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.814
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-pNjujb7mEoew
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
```

```
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 2.429
  Reference Id: 2 - (Root) AWS::ApiGateway::Stage - sam-app/Prod - Edges: [0]
   Summary_statistics:
    - total requests: 1
    - ok count(2XX): 1
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 2.839
  Reference Id: 3 - client - sam-app/Prod - Edges: [2]
   Summary_statistics:
    - total requests: 0
    - ok count(2XX): 0
    - error count(4XX): 0
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 3] at (2023-02-20T23:05:16.521000) with id
 (1-63f38c2c-270200bf1d292a442c8e8a00) and duration (2.877s)
 - 2.839s - sam-app/Prod [HTTP: 200]
   - 2.836s - Lambda [HTTP: 200]
 - 2.814s - sam-app-HelloWorldFunction-pNjujb7mEoew [HTTP: 200]
 - 2.429s - sam-app-HelloWorldFunction-pNjujb7mEoew
   - 0.230s - Initialization
   - 2.389s - Invocation
     - 0.600s - ## FunctionHandler
       - 0.517s - Get Calling IP
   - 0.039s - Overhead
```

8. This is a public API endpoint that is accessible over the internet. We recommend that you delete the endpoint after testing.

```
sam delete
```

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

Using the X-Ray SDK to instrument your .NET functions

You can instrument your function code to record metadata and trace downstream calls. To record detail about calls that your function makes to other resources and services, use the AWS X-Ray SDK for .NET. To get the SDK, add the `AWSXRayRecorder` packages to your project file.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.SQSEvents" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.1.0" />
    <PackageReference Include="AWSSDK.Core" Version="3.7.103.24" />
    <PackageReference Include="AWSSDK.Lambda" Version="3.7.104.3" />
    <PackageReference Include="AWSXRayRecorder.Core" Version="2.13.0" />
    <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.11.0" />
  </ItemGroup>
</Project>
```

There are a range of Nuget packages that provide auto-instrumentation for AWS SDKs, Entity Framework and HTTP requests. To see the complete set of configuration options refer to AWS X-Ray SDK for .NET in the *AWS X-Ray Developer Guide*.

Once you have added the desired Nuget packages, configure auto-instrumentation. Best practice is to perform this configuration outside of your function's handler function. This allows you to take advantage of execution environment re-use to improve the performance of your function. In the following code example, the `RegisterXRayForAllServices` method is called in the function constructor to add instrumentation for all AWS SDK calls.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;
```

```
    public Function()
    {
        // Add auto instrumentation for all AWS SDK calls
        // It is important to call this method before initializing any SDK clients
        AWSSDKHandler.RegisterXRayForAllServices();
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

Activating tracing with the Lambda console

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Under **Additional monitoring tools**, choose **Edit**.
5. Under **CloudWatch Application Signals and AWS X-Ray**, choose **Enable** for **Lambda service traces**.
6. Choose **Save**.

Activating tracing with the Lambda API

Configure tracing on your Lambda function with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Activating tracing with CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an CloudFormation template, use the `TracingConfig` property.

Example function-inline.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example template.yml – Tracing configuration

```
Resources:
  function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Interpreting an X-Ray trace

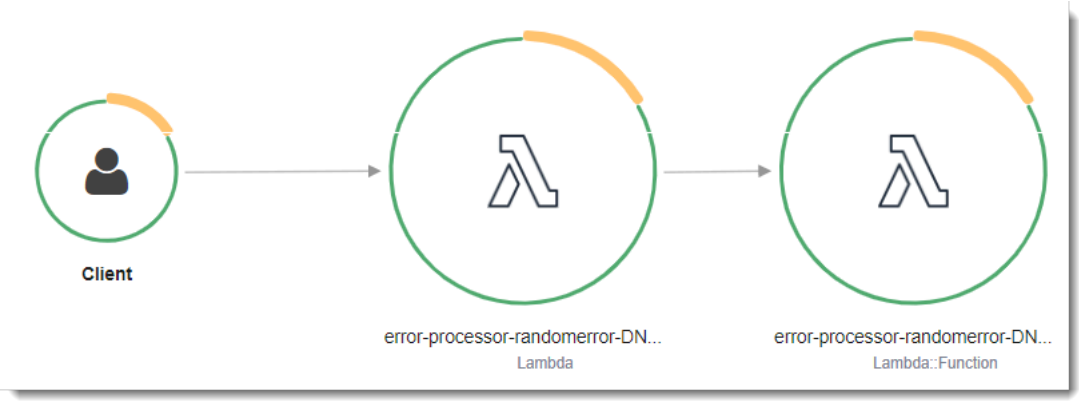
Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

After you've configured active tracing, you can observe specific requests through your application. The X-Ray service graph shows information about your application and all its components. The following example shows an application with two functions. The primary function processes events and sometimes returns errors. The second function at the top processes errors that appear in the first's log group and uses the AWS SDK to call X-Ray, Amazon Simple Storage Service (Amazon S3), and Amazon CloudWatch Logs.

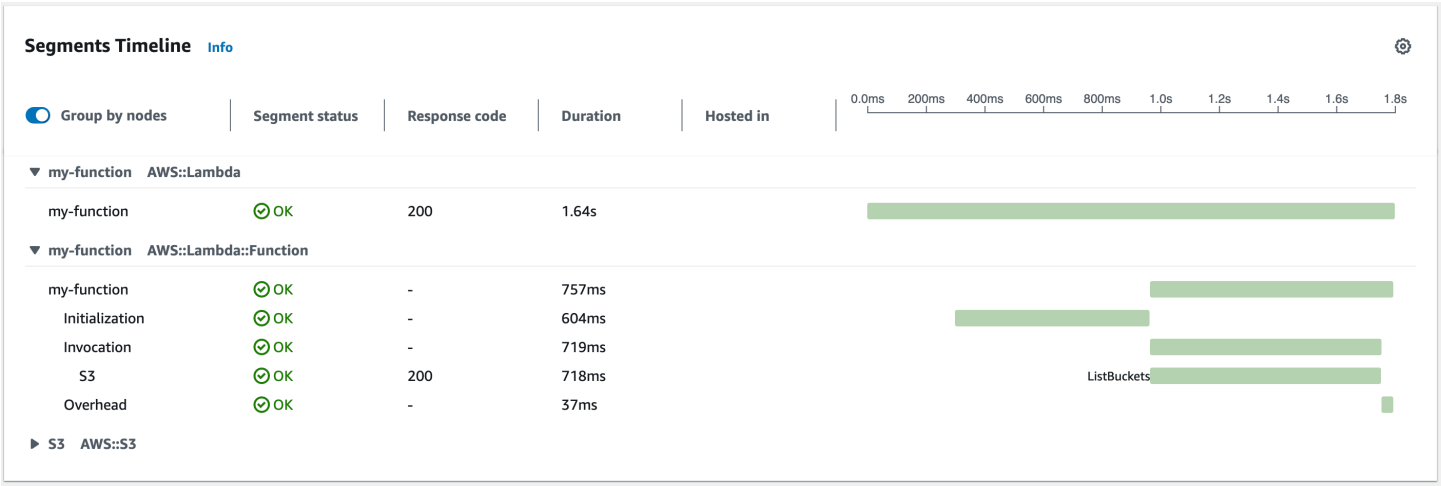


X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

In X-Ray, a *trace* records information about a request that is processed by one or more *services*. Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes:



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function. The following example shows a trace with these two segments. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.



This example expands the `AWS::Lambda::Function` segment to show its three subsegments.

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
> The example trace shown here illustrates the old-style function segment. The differences between the old- and new-style segments are described in the following paragraphs.

These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

The old-style function segment contains the following subsegments:

- **Initialization** – Represents time spent loading your function and running initialization code. This subsegment only appears for the first event that each instance of your function processes.
- **Invocation** – Represents the time spent running your handler code.
- **Overhead** – Represents the time the Lambda runtime spends preparing to handle the next event.

The new-style function segment doesn't contain an `Invocation` subsegment. Instead, customer subsegments are attached directly to the function segment. For more information about the structure of the old- and new-style function segments, see the section called "Understanding X-Ray traces".

You can also instrument HTTP clients, record SQL queries, and create custom subsegments with annotations and metadata. For more information, see the AWS X-Ray SDK for .NET in the *AWS X-Ray Developer Guide*.

Pricing

You can use X-Ray tracing for free each month up to a certain limit as part of the AWS Free Tier. Beyond that threshold, X-Ray charges for trace storage and retrieval. For more information, see AWS X-Ray pricing.

AWS Lambda function testing in C#

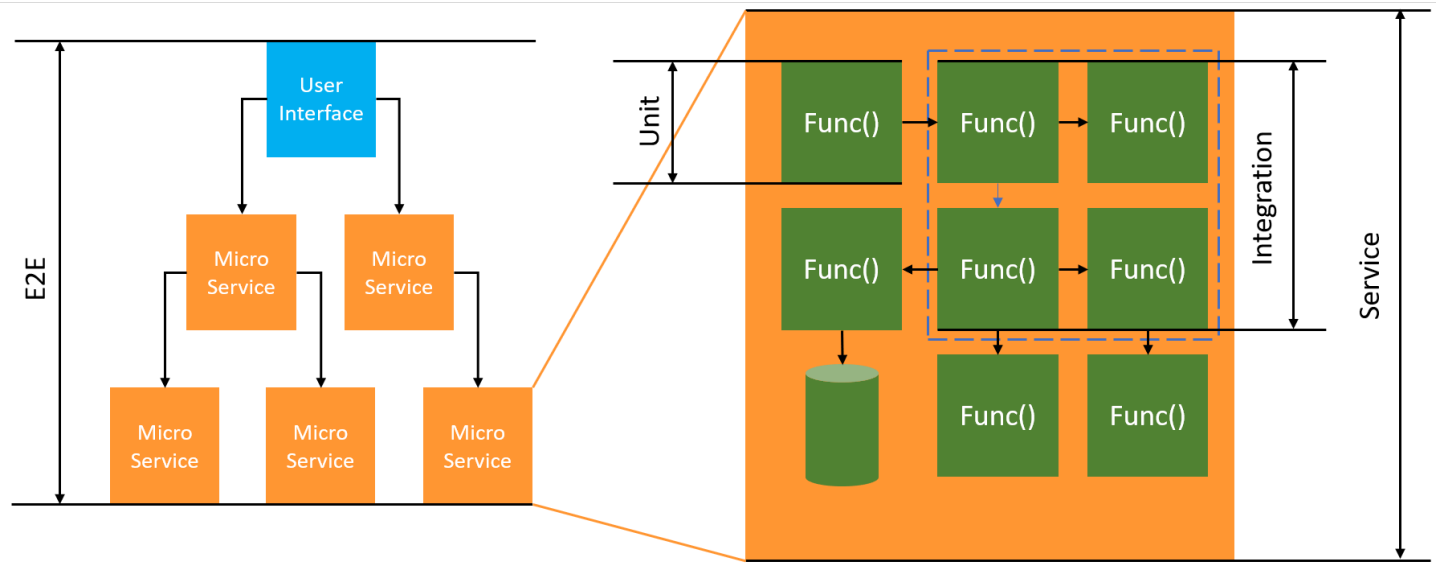
> **Note**
>
> See the Testing functions chapter for a complete introduction to techniques and best practices for testing serverless solutions.

Testing serverless functions uses traditional test types and techniques, but you must also consider testing serverless applications as a whole. Cloud-based tests will provide the **most accurate** measure of quality of both your functions and serverless applications.

A serverless application architecture includes managed services that provide critical application functionality through API calls. For this reason, your development cycle should include automated tests that verify functionality when your function and services interact.

If you do not create cloud-based tests, you could encounter issues due to differences between your local environment and the deployed environment. Your continuous integration process should run tests against a suite of resources provisioned in the cloud before promoting your code to the next deployment environment, such as QA, Staging, or Production.

Continue reading this short guide to learn about testing strategies for serverless applications, or visit the Serverless Test Samples repository to dive in with practical examples, specific to your chosen language and runtime.



For serverless testing, you will still write *unit*, *integration* and *end-to-end* tests.

- **Unit tests** - Tests that run against an isolated block of code. For example, verifying the business logic to calculate the delivery charge given a particular item and destination.
- **Integration tests** - Tests involving two or more components or services that interact, typically in a cloud environment. For example, verifying a function processes events from a queue.
- **End-to-end tests** - Tests that verify behavior across an entire application. For example, ensuring infrastructure is set up correctly and that events flow between services as expected to record a customer's order.

Testing your serverless applications

You will generally use a mix of approaches to test your serverless application code, including testing in the cloud, testing with mocks, and occasionally testing with emulators.

Testing in the cloud

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. You run tests against code deployed in the cloud and interacting with cloud-based services. This approach provides the **most accurate** measure of quality of your code.

A convenient way to debug your Lambda function in the cloud is through the console with a test event. A *test event* is a JSON input to your function. If your function does not require input, the event can be an empty JSON document (`{}`). The console provides sample events for a variety of service integrations. After creating an event in the console, you can share it with your team to make testing easier and consistent.

> **Note**
>
> Testing a function in the console is a quick way to get started, but automating your test cycles ensures application quality and development speed.

Testing tools

To accelerate your development cycle, there are a number of tools and techniques you can use when testing your functions. For example, AWS SAM Accelerate and AWS CDK watch mode both decrease the time required to update cloud environments.

The way you define your Lambda function code makes it simple to add unit tests. Lambda requires a public, parameterless constructor to initialize your class. Introducing a second, internal constructor gives you control of the dependencies your application uses.

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(): this(null)
    {
    }

    internal Function(IDatabaseRepository repo)
    {
        this._repo = repo ?? new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

To write a test for this function, you can initialize a new instance of your `Function` class and pass in a mocked implementation of the `IDatabaseRepository`. The below examples uses `XUnit`, `Moq`, and `FluentAssertions` to write a simple test ensuring the `FunctionHandler` returns a 200 status code.

```
using Xunit;
using Moq;
using FluentAssertions;

public class FunctionTests
{
    [Fact]
    public async Task TestLambdaHandler_WhenInputIsValid_ShouldReturn200StatusCode()
    {
        // Arrange
        var mockDatabaseRepository = new Mock<IDatabaseRepository>();

        var functionUnderTest = new Function(mockDatabaseRepository.Object);

        // Act
        var response = await functionUnderTest.FunctionHandler(new
 APIGatewayProxyRequest());

        // Assert
        response.StatusCode.Should().Be(200);
    }
}
```

For more detailed examples, including examples of asynchronous tests, see the .NET testing samples repository on GitHub.

Building Lambda functions with PowerShell

The following sections explain how common programming patterns and core concepts apply when you author Lambda function code in PowerShell.

Lambda provides the following sample applications for PowerShell:

- blank-powershell – A PowerShell function that shows the use of logging, environment variables, and the AWS SDK.

Before you get started, you must first set up a PowerShell development environment. For instructions on how to do this, see Setting Up a PowerShell Development Environment.

To learn about how to use the AWSLambdaPSCore module to download sample PowerShell projects from templates, create PowerShell deployment packages, and deploy PowerShell functions to the AWS Cloud, see Deploy PowerShell Lambda functions with .zip file archives.

Lambda provides the following runtimes for .NET languages:

Name	Identifier	Operating system	Deprecation date	Block function create	Block function update
.NET 10	`dotnet10`	Amazon Linux 2023	Nov 14, 2028	Dec 14, 2028	Jan 15, 2029
.NET 9 (container only)	`dotnet9`	Amazon Linux 2023	Nov 10, 2026	Not scheduled	Not scheduled
.NET 8	`dotnet8`	Amazon Linux 2023	Nov 10, 2026	Dec 10, 2026	Jan 11, 2027

Topics

- Setting Up a PowerShell Development Environment
- Deploy PowerShell Lambda functions with .zip file archives

- Define Lambda function handler in PowerShell
- Using the Lambda context object to retrieve PowerShell function information
- Log and monitor Powershell Lambda functions

Setting Up a PowerShell Development Environment

Lambda provides a set of tools and libraries for the PowerShell runtime. For installation instructions, see Lambda tools for PowerShell on GitHub.

The AWSLambdaPSCore module includes the following cmdlets to help author and publish PowerShell Lambda functions:

- **Get-AWSPowerShellLambdaTemplate** – Returns a list of getting started templates.
- **New-AWSPowerShellLambda** – Creates an initial PowerShell script based on a template.
- **Publish-AWSPowerShellLambda** – Publishes a given PowerShell script to Lambda.
- **New-AWSPowerShellLambdaPackage** – Creates a Lambda deployment package that you can use in a CI/CD system for deployment.

Deploy PowerShell Lambda functions with .zip file archives

A deployment package for the PowerShell runtime contains your PowerShell script, PowerShell modules that are required for your PowerShell script, and the assemblies needed to host PowerShell Core.

Creating the Lambda function

To get started writing and invoking a PowerShell script with Lambda, you can use the `New-AWSPowerShellLambda` cmdlet to create a starter script based on a template. You can use the `Publish-AWSPowerShellLambda` cmdlet to deploy your script to Lambda. Then you can test your script either through the command line or the Lambda console.

To create a new PowerShell script, upload it, and test it, do the following:

1. To view the list of available templates, run the following command:

```
PS C:\> Get-AWSPowerShellLambdaTemplate

Template               Description
--------               -----------
Basic                  Bare bones script
CodeCommitTrigger      Script to process AWS CodeCommit Triggers
...
```

2. To create a sample script based on the `Basic` template, run the following command:

```
New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
```

   A new file named `MyFirstPSScript.ps1` is created in a new subdirectory of the current directory. The name of the directory is based on the `-ScriptName` parameter. You can use the `-Directory` parameter to choose an alternative directory.

   You can see that the new file has the following contents:

```
# PowerShell script file to run as a Lambda function
#
# When executing in Lambda the following variables are predefined.
#   $LambdaInput - A PSObject that contains the Lambda function input data.
#   $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains
 information about the currently running Lambda environment.
```

```
#
# The last item in the PowerShell pipeline is returned as the result of the Lambda
 function.
#
# To include PowerShell modules with your Lambda function, like the
 AWSPowerShell.NetCore module, add a "#Requires" statement
# indicating the module and version.

#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}

# Uncomment to send the input to CloudWatch Logs
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. To see how log messages from your PowerShell script are sent to Amazon CloudWatch Logs, uncomment the `Write-Host` line of the sample script.

   To demonstrate how you can return data back from your Lambda functions, add a new line at the end of the script with `$PSVersionTable`. This adds the `$PSVersionTable` to the PowerShell pipeline. After the PowerShell script is complete, the last object in the PowerShell pipeline is the return data for the Lambda function. `$PSVersionTable` is a PowerShell global variable that also provides information about the running environment.

   After making these changes, the last two lines of the sample script look like this:

   ```
   Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
   $PSVersionTable
   ```

4. After editing the `MyFirstPSScript.ps1` file, change the directory to the script's location. Then run the following command to publish the script to Lambda:

   ```
   Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
    MyFirstPSScript -Region us-east-2
   ```

   Note that the `-Name` parameter specifies the Lambda function name, which appears in the Lambda console. You can use this function to invoke your script manually.

5. Invoke your function using the AWS Command Line Interface (AWS CLI) `invoke` command.

   ```
   > aws lambda invoke --function-name MyFirstPSScript out
   ```

Define Lambda function handler in PowerShell

When a Lambda function is invoked, the Lambda handler invokes the PowerShell script.

When the PowerShell script is invoked, the following variables are predefined:

- *$LambdaInput* – A PSObject that contains the input to the handler. This input can be event data (published by an event source) or custom input that you provide, such as a string or any custom data object.
- *$LambdaContext* – An Amazon.Lambda.Core.ILambdaContext object that you can use to access information about the current invocation—such as the name of the current function, the memory limit, execution time remaining, and logging.

For example, consider the following PowerShell example code.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

This script returns the FunctionName property that's obtained from the $LambdaContext variable.

> **Note**
>
> You're required to use the `#Requires` statement within your PowerShell scripts to indicate the modules that your scripts depend on. This statement performs two important tasks. 1) It communicates to other developers which modules the script uses, and 2) it identifies the dependent modules that AWS PowerShell tools need to package with the script, as part of the deployment. For more information about the `#Requires` statement in PowerShell, see About requires. For more information about PowerShell deployment packages, see Deploy PowerShell Lambda functions with .zip file archives.
>
> When your PowerShell Lambda function uses the AWS PowerShell cmdlets, be sure to set a `#Requires` statement that references the `AWSPowerShell.NetCore` module, which supports PowerShell Core—and not the `AWSPowerShell` module, which only supports Windows PowerShell. Also, be sure to use version 3.3.270.0 or newer of `AWSPowerShell.NetCore` which optimizes the cmdlet import process. If you use an older version, you'll experience longer cold starts. For more information, see AWS Tools for PowerShell.

Returning data

Some Lambda invocations are meant to return data back to their caller. For example, if an invocation was in response to a web request coming from API Gateway, then our Lambda function needs to return back the response. For PowerShell Lambda, the last object that's added to the PowerShell pipeline is the return data from the Lambda invocation. If the object is a string, the data is returned as is. Otherwise the object is converted to JSON by using the `ConvertTo-Json` cmdlet.

For example, consider the following PowerShell statement, which adds `$PSVersionTable` to the PowerShell pipeline:

```
$PSVersionTable
```

After the PowerShell script is finished, the last object in the PowerShell pipeline is the return data for the Lambda function. `$PSVersionTable` is a PowerShell global variable that also provides information about the running environment.

Using the Lambda context object to retrieve PowerShell function information

When Lambda runs your function, it passes context information by making a `$LambdaContext` variable available to the handler. This variable provides methods and properties with information about the invocation, function, and execution environment.

Context properties

- `FunctionName` – The name of the Lambda function.
- `FunctionVersion` – The version of the function.
- `InvokedFunctionArn` – The Amazon Resource Name (ARN) that's used to invoke the function. Indicates if the invoker specified a version number or alias.
- `MemoryLimitInMB` – The amount of memory that's allocated for the function.
- `AwsRequestId` – The identifier of the invocation request.
- `LogGroupName` – The log group for the function.
- `LogStreamName` – The log stream for the function instance.
- `RemainingTime` – The number of milliseconds left before the execution times out.
- `Identity` – (mobile apps) Information about the Amazon Cognito identity that authorized the request.
- `ClientContext` – (mobile apps) Client context that's provided to Lambda by the client application.
- `Logger` – The logger object for the function.

The following PowerShell code snippet shows a simple handler function that prints some of the context information.

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```

Log and monitor Powershell Lambda functions

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see Sending Lambda function logs to CloudWatch Logs.

This page describes how to produce log output from your Lambda function's code, and access logs using the AWS Command Line Interface, the Lambda console, or the CloudWatch console.

Sections

- Creating a function that returns logs
- Viewing logs in the Lambda console
- Viewing logs in the CloudWatch console
- Viewing logs using the AWS Command Line Interface (AWS CLI)
- Deleting logs

Creating a function that returns logs

To output logs from your function code, you can use cmdlets on Microsoft.PowerShell.Utility , or any logging module that writes to `stdout` or `stderr`. The following example uses `Write-Host`.

Example function/Handler.ps1 – Logging

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

Example log format

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
```

```
Importing module ./Modules/AWSPowerShell.NetCore/3.3.618.0/AWSPowerShell.NetCore.psd1
[Information] - ## Environment variables
[Information] - AWS_LAMBDA_FUNCTION_VERSION=$LATEST
[Information] - AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/blank-powershell-
function-18CIXMPLHFAJJ
[Information] - AWS_LAMBDA_LOG_STREAM_NAME=2020/04/01/
[$LATEST]53c5xmpl52d64ed3a744724d9c201089
[Information] - AWS_EXECUTION_ENV=AWS_Lambda_dotnet6_powershell_1.0.0
[Information] - AWS_LAMBDA_FUNCTION_NAME=blank-powershell-function-18CIXMPLHFAJJ
[Information] - PATH=/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin
[Information] - ## Event
[Information] -
{
    "Records": [
        {
            "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
            "receiptHandle": "MessageReceiptHandle",
            "body": "Hello from SQS!",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1523232000000",
                "SenderId": "123456789012",
                "ApproximateFirstReceiveTimestamp": "1523232000001"
            },
            ...
END RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed
REPORT RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Duration: 3906.38 ms Billed
 Duration: 9867 ms Memory Size: 512 MB Max Memory Used: 367 MB Init Duration: 5960.19
 ms
XRAY TraceId: 1-5e843da6-733cxmple7d0c3c020510040 SegmentId: 3913xmpl20999446 Sampled:
 true
```

The .NET runtime logs the START, END, and REPORT lines for each invocation. The report line provides the following details.

REPORT line data fields

- **RequestId** – The unique request ID for the invocation.
- **Duration** – The amount of time that your function's handler method spent processing the event.
- **Billed Duration** – The amount of time billed for the invocation.
- **Memory Size** – The amount of memory allocated to the function.

- **Max Memory Used** – The amount of memory used by the function. When invocations share an execution environment, Lambda reports the maximum memory used across all invocations. This behavior might result in a higher than expected reported value.
- **Init Duration** – For the first request served, the amount of time it took the runtime to load the function and run code outside of the handler method.
- **XRAY TraceId** – For traced requests, the AWS X-Ray trace ID.
- **SegmentId** – For traced requests, the X-Ray segment ID.
- **Sampled** – For traced requests, the sampling result.

Viewing logs in the Lambda console

You can use the Lambda console to view log output after you invoke a Lambda function.

If your code can be tested from the embedded **Code** editor, you will find logs in the **execution results**. When you use the console test feature to invoke a function, you'll find **Log output** in the **Details** section.

Viewing logs in the CloudWatch console

You can use the Amazon CloudWatch console to view logs for all Lambda function invocations.

To view logs on the CloudWatch console

1. Open the Log groups page on the CloudWatch console.
2. Choose the log group for your function (**/aws/lambda/*your-function-name***).
3. Choose a log stream.

Each log stream corresponds to an instance of your function. A log stream appears when you update your Lambda function, and when additional instances are created to handle concurrent invocations. To find logs for a specific invocation, we recommend instrumenting your function with AWS X-Ray. X-Ray records details about the request and the log stream in the trace.

Viewing logs using the AWS Command Line Interface (AWS CLI)

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
```

```
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
```

```
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

Deleting logs

Log groups aren't deleted automatically when you delete a function. To avoid storing logs indefinitely, delete the log group, or configure a retention period after which logs are deleted automatically.

Building Lambda functions with Rust

Because Rust compiles to native code, you don't need a dedicated runtime to run Rust code on Lambda. Instead, use the Rust runtime client to build your project locally, and then deploy it to Lambda using an OS-only runtime. When you use an OS-only runtime, Lambda automatically keeps the operating system up to date with the latest patches.

Tools and libraries for Rust

- AWS SDK for Rust: The AWS SDK for Rust provides Rust APIs to interact with Amazon Web Services infrastructure services.
- Rust runtime client for Lambda: The Rust runtime client makes it easy to run Lambda functions written in Rust.
- Cargo Lambda: This is a third-party open-source extension to the Cargo command-line tool that simplifies building and deploying Rust Lambda functions.
- Lambda HTTP: This library provides a wrapper to work with HTTP events.
- Lambda Extension: This library provides support to write Lambda Extensions with Rust.
- AWS Lambda Events: This library provides type definitions for common event source integrations.

Sample Lambda applications for Rust

- Basic Lambda function: A Rust function that shows how to process basic events.
- Lambda function with error handling: A Rust function that shows how to handle custom Rust errors in Lambda.
- Lambda function with shared resources: A Rust project that initializes shared resources before creating the Lambda function.
- Lambda HTTP events: A Rust function that handles HTTP events.
- Lambda HTTP events with CORS headers: A Rust function that uses Tower to inject CORS headers.
- Lambda REST API: A REST API that uses Axum and Diesel to connect to a PostgreSQL database.
- Serverless Rust demo: A Rust project that shows the use of Lambda's Rust libraries, logging, environment variables, and the AWS SDK.
- Basic Lambda Extension: A Rust extension that shows how to process basic extension events.

- Lambda Logs Amazon Data Firehose Extension: A Rust extension that shows how to send Lambda logs to Firehose.

Topics

- Define Lambda function handlers in Rust
- Using the Lambda context object to retrieve Rust function information
- Processing HTTP events with Rust
- Deploy Rust Lambda functions with .zip file archives
- Working with layers for Rust Lambda functions
- Log and monitor Rust Lambda functions

Define Lambda function handlers in Rust

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

This page describes how to work with Lambda function handlers in Rust, including project initialization, naming conventions, and best practices. This page also includes an example of a Rust Lambda function that takes in information about an order, produces a text file receipt, and puts this file in an Amazon Simple Storage Service (S3) bucket. For more information about how to deploy your function after writing it, see the section called "Deploy .zip file archives".

Topics

- Setting up your Rust handler project
- Example Rust Lambda function code
- Valid class definitions for Rust handlers
- Handler naming conventions
- Defining and accessing the input event object
- Accessing and using the Lambda context object
- Using the AWS SDK for Rust in your handler
- Accessing environment variables
- Using shared state
- Code best practices for Rust Lambda functions

Setting up your Rust handler project

When working with Lambda functions in Rust, the process involves writing your code, compiling it, and deploying the compiled artifacts to Lambda. The simplest way to set up a Lambda handler project in Rust is to use the AWS Lambda Runtime for Rust. Despite its name, the AWS Lambda Runtime for Rust is not a managed runtime in the same sense as it is in Lambda for Python, Java, or Node.js. Instead, the AWS Lambda Runtime for Rust is a crate (`lambda_runtime`) that supports writing Lambda functions in Rust and interfacing with AWS Lambda's execution environment.

Use the following command to install Cargo Lambda, a third-party open-source extension to the Cargo command-line tool that simplifies building and deploying Rust Lambda functions:

```
cargo install cargo-lambda
```

After you successfully install `cargo-lambda`, use the following command to initialize a new Rust Lambda function handler project:

```
cargo lambda new example-rust
```

When you run this command, the command line interface (CLI) asks you a couple of questions about your Lambda function:

- **HTTP function** – If you intend to invoke your function via API Gateway or a function URL, answer **Yes**. Otherwise, answer **No**. In the example code on this page, we invoke our function with a custom JSON event, so we answer **No**.
- **Event type** – If you intend to use a predefined event shape to invoke your function, select the correct expected event type. Otherwise, leave this option blank. In the example code on this page, we invoke our function with a custom JSON event, so we leave this option blank.

After the command runs successfully, enter the main directory of your project:

```
cd example-rust
```

This command generates a `generic_handler.rs` file and a `main.rs` file in the `src` directory. The `generic_handler.rs` can be used to customize a generic event handler. The `main.rs` file contains your main application logic. The `Cargo.toml` file contains metadata about your package and lists its external dependencies.

Example Rust Lambda function code

The following example Rust Lambda function code takes in information about an order, produces a text file receipt, and puts this file in an Amazon S3 bucket.

Example `main.rs` Lambda function

```
use aws_sdk_s3::{Client, primitives::ByteStream};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};
use serde_json::Value;
use std::env;
```

```
#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}

async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let payload = event.payload;

    // Deserialize the incoming event into Order struct
    let order: Order = serde_json::from_value(payload)?;

    let bucket_name = env::var("RECEIPT_BUCKET")
        .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;

    let receipt_content = format!(
        "OrderID: {}\nAmount: ${:.2}\nItem: {}",
        order.order_id, order.amount, order.item
    );
    let key = format!("receipts/{}.txt", order.order_id);

    let config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
    let s3_client = Client::new(&config);

    upload_receipt_to_s3(&s3_client, &bucket_name, &key, &receipt_content).await?;

    Ok("Success".to_string())
}

async fn upload_receipt_to_s3(
    client: &Client,
    bucket_name: &str,
    key: &str,
    content: &str,
) -> Result<(), Error> {
    client
        .put_object()
        .bucket(bucket_name)
        .key(key)
        .body(ByteStream::from(content.as_bytes().to_vec()))  // Fixed conversion
        .content_type("text/plain")
        .send()
```

```
        .await?;

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

This `main.rs` file contains the following sections of code:

- `use` statements: Use these to import Rust crates and methods that your Lambda function requires.
- `#[derive(Deserialize, Serialize)]`: Define the shape of the expected input event in this Rust struct.
- `async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>`: This is the **main handler method**, which contains your main application logic.
- `async fn upload_receipt_to_s3 (...)`: This is a helper method that's referenced by the main `function_handler` method.
- `#[tokio::main]`: This is a macro that marks the entry point of a Rust program. It also sets up a Tokio runtime, which allows your `main()` method to use `async/await` and run asynchronously.
- `async fn main() -> Result<(), Error>`: The `main()` function is the entry point of your code. Within it, we specify `function_handler` as the main handler method.

Sample Cargo.toml file

The following `Cargo.toml` file accompanies this function.

```
[package]
name = "example-rust"
version = "0.1.0"
edition = "2024"

[dependencies]
aws-config = "1.5.18"
aws-sdk-s3 = "1.78.0"
lambda_runtime = "0.13.0"
serde = { version = "1", features = ["derive"] }
```

```
serde_json = "1"
tokio = { version = "1", features = ["full"] }
```

For this function to work properly, its execution role must allow the `s3:PutObject` action. Also, ensure that you define the `RECEIPT_BUCKET` environment variable. After a successful invocation, the Amazon S3 bucket should contain a receipt file.

Valid class definitions for Rust handlers

In most cases, Lambda handler signatures that you define in Rust will have the following format:

```
async fn function_handler(event: LambdaEvent<T>) -> Result<U, Error>
```

For this handler:

- The name of this handler is `function_handler`.
- The singular input to the handler is event, and is of type `LambdaEvent<T>`.
  - `LambdaEvent` is a wrapper that comes from the `lambda_runtime` crate. Using this wrapper gives you access to the context object, which includes Lambda-specific metadata such as the request ID of the invocation.
  - T is the deserialized event type. For example, this can be `serde_json::Value`, which allows the handler to take in any generic JSON input. Alternatively, this can be a type like `ApiGatewayProxyRequest` if your function expects a specific, pre-defined input type.
- The return type of the handler is `Result<U, Error>`.
  - U is the deserialized output type. U must implement the `serde::Serialize` trait so Lambda can convert the return value to JSON. For example, U can be a simple type like `String`, `serde_json::Value`, or a custom struct as long as it implements `Serialize`. When your code reaches an Ok(U) statement, this indicates successful execution, and your function returns a value of type U.
  - When your code encounters an error (i.e. `Err(Error)`), your function logs the error in Amazon CloudWatch and returns an error response of type `Error`.

In our example, the handler signature looks like the following:

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>
```

Other valid handler signatures can feature the following:

- Omitting the `LambdaEvent` wrapper – If you omit `LambdaEvent`, you lose access to the Lambda context object within your function. The following is an example of this type of signature:

```
async fn handler(event: serde_json::Value) -> Result<String, Error>
```

- Using the unit type as an input – For Rust, you can use the unit type to represent an empty input. This is commonly used for functions with periodic, scheduled invocations. The following is an example of this type of signature:

```
async fn handler(_: ()) -> Result<Value, Error>
```

Handler naming conventions

Lambda handlers in Rust don't have strict naming restrictions. Although you can use any name for your handler, function names in Rust are generally in `snake_case`.

For smaller applications, such as in this example, you can use a single `main.rs` file to contain all of your code. For larger projects, `main.rs` should contain the entry point to your function, but you can have additional files for that separate your code into logical modules. For example, you might have the following file structure:

```
/example-rust
### src/
#   ### main.rs        # Entry point
#   ### handler.rs     # Contains main handler
#   ### services.rs    # [Optional] Back-end service calls
#   ### models.rs      # [Optional] Data models
### Cargo.toml
```

Defining and accessing the input event object

JSON is the most common and standard input format for Lambda functions. In this example, the function expects an input similar to the following:

```
{
    "order_id": "12345",
    "amount": 199.99,
```

```
    "item": "Wireless Headphones"
}
```

In Rust, you can define the shape of the expected input event in a struct. In this example, we define the following struct to represent an `Order`:

```
#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}
```

This struct matches the expected input shape. In this example, the `#[derive(Deserialize, Serialize)]` macro automatically generates code for serialization and deserialization.
This means that we can deserialize the generic input JSON type into our struct using the `serde_json::from_value()` method. This is illustrated in the first few lines of the handler:

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let payload = event.payload;

    // Deserialize the incoming event into Order struct
    let order: Order = serde_json::from_value(payload)?;
    ...
}
```

You can then access the fields of the object. For example, `order.order_id` retrieves the value of `order_id` from the original input.

Pre-defined input event types

There are many pre-defined input event types available in the `aws_lambda_events` crate. For example, if you intend to invoke your function with API Gateway, including the following import:

```
use aws_lambda_events::event::apigw::ApiGatewayProxyRequest;
```

Then, make sure your main handler uses the following signature:

```
async fn handler(event: LambdaEvent<ApiGatewayProxyRequest>) -> Result<String, Error> {
    let body = event.payload.body.unwrap_or_default();
```

```
    ...
}
```

Refer to the aws_lambda_events crate for more information about other pre-defined input event types.

Accessing and using the Lambda context object

The Lambda context object contains information about the invocation, function, and execution environment. In Rust, the `LambdaEvent` wrapper includes the context object. For example, you can use the context object to retrieve the request ID of the current invocation with the following code:

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let request_id = event.context.request_id;
    ...
}
```

For more information about the context object, see the section called “Context”.

Using the AWS SDK for Rust in your handler

Often, you'll use Lambda functions to interact with or make updates to other AWS resources. The simplest way to interface with these resources is to use the AWS SDK for Rust.

To add SDK dependencies to your function, add them in your `Cargo.toml` file. We recommend only adding the libraries that you need for your function. In the example code earlier, we used the `aws_sdk_s3::Client`. In the `Cargo.toml` file, you can add this dependency by adding the following line under the `[dependencies]` section:

```
aws-sdk-s3 = "1.78.0"
```

> **Note**
>
> This may not be the most recent version. Choose the appropriate version for your application.

The, import the dependencies directly in your code:

```
use aws_sdk_s3::{Client, primitives::ByteStream};
```

The example code then initializes an Amazon S3 client as follows:

```
let config = aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let s3_client = Client::new(&config);
```

After you initialize your SDK client, you can then use it to interact with other AWS services. The example code calls the Amazon S3 `PutObject` API in the `upload_receipt_to_s3` helper function.

Accessing environment variables

In your handler code, you can reference any environment variables by using the `env::var` method. In this example, we reference the defined `RECEIPT_BUCKET` environment variable using the following line of code:

```
let bucket_name = env::var("RECEIPT_BUCKET")
    .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;
```

Using shared state

You can declare shared variables that are independent of your Lambda function's handler code. These variables can help you load state information during the Init phase, before your function receives any events. For example, you can modify the code on this page to use shared state when initializing the Amazon S3 client by updating the `main` function and handler signature:

```
async fn function_handler(client: &Client, event: LambdaEvent<Value>) -> Result<String,
 Error> {
    ...
    upload_receipt_to_s3(client, &bucket_name, &key, &receipt_content).await?;
    ...
}

...

#[tokio::main]
async fn main() -> Result<(), Error> {
    let shared_config = aws_config::from_env().load().await;
```

```
    let client = Client::new(&shared_config);
    let shared_client = &client;
    lambda_runtime::run(service_fn(move |event: LambdaEvent<Request>| async move {
        handler(&shared_client, event).await
    }))
    .await
```

Code best practices for Rust Lambda functions

Adhere to the guidelines in the following list to use best coding practices when building your Lambda functions:

- **Separate the Lambda handler from your core logic.** This allows you to make a more unit-testable function.
- **Minimize the complexity of your dependencies.** Prefer simpler frameworks that load quickly on execution environment startup.
- **Minimize your deployment package size to its runtime necessities.** This will reduce the amount of time that it takes for your deployment package to be downloaded and unpacked ahead of invocation.

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of function invocations and escalated costs. If you see an unintended volume of invocations, set the function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

Using the Lambda context object to retrieve Rust function information

When Lambda runs your function, it adds a context object to the LambdaEvent that the handler receives. This object provides properties with information about the invocation, function, and execution environment.

Context properties

- `request_id`: The AWS request ID generated by the Lambda service.
- `deadline`: The execution deadline for the current invocation in milliseconds.
- `invoked_function_arn`: The Amazon Resource Name (ARN) of the Lambda function being invoked.
- `xray_trace_id`: The AWS X-Ray trace ID for the current invocation.
- `client_content`: The client context object sent by the AWS mobile SDK. This field is empty unless the function is invoked using an AWS mobile SDK.
- `identity`: The Amazon Cognito identity that invoked the function. This field is empty unless the invocation request to the Lambda APIs was made using AWS credentials issued by Amazon Cognito identity pools.
- `env_config`: The Lambda function configuration from the local environment variables. This property includes information such as the function name, memory allocation, version, and log streams.

Accessing invoke context information

Lambda functions have access to metadata about their environment and the invocation request. The `LambaEvent` object that your function handler receives includes the `context` metadata:

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let invoked_function_arn = event.context.invoked_function_arn;
    Ok(json!({ "message": format!("Hello, this is function
 {invoked_function_arn}!") }))
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Processing HTTP events with Rust

Amazon API Gateway APIs, Application Load Balancers, and Lambda function URLs can send HTTP events to Lambda. You can use the aws_lambda_events crate from crates.io to process events from these sources.

Example— Handle API Gateway proxy request

Note the following:

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`: The aws_lambda_events crate includes many Lambda events. To reduce compilation time, use feature flags to activate the events you need. Example: `aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`.
- `use http::HeaderMap`: This import requires you to add the http crate to your dependencies.

```
use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};
use http::HeaderMap;
use lambda_runtime::{service_fn, Error, LambdaEvent};

async fn handler(
    _event: LambdaEvent<ApiGatewayProxyRequest>,
) -> Result<ApiGatewayProxyResponse, Error> {
    let mut headers = HeaderMap::new();
    headers.insert("content-type", "text/html".parse().unwrap());
    let resp = ApiGatewayProxyResponse {
        status_code: 200,
        multi_value_headers: headers.clone(),
        is_base64_encoded: false,
        body: Some("Hello AWS Lambda HTTP request".into()),
        headers,
    };
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

The Rust runtime client for Lambda also provides an abstraction over these event types that allows you to work with native HTTP types, regardless of which service sends the events. The following code is equivalent to the previous example, and it works out of the box with Lambda function URLs, Application Load Balancers, and API Gateway.

> **Note**
>
> The lambda_http crate uses the lambda_runtime crate underneath. You don't have to import `lambda_runtime` separately.

Example— Handle HTTP requests

```
use lambda_http::{service_fn, Error, IntoResponse, Request, RequestExt, Response};

async fn handler(event: Request) -> Result<impl IntoResponse, Error> {
    let resp = Response::builder()
        .status(200)
        .header("content-type", "text/html")
        .body("Hello AWS Lambda HTTP request")
        .map_err(Box::new)?;
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_http::run(service_fn(handler)).await
}
```

For another example of how to use `lambda_http`, see the http-axum code sample on the AWS Labs GitHub repository.

Sample HTTP Lambda events for Rust

- Lambda HTTP events: A Rust function that handles HTTP events.
- Lambda HTTP events with CORS headers: A Rust function that uses Tower to inject CORS headers.
- Lambda HTTP events with shared resources: A Rust function that uses shared resources initialized before the function handler is created.

Deploy Rust Lambda functions with .zip file archives

This page describes how to compile your Rust function, and then deploy the compiled binary to AWS Lambda using Cargo Lambda. It also shows how to deploy the compiled binary with the AWS Command Line Interface and the AWS Serverless Application Model CLI.

Sections

- Prerequisites
- Building Rust functions on macOS, Windows, or Linux
- Deploying the Rust function binary with Cargo Lambda
- Invoking your Rust function with Cargo Lambda

Prerequisites

- Rust
- AWS CLI version 2

Building Rust functions on macOS, Windows, or Linux

The following steps demonstrate how to create the project for your first Lambda function with Rust and compile it with Cargo Lambda, a third-party open-source extension to the Cargo command-line tool that simplifies building and deploying Rust Lambda functions.

1. Install Cargo Lambda, a third-party open-source extension to the Cargo command-line tool that simplifies building and deploying Rust Lambda functions:

   ```
   cargo install cargo-lambda
   ```

   For other installation options, see Installation in the Cargo Lambda documentation.
2. Create the package structure. This command creates some basic function code in `src/main.rs`. You can use this code for testing or replace it with your own.

   ```
   cargo lambda new my-function
   ```
3. Inside the package's root directory, run the build subcommand to compile the code in your function.

```
cargo lambda build --release
```

(Optional) If you want to use AWS Graviton2 on Lambda, add the `--arm64` flag to compile your code for ARM CPUs.

```
cargo lambda build --release --arm64
```

4. Before deploying your Rust function, configure AWS credentials on your machine.

```
aws configure
```

Deploying the Rust function binary with Cargo Lambda

Use the deploy subcommand to deploy the compiled binary to Lambda. This command creates an execution role and then creates the Lambda function. To specify an existing execution role, use the --iam-role flag.

```
cargo lambda deploy my-function
```

Deploying your Rust function binary with the AWS CLI

You can also deploy your binary with the AWS CLI.

1. Use the build subcommand to build the .zip deployment package.

```
cargo lambda build --release --output-format zip
```

2. To deploy the .zip package to Lambda, run the create-function command.

   - For `--runtime`, specify `provided.al2023`. This is an OS-only runtime. OS-only runtimes are used to deploy compiled binaries and custom runtimes to Lambda.
   - For `--role`, specify the ARN of the execution role.

```
aws lambda create-function \
     --function-name my-function \
     --runtime provided.al2023 \
     --role arn:aws:iam::111122223333:role/lambda-role \
```

```
    --handler rust.handler \
    --zip-file fileb://target/lambda/my-function/bootstrap.zip
```

Deploying your Rust function binary with the AWS SAM CLI

You can also deploy your binary with the AWS SAM CLI.

1. Create an AWS SAM template with the resource and property definition. For `Runtime`, specify `provided.al2023`. This is an OS-only runtime. OS-only runtimes are used to deploy compiled binaries and custom runtimes to Lambda.

   For more information about deploying Lambda functions using AWS SAM, see AWS::Serverless::Function in the *AWS Serverless Application Model Developer Guide*.

   Example SAM resource and property definition for a Rust binary

   ```
   AWSTemplateFormatVersion: '2010-09-09'
   Transform: AWS::Serverless-2016-10-31
   Description: SAM template for Rust binaries
   Resources:
     RustFunction:
       Type: AWS::Serverless::Function
       Properties:
         CodeUri: target/lambda/my-function/
         Handler: rust.handler
         Runtime: provided.al2023
   Outputs:
     RustFunction:
       Description: "Lambda Function ARN"
       Value: !GetAtt RustFunction.Arn
   ```

2. Use the build subcommand to compile the function.

   ```
   cargo lambda build --release
   ```

3. Use the sam deploy command to deploy the function to Lambda.

   ```
   sam deploy --guided
   ```

For more information about building Rust functions with the AWS SAM CLI, see Building Rust Lambda functions with Cargo Lambda in the *AWS Serverless Application Model Developer Guide*.

Invoking your Rust function with Cargo Lambda

Use the invoke subcommand to test your function with a payload.

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

Invoking your Rust function with the AWS CLI

You can also use the AWS CLI to invoke the function.

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

Working with layers for Rust Lambda functions

We don't recommend using layers to manage dependencies for Lambda functions written in Rust. This is because Lambda functions in Rust compile into a single executable, which you provide to Lambda when you deploy your function. This executable contains your compiled function code, along with all of its dependencies. Using layers not only complicates this process, but also leads to increased cold start times because your functions need to manually load extra assemblies into memory during the init phase.

To use external dependencies with your Rust handlers, include them directly in your deployment package. By doing so, you simplify the deployment process and also take advantage of built-in Rust compiler optimizations. For an example of how to import and use a dependency like the AWS SDK for Rust in your function, see the section called “Handler”.

Log and monitor Rust Lambda functions

AWS Lambda automatically monitors Lambda functions on your behalf and sends logs to Amazon CloudWatch. Your Lambda function comes with a CloudWatch Logs log group and a log stream for each instance of your function. The Lambda runtime environment sends details about each invocation to the log stream, and relays logs and other output from your function's code. For more information, see Sending Lambda function logs to CloudWatch Logs. For information about configuring log formats, see Configuring JSON and plain text log formats. This page describes how to produce log output from your Lambda function's code.

Creating a function that writes logs

To output logs from your function code, you can use any logging function that writes to `stdout` or `stderr`, such as the `println!` macro. The following example uses `println!` to print a message when the function handler starts and before it finishes.

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    println!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    println!("Rust function responds to {}", &first_name);
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Implementing advanced logging with the Tracing crate

Tracing is a framework for instrumenting Rust programs to collect structured, event-based diagnostic information. This framework provides utilities to customize logging output levels and formats, like creating structured JSON log messages. To use this framework, you must initialize a `subscriber` before implementing the function handler. Then, you can use tracing macros like debug, `info`, and `error`, to specify the level of logging that you want for each scenario.

Example— Using the Tracing crate

Note the following:

- `tracing_subscriber::fmt().json()`: When this option is included, logs are formatted in JSON. To use this option, you must include the `json` feature in the `tracing-subscriber` dependency (for example,`tracing-subscriber = { version = "0.3.11", features = ["json"] }`).
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]`: This annotation generates a span every time the handler is invoked. The span adds the request ID to each log line.
- `{ %first_name }`: This construct adds the `first_name` field to the log line where it's used. The value for this field corresponds to the variable with the same name.

```
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    tracing::info!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    tracing::info!({ %first_name }, "Rust function responds to event");
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt().json()
        .with_max_level(tracing::Level::INFO)
        // this needs to be set to remove duplicated information in the log.
        .with_current_span(false)
        // this needs to be set to false, otherwise ANSI color codes will
        // show up in a confusing manner in CloudWatch logs.
        .with_ansi(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
        .without_time()
        // remove the name of the function from every log entry
        .with_target(false)
        .init();
    lambda_runtime::run(service_fn(handler)).await
```

```
}
```

When this Rust function is invoked, it prints two log lines similar to the following:

```
{"level":"INFO","fields":{"message":"Rust function invoked"},"spans":
[{"req_id":"45daaaa7-1a72-470c-9a62-e79860044bb5","name":"handler"}]}
{"level":"INFO","fields":{"message":"Rust function responds to
 event","first_name":"David"},"spans":[{"req_id":"45daaaa7-1a72-470c-9a62-
e79860044bb5","name":"handler"}]}
```

Best practices for working with AWS Lambda functions

The following are recommended best practices for using AWS Lambda:

Topics

- Function code
- Function configuration
- Function scalability
- Metrics and alarms
- Working with streams
- Security best practices

Function code

Take advantage of execution environment reuse to improve the performance of your function. Initialize SDK clients and database connections outside of the function handler, and cache static assets locally in the `/tmp` directory. Subsequent invocations processed by the same instance of your function can reuse these resources. This saves cost by reducing function run time.

To avoid potential data leaks across invocations, don't use the execution environment to store user data, events, or other information with security implications. If your function relies on a mutable state that can't be stored in memory within the handler, consider creating a separate function or separate versions of a function for each user.

Use a keep-alive directive to maintain persistent connections. Lambda purges idle connections over time. Attempting to reuse an idle connection when invoking a function will result in a connection error. To maintain your persistent connection, use the keep-alive directive associated with your runtime. For an example, see Reusing Connections with Keep-Alive in Node.js.

Use environment variables to pass operational parameters to your function. For example, if you are writing to an Amazon S3 bucket, instead of hard-coding the bucket name you are writing to, configure the bucket name as an environment variable.

Avoid using recursive invocations in your Lambda function, where the function invokes itself or initiates a process that may invoke the function again. This could lead to unintended volume of function invocations and escalated costs. If you see an unintended volume of invocations, set the

function reserved concurrency to `0` immediately to throttle all invocations to the function, while you update the code.

Do not use non-documented, non-public APIs in your Lambda function code. For AWS Lambda managed runtimes, Lambda periodically applies security and functional updates to Lambda's internal APIs. These internal API updates may be backwards-incompatible, leading to unintended consequences such as invocation failures if your function has a dependency on these non-public APIs. See the API reference for a list of publicly available APIs.

Write idempotent code. Writing idempotent code for your functions ensures that duplicate events are handled the same way. Your code should properly validate events and gracefully handle duplicate events. For more information, see How do I make my Lambda function idempotent?.

> **Note**
>
> You can use Powertools for AWS Lambda to make functions idempotent. For more information, see:
>
> - Python - Idempotency utility
> - TypeScript - Idempotency utility
> - Java - Idempotency utility
> - .NET - Idempotency utility

For language-specific code best practices, refer to the following sections:

- Code best practices for Node.js Lambda functions
- Code best practices for TypeScript Lambda functions
- Code best practices for Python Lambda functions
- Code best practices for Ruby Lambda functions
- Code best practices for Java Lambda functions
- Code best practices for Go Lambda functions
- Code best practices for C# Lambda functions
- Code best practices for Rust Lambda functions

Function configuration

Performance testing your Lambda function is a crucial part in ensuring you pick the optimum memory size configuration. Any increase in memory size triggers an equivalent increase in CPU available to your function. The memory usage for your function is determined per-invoke and can be viewed in Amazon CloudWatch. On each invoke a `REPORT:` entry will be made, as shown below:

```
REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed
 Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
```

By analyzing the `Max Memory Used:` field, you can determine if your function needs more memory or if you over-provisioned your function's memory size.

To find the right memory configuration for your functions, we recommend using the open source AWS Lambda Power Tuning project. For more information, see AWS Lambda Power Tuning on GitHub.

To optimize function performance, we also recommend deploying libraries that can leverage Advanced Vector Extensions 2 (AVX2). This allows you to process demanding workloads, including machine learning inferencing, media processing, high performance computing (HPC), scientific simulations, and financial modeling. For more information, see Creating faster AWS Lambda functions with AVX2.

Load test your Lambda function to determine an optimum timeout value. It is important to analyze how long your function runs so that you can better determine any problems with a dependency service that may increase the concurrency of the function beyond what you expect. This is especially important when your Lambda function makes network calls to resources that may not handle Lambda's scaling. For more information about load testing your application, see Distributed Load Testing on AWS.

Use most-restrictive permissions when setting IAM policies. Understand the resources and operations your Lambda function needs, and limit the execution role to these permissions. For more information, see Managing permissions in AWS Lambda.

Be familiar with Lambda quotas. Payload size, file descriptors and /tmp space are often overlooked when determining runtime resource limits.

Delete Lambda functions that you are no longer using. By doing so, the unused functions won't needlessly count against your deployment package size limit.

If you are using Amazon Simple Queue Service as an event source, make sure the value of the function's expected invocation time does not exceed the Visibility Timeout value on the queue. This applies both to CreateFunction and UpdateFunctionConfiguration.

- In the case of **CreateFunction**, AWS Lambda will fail the function creation process.
- In the case of **UpdateFunctionConfiguration**, it could result in duplicate invocations of the function.

Function scalability

Be familiar with your upstream and downstream throughput constraints. While Lambda functions scale seamlessly with load, upstream and downstream dependencies may not have the same throughput capabilities. If you need to limit how high your function can scale, you can configure reserved concurrency on your function.

Build in throttle tolerance. If your synchronous function experiences throttling due to traffic exceeding Lambda's scaling rate, you can use the following strategies to improve throttle tolerance:

- **Use timeouts, retries, and backoff with jitter**. Implementing these strategies smooth out retried invocations, and helps ensure Lambda can scale up within seconds to minimize end-user throttling.
- **Use provisioned concurrency**. Provisioned concurrency is the number of pre-initialized execution environments that Lambda allocates to your function. Lambda handles incoming requests using provisioned concurrency when available. Lambda can also scale your function above and beyond your provisioned concurrency setting if required. Configuring provisioned concurrency incurs additional charges to your AWS account.

Metrics and alarms

Use Using CloudWatch metrics with Lambda and CloudWatch Alarms instead of creating or updating a metric from within your Lambda function code. It's a much more efficient way to track the health of your Lambda functions, allowing you to catch issues early in the development process. For instance, you can configure an alarm based on the expected duration of your Lambda function invocation in order to address any bottlenecks or latencies attributable to your function code.

Emit custom metrics asynchronously using Embedded Metric Format (EMF). Instead of making synchronous API calls to CloudWatch, use EMF to emit metrics through your function's logs. This approach reduces latency and improves performance. The Metrics utility in Powertools for AWS Lambda handles EMF formatting automatically. For more information, see Python, TypeScript, Java, or .NET Metrics utilities in the Powertools for AWS Lambda documentation. For information on using EMF to generate metric format logs, see Publishing logs with the embedded metric format in the Amazon CloudWatch User Guide.

Use structured JSON logging for better observability. Structured logging makes it easier to search, filter, and analyze your function's logs. Consider using the Logger utility from Powertools for AWS Lambda to automatically format logs in JSON. For more information, see Python, TypeScript, Java, or .NET Logger utilities in the Powertools for AWS Lambda documentation.

Leverage your logging library and AWS Lambda Metrics and Dimensions to catch app errors (e.g. ERR, ERROR, WARNING, etc.)

Use AWS Cost Anomaly Detection to detect unusual activity on your account. Cost Anomaly Detection uses machine learning to continuously monitor your cost and usage while minimizing false positive alerts. Cost Anomaly Detection uses data from AWS Cost Explorer, which has a delay of up to 24 hours. As a result, it can take up to 24 hours to detect an anomaly after usage occurs. To get started with Cost Anomaly Detection, you must first sign up for Cost Explorer. Then, access Cost Anomaly Detection.

Working with streams

Test with different batch and record sizes so that the polling frequency of each event source is tuned to how quickly your function is able to complete its task. The CreateEventSourceMapping BatchSize parameter controls the maximum number of records that can be sent to your function with each invoke. A larger batch size can often more efficiently absorb the invoke overhead across a larger set of records, increasing your throughput.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a *batching window*. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see Batching behavior.

> **Warning**
>
> Lambda event source mappings process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

Enable partial batch response for stream processing. When processing batches of records from streams like Kinesis or DynamoDB Streams, enable partial batch response to allow Lambda to retry only the failed records instead of the entire batch. This improves processing efficiency and reduces unnecessary reprocessing. You can optionally use the Batch utility from Powertools for AWS Lambda to simplify batch processing patterns.

> **Note**
>
> You can use Powertools for AWS Lambda for batch processing. For more information, see:
>
> - Python - Batch Processing
> - TypeScript - Batch Processing
> - Java - Batch Processing
> - .NET - Batch Processing

Increase Kinesis stream processing throughput by adding shards. A Kinesis stream is composed of one or more shards. The rate at which Lambda can read data from Kinesis scales linearly with the number of shards. Increasing the number of shards will directly increase the number of maximum concurrent Lambda function invocations and can increase your Kinesis stream processing throughput. For more information about the relationship between shards and function invocations, see the section called “Polling and batching streams”. If you are increasing the number of shards in a Kinesis stream, make sure you have picked a good partition key (see Partition Keys) for your data, so that related records end up on the same shards and your data is well distributed.

Use Amazon CloudWatch on IteratorAge to determine if your Kinesis stream is being processed. For example, configure a CloudWatch alarm with a maximum setting to 30000 (30 seconds).

Security best practices

Monitor your usage of AWS Lambda as it relates to security best practices by using AWS Security Hub CSPM. Security Hub CSPM uses security controls to evaluate resource configurations and security standards to help you comply with various compliance frameworks. For more information about using Security Hub CSPM to evaluate Lambda resources, see AWS Lambda controls in the AWS Security Hub CSPM User Guide.

Monitor Lambda network activity logs using Amazon GuardDuty Lambda Protection. GuardDuty Lambda protection helps you identify potential security threats when Lambda functions are invoked in your AWS account. For example, if one of your functions queries an IP address that is associated with cryptocurrency-related activity. GuardDuty monitors the network activity logs that are generated when a Lambda function is invoked. To learn more, see Lambda protection in the *Amazon GuardDuty User Guide*.

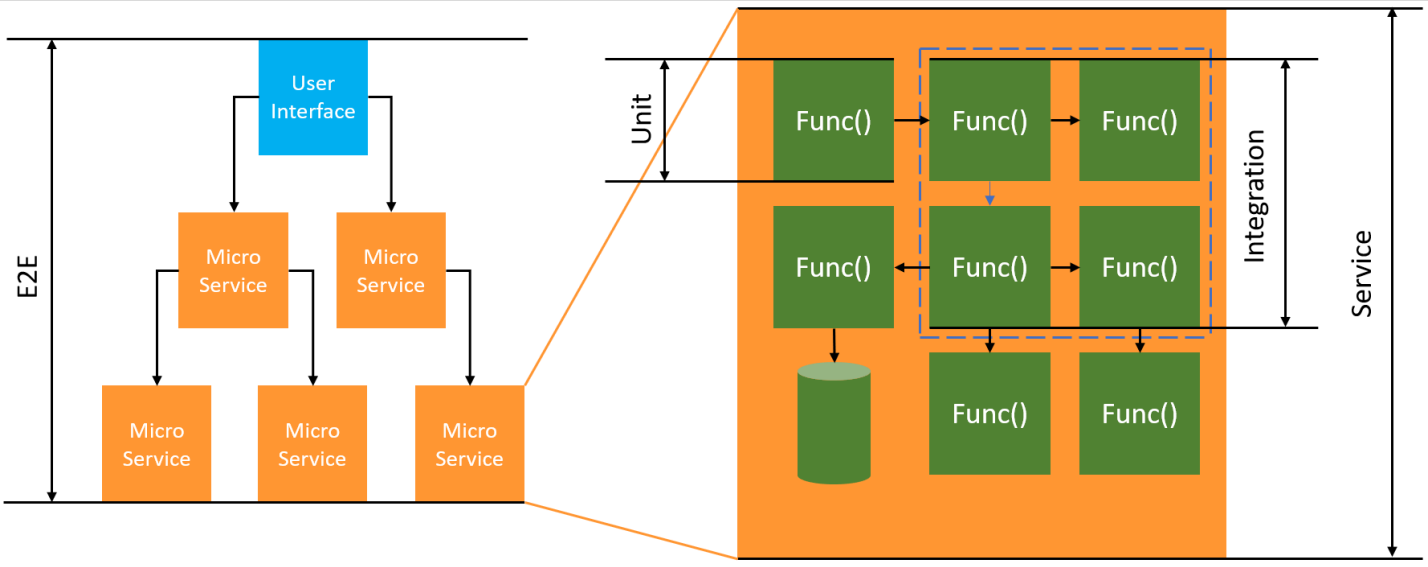
How to test serverless functions and applications

Testing serverless functions uses traditional test types and techniques, but you must also consider testing serverless applications as a whole. Cloud-based tests will provide the **most accurate** measure of quality of both your functions and serverless applications.

A serverless application architecture includes managed services that provide critical application functionality through API calls. For this reason, your development cycle should include automated tests that verify functionality when your function and services interact.

If you do not create cloud-based tests, you could encounter issues due to differences between your local environment and the deployed environment. Your continuous integration process should run tests against a suite of resources provisioned in the cloud before promoting your code to the next deployment environment, such as QA, Staging, or Production.

Continue reading this short guide to learn about testing strategies for serverless applications, or visit the [Serverless Test Samples repository] to dive in with practical examples, specific to your chosen language and runtime.



For serverless testing, you will still write *unit*, *integration* and *end-to-end* tests.

- **Unit tests** - Tests that run against an isolated block of code. For example, verifying the business logic to calculate the delivery charge given a particular item and destination.
- **Integration tests** - Tests involving two or more components or services that interact, typically in a cloud environment. For example, verifying a function processes events from a queue.

- **End-to-end tests** - Tests that verify behavior across an entire application. For example, ensuring infrastructure is set up correctly and that events flow between services as expected to record a customer's order.

Targeted business outcomes

Testing serverless solutions may require slightly more time to set up tests that verify event-driven interactions between services. Keep the following practical business reasons in mind as you read this guide:

- Increase the quality of your application
- Decrease time to build features and fix bugs

The quality of an application depends on testing a variety of scenarios to verify functionality. Carefully considering the business scenarios and automating those tests to run against cloud services will raise the quality of your application.

Software bugs and configuration problems have the least impact on cost and schedule when caught during an iterative development cycle. If issues remain undetected during development, finding and fixing in production requires more effort by more people.

A well planned serverless testing strategy will increase software quality and improve iteration time by verifying your Lambda functions and applications perform as expected in a cloud environment.

What to test

We recommend adopting a testing strategy that tests managed service *behaviors*, cloud configuration, security policies, and the integration with your code to improve software quality. *Behavior testing*, also known as black box testing, verifies a system works as expected without knowing all the internals.

- Run unit tests to check business logic inside Lambda functions.
- Verify integrated services are actually invoked, and input parameters are correct.
- Check that an event goes through all expected services end-to-end in a workflow.

In traditional server-based architecture, teams often define a scope for testing to only include code that runs on the application server. Other components, services, or dependencies are often considered external and out of scope for testing.

Serverless applications often consist of small units of work, such as Lambda functions that retrieve products from a database, or process items from a queue, or resize an image in storage. Each component runs in their own environment. Teams will likely be responsible for many of these small units within a single application.

Some application functionality can be delegated entirely to managed services such as Amazon S3, or created without using any internally developed code. There is no need to test these managed services, but you do need to test the integration with these services.

How to test serverless

You are probably familiar with how to test applications deployed locally: You write tests that run against code running entirely on your desktop operating system, or inside containers. For example, you might invoke a local web service component with a request and then make assertions about the response.

Serverless solutions are built from your function code and cloud-based managed services, such as queues, databases, event buses, and messaging systems. These components are all connected through an *event-driven architecture*, where messages, called *events*, flow from one resource to another. These interactions can be synchronous, such as when a web service returns results immediately, or an asynchronous action which completes at a later time, such as placing items in a queue or starting a workflow step. Your testing strategy must include both scenarios and test the interactions between services. For asynchronous interactions, you may need to detect side effects in downstream components that may not be immediately observable.

Replicating an entire cloud environment, including queues, database tables, event buses, security policies, and more, is not practical. You will inevitably encounter issues due to differences between your local environment and your deployed environments in the cloud. The variations between your environments will increase the time to reproduce and fix bugs.

In serverless applications, architecture components commonly exist entirely in the cloud, so testing against code and services in the cloud is necessary to develop features and fix bugs.

Testing techniques

In reality, your testing strategy will likely include a mix of techniques to increase quality of your solutions. You will use quick interactive tests to debug functions in the console, automated unit tests to check isolated business logic, verification of calls to external services with mocks, and occasional testing against emulators that mimic a service.

- the section called "Testing in the cloud": You deploy infrastructure and code to test with actual services, security policies, configurations and infrastructure specific parameters. Cloud-based tests provide the **most accurate** measure of quality of your code.

  Debugging a function in the console is a quick way to test in the cloud. You can choose from a library of sample test events or create a custom event to test a function in isolation. You can also share test events through the console with your team.

  To **automate** testing in the development and build lifecycle, you will need to test outside of the console. See the language specific testing sections in this guide for automation strategies and resources.
- the section called "Testing with mocks": Mocks are objects within your code that simulate and stand-in for an external service. Mocks provide pre-defined behavior to verify service calls and parameters. A *fake* is a mock implementation that takes shortcuts to simplify or improve performance. For example, a fake data access object might return data from an in-memory datastore. Mocks can mimic and simplify complex dependencies, but can also lead to more mocks in order to replace nested dependencies.
- the section called "Testing locally using AWS SAM CLI": Use AWS SAM CLI to locally invoke Lambda functions in Docker containers that use the same runtime environment as AWS Lambda. You can test function logic and event processing without deploying to the cloud.
- the section called "Testing with emulation": Use the LocalStack integration in VS Code to emulate multiple AWS services locally for testing service integrations.

Testing in the cloud

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. When you run tests against cloud-based code that also interacts with cloud-based services, you get the **most accurate** measure of quality of your code.

A convenient way to run a Lambda function in the cloud is with a test event in the AWS Management Console. A *test event* is a JSON input to your function. If your function does not

require input, the event can be an empty JSON document (`{}`). The console provides sample events for a variety of service integrations. After creating an event in the console, you can also share it with your team to make testing easier and consistent.

Learn how to debug a sample function in the console.

> **Note**
>
> Although running functions in the console is a quick way to debug, **automating** your test cycles is essential to increase application quality and development speed.

Test automation samples are available in the Serverless Test Samples repository. The following command line runs an automated Python integration test example:

```
python -m pytest -s tests/integration -v
```

Although the test runs locally, it interacts with cloud-based resources. These resources have been deployed using the AWS Serverless Application Model and AWS SAM command line tool. The test code first retrieves the deployed stack outputs, which includes the API endpoint, function ARN, and security role. Next, the test sends a request to the API endpoint, which responds with a list of Amazon S3 buckets. This test runs entirely against cloud-based resources to verify those resources are deployed, secured, and work as expected.

```
========================= test session starts ==========================
      platform darwin -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0
      -- /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda/
venv/bin/python
      cachedir: .pytest_cache
      rootdir: /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-
lambda
      plugins: mock-3.10.0
      collected 1 item


      tests/integration/test_api_gateway.py::TestApiGateway::test_api_gateway

      --> Stack outputs:

        HelloWorldApi
        = https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
```

```
        > API Gateway endpoint URL for Prod stage for Hello World function

        PythonTestDemo
        = arn:aws:lambda:us-east-2:123456789012:function:testing-apigw-lambda-
PythonTestDemo-iSij8evaTdxl
        > Hello World Lambda Function ARN

        PythonTestDemoIamRole
        = arn:aws:iam::123456789012:role/testing-apigw-lambda-PythonTestDemoRole-
IZELQQ9MG4HQ
        > Implicit IAM Role created for Hello World function

      --> Found API endpoint for "testing-apigw-lambda" stack...
      --> https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
      API Gateway response:
      amplify-dev-123456789-deployment|myapp-prod-p-loggingbucket-123456|s3-java-
bucket-123456789
      PASSED

      ========================= 1 passed in 1.53s =========================
```

For cloud-native application development, testing in the cloud provides the following benefits:

- You can test **every** available service.
- You are always using the most recent service APIs and return values.
- A cloud test environment closely resembles your production environment.
- Tests can cover security policies, service quotas, configurations and infrastructure specific parameters.
- Every developer can quickly create one or more testing environments in the cloud.
- Cloud tests increase confidence your code will run correctly in production.

Testing in the cloud does have some disadvantages. The most obvious negative of testing in the cloud is that deployments to cloud environments typically take longer than deployments to a local desktop environments.

Fortunately, tools such as AWS Serverless Application Model (AWS SAM) Accelerate, AWS Cloud Development Kit (AWS CDK) watch mode, and SST (3rd party) reduce the latency involved with cloud deployment iterations. These tools can monitor your infrastructure and code and automatically deploy incremental updates into your cloud environment.

> **Note**
>
> See how to create infrastructure as code in the *Serverless Developer Guide* to learn more about AWS Serverless Application Model, CloudFormation, and AWS Cloud Development Kit (AWS CDK).

Unlike local testing, testing in the cloud requires additional resources which may incur service costs. Creating isolated testing environments may increase the burden on your DevOps teams, especially in organizations with strict controls around accounts and infrastructure. Even so, when working with complex infrastructure scenarios, the cost in developer time to set up and maintain an intricate local environment could be similar (or more costly) than using disposable testing environments created with Infrastructure as Code automation tools.

Testing in the cloud, even with these considerations, is still the **best way** to guarantee the quality of your serverless solutions.

Testing with mocks

Testing with mocks is a technique where you create replacement objects in your code to simulate the behavior of a cloud service.

For example, you could write a test that uses a mock of the Amazon S3 service that returns a specific response whenever the **CreateObject** method is called. When a test runs, the mock returns that programmed response without calling Amazon S3, or any other service endpoints.

Mock objects are often generated by a mock framework to reduce development effort. Some mock frameworks are generic and others are designed specifically for AWS SDKs, such as Moto, a Python library for mocking AWS services and resources.

Note that mock objects differ from emulators in that mocks are typically created or configured by a developer as part of the test code, whereas emulators are standalone applications that expose functionality in the same manner as the systems they emulate.

The advantages of using mocks include the following:

- Mocks can simulate third-party services that are beyond the control of your application, such as APIs and software as a service (SaaS) providers, without needing direct access to those services.
- Mocks are useful for testing failure conditions, especially when such conditions are hard to simulate, like a service outage.

- Mock can provide fast local testing once configured.
- Mocks can provide substitute behavior for virtually any kind of object, so mocking strategies can create coverage for a wider variety of services than emulators.
- When new features or behaviors become available, mock testing can react more quickly. By using a generic mock framework, you can simulate new features as soon as the updated AWS SDK become available.

Mock testing has these disadvantages:

- Mocks generally require a non-trivial amount of setup and configuration effort, specifically when trying to determine return values from different services in order to properly mock responses.
- Mocks are written, configured, and must be maintained by developers, increasing their responsibilities.
- You might need to have access to the cloud in order to understand the APIs and return values of services.
- Mocks can be difficult to maintain. When mocked cloud API signatures change, or return value schemas evolve, you need to update your mocks. Mocks also require updates if you extend your application logic to make calls to new APIs.
- Tests that use mocks might pass in desktop environments but fail in the cloud. Results may not match the current API. Service configuration and quotas cannot be tested.
- Mock frameworks are limited in testing or detecting AWS Identity and Access Management (IAM) policy or quota limitations. Although mocks are better at simulating when authorization fails or a quota is exceeded, testing cannot determine which outcome will actually occur in a production environment.

Testing locally using AWS SAM CLI

Use AWS SAM CLI to test your functions in Docker containers using the same runtime environment as AWS Lambda. You can test function logic and event processing locally without deploying to the cloud. If your function makes API calls to other AWS services, those calls will reach real AWS resources.

The advantages of testing with local containers include the following:

- Uses AWS Lambda runtime environments for accurate testing.
- Enables fast local development iterations without cloud deployment.
- Supports debugging with familiar local development tools.

Testing with local containers has these limitations:

- AWS service calls from your function will interact with real AWS resources, which may incur costs and affect production data.
- Requires Docker to be installed and running locally.

Testing with emulation

Emulators are locally running applications that mimic AWS services by providing similar APIs and return values. LocalStack is a popular emulation tool that provides a complete local development environment for testing service integrations.

LocalStack is an AWS Cloud emulator that you can use to test serverless applications locally. You can test Lambda functions that integrate with services like DynamoDB, Amazon S3, and Amazon SQS without connecting to actual AWS services. You can use LocalStack in the AWS Toolkit for VS Code.

The advantages of test with emulators include the following:

- Emulators can facilitate fast local development iterations and testing.
- Emulators provide a familiar environment for developers used to developing code in a local environment. For example, if you're familiar with the development of an *n*-tier application, you might have a database engine and web server, similar to those running in production, running on your local machine to provide quick, local, isolated test capability.
- Emulators do not require any changes to cloud infrastructure (such as developer cloud accounts), so it's easy to implement with existing testing patterns.
- Because emulators don't use actual AWS resources, you won't get unexpected charges when starting multiple services or for letting some resources run for extended periods of time.

Testing with emulators has these disadvantages:

- Emulators can be difficult to set up and replicate, especially when used in CI/CD pipelines. This can increase the workload of IT staff or developers who manage their own software.
- Emulated features and APIs typically lag behind service updates. This can lead to errors because tested code does not match the actual API, and impede the adoption of new features.
- Emulators require support, updates, bug fixes, and feature parity enhancements. These are the responsibility of the emulator author, which could be a third-party company.

- Tests that rely on emulators may provide successful results locally, but fail in the cloud due to production security policies, inter-service configurations, or exceeding Lambda quotas.

Best practices

The following sections provide recommendations for successful serverless application testing.

You can find practical examples of tests and test automation in the Serverless Test Samples repository.

Prioritize testing in the cloud

Testing in the cloud provides the most reliable, accurate, and complete test coverage. Performing tests in the context of the cloud will comprehensively test not only business logic but also security policies, service configurations, quotas, and the most up to date API signatures and return values.

Structure your code for testability

Simplify your tests and Lambda functions by separating Lambda-specific code from your core business logic.

Your Lambda function *handler* should be a slim adapter that takes in event data and passes only the details that matter to your business logic method(s). With this strategy, you can wrap comprehensive tests around your business logic without worrying about Lambda-specific details. Your AWS Lambda functions should not require setting up a complex environment or large amount of dependencies to create and initialize the component under test.

Generally speaking, you should write a handler that extracts and validates data from the incoming *event* and *context* objects, then sends that input to methods that perform your business logic.

Accelerate development feedback loops

There are tools and techniques to accelerate development feedback loops. For example, AWS SAM Accelerate and AWS CDK watch mode both decrease the time required to update cloud environments.

The samples in the GitHub Serverless Test Samples repository explore some of these techniques.

We also recommend that you create and test cloud resources as early as possible during development—not only after a check-in to source control. This practice enables quicker exploration

and experimentation when developing solutions. In addition, automating deployment from a development machine helps you discover cloud configuration problems more quickly and reduces wasted effort for updates and code review processes.

Focus on integration tests

When building applications with Lambda, testing components together is a best practice.

Tests that run against two or more architectural components are called *integration tests*. The goal of integration tests is to understand not only how your code will execute across components, but how the environment hosting your code will behave. *End-to-end tests* are special types of integration tests that verify behaviors across an entire application.

To build integration tests, deploy your application to a cloud environment. This can be done from a local environment or through a CI/CD pipeline. Then, write tests to exercise the system under test (SUT) and validate expected behavior.

For example, the system under test could be an application that uses API Gateway, Lambda and DynamoDB. A test could make a synthetic HTTP call to an API Gateway endpoint and validate that the response included the expected payload. This test validates that the AWS Lambda code is correct, and that each service is correctly configured to handle the request, including the IAM permissions between them. Further, you could design the test to write records of various sizes to verify your service quotas, such as max record size in DynamoDB, are set up correctly.



Create isolated test environments

Testing in the cloud typically requires isolated developer environments, so that tests, data, and events do not overlap.

One approach is to provide each developer a dedicated AWS account. This will avoid conflicts with resource naming that can occur when multiple developers working in a shared code base, attempt to deploy resources or invoke an API.

Automated test processes should create uniquely named resources for each stack. For example, you can set up scripts or TOML configuration files so that AWS SAM CLI sam deploy or sam sync commands will automatically specify a stack with a unique prefix.

In some cases, developers share an AWS account. This may be due to having resources in your stack that are expensive to operate, or to provision and configure. For example, a database may be shared to make it easier to set up and seed the data properly

If developers share an account, you should set boundaries to identify ownership and eliminate overlap. One way to do this is by prefixing stack names with developer user IDs. Another popular approach is to set up stacks based on **code branches**. With branch boundaries, environments are isolated, but developers can still share resources, such as a relational database. This approach is a best practice when developers work on more than one branch at a time.

Testing in the cloud is valuable for all phases of testing, including unit tests, integration tests, and end-to-end tests. Maintaining proper isolation is essential; but you still want your QA environment to resemble your production environment as closely as possible. For this reason, teams add change control processes for QA environments.

For pre-production and production environments, boundaries are typically drawn at the account level to insulate workloads from noisy neighbor problems and implement least privilege security controls to protect sensitive data. Workloads have quotas. You don't want your testing to consume quotas allocated for production (noisy neighbor) or have access to customer data. Load testing is another activity you should isolate from your production stack.

In all cases, environments should be configured with alerts and controls to avoid unnecessary spending. For example, you can limit the type, tier, or size of resources that can be created, and set up email alerts when estimated costs exceed a given threshold.

Use mocks for isolated business logic

Mock frameworks are a valuable tool for writing fast unit tests. They are especially beneficial when tests cover complex internal business logic, such as mathematical or financial calculations or simulations. Look for unit tests that have a large number of test cases or input variations, where those inputs do not change the pattern or the content of calls to other cloud services.

Code that is covered by unit tests with mocks should also be covered by testing in the cloud. This is recommended because a developer laptop or build machine environment could be configured differently than a production environment in the cloud. For example, your Lambda functions could use more memory or time than allocated when run with certain input parameters. Or your code

might include environment variables that are not configured in the same way (or at all), and the differences could cause the code to behave differently or fail.

The benefit of mocks is less for integration tests, because the level of effort to implement the necessary mocks increases with the number of connection points. End-to-end testing should not use mocks, because these tests generally deal with states and complex logic that cannot be easily simulated with mock frameworks.

Lastly, avoid using mocked cloud services to validate the proper implementation of service calls. Instead, make cloud service calls in the cloud to validate behavior, configuration, and functional implementation.

Use emulators sparingly

Emulators can be convenient for some use cases, for example, for a development team with limited, unreliable, or slow internet access. But, in most circumstances, choose to use emulators sparingly.

By avoiding emulators, you will be able to build and innovate with the latest service features and up to date APIs. You will not be stuck waiting on vendor releases to achieve feature parity. You will reduce your upfront and ongoing expenses for purchasing and configuration on multiple development systems and build machines. Moreover, you will avoid the problem that many cloud services simply do not have emulators available. A testing strategy that depends on emulation will make it impossible to use those services (leading to potentially more expensive workarounds) or produce code and configurations that aren't well tested.

When you do use emulation for testing, you must still test in the cloud to verify configuration and to test interactions with cloud services that can only be simulated or mocked in an emulated environment.

Challenges testing locally

When you use emulators and mocked calls to test on your local desktop you might experience testing inconsistencies as your code progresses from environment to environment in your CI/CD pipeline. Unit tests to validate your application's business logic on your desktop may not accurately test critical aspects of the cloud services.

The following examples provide cases to watch out for when testing locally with mocks and emulators:

Example: Lambda function creates an S3 bucket

If a Lambda function's logic depends on creating an S3 bucket, a complete test should confirm that Amazon S3 was called and the bucket was successfully created.

- In a mock testing setup, you might mock a success response and potentially add a test case to handle a failure response.
- In an emulation testing scenario, the **CreateBucket** API might be called, but you need to be aware that the identity making the local call will **not** originate from the Lambda service. The calling identity will not assume a security role as it would in the cloud, so a placeholder authentication will be used instead, possibly with a more permissive role or user identity that will be different when run in the cloud.

The mock and emulation setups will test what the Lambda function will do if it calls Amazon S3; however, those tests will not verify that the Lambda function, as configured, is capable of successfully creating the Amazon S3 bucket. You must make sure the role assigned to the function has an attached security policy that allows the function to perform the `s3:CreateBucket` action. If not, the function will likely fail when deployed to a cloud environment.

Example: Lambda function processes messages from an Amazon SQS queue

If an Amazon SQS queue is the source of a Lambda function, a complete test should verify that the Lambda function is successfully invoked when a message is put in a queue.

Emulation testing and mock testing are generally set up to run the Lambda function code directly, and to simulate the Amazon SQS integration by passing a JSON event payload (or a deserialized object) as the function handler's input.

Local testing that simulates the Amazon SQS integration will test what the Lambda function will do when it's called by Amazon SQS with a given payload, but the test will not verify that Amazon SQS will successfully invoke the Lambda function when it is deployed to a cloud environment.

Some examples of configuration problems you might encounter with Amazon SQS and Lambda include the following:

- Amazon SQS visibility timeout is too low, resulting in multiple invocations when only one was intended.

- The Lambda function's execution role doesn't allow reading messages from the queue (through `sqs:ReceiveMessage`, `sqs:DeleteMessage`, or`sqs:GetQueueAttributes`).
- The sample event that is passed to the Lambda function exceeds the Amazon SQS message size quota. Therefore, the test is invalid because Amazon SQS would never be able to send a message of that size.

As these examples show, tests that cover business logic but not the configurations between cloud services are likely to provide unreliable results.

FAQ

I have a Lambda function that performs calculations and returns a result without calling any other services. Do I really need to test it in the cloud?

Yes. Lambda functions have configuration parameters that could change the outcome of the test. All Lambda function code has a dependency on timeout and memory settings, which could cause the function to fail if those settings are not set properly. Lambda policies also enable standard output logging to Amazon CloudWatch. Even if your code does not call CloudWatch directly, permission is needed to enable logging. This required permission cannot be accurately mocked or emulated.

How can testing in the cloud help with unit testing? If it's in the cloud and connects to other resources, isn't that an integration test?

We define *unit tests* as tests that operate on architectural components in isolation, but this does not prevent tests from including components that may call other services or use some network communication.

Many serverless applications have architectural components that can be tested in isolation, even in the cloud. One example is a Lambda function that takes input, processes the data, and sends a message to an Amazon SQS queue. A unit test of this function would likely test whether input values result in certain values being present in the queued message.

Consider a test that is written by using the Arrange, Act, Assert pattern:

- *Arrange*: Allocate resources (a queue to receive messages, and the function under test).
- *Act*: Call the function under test.
- *Assert*: Retrieve the message sent by the function, and validate the output.

A mock testing approach would involve mocking the queue with an in-process mock object, and creating an in-process instance of the class or module that contains the Lambda function code. During the Assert phase, the queued message would be retrieved from the mocked object.

In a cloud-based approach, the test would create an Amazon SQS queue for the purposes of the test, and would deploy the Lambda function with environment variables that are configured to use the isolated Amazon SQS queue as the output destination. After running the Lambda function, the test would retrieve the message from the Amazon SQS queue.

The cloud-based test would run the same code, assert the same behavior, and validate the application's functional correctness. However, it would have the added advantage of being able to validate the settings of the Lambda function: the IAM role, IAM policies, and the function's timeout and memory settings.

Next steps and resources

Use the following resources to learn more and explore practical examples of testing.

Sample implementations

The Serverless Test Samples repository on GitHub contains concrete examples of tests that follow the patterns and best practices described in this guide. The repository contains sample code and guided walkthroughs of the mock, emulation, and cloud testing processes described in previous sections. Use this repository to get up to speed on the latest serverless testing guidance from AWS.

Further reading

Visit Serverless Land to access the latest blogs, videos, and training for AWS serverless technologies.

The following AWS blog posts are also recommended reading:

- Accelerating serverless development with AWS SAM Accelerate (AWS blog post)
- Increasing development speed with CDK Watch (AWS blog post)
- Mocking service integrations with AWS Step Functions Local (AWS blog post)
- Getting started with testing serverless applications (AWS blog post)

Tools

- AWS SAM – Testing and debugging serverless applications
- AWS SAM – Integrating with automated tests
- Lambda – Testing Lambda functions in the Lambda console

Improving startup performance with Lambda SnapStart

Lambda SnapStart can provide as low as sub-second startup performance, typically with no changes to your function code. SnapStart makes it easier to build highly responsive and scalable applications without provisioning resources or implementing complex performance optimizations.

The largest contributor to startup latency (often referred to as cold start time) is the time that Lambda spends initializing the function, which includes loading the function's code, starting the runtime, and initializing the function code. With SnapStart, Lambda initializes your function when you publish a function version. Lambda takes a Firecracker microVM snapshot of the memory and disk state of the initialized execution environment, encrypts the snapshot, and intelligently caches it to optimize retrieval latency.

To ensure resiliency, Lambda maintains several copies of each snapshot. Lambda automatically patches snapshots and their copies with the latest runtime and security updates. When you invoke the function version for the first time, and as the invocations scale up, Lambda resumes new execution environments from the cached snapshot instead of initializing them from scratch, improving startup latency.

> **Important**
>
> If your applications depend on uniqueness of state, you must evaluate your function code and verify that it is resilient to snapshot operations. For more information, see Handling uniqueness with Lambda SnapStart.

Topics

- When to use SnapStart
- Supported features and limitations
- Supported Regions
- Compatibility considerations
- SnapStart pricing
- Activating and managing Lambda SnapStart
- Handling uniqueness with Lambda SnapStart

- Implement code before or after Lambda function snapshots
- Monitoring for Lambda SnapStart
- Security model for Lambda SnapStart
- Maximize Lambda SnapStart performance
- Troubleshooting SnapStart errors for Lambda functions

When to use SnapStart

Lambda SnapStart is designed to address the latency variability introduced by one-time initialization code, such as loading module dependencies or frameworks. These operations can sometimes take several seconds to complete during the initial invocation. Use SnapStart to reduce this latency from several seconds to as low as sub-second, in optimal scenarios. SnapStart works best when used with function invocations at scale. Functions that are invoked infrequently might not experience the same performance improvements.

SnapStart is particularly beneficial for two main types of applications:

- **Latency-sensitive APIs and user flows:** Functions that are part of critical API endpoints or user-facing flows can benefit from SnapStart's reduced latency and improved response times.
- **Latency-sensitive data processing workflows:** Time-bound data processing workflows that use Lambda functions can achieve better throughput by reducing outlier function initialization latency.

Provisioned concurrency keeps functions initialized and ready to respond in double-digit milliseconds. Use provisioned concurrency if your application has strict cold start latency requirements that can't be adequately addressed by SnapStart.

Supported features and limitations

SnapStart is available for the following Lambda managed runtimes:

- Java 11 and later
- Python 3.12 and later
- .NET 8 and later. If you're using the Lambda Annotations framework for .NET, upgrade to Amazon.Lambda.Annotations version 1.6.0 or later to ensure compatibility with SnapStart.

Other managed runtimes (such as `nodejs24.x` and `ruby3.4`), OS-only runtimes, and container images are not supported.

SnapStart does not support provisioned concurrency, Amazon Elastic File System (Amazon EFS), or ephemeral storage greater than 512 MB.

> **Note**
>
> You can use SnapStart only on published function versions and aliases that point to versions. You can't use SnapStart on a function's unpublished version ($LATEST).

Supported Regions

Lambda SnapStart is available in all commercial Regions except Asia Pacific (New Zealand) and Asia Pacific (Taipei).

Compatibility considerations

With SnapStart, Lambda uses a single snapshot as the initial state for multiple execution environments. If your function uses any of the following during the initialization phase, then you might need to make some changes before using SnapStart:

Uniqueness

If your initialization code generates unique content that is included in the snapshot, then the content might not be unique when it is reused across execution environments. To maintain uniqueness when using SnapStart, you must generate unique content after initialization. This includes unique IDs, unique secrets, and entropy that's used to generate pseudorandomness. To learn how to restore uniqueness, see Handling uniqueness with Lambda SnapStart.

Network connections

The state of connections that your function establishes during the initialization phase isn't guaranteed when Lambda resumes your function from a snapshot. Validate the state of your network connections and re-establish them as necessary. In most cases, network connections that an AWS SDK establishes automatically resume. For other connections, review the best practices.

Temporary data

Some functions download or initialize ephemeral data, such as temporary credentials or cached timestamps, during the initialization phase. Refresh ephemeral data in the function handler before using it, even when not using SnapStart.

SnapStart pricing

> **Note**
>
> For Java managed runtimes, there's no additional cost for SnapStart. You're charged based on the number of requests for your functions, the time that it takes your code to run, and the memory configured for your function.

The cost of using SnapStart includes the following:

- **Caching:** For every function version that you publish with SnapStart enabled, you pay for the cost of caching and maintaining the snapshot. The price depends on the amount of memory that you allocate to your function. You're charged for a minimum of 3 hours. You will continue to be charged as long as your function remains active. Use the ListVersionsByFunction API action to identify function versions, and then use DeleteFunction to delete unused versions. To automatically delete unused function versions, see the Lambda Version Cleanup pattern on Serverless Land.
- **Restoration:** Each time a function instance is restored from a snapshot, you pay a restoration charge. The price depends on the amount of memory you allocate to your function.

As with all Lambda functions, duration charges apply to code that runs in the function handler. For SnapStart functions, duration charges also apply to initialization code that's declared outside of the handler, the time it takes for the runtime to load, and any code that runs in a runtime hook. Duration is calculated from the time that your code begins running until it returns or otherwise ends, rounded up to the nearest 1 ms. Lambda maintains cached copies of your snapshot for resiliency and automatically applies software updates, such as runtime upgrades and security patches to them. Charges apply each time that Lambda re-runs your initialization code to apply software updates.

For more information about the cost of using SnapStart, see AWS Lambda Pricing.

Activating and managing Lambda SnapStart

To use SnapStart, activate SnapStart on a new or existing Lambda function. Then, publish and invoke a function version.

Topics

- Activating SnapStart (console)
- Activating SnapStart (AWS CLI)
- Activating SnapStart (API)
- Lambda SnapStart and function states
- Updating a snapshot
- Using SnapStart with AWS SDKs
- Using SnapStart with CloudFormation, AWS SAM, and AWS CDK
- Deleting snapshots

Activating SnapStart (console)

To activate SnapStart for a function

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose **Configuration**, and then choose **General configuration**.
4. On the **General configuration** pane, choose **Edit**.
5. On the **Edit basic settings** page, for **SnapStart**, choose **Published versions**.
6. Choose **Save**.
7. Publish a function version. Lambda initializes your code, creates a snapshot of the initialized execution environment, and then caches the snapshot for low-latency access.
8. Invoke the function version.

Activating SnapStart (AWS CLI)

To activate SnapStart for an existing function

1. Update the function configuration by running the update-function-configuration command with the **--snap-start** option.

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. Publish a function version with the publish-version command.

```
aws lambda publish-version \
  --function-name my-function
```

3. Confirm that SnapStart is activated for the function version by running the get-function-configuration command and specifying the version number. The following example specifies version 1.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

If the response shows that OptimizationStatus is On and State is `Active`, then SnapStart is activated and a snapshot is available for the specified function version.

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. Invoke the function version by running the invoke command and specifying the version. The following example invokes version 1.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

To activate SnapStart when you create a new function

1. Create a function by running the create-function command with the **--snap-start** option. For **--role**, specify the Amazon Resource Name (ARN) of your execution role.

```
aws lambda create-function \
  --function-name my-function \
  --runtime "java25" \
  --zip-file fileb://my-function.zip \
  --handler my-function.handler \
  --role arn:aws:iam::111122223333:role/lambda-ex \
  --snap-start ApplyOn=PublishedVersions
```

2. Create a version with the publish-version command.

```
aws lambda publish-version \
  --function-name my-function
```

3. Confirm that SnapStart is activated for the function version by running the get-function-configuration command and specifying the version number. The following example specifies version 1.

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

If the response shows that OptimizationStatus is On and State is `Active`, then SnapStart is activated and a snapshot is available for the specified function version.

```
"SnapStart": {
     "ApplyOn": "PublishedVersions",
     "OptimizationStatus": "On"
  },
  "State": "Active",
```

4. Invoke the function version by running the invoke command and specifying the version. The following example invokes version 1.

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

Activating SnapStart (API)

To activate SnapStart

1. Do one of the following:
   - Create a new function with SnapStart activated by using the CreateFunction API action with the SnapStart parameter.
   - Activate SnapStart for an existing function by using the UpdateFunctionConfiguration action with the SnapStart parameter.
2. Publish a function version with the PublishVersion action. Lambda initializes your code, creates a snapshot of the initialized execution environment, and then caches the snapshot for low-latency access.
3. Confirm that SnapStart is activated for the function version by using the GetFunctionConfiguration action. Specify a version number to confirm that SnapStart is activated for that version. If the response shows that OptimizationStatus is On and State is `Active`, then SnapStart is activated and a snapshot is available for the specified function version.

```
"SnapStart": {
        "ApplyOn": "PublishedVersions",
        "OptimizationStatus": "On"
    },
    "State": "Active",
```

4. Invoke the function version with the Invoke action.

Lambda SnapStart and function states

The following function states can occur when you use SnapStart.

Pending

Lambda is initializing your code and taking a snapshot of the initialized execution environment. Any invocations or other API actions that operate on the function version will fail.

Active

Snapshot creation is complete and you can invoke the function. To use SnapStart, you must invoke the published function version, not the unpublished version ($LATEST).

Inactive

The `Inactive` state can occur when Lambda periodically regenerates function snapshots to apply software updates. In this instance, if your function fails to initialize, the function can enter an `Inactive` state.

For functions using a Java runtime, Lambda deletes snapshots after 14 days without an invocation. If you invoke the function version after 14 days, Lambda returns a `SnapStartNotReadyException` response and begins initializing a new snapshot. Wait until the function version reaches the `Active` state, and then invoke it again.

Failed

Lambda encountered an error when running the initialization code or creating the snapshot.

Updating a snapshot

Lambda creates a snapshot for each published function version. To update a snapshot, publish a new function version.

Using SnapStart with AWS SDKs

To make AWS SDK calls from your function, Lambda generates an ephemeral set of credentials by assuming your function's execution role. These credentials are available as environment variables during your function's invocation. You don't need to provide credentials for the SDK

directly in code. By default, the credential provider chain sequentially checks each place where you can set credentials and chooses the first available—usually the environment variables (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`).

> **Note**
>
> When SnapStart is activated, the Lambda runtime automatically uses the container credentials (`AWS_CONTAINER_CREDENTIALS_FULL_URI` and `AWS_CONTAINER_AUTHORIZATION_TOKEN`) instead of the access key environment variables. This prevents credentials from expiring before the function is restored.

Using SnapStart with CloudFormation, AWS SAM, and AWS CDK

- **AWS CloudFormation:** Declare the SnapStart entity in your template.
- **AWS Serverless Application Model (AWS SAM):** Declare the SnapStart property in your template.
- **AWS Cloud Development Kit (AWS CDK):** Use the SnapStartProperty type.

Deleting snapshots

Lambda deletes snapshots when:

- You delete the function or function version.
- **Java runtimes only** — You don't invoke the function version for 14 days. After 14 days without an invocation, the function version transitions to the Inactive state. If you invoke the function version after 14 days, Lambda returns a `SnapStartNotReadyException` response and begins initializing a new snapshot. Wait until the function version reaches the Active state, and then invoke it again.

Lambda removes all resources associated with deleted snapshots in compliance with the General Data Protection Regulation (GDPR).

Handling uniqueness with Lambda SnapStart

When invocations scale up on a SnapStart function, Lambda uses a single initialized snapshot to resume multiple execution environments. If your initialization code generates unique content that is included in the snapshot, then the content might not be unique when it is reused across execution environments. To maintain uniqueness when using SnapStart, you must generate unique content after initialization. This includes unique IDs, unique secrets, and entropy that's used to generate pseudorandomness.

We recommend the following best practices to help you maintain uniqueness in your code. For Java functions, Lambda also provides an open-source SnapStart scanning tool to help check for code that assumes uniqueness. If you generate unique data during the initialization phase, then you can use a runtime hook to restore uniqueness. With runtime hooks, you can run specific code immediately before Lambda takes a snapshot or immediately after Lambda resumes a function from a snapshot.

Avoid saving state that depends on uniqueness during initialization

During the initialization phase of your function, avoid caching data that's intended to be unique, such as generating a unique ID for logging or setting seeds for random functions. Instead, we recommend that you generate unique data or set seeds for random functions inside your function handler—or use a runtime hook.

The following examples demonstrate how to generate a UUID in the function handler.

Java

Example– Generating a unique ID in function handler

```
import java.util.UUID;
  public class Handler implements RequestHandler<String, String> {
    private static UUID uniqueSandboxId = null;
    @Override
    public String handleRequest(String event, Context context) {
      if (uniqueSandboxId == null)
        uniqueSandboxId = UUID.randomUUID();
      System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
      return "Hello, World!";
    }
  }
```

Python

Example– Generating a unique ID in function handler

```
import json
import random
import time

unique_number = None

def lambda_handler(event, context):
    seed = int(time.time() * 1000)
    random.seed(seed)
    global unique_number
    if not unique_number:
        unique_number = random.randint(1, 10000)

    print("Unique number: ", unique_number)

    return "Hello, World!"
```

.NET

Example– Generating a unique ID in function handler

```
namespace Example;
public class SnapstartExample
{
    private Guid _myExecutionEnvironmentGuid;
    public SnapstartExample()
    {
        // This GUID is set for non-restore use cases, such as testing or if
 SnapStart is turned off
        _myExecutionEnvironmentGuid = new Guid();
        // Register the method which will run after each restore. You may need to
 update Amazon.Lambda.Core to see this
        Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(MyAfterRestore);
    }

    private ValueTask MyAfterRestore()
    {
        // After restoring this snapshot to a new execution environment, update the
 GUID
```

```
        _myExecutionEnvironmentGuid = new Guid();
        return ValueTask.CompletedTask;
    }

    public string Handler()
    {
        return $"Hello World! My Execution Environment GUID is
 {_myExecutionEnvironmentGuid}";
    }
}
```

Use cryptographically secure pseudorandom number generators (CSPRNGs)

If your application depends on randomness, we recommend that you use cryptographically secure random number generators (CSPRNGs). In addition to OpenSSL 1.0.2, the Lambda managed runtimes also include the following built-in CSPRNGs:

- **Java:** `java.security.SecureRandom`
- **Python:** `random.SystemRandom`
- **.NET:** `System.Security.Cryptography.RandomNumberGenerator`

Software that always gets random numbers from `/dev/random` or `/dev/urandom` also maintains randomness with SnapStart.

AWS cryptography libraries automatically maintain randomness with SnapStart beginning with the minimum versions specified in the following table. If you use these libraries with your Lambda functions, make sure that you use the following minimum versions or later versions:

Library	Minimum supported version (x86)	Minimum supported version (ARM)
AWS libcrypto (AWS-LC)	1.16.0	1.30.0
AWS libcrypto FIPS	2.0.13	2.0.13

If you package the preceding cryptographic libraries with your Lambda functions as transitive dependencies through the following libraries, make sure that you use the following minimum versions or later versions:

Library	Minimum supported version (x86)	Minimum supported version (ARM)
AWS SDK for Java 2.x	2.23.20	2.26.12
AWS Common Runtime for Java	0.29.8	0.29.25
Amazon Corretto Crypto Provider	2.4.1	2.4.1
Amazon Corretto Crypto Provider FIPS	2.4.1	2.4.1

The following examples demonstrate how to use CSPRNGs to guarantee unique number sequences even when the function is restored from a snapshot.

Java

Example– java.security.SecureRandom

```
import java.security.SecureRandom;
  public class Handler implements RequestHandler<String, String> {
    private static SecureRandom rng = new SecureRandom();
    @Override
    public String handleRequest(String event, Context context) {
      for (int i = 0; i < 10; i++) {
        System.out.println(rng.next());
      }
      return "Hello, World!";
    }
  }
```

Python

Example– random.SystemRandom

```
import json
import random

secure_rng = random.SystemRandom()

def lambda_handler(event, context):
    random_numbers = [secure_rng.random() for _ in range(10)]

    for number in random_numbers:
        print(number)

    return "Hello, World!"
```

.NET

Example– RandomNumberGenerator

```
using Amazon.Lambda.Core;
using System.Security.Cryptography;
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

namespace DotnetSecureRandom;

public class Function
{
    public string FunctionHandler()
    {
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            byte[] randomUnsignedInteger32Bytes = new byte[4];
            for (int i = 0; i < 10; i++)
            {
                rng.GetBytes(randomUnsignedInteger32Bytes);
                int randomInt32 = BitConverter.ToInt32(randomUnsignedInteger32Bytes,
 0);
                Console.WriteLine("{0:G}", randomInt32);
            }
        }
```

```
        return "Hello World!";
    }
}
```

SnapStart scanning tool (Java only)

Lambda provides a scanning tool for Java to help you check for code that assumes uniqueness. The SnapStart scanning tool is an open-source SpotBugs plugin that runs a static analysis against a set of rules. The scanning tool helps identify potential code implementations that might break assumptions regarding uniqueness. For installation instructions and a list of checks that the scanning tool performs, see the aws-lambda-snapstart-java-rules repository on GitHub.

To learn more about handling uniqueness with SnapStart, see Starting up faster with AWS Lambda SnapStart on the AWS Compute Blog.

Implement code before or after Lambda function snapshots

You can use runtime hooks to implement code before Lambda creates a snapshot or after Lambda resumes a function from a snapshot. Runtime hooks are useful for a variety of purposes, such as:

- **Cleanup and initialization:** Before a snapshot is created, you can use a runtime hook to perform cleanup or resource release operations. After a snapshot is restored, you can use a runtime hook to re-initialize any resources or state that were not captured in the snapshot.
- **Dynamic configuration:** You can use runtime hooks to dynamically update configuration or other metadata before a snapshot is created or after it is restored. This can be useful if your function needs to adapt to changes in the runtime environment.
- **External integrations:** You can use runtime hooks to integrate with external services or systems, such as sending notifications or updating external state, as part of the checkpointing and restoration process.
- **Performance tuning:** You can use runtime hooks to fine-tune your function's startup sequence, such as by preloading dependencies. For more information, see Performance tuning.

The following pages explain how to implement runtime hooks for your preferred runtime.

Topics

- Lambda SnapStart runtime hooks for Java
- Lambda SnapStart runtime hooks for Python
- Lambda SnapStart runtime hooks for .NET

Lambda SnapStart runtime hooks for Java

You can use runtime hooks to implement code before Lambda creates a snapshot or after Lambda resumes a function from a snapshot. Runtime hooks are available as part of the open-source Coordinated Restore at Checkpoint (CRaC) project. CRaC is in development for the Open Java Development Kit (OpenJDK). For an example of how to use CRaC with a reference application, see the CRaC repository on GitHub. CRaC uses three main elements:

- `Resource` – An interface with two methods, `beforeCheckpoint()` and `afterRestore()`. Use these methods to implement the code that you want to run before a snapshot and after a restore.

- `Context <R extends Resource>` – To receive notifications for checkpoints and restores, a `Resource` must be registered with a `Context`.
- `Core` – The coordination service, which provides the default global `Context` via the static method `Core.getGlobalContext()`.

For more information about `Context` and `Resource`, see Package org.crac in the CRaC documentation.

Use the following steps to implement runtime hooks with the org.crac package. The Lambda runtime contains a customized CRaC context implementation that calls your runtime hooks before checkpointing and after restoring.

Runtime hook registration and execution

The order that Lambda executes your runtime hooks is determined by the order of registration. Registration order follows the order of import, definition, or execution in your code.

- `beforeCheckpoint()`: Executed in the reverse order of registration
- `afterRestore()`: Executed in the order of registration

Make sure that all registered hooks are properly imported and included in your function's code. If you register runtime hooks in a separate file or module, you must ensure that the module is imported, either directly or as part of a larger package, in your function's handler file. If the file or module is not imported in the function handler, Lambda ignores the runtime hooks.

> **Note**
>
> When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the configured function timeout (maximum 900 seconds), whichever is higher. Your `beforeCheckpoint()` runtime hooks count towards the initialization code time limit. When Lambda restores a snapshot, the runtime must load and `afterRestore()` runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException.

Step 1: Update the build configuration

Add the `org.crac` dependency to the build configuration. The following example uses Gradle. For examples for other build systems, see the Apache Maven documentation.

```
dependencies {
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'
    # All other project dependecies go here:
    # ...
    # Then, add the org.crac dependency:
 implementation group: 'org.crac', name: 'crac', version: '1.4.0'
}
```

Step 2: Update the Lambda handler

The Lambda function *handler* is the method in your function code that processes events. When your function is invoked, Lambda runs the handler method. Your function runs until the handler returns a response, exits, or times out.

For more information, see Define Lambda function handler in Java.

The following example handler shows how to run code before checkpointing (`beforeCheckpoint()`) and after restoring (`afterRestore()`). This handler also registers the `Resource` to the runtime-managed global `Context`.

> **Note**
>
> When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the configured function timeout (maximum 900 seconds), whichever is higher. Your `beforeCheckpoint()` runtime hooks count towards the initialization code time limit. When Lambda restores a snapshot, the runtime (JVM) must load and `afterRestore()` runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException.

```
...
  import org.crac.Resource;
  import org.crac.Core;
  ...
```

```
public class CRaCDemo implements RequestStreamHandler, Resource {
    public CRaCDemo() {
      Core.getGlobalContext().register(this);
    }
    public String handleRequest(String name, Context context) throws IOException {
      System.out.println("Handler execution");
      return "Hello " + name;
    }
    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("Before checkpoint");
    }
    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("After restore");
```

`Context` maintains only a `WeakReference` to the registered object. If a `Resource` is garbage collected, runtime hooks do not run. Your code must maintain a strong reference to the `Resource` to guarantee that the runtime hook runs.

Here are two examples of patterns to avoid:

Example– Object without a strong reference

```
Core.getGlobalContext().register( new MyResource() );
```

Example– Objects of anonymous classes

```
Core.getGlobalContext().register( new Resource() {

   @Override
   public void afterRestore(Context<? extends Resource> context) throws Exception {
    // ...
   }

   @Override
   public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
    // ...
   }
```

```
} );
```

Instead, maintain a strong reference. In the following example, the registered resource isn't garbage collected and runtime hooks run consistently.

Example– Object with a strong reference

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

Lambda SnapStart runtime hooks for Python

You can use runtime hooks to implement code before Lambda creates a snapshot or after Lambda resumes a function from a snapshot. Python runtime hooks are available as part of the open-source Snapshot Restore for Python library, which is included in Python managed runtimes. This library provides two decorators that you can use to define your runtime hooks:

- `@register_before_snapshot`: For functions you want to run before Lambda creates a snapshot.
- `@register_after_restore`: For functions you want to run when Lambda resumes a function from a snapshot.

Alternatively, you can use the following methods to register callables for runtime hooks:

- `register_before_snapshot(func, *args, **kwargs)`
- `register_after_restore(func, *args, **kwargs)`

Runtime hook registration and execution

The order that Lambda executes your runtime hooks is determined by the order of registration:

- Before snapshot: Executed in the reverse order of registration
- After snapshot: Executed in the order of registration

The order of runtime hook registration depends on how you define the hooks. When using decorators (`@register_before_snapshot` and `@register_after_restore`), the

registration order follows the order of import, definition, or execution in your code. If you need more control over the registration order, use the `register_before_snapshot()` and `register_after_restore()` methods instead of decorators.

Make sure that all registered hooks are properly imported and included in your function's code. If you register runtime hooks in a separate file or module, you must ensure that the module is imported, either directly or as part of a larger package, in your function's handler file. If the file or module is not imported in the function handler, Lambda ignores the runtime hooks.

> **Note**
>
> When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the configured function timeout (maximum 900 seconds), whichever is higher. Your `@register_before_snapshot` runtime hooks count towards the initialization code time limit. When Lambda restores a snapshot, the runtime must load and `@register_after_restore` runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException.

Example

The following example handler shows how to run code before checkpointing (`@register_before_snapshot`) and after restoring (`@register_after_restore`).

```
from snapshot_restore_py import register_before_snapshot, register_after_restore

def lambda_handler(event, context):
    # Handler code

@register_before_snapshot
def before_checkpoint():
    # Logic to be executed before taking snapshots

@register_after_restore
def after_restore():
    # Logic to be executed after restore
```

For more examples, see Snapshot Restore for Python in the AWS GitHub repository.

Lambda SnapStart runtime hooks for .NET

You can use runtime hooks to implement code before Lambda creates a snapshot or after Lambda resumes a function from a snapshot. .NET runtime hooks are available as part of the Amazon.Lambda.Core package (version 2.5.0 or later). This library provides two methods that you can use to define your runtime hooks:

- `RegisterBeforeSnapshot()`: Code to run before snapshot creation
- `RegisterAfterSnapshot()`: Code to run after resuming a function from a snapshot

> **Note**
>
> If you're using the Lambda Annotations framework for .NET, upgrade to Amazon.Lambda.Annotations version 1.6.0 or later to ensure compatibility with SnapStart.

Runtime hook registration and execution

Register your hooks in your initialization code. Consider the following guidelines based on your Lambda function's execution model:

- For the executable assembly approach, register your hooks before you start the Lambda bootstrap with `RunAsync`.
- For the class library approach, register your hooks in the handler class constructor.
- For ASP.NET Core applications, register your hooks before calling the `WebApplications.Run` method.

To register runtime hooks for SnapStart in .NET, use the following methods:

```
Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
```

When multiple hook types are registered, the order that Lambda executes your runtime hooks is determined by the order of registration:

- `RegisterBeforeSnapshot()`: Executed in the reverse order of registration

- `RegisterAfterSnapshot()`: Executed in the order of registration

> **Note**
>
> When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the configured function timeout (maximum 900 seconds), whichever is higher. Your `RegisterBeforeSnapshot()` runtime hooks count towards the initialization code time limit. When Lambda restores a snapshot, the runtime must load and `RegisterAfterSnapshot()` runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException.

Example

The following example function shows how to run code before checkpointing (`RegisterBeforeSnapshot`) and after restoring (`RegisterAfterRestore`).

```
public class SampleClass
{
    public SampleClass()
    {
        Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
        Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
    }

    private ValueTask BeforeCheckpoint()
    {
        // Add logic to be executed before taking the snapshot
        return ValueTask.CompletedTask;
    }

    private ValueTask AfterCheckpoint()
    {
        // Add logic to be executed after restoring the snapshot
        return ValueTask.CompletedTask;
    }

    public APIGatewayProxyResponse FunctionHandler(APIGatewayProxyRequest request,
 ILambdaContext context)
    {
        // Add business logic
```

```
        return new APIGatewayProxyResponse
        {
            StatusCode = 200
        };
    }
}
```

Monitoring for Lambda SnapStart

You can monitor your Lambda SnapStart functions using Amazon CloudWatch, AWS X-Ray, and the Accessing real-time telemetry data for extensions using the Telemetry API.

> **Note**
>
> The `AWS_LAMBDA_LOG_GROUP_NAME` and `AWS_LAMBDA_LOG_STREAM_NAME` environment variables are not available in Lambda SnapStart functions.

Understanding logging and billing behavior with SnapStart

There are a few differences with the CloudWatch log stream format for SnapStart functions:

- **Initialization logs** – When a new execution environment is created, the `REPORT` doesn't include the `Init Duration` field. That's because Lambda initializes SnapStart functions when you create a version instead of during function invocation. For SnapStart functions, the `Init Duration` field is in the `INIT_REPORT` record. This record shows duration details for the Init phase, including the duration of any `beforeCheckpoint` runtime hooks.
- **Invocation logs** – When a new execution environment is created, the `REPORT` includes the `Restore Duration` and `Billed Restore Duration` fields:
  - `Restore Duration`: The time it takes for Lambda to restore a snapshot, load the runtime, and run any after-restore runtime hooks. The process of restoring snapshots can include time spent on activities outside the MicroVM. This time is reported in `Restore Duration`.
  - `Billed Restore Duration`: The time it takes for Lambda to load the runtime and run any after-restore runtime hooks.

> **Note**
>
> As with all Lambda functions, duration charges apply to code that runs in the function handler. For SnapStart functions, duration charges also apply to initialization code that's declared outside of the handler, the time it takes for the runtime to load, and any code that runs in a runtime hook.

The cold start duration is the sum of `Restore Duration` + `Duration`.

The following example is a Lambda Insights query that returns the latency percentiles for SnapStart functions. For more information about Lambda Insights queries, see Example workflow using queries to troubleshoot a function.

```
filter @type = "REPORT"
  | parse @log /\d+:\/aws\/lambda\/(?<function>.*)/
  | parse @message /Restore Duration: (?<restoreDuration>.*?) ms/
  | stats
count(*) as invocations,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 50) as p50,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 90) as p90,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99) as p99,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99.9) as p99.9
group by function, (ispresent(@initDuration) or ispresent(restoreDuration)) as
 coldstart
  | sort by coldstart desc
```

X-Ray active tracing for SnapStart

You can use X-Ray to trace requests to Lambda SnapStart functions. There are a few differences with the X-Ray subsegments for SnapStart functions:

- There is no `Initialization` subsegment for SnapStart functions.
- The `Restore` subsegment shows the time it takes for Lambda to restore a snapshot, load the runtime, and run any after-restore runtime hooks. The process of restoring snapshots can include time spent on activities outside the MicroVM. This time is reported in the `Restore` subsegment. You aren't charged for the time spent outside the microVM to restore a snapshot.

Telemetry API events for SnapStart

Lambda sends the following SnapStart events to the Telemetry API:

- `platform.restoreStart` – Shows the time when the `Restore` phase started.
- `platform.restoreRuntimeDone` – Shows whether the `Restore` phase was successful. Lambda sends this message when the runtime sends a `restore/next` runtime API request. There are three possible statuses: success, failure, and timeout.
- `platform.restoreReport` – Shows how long the `Restore` phase lasted and how many milliseconds you were billed for during this phase.

Amazon API Gateway and function URL metrics

If you create a web API using API Gateway, then you can use the IntegrationLatency metric to measure end-to-end latency (the time between when API Gateway relays a request to the backend and when it receives a response from the backend).

If you're using a Lambda function URL, then you can use the UrlRequestLatency metric to measure end-to-end latency (the time between when the function URL receives a request and when the function URL returns a response).

Security model for Lambda SnapStart

Lambda SnapStart supports encryption at rest. Lambda encrypts snapshots with an AWS KMS key. By default, Lambda uses an AWS managed key. If this default behavior suits your workflow, then you don't need to set up anything else. Otherwise, you can use the `--kms-key-arn` option in the create-function or update-function-configuration command to provide an AWS KMS customer managed key. You might do this to control rotation of the KMS key or to meet the requirements of your organization for managing KMS keys. Customer managed keys incur standard AWS KMS charges. For more information, see AWS Key Management Service pricing.

When you delete a SnapStart function or function version, all `Invoke` requests to that function or function version fail. Lambda removes all resources associated with deleted snapshots in compliance with the General Data Protection Regulation (GDPR).

Maximize Lambda SnapStart performance

Topics

- Performance tuning
- Networking best practices

Performance tuning

To maximize the benefits of SnapStart, consider the following code optimization recommendations for your runtime.

> **Note**
>
> SnapStart works best when used with function invocations at scale. Functions that are invoked infrequently might not experience the same performance improvements.

Java

To maximize the benefits of SnapStart, we recommend that you preload dependencies and initialize resources that contribute to startup latency in your initialization code instead of in the function handler. This moves the latency associated with heavy class loading out of the invocation path, optimizing startup performance with SnapStart.

If you can't preload dependencies or resources during initialization, then we recommend that you preload them with dummy invocations. To do this, update the function handler code, as shown in the following example from the pet store function on the AWS Labs GitHub repository.

```
private static SpringLambdaContainerHandler<AwsProxyRequest, AwsProxyResponse> handler;
  static {
      try {
          handler =
 SpringLambdaContainerHandler.getAwsProxyHandler(PetStoreSpringAppConfig.class);

          // Use the onStartup method of the handler to register the custom filter
          handler.onStartup(servletContext -> {
              FilterRegistration.Dynamic registration =
 servletContext.addFilter("CognitoIdentityFilter", CognitoIdentityFilter.class);
```

```
            registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
false, "/*");
        });

        // Send a fake Amazon API Gateway request to the handler to load classes
ahead of time
        ApiGatewayRequestIdentity identity = new ApiGatewayRequestIdentity();
        identity.setApiKey("foo");
        identity.setAccountId("foo");
        identity.setAccessKey("foo");

        AwsProxyRequestContext reqCtx = new AwsProxyRequestContext();
        reqCtx.setPath("/pets");
        reqCtx.setStage("default");
        reqCtx.setAuthorizer(null);
        reqCtx.setIdentity(identity);

        AwsProxyRequest req = new AwsProxyRequest();
        req.setHttpMethod("GET");
        req.setPath("/pets");
        req.setBody("");
        req.setRequestContext(reqCtx);

        Context ctx = new TestContext();
        handler.proxy(req, ctx);


    } catch (ContainerInitializationException e) {
        // if we fail here. We re-throw the exception to force another cold start
        e.printStackTrace();
        throw new RuntimeException("Could not initialize Spring framework", e);
    }
 }
```

Python

To maximize the benefits of SnapStart, focus on efficient code organization and resource management within your Python functions. As a general guideline, perform heavy computational tasks during the initialization phase. This approach moves time-consuming operations out of the invocation path, improving overall function performance. To implement this strategy effectively, we recommend the following best practices:

- Import dependencies outside of the function handler.

- Create `boto3` instances outside of the handler.
- Initialize static resources or configurations before the handler is invoked.
- Consider using a before-snapshot runtime hook for resource-intensive tasks such as downloading external files, pre-loading frameworks like Django, or loading machine learning models.

Example— Optimize Python function for SnapStart

```
# Import all dependencies outside of Lambda handler
from snapshot_restore_py import register_before_snapshot
import boto3
import pandas
import pydantic

# Create S3 and SSM clients outside of Lambda handler
s3_client = boto3.client("s3")

# Register the function to be called before snapshot
@register_before_snapshot
def download_llm_models():
    # Download an object from S3 and save to tmp
    # This files will persist in this snapshot
    with open('/tmp/FILE_NAME', 'wb') as f:
        s3_client.download_fileobj('amzn-s3-demo-bucket', 'OBJECT_NAME', f)
    ...

def lambda_handler(event, context):
    ...
```

.NET

To reduce just-in-time (JIT) compilation and assembly loading time, consider invoking your function handler from a `RegisterBeforeCheckpoint` runtime hook. Because of how .NET tiered compilation works, you'll get optimal results by invoking the handler multiple times, as shown in the following example.

> **Important**
>
> Make sure that your dummy function invocation does not produce unintended side effects, such as initiating business transactions.

Example

```
public class Function
{
    public Function()
    {
        Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(FunctionWarmup);
    }

    // Warmup method that calls the function handler before snapshot to warm up
 the .NET code and runtime.
    // This speeds up future cold starts after restoring from a snapshot.

    private async ValueTask FunctionWarmup()
    {
        var request = new APIGatewayProxyRequest
        {
            Path = "/heathcheck",
            HttpMethod = "GET"
        };

        for (var i = 0; i < 10; i++)
        {
            await FunctionHandler(request, null);
        }
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        //
        // Process HTTP request
        //

        var response = new APIGatewayProxyResponse
        {
            StatusCode = 200
        };

        return await Task.FromResult(response);
    }
}
```

Networking best practices

The state of connections that your function establishes during the initialization phase isn't guaranteed when Lambda resumes your function from a snapshot. In most cases, network connections that an AWS SDK establishes automatically resume. For other connections, we recommend the following best practices.

Re-establish network connections

Always re-establish your network connections when your function resumes from a snapshot. We recommend that you re-establish network connections in the function handler. Alternatively, you can use an after-restore runtime hook.

Don't use hostname as a unique execution environment identifier

We recommend against using `hostname` to identify your execution environment as a unique node or container in your applications. With SnapStart, a single snapshot is used as the initial state for multiple execution environments. All execution environments return the same `hostname` value for `InetAddress.getLocalHost()` (Java), `socket.gethostname()` (Python), and `Dns.GetHostName()` (.NET). For applications that require a unique execution environment identity or `hostname` value, we recommend that you generate a unique ID in the function handler. Or, use an after-restore runtime hook to generate a unique ID, and then use the unique ID as the identifier for the execution environment.

Avoid binding connections to fixed source ports

We recommend that you avoid binding network connections to fixed source ports. Connections are re-established when a function resumes from a snapshot, and network connections that are bound to a fixed source port might fail.

Avoid using Java DNS cache

Lambda functions already cache DNS responses. If you use another DNS cache with SnapStart, then you might experience connection timeouts when the function resumes from a snapshot.

The `java.util.logging.Logger` class can indirectly enable the JVM DNS cache. To override the default settings, set networkaddress.cache.ttl to 0 before initializing `logger`. Example:

```
public class MyHandler {
  // first set TTL property
```

```
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

To prevent UnknownHostException failures in the Java 11 runtime, we recommend setting networkaddress.cache.negative.ttl to 0. In Java 17 and later runtimes, this step isn't necessary. You can set this property for a Lambda function with the AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0 environment variable.

Disabling the JVM DNS cache does not disable Lambda's managed DNS caching.

Troubleshooting SnapStart errors for Lambda functions

This page addresses common issues that occur when using Lambda SnapStart, including snapshot creation errors, timeout errors, and internal service errors.

SnapStartNotReadyException

Error: An error occurred (SnapStartNotReadyException) when calling the Invoke20150331 operation: Lambda is initializing your function. It will be ready to invoke once your function state becomes ACTIVE.

Common causes

This error occurs when you try to invoke a function version that is in the `Inactive` state. Your function version becomes `Inactive` when it hasn't been invoked for 14 days or when Lambda periodically recycles the execution environment

Resolution

Wait until the function version reaches the `Active` state, and then invoke it again.

SnapStartTimeoutException

Issue: You receive a `SnapStartTimeoutException` when you try to invoke a SnapStart function version.

Common cause

During the Restore phase, Lambda restores the Java runtime and runs any after-restore runtime hooks. If an after-restore runtime hook runs for longer than 10 seconds, the `Restore` phase times out and you get an error when you try to invoke the function. Network connection and credentials issues can also cause `Restore` phase timeouts.

Resolution

Check the function's CloudWatch logs for timeout errors that happened during the Restore phase. Make sure that all after-restore hooks complete in less than 10 seconds.

Example CloudWatch log

```
{ "cause": "Lambda couldn't restore the snapshot within the timeout limit. (Service:
 Lambda, Status Code: 408, Request ID: 11a222c3-410f-427c-ab22-931d6bcbf4f2)", "error":
 "Lambda.SnapStartTimeoutException"}
```

500 Internal Service Error

Error: Lambda was unable to create a new snapshot because you have reached your concurrent snapshot creation limit.

Common cause

A 500 error is an internal error within the Lambda service itself, rather than an issue with your function or code. These errors are often intermittent.

Resolution

Try to publish the function version again.

401 Unauthorized

Error: Bad session token or header key

Common cause

This error occurs when using the AWS Systems Manager Parameter Store and AWS Secrets Manager extension with Lambda SnapStart.

Resolution

The AWS Systems Manager Parameter Store and AWS Secrets Manager extension isn't compatible with SnapStart. The extension generates credentials for communicating with AWS Secrets Manager during function initialization, which causes expired credential errors when used with SnapStart.

UnknownHostException (Java)

Error: Unable to execute HTTP request: Certificate for `abc.us-east-1.amazonaws.com` doesn't match any of the subject alternative names.

Common cause

Lambda functions already cache DNS responses. If you use another DNS cache with SnapStart, then you might experience connection timeouts when the function resumes from a snapshot.

Resolution

To prevent `UnknownHostException` failures in the Java 11 runtime, we recommend setting `networkaddress.cache.negative.ttl` to 0. In Java 17 and later runtimes, this step isn't necessary. You can set this property for a Lambda function with the `AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0` environment variable.

Snapshot creation failures

Error: AWS Lambda could not invoke your SnapStart function. If this error persists, check your function's CloudWatch logs for initialization errors.

Resolution

Review your function's Amazon CloudWatch logs for before-checkpoint runtime hook timeouts. You can also try publishing a new function version, which can sometimes resolve the issue.

Snapshot creation latency

Issue: When you publish a new function version, the function stays in the `Pending` state for a long time.

Common cause

When Lambda creates a snapshot, your initialization code can run for up to 15 minutes. The time limit is 130 seconds or the configured function timeout (maximum 900 seconds), whichever is higher.

If your function is attached to a VPC, Lambda might also need to create network interfaces before the function becomes `Active`. If you try to invoke the function version while the function is `Pending`, you might get a 409 `ResourceConflictException`. If the function is invoked using an Amazon API Gateway endpoint, you might get a 500 error in API Gateway.

Resolution

Wait at least 15 minutes for the function version to initialize before invoking it.

Tenant isolation

Use tenant isolation mode when you need isolated request processing for individual end-users or tenants invoking a Lambda function. This capability simplifies building multi-tenant applications that process tenant-specific code or data, such as SaaS platforms for workflow automation or code execution, by removing the need to manage tenant-specific function resources and request routing logic.

Multi-tenant applications have strict isolation requirements when running code or processing data for individual tenants or end-users. With tenant isolation mode, Lambda uses a customer-specified tenant identifier to route requests to underlying function execution environments, ensuring that a function's execution environments are only used to serve invocations from the specified end-user or tenant. Lambda's function execution environments leverage Firecracker virtualization to provide workload isolation.

When a function using tenant isolation mode receives an invoke with a tenant identifier, Lambda first attempts to locate an available execution environment associated with that tenant identifier. If no execution environments exist, Lambda creates and assigns a new execution environment to that tenant. As function invocations with the specified tenant identifier scale up, Lambda locates or creates new execution environments as necessary.

Topics

- When to use tenant isolation mode
- Supported features and limitations
- Supported AWS Regions
- Considerations
- Pricing
- Isolation mode
- Enabling tenant isolation for Lambda functions
- Invoking Lambda functions with tenant isolation
- Accessing tenant identifier in Lambda function code
- Monitoring Lambda functions with tenant isolation
- Troubleshooting tenant isolation for Lambda functions

When to use tenant isolation mode

Use tenant isolation mode when you need to serve multiple end-users or tenants using a single Lambda function, while ensuring that the execution environments used to process invocations for individual tenants remain isolated from one another. This strict isolation of execution environments is required for multi-tenant applications that:

- **Execute end-user supplied code**: Maintaining separate execution environments for individual tenants can limit the impact of executing user-supplied code that may be incorrect or malicious.
- **Process tenant-specific data**: Maintaining separate execution environments for individual tenants can prevent exposure of sensitive tenant-specific data to other tenants.

Multiple invocation requests from the same tenant can re-use the same function execution environment, reducing cold-starts and improving response times for latency sensitive applications.

Supported features and limitations

Tenant isolation mode is not supported with functions that use function URLs, provisioned concurrency, or SnapStart. You can send requests to a tenant-isolated function using synchronous invocations, asynchronous invocations, or by using Amazon API Gateway as an event-trigger.

Supported AWS Regions

Tenant isolation mode is supported in all commercial Regions except Asia Pacific (New Zealand).

Considerations

When using tenant isolation with your Lambda functions, keep the following in mind:

- **Immutable configuration**: Tenant isolation is an immutable function property. It can only be enabled when creating a function.
- **Required tenant-id parameter**: Functions using tenant isolation mode must be invoked with a `tenant-id` parameter. Omitting this parameter will cause function invocations to fail.
- **Execution role applies to all tenants**: Invocations from all tenants use the permissions defined in your Lambda function's execution role.

- **Concurrency**: There are no changes to your function's concurrency or scaling behavior when using tenant isolation. Lambda imposes a limit of 2,500 tenant-isolated execution environments (active or idle) for every 1,000 concurrent executions configured for your Lambda function.

Pricing

You are charged when Lambda creates a new tenant-isolated execution environment. The price depends on the amount of memory that you allocate to your function and the CPU architecture that you use. For more information, see AWS Lambda pricing.

Isolation mode

The following table outlines differences between Lambda functions with and without tenant isolation.

Feature	With tenant isolation	Without tenant isolation
Isolation type	Tenant-level isolation	Function-level isolation
Environment reuse	Execution environments are never reused across different tenants	Execution environments might be reused across invocations of the same function
Data isolation	Data from other tenants is not accessible	Data from previous invocations of the same function version may be accessible
Cold starts	More cold starts due to tenant-specific environments	Fewer cold starts due to environment reuse
Pricing	Additional charge besides the standard Lambda pricing	Standard Lambda pricing

Enabling tenant isolation for Lambda functions

To activate tenant isolation mode, create a new Lambda function. You cannot enable tenant isolation on existing functions.

Topics

- Enabling tenant isolation (console)
- Enabling tenant isolation (AWS CLI)
- Enabling tenant isolation (API)
- Enabling tenant isolation (CloudFormation)

Enabling tenant isolation (console)

To create a Lambda function using the console

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Select **Author from scratch**.
4. In the **Basic information** pane, for **Function name**, enter **`image-analysis`**.
5. For **Runtime**, choose any of the supported Lambda runtimes.
6. Under additional configurations, **Tenant isolation mode**, select **Enable**.
7. Review your settings, and choose **Create function**.

Enabling tenant isolation (AWS CLI)

Create function with tenant isolation

When creating a new function using the CLI, add the `--tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'` option to your create-function request. Example:

```
aws lambda create-function \
    --function-name image-analysis \
    --runtime nodejs24.x \
    --zip-file fileb://image-analysis-function.zip \
    --handler image-analysis-function.handler \
```

```
    --role arn:aws:iam:123456789012:role/execution-role \
    --tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'
```

Enabling tenant isolation (API)

To enable tenant isolation using the Lambda API

1. Create a new function with tenant isolation enabled by using the CreateFunction API action with the `TenancyConfig` parameter.
2. Confirm that tenant isolation is enabled for the function by using the GetFunctionConfiguration action. If the response shows that `TenantIsolationMode` is PER_TENANT, then tenant isolation is enabled for the function:

```
"TenancyConfig": {
        "TenantIsolationMode": "PER_TENANT"
    }
```

Invoke the function version with the Invoke action. For more information, see Invoking Lambda functions with tenant isolation.

Enabling tenant isolation (CloudFormation)

The following CloudFormation template creates a new Lambda function with tenant isolation enabled:

```
MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-sample-python-lambda
      Runtime: python3.14
      Role: !GetAtt LambdaExecutionRole.Arn
      Handler: index.lambda_handler
      TenancyConfig:
        TenantIsolationMode: PER_TENANT
      Code:
        ZipFile: |
          import json

          def lambda_handler(event, context):
              return {
```

```
                'statusCode': 200,
                'body': json.dumps(f'Hello from Lambda! Tenant-ID:
{context.tenant_id}')
            }
     Timeout: 10
     MemorySize: 128
```

Invoking Lambda functions with tenant isolation

When invoking a function that has tenant isolation enabled, you must provide a `tenant-id` parameter. This parameter ensures that your function invocation is processed in an execution environment dedicated to that specific tenant.

Invoking functions with tenant isolation (AWS CLI)

Synchronous invocation

For synchronous invocations, add the `--tenant-id` parameter to your Invoke command:

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    response.json
```

Asynchronous invocation

For asynchronous invocations, include both the `--tenant-id` and `--invocation-type Event` parameters:

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    --invocation-type Event \
    response.json
```

Invoking functions with tenant isolation (API)

When using the Invoke API action directly, include the tenant identifier using the `X-Amzn-Tenant-Id` parameter in your request.

Example API request

```
POST /2015-03-31/functions/image-analysis/invocations HTTP/1.1
Host: lambda.us-east-1.amazonaws.com
Content-Type: application/json
Authorization: AWS4-HMAC-SHA256 Credential=...
X-Amz-Tenant-Id: blue
```

```
{
    "key1": "value1",
    "key2": "value2"
}
```

Invoking functions with tenant isolation (API Gateway)

When using API Gateway REST APIs to trigger tenant-isolated Lambda functions, you must configure API Gateway to map client request properties to the `X-Amz-Tenant-Id` header that Lambda expects. API Gateway uses Lambda's Invoke API action, which requires the tenant ID to be passed using the `X-Amz-Tenant-Id` HTTP header. You can configure API Gateway to inject this HTTP header into the Lambda invocation request with a value obtained from client request properties such as HTTP headers, query parameters, or path parameters. You must first map the client request property before you can override the `X-Amz-Tenant-Id` header.

> **Note**
>
> You cannot use HTTP APIs to invoke tenant-isolated Lambda functions because it is not possible to override the `X-Amz-Tenant-Id` header.

Using request headers

Configure your API Gateway integration to map a custom header from the client request to the `X-Amz-Tenant-Id` header. The following example shows a client request with an `x-tenant-id` header:

```
POST /api/process HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
x-tenant-id: blue

{
    "data": "sample payload"
}
```

In your API Gateway method configuration, you must:

1. Enable the client request header parameter (for example, `method.request.header.x-tenant-id`)

2. Map the client header to the Lambda integration header using `integration.request.header.X-Amz-Tenant-Id`

Using query parameters

Similarly, you can map query parameters to the `X-Amz-Tenant-Id` header:

```
GET /api/process?tenant-id=blue&data=sample HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
```

Configure the method to enable the query parameter and map it to the integration header.

Invoking functions with tenant isolation (SDK)

When using AWS SDKs to invoke tenant-isolated functions, include the tenant identifier in your invocation request.

Python

```
import boto3
import json

lambda_client = boto3.client('lambda')

response = lambda_client.invoke(
    FunctionName='image-analysis',
    TenantId='blue',
    Payload=json.dumps({
        'key1': 'value1',
        'key2': 'value2'
    })
)

result = json.loads(response['Payload'].read())
```

Node.js

```
const AWS = require('aws-sdk');
const lambda = new AWS.Lambda();

const params = {
```

```
    FunctionName: 'image-analysis',
    TenantId: 'blue',
    Payload: JSON.stringify({
        key1: 'value1',
        key2: 'value2'
    })
};

lambda.invoke(params, (err, data) => {
    if (err) {
        console.error(err);
    } else {
        const result = JSON.parse(data.Payload);
        console.log(result);
    }
});
```

Java

```
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.InvokeRequest;
import software.amazon.awssdk.services.lambda.model.InvokeResponse;
import software.amazon.awssdk.core.SdkBytes;

public class TenantIsolationExample {

    public static void main(String[] args) {
        LambdaClient lambdaClient = LambdaClient.create();

        String payload = "{\"key1\": \"value1\", \"key2\": \"value2\"}";

        InvokeRequest request = InvokeRequest.builder()
                .functionName("image-analysis")
                .tenantId("blue")
                .payload(SdkBytes.fromUtf8String(payload))
                .build();

        InvokeResponse response = lambdaClient.invoke(request);
    }
}
```

Accessing tenant identifier in Lambda function code

When your Lambda function has tenant isolation enabled, the tenant identifier used to invoke your function is made available within the context object passed to your function handler. You can use this identifier to implement tenant-specific logic, monitoring, and debugging capabilities.

Topics

- Accessing the tenant identifier
- Common usage patterns
- Monitoring and debugging

Accessing the tenant identifier

The tenant identifier is available through the `tenantId` property of the context object. Note that this property is available during the invocation phase, not during the initialization phase.

Python

```
def lambda_handler(event, context):
    tenant_id = context.tenant_id
    print(f"Processing request for tenant: {tenant_id}")

    # Implement tenant-specific logic
    if tenant_id == "blue":
        return process_blue_tenant(event)
    elif tenant_id == "green":
        return process_green_tenant(event)
    else:
        return process_default_tenant(event)
```

Node.js

```
exports.handler = async (event, context) => {
    const tenantId = context.tenantId;
    console.log(`Processing request for tenant: ${tenantId}`);

    // Implement tenant-specific logic
    switch (tenantId) {
        case 'blue':
```

```
            return processBlueTenant(event);
        case 'green':
            return processGreenTenant(event);
        default:
            return processDefaultTenant(event);
    }
};
```

Java

```
public class TenantHandler implements RequestHandler<Map<String, Object>, String> {

    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        String tenantId = context.getTenantId();
        System.out.println("Processing request for tenant: " + tenantId);

        // Implement tenant-specific logic
        switch (tenantId) {
            case "blue":
                return processBlueTenant(event);
            case "green":
                return processGreenTenant(event);
            default:
                return processDefaultTenant(event);
        }
    }
}
```

Common usage patterns

Here are common ways to use the tenant identifier in your function code:

Tenant-specific configuration

Use the tenant ID to load tenant-specific configuration or settings:

```
def lambda_handler(event, context):
    tenant_id = context.tenant_id

    # Load tenant-specific configuration
    config = load_tenant_config(tenant_id)
```

```
    database_url = config['database_url']
    api_key = config['api_key']

    # Process with tenant-specific settings
    return process_request(event, database_url, api_key)
```

Tenant-specific data access

Use the tenant ID to ensure data isolation and access control:

```
import boto3

def lambda_handler(event, context):
    tenant_id = context.tenant_id

    # Ensure data access is scoped to the tenant
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table('user_data')

    user_id = event.get('userId')

    response = table.get_item(
        Key={
            'tenant_id': tenant_id,
            'user_id': user_id
        }
    )

    return process_results(response.get('Item'), tenant_id)
```

Monitoring and debugging

The tenant identifier is automatically included in Lambda logs when you have JSON logging enabled, making it easier to monitor and debug tenant-specific issues. You can also use the tenant ID for custom metrics and tracing.

Example Custom metrics with tenant ID

The following example demonstrates how to use the tenant ID to create tenant-specific CloudWatch metrics for monitoring usage patterns and performance by tenant:

```
import boto3
```

```
def lambda_handler(event, context):
    tenant_id = context.tenant_id
    cloudwatch = boto3.client('cloudwatch')

    # Record tenant-specific metrics
    cloudwatch.put_metric_data(
        Namespace='MyApp/TenantMetrics',
        MetricData=[
            {
                'MetricName': 'RequestCount',
                'Dimensions': [
                    {
                        'Name': 'TenantId',
                        'Value': tenant_id
                    }
                ],
                'Value': 1,
                'Unit': 'Count'
            }
        ]
    )

    return process_request(event, tenant_id)
```

Monitoring Lambda functions with tenant isolation

You can monitor your tenant-isolated Lambda functions using Amazon CloudWatch, AWS X-Ray, and by accessing real-time telemetry data for extensions using the Telemetry API.

Understanding logging for tenant isolated mode

For functions using tenant isolation, Lambda automatically includes the tenant identifier in function logs when you have JSON logging enabled, making it easier to monitor and debug tenant-specific issues. Lambda creates a separate CloudWatch log stream for each execution environment. You can use CloudWatch Logs Insights to find log streams that belong to a particular tenant by filtering by tenant identifier:

```
fields @logStream, @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| stats count() as logCount by @logStream
| sort @timestamp desc
```

You can also use this parameter to retrieve tenant-specific logs across all log streams:

```
fields @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| limit 1000
```

The `tenantId` property is included for platforms events (like `platform.start` and `platform.report`) and custom logs you print in your function code, as shown below:

```
{
    "time": "2025-10-13T19:48:06.990Z",
    "type": "platform.start",
    "record": {
        "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
        "functionArn": "arn:aws:lambda:us-east-1:xxxxxx:function:multitenant-
function-1",
        "version": "$LATEST",
        "tenantId": "BlueTenant""
    }
}
{
    "timestamp": "2025-10-13T19:48:06.992Z",
```

```
    "level": "INFO",
    "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
    "tenantId": "BlueTenant"",
    "message": "custom log line1"
}
{
    "timestamp": "2025-10-13T19:48:07.022Z",
    "level": "WARN",
    "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
    "tenantId": "BlueTenant"",
    "message": "custom log line2"
}
```

Troubleshooting tenant isolation for Lambda functions

This page addresses common issues that occur when using tenant isolation for AWS Lambda.

InvalidParameterValueException

Error : Tenant ID configuration not specified or passed to a function when tenant isolation is not enabled.

Common causes

This error occurs when invoking a tenant-isolated function without a tenant ID, or invoking a non-tenant-isolated function with a tenant ID.

Resolution

Add a tenant ID if the function has tenant isolation enabled, or remove the tenant ID if the function doesn't have tenant isolation enabled.

TooManyRequestsException

Error: Rate exceeded

Common causes

In addition to rate limiting based on maximum concurrent executions and function scaling rate, Lambda limits the maximum number of tenant-aware execution environments (active or idle) that can exist at a time to 2,500 for every 1,000 concurrent executions of your function.

Resolution

To fix this issue, you can either lower the rate at which invocation requests with unique tenant identifiers are made, implement retries with backoff and jitter, or request a function concurrency limit increase.

Invoking Lambda with events from other AWS services

Some AWS services can directly invoke Lambda functions using *triggers*. These services push events to Lambda, and the function is invoked immediately when the specified event occurs. Triggers are suitable for discrete events and real-time processing. When you create a trigger using the Lambda console, the console interacts with the corresponding AWS service to configure the event notification on that service. The trigger is actually stored and managed by the service that generates the events, not by Lambda.

The events are data structured in JSON format. The JSON structure varies depending on the service that generates it and the event type, but they all contain the data that the function needs to process the event.

A function can have multiple triggers. Each trigger acts as a client invoking your function independently, and each event that Lambda passes to your function has data from only one trigger. Lambda converts the event document into an object and passes it to your function handler.

Depending on the service, the event-driven invocation can be synchronous or asynchronous.

- For synchronous invocation, the service that generates the event waits for the response from your function. That service defines the data that the function needs to return in the response. The service controls the error strategy, such as whether to retry on errors.
- For asynchronous invocation, Lambda queues the event before passing it to your function. When Lambda queues the event, it immediately sends a success response to the service that generated the event. After the function processes the event, Lambda doesn't return a response to the event-generating service.

Creating a trigger

The easiest way to create a trigger is to use the Lambda console. When you create a trigger using the console, Lambda automatically adds the required permissions to the function's resource-based policy.

To create a trigger using the Lambda console

1. Open the Functions page of the Lambda console.
2. Select the function you want to create a trigger for.

3. In the **Function overview** pane, choose **Add trigger**.
4. Select the AWS service you want to invoke your function.
5. Fill out the options in the **Trigger configuration** pane and choose **Add**. Depending on the AWS service you choose to invoke your function, the trigger configuration options will be different.

Services that can invoke Lambda functions

The following table lists services that can invoke Lambda functions.

Service	Method of invocation
Amazon Managed Streaming for Apache Kafka	Event source mapping
Self-managed Apache Kafka	Event source mapping
Amazon API Gateway	Event-driven; synchronous invocation
AWS CloudFormation	Event-driven; asynchronous invocation
Amazon CloudWatch Logs	Event-driven; asynchronous invocation
AWS CodeCommit	Event-driven; asynchronous invocation
AWS CodePipeline	Event-driven; asynchronous invocation
Amazon Cognito	Event-driven; synchronous invocation
AWS Config	Event-driven; asynchronous invocation
Amazon Connect	Event-driven; synchronous invocation
Amazon DocumentDB	Event source mapping
Amazon DynamoDB	Event source mapping
Elastic Load Balancing (Application Load Balancer)	Event-driven; synchronous invocation

Service	Method of invocation
Amazon EventBridge (CloudWatch Events)	Event-driven; asynchronous invocation (event buses), synchronous or asynchronous invocation (pipes and schedules)
AWS IoT	Event-driven; asynchronous invocation
Amazon Kinesis	Event source mapping
Amazon Data Firehose	Event-driven; synchronous invocation
Amazon Lex	Event-driven; synchronous invocation
Amazon MQ	Event source mapping
Amazon Simple Email Service	Event-driven; asynchronous invocation
Amazon Simple Notification Service	Event-driven; asynchronous invocation
Amazon Simple Queue Service	Event source mapping
Amazon Simple Storage Service (Amazon S3)	Event-driven; asynchronous invocation
Amazon Simple Storage Service Batch	Event-driven; synchronous invocation
Secrets Manager	Secret rotation
AWS Step Functions	Event-driven; synchronous or asynchronous invocation
Amazon VPC Lattice	Event-driven; synchronous invocation

Using Lambda with Apache Kafka

Lambda supports Apache Kafka as an event source. Apache Kafka is an open-source event streaming platform designed to handle high-throughput, real-time data pipelines and streaming applications. There are two main ways to use Lambda with Apache Kafka:

- the section called "MSK" – Amazon Managed Streaming for Apache Kafka (Amazon MSK) is a fully-managed service by AWS. Amazon MSK helps automate management of your Kafka infrastructure, including provisioning, patching, and scaling.
- the section called "Self-managed Apache Kafka" – In AWS terminology, a self-managed cluster includes non-AWS hosted Kafka clusters. For example, you can still use Lambda with a Kafka cluster hosted with a non-AWS cloud provider such as Confluent Cloud or Redpanda.

When deciding between Amazon MSK and self-managed Apache Kafka, consider your operational needs and control requirements. Amazon MSK is a better choice if you want AWS to quickly help you manage a scalable, production-ready Kafka setup with minimal operational overhead. It simplifies security, monitoring, and high availability, helping you focus on application development rather than infrastructure management. On the other hand, self-managed Apache Kafka is better suited for use cases running on non-AWS hosted environments, including on-premises clusters.

Topics

- Using Lambda with Amazon MSK
- Using Lambda with self-managed Apache Kafka
- Apache Kafka event poller scaling modes in Lambda
- Apache Kafka polling and stream starting positions in Lambda
- Customizable consumer group ID in Lambda
- Filtering events from Amazon MSK and self-managed Apache Kafka event sources
- Using schema registries with Kafka event sources in Lambda
- Low latency processing for Kafka event sources
- Configuring error handling controls for Kafka event sources
- Capturing discarded batches for Amazon MSK and self-managed Apache Kafka event sources
- Using a Kafka topic as an on-failure destination
- Kafka event source mapping logging
- Troubleshooting Kafka event source mapping errors

Using Lambda with Amazon MSK

Amazon Managed Streaming for Apache Kafka (Amazon MSK) is a fully-managed service that you can use to build and run applications that use Apache Kafka to process streaming data. Amazon MSK simplifies the setup, scaling, and management of Kafka clusters. Amazon MSK also makes it easier to configure your application for multiple Availability Zones and for security with AWS Identity and Access Management (IAM).

This chapter explains how to use an Amazon MSK cluster as an event source for your Lambda function. The general process for integrating Amazon MSK with Lambda involves the following steps:

1. **Cluster and network setup** – First, set up your Amazon MSK cluster. This includes the correct networking configuration to allow Lambda to access your cluster.
2. **Event source mapping setup** – Then, create the event source mapping resource that Lambda needs to securely connect your Amazon MSK cluster to your function.
3. **Function and permissions setup** – Finally, ensure that your function is correctly set up, and has the necessary permissions in its execution role.

> **Note**
>
> You can now create and manage your Amazon MSK event source mappings directly from either the Lambda or the Amazon MSK console. Both consoles offer the option to automatically handle the setup of the necessary Lambda execution role permissions for a more streamlined configuration process.

For examples on how to set up a Lambda integration with an Amazon MSK cluster, see the section called "Tutorial", Using Amazon MSK as an event source for AWS Lambda on the AWS Compute Blog, and Amazon MSK Lambda Integration in the Amazon MSK Labs.

Topics

- Example event
- Configuring your Amazon MSK cluster and Amazon VPC network for Lambda
- Configuring Lambda permissions for Amazon MSK event source mappings
- Configuring Amazon MSK event sources for Lambda

- Tutorial: Using an Amazon MSK event source mapping to invoke a Lambda function

Example event

Lambda sends the batch of messages in the event parameter when it invokes your function. The event payload contains an array of messages. Each array item contains details of the Amazon MSK topic and partition identifier, together with a timestamp and a base64-encoded message.


```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
```

```
            }
        ]
    }
}
```

Configuring your Amazon MSK cluster and Amazon VPC network for Lambda

To connect your AWS Lambda function to your Amazon MSK cluster, you need to correctly configure your cluster and the Amazon Virtual Private Cloud (VPC) it resides in. This page describes how to configure your cluster and VPC. If your cluster and VPC are already configured properly, see the section called "Configure event source" to configure the event source mapping.

Topics

- Overview of network configuration requirements for Lambda and MSK integrations
- Configuring a NAT gateway for an MSK event source
- Configuring AWS PrivateLink endpoints for an MSK event source

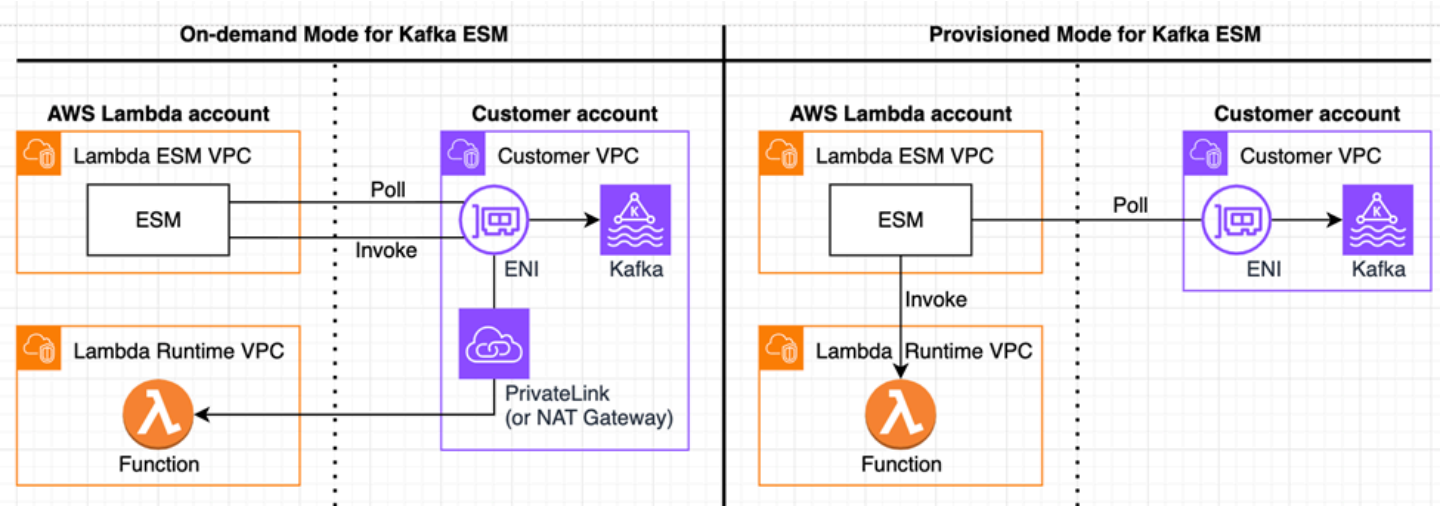
Overview of network configuration requirements for Lambda and MSK integrations

The networking configuration required for a Lambda and MSK integration depends on the network architecture of your application. There are three main resources involved in this integration: the Amazon MSK cluster, the Lambda function, and the Lambda event source mapping. Each of these resources resides in a different VPC:

- Your Amazon MSK cluster typically resides in a private subnet of a VPC that you manage.
- Your Lambda function resides in an AWS-managed VPC owned by Lambda.
- Your Lambda event source mapping resides in another AWS-managed VPC owned by Lambda, separate from the VPC that contains your function.

The event source mapping is the intermediary resource between the MSK cluster and the Lambda function. The event source mapping has two primary jobs. First, it polls your MSK cluster for new messages. Then, it invokes your Lambda function with those messages. Since these three resources are in different VPCs, both the poll and invoke operations require cross-VPC network calls.

The network configuration requirements for your event source mapping depends on whether it uses provisioned mode or on-demand mode, as shown in the following diagram:



The way that the Lambda event source mapping polls your MSK cluster for new messages is the same in both modes. To establish a connection between your event source mapping and your MSK cluster, Lambda creates a hyperplane ENI (or reuses an existing one, if available) in your private subnet to establish a secure connection. As illustrated in the diagram, this hyperplane ENI uses the subnet and security group configuration of your MSK cluster, not your Lambda function.

After polling the message from the cluster, the way Lambda invokes your function is different in each mode:

- In provisioned mode, Lambda automatically handles the connection between the event source mapping VPC and the function VPC. So, you don't need any additional networking components to successfully invoke your function.
- In on-demand mode, your Lambda event source mapping invokes your function via a path through your customer-managed VPC. Because of this, you need to configure either a NAT gateway in the public subnet of your VPC, or AWS PrivateLink endpoints in the private subnet of the VPC that provide access to Lambda, AWS Security Token Service (STS), and optionally, AWS Secrets Manager. Correctly configuring either one of these options allows a connection between your VPC and the Lambda-managed runtime VPC, which is necessary to invoke your function.

A NAT gateway allows resources in your private subnet to access the public internet. Using this configuration means your traffic traverses the internet before invoking the Lambda function. AWS PrivateLink endpoints allow private subnets to securely connect to AWS services or other private VPC resources without traversing the public internet. See the section called "Configuring a NAT gateway for an MSK event source" or the section called "Configuring AWS PrivateLink endpoints for an MSK event source" for details on how to configure these resources.

So far, we've assumed that your MSK cluster resides in a private subnet within your VPC, which is the more common case. However, even if your MSK cluster is in a public subnet within your VPC, you must configure AWS PrivateLink endpoints to enable a secure connection. The following table summarizes the networking configuration requirements based on how you configure your MSK cluster and Lambda event source mapping:

MSK cluster location (in customer-managed VPC)	Lambda event source mapping scaling mode	Required networking configuration
Private subnet	On-demand mode	NAT gateway (in your VPC's public subnet), or AWS PrivateLink endpoints (in your VPC's private subnet) to enable access to Lambda, AWS STS, and optionally, Secrets Manager.
Public subnet	On-demand mode	AWS PrivateLink endpoints (in your VPC's public subnet) to enable access to Lambda, AWS STS, and optionally, Secrets Manager.
Private subnet	Provisioned mode	None
Public subnet	Provisioned mode	None

In addition, the security groups associated with your MSK cluster must allow traffic over the correct ports. Ensure that you have the following security group rules configured:

- **Inbound rules** – Allow all traffic on the default broker port. The port that MSK uses depends on the type of authentication on the cluster: 9098 for IAM authentication, 9096 for SASL/SCRAM, and 9094 for TLS. Alternatively, you can use a self-referencing security group rule to allow access from instances within the same security group.
- **Outbound rules** – Allow all traffic on port 443 for external destinations if your function needs to communicate with other AWS services. Alternatively, you can use a self-referencing security

group rule to limit access to the broker if you don't need to communicate with other AWS services.

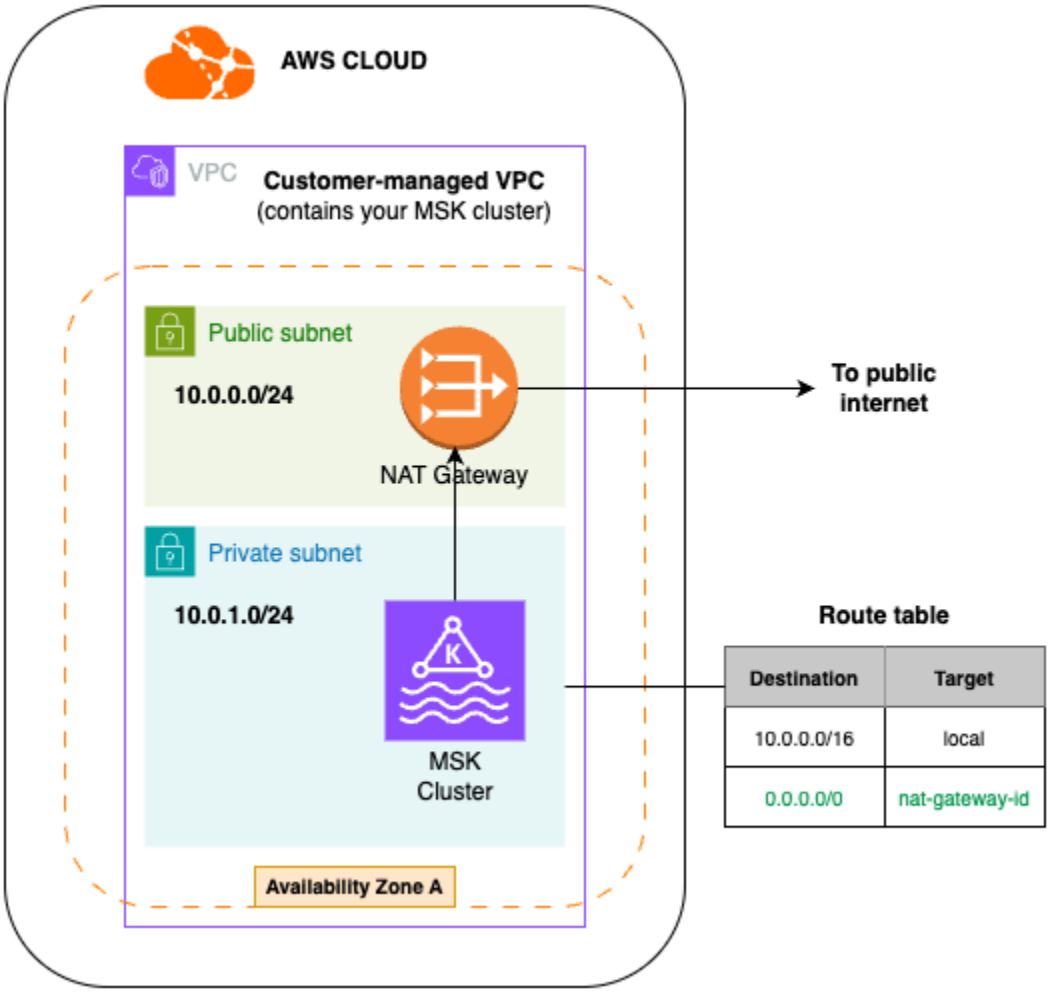
- **Amazon VPC endpoint inbound rules** – If you're using an Amazon VPC endpoint, the security group associated with the endpoint must allow inbound traffic on port 443 from the cluster's security group.

Configuring a NAT gateway for an MSK event source

You can configure a NAT gateway to allow your event source mapping to poll messages from your cluster, and invoke the function via a path through your VPC. This is required only if your event source mapping uses on-demand mode, and your cluster resides within a private subnet of your VPC. If your cluster resides in a public subnet of your VPC, or your event source mapping uses provisioned mode, you don't need to configure a NAT gateway.

A NAT gateway allows resources in a private subnet to access the public internet. If you need private connectivity to Lambda, see the section called "Configuring AWS PrivateLink endpoints for an MSK event source" instead.

After you configure your NAT gateway, you must configure the appropriate route tables. This allows traffic from your private subnet to route to the public internet via the NAT gateway.



The following steps guide you through configuring a NAT gateway using the console. Repeat these steps as necessary for each Availability Zone (AZ).

To configure a NAT gateway and proper routing (console)

1. Follow the steps in Create a NAT gateway, noting the following:
   - NAT gateways should always reside in a public subnet. Create NAT gateways with public connectivity.
   - If your MSK cluster is replicated across multiple AZs, create one NAT gateway per AZ. For example, in each AZ, your VPC should have one private subnet containing your cluster, and one public subnet containing your NAT gateway. For a setup with three AZs, you'll have three private subnets, three public subnets, and three NAT gateways.
2. After you create your NAT gateway, open the Amazon VPC console and choose **Route tables** in the left menu.

3. Choose **Create route table**.
4. Associate this route table with the VPC that contains your MSK cluster. Optionally, enter a name for your route table.
5. Choose **Create route table**.
6. Choose the route table you just created.
7. Under the **Subnet associations** tab, choose **Edit subnet associations**.
   - Associate this route table with the private subnet that contains your MSK cluster.
8. Choose **Edit routes**.
9. Choose **Add route**:
   1. For **Destination**, choose `0.0.0.0/0`.
   2. For **Target**, choose **NAT gateway**.
   3. In the search box, choose the NAT gateway you created in step 1. This should be the NAT gateway in the same AZ as the private subnet that contains your MSK cluster (the private subnet that you associated with this route table in step 6).
10. Choose **Save changes**.

Configuring AWS PrivateLink endpoints for an MSK event source

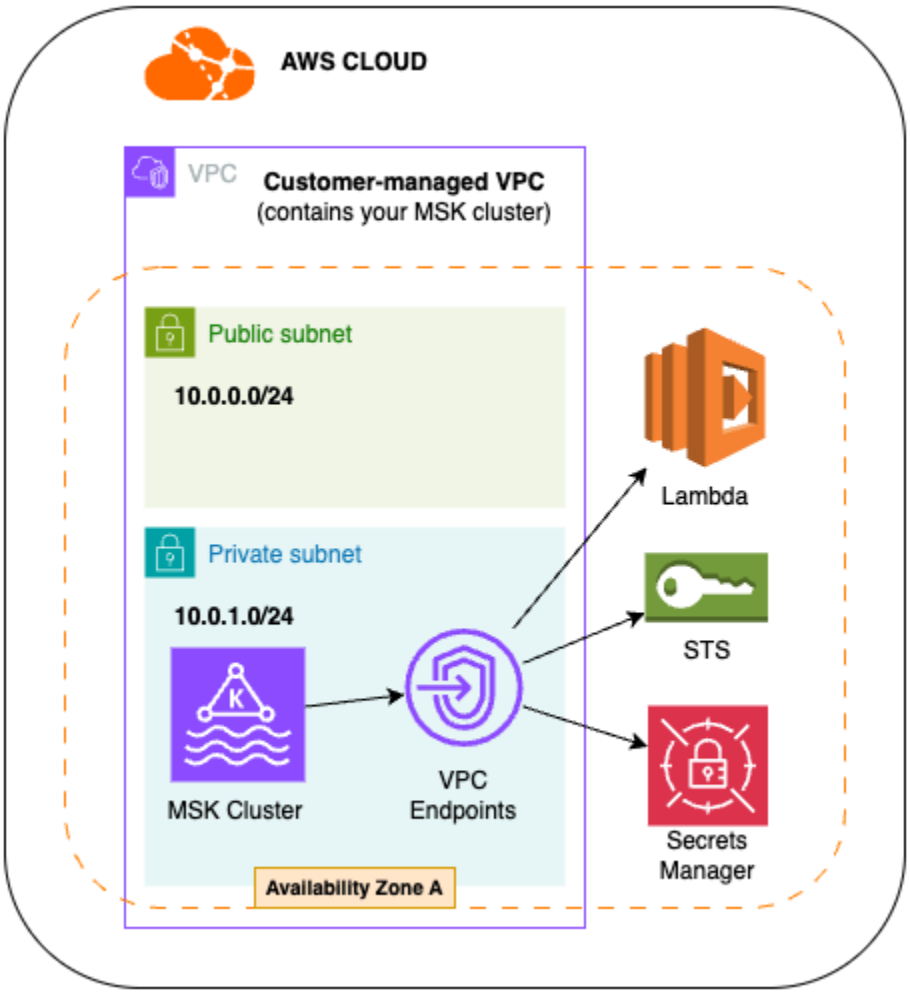
You can configure AWS PrivateLink endpoints to poll messages from your cluster, and invoke the function via a path through your VPC. These endpoints should allow your MSK cluster to access the following:

- The Lambda service
- The AWS Security Token Service (STS)
- Optionally, the AWS Secrets Manager service. This is required if the secret required for cluster authentication is stored in Secrets Manager.

Configuring PrivateLink endpoints is required only if your event source mapping uses on-demand mode. If your event source mapping uses provisioned mode, Lambda establishes the required connections for you.

PrivateLink endpoints allow secure, private access to AWS services over AWS PrivateLink. Alternatively, to configure a NAT gateway to give your MSK cluster access to the public internet, see the section called “Configuring a NAT gateway for an MSK event source”.

After you configure your VPC endpoints, your MSK cluster should have direct and private access to Lambda, STS, and optionally, Secrets Manager.



The following steps guide you through configuring a PrivateLink endpoint using the console. Repeat these steps as necessary for each endpoint (Lambda, STS, Secrets Manager).

To configure VPC PrivateLink endpoints (console)

1. Open the Amazon VPC console and choose **Endpoints** in the left menu.
2. Choose **Create endpoint**.
3. Optionally, enter a name for your endpoint.
4. For **Type**, choose **AWS services**.
5. Under **Services**, start typing the name of the service. For example, to create an endpoint to connect to Lambda, type `lambda` in the search box.

6. In the results, you should see the service endpoint in the current region. For example, in the US East (N. Virginia) region, you should see `com.amazonaws.us-east-2.lambda`. Select this service.
7. Under **Network settings**, select the VPC that contains your MSK cluster.
8. Under **Subnets**, select the AZs that your MSK cluster is in.
   - For each AZ, under **Subnet ID**, choose the private subnet that contains your MSK cluster.
9. Under **Security groups**, select the security groups associated with your MSK cluster.
10. Choose **Create endpoint**.

By default, Amazon VPC endpoints have open IAM policies that allow broad access to resources. Best practice is to restrict these policies to perform the needed actions using that endpoint. For example, for your Secrets Manager endpoint, you can modify its policy such that it allows only your function's execution role to access the secret.

Example VPC endpoint policy – Secrets Manager endpoint

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

For the AWS STS and Lambda endpoints, you can restrict the calling principal to the Lambda service principal. However, ensure that you use `"Resource": "*"` in these policies.

Example VPC endpoint policy – AWS STS endpoint

```
{
    "Statement": [
```

```
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC endpoint policy – Lambda endpoint

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Configuring Lambda permissions for Amazon MSK event source mappings

To access the Amazon MSK cluster, your function and event source mapping need permissions to perform various Amazon MSK API actions. Add these permissions to the function's execution role. If your users need access, add the required permissions to the identity policy for the user or role.

The AWSLambdaMSKExecutionRole managed policy contains the minimum required permissions for Amazon MSK Lambda event source mappings. To simplify the permissions process, you can:

- Attach the AWSLambdaMSKExecutionRole managed policy to your execution role.
- Let the Lambda console generate the permissions for you. When you create an Amazon MSK event source mapping in the console, Lambda evaluates your execution role and alerts you if any

permissions are missing. Choose **Generate permissions** to automatically update your execution role. This doesn't work if you manually created or modified your execution role policies, or if the policies are attached to multiple roles. Note that additional permissions may still be required in your execution role when using advanced features such as On-Failure Destination or AWS Glue Schema Registry.

Topics

- Required permissions
- Optional permissions

Required permissions

Your Lambda function execution role must have the following required permissions for Amazon MSK event source mappings. These permissions are included in the AWSLambdaMSKExecutionRole managed policy.

CloudWatch Logs permissions

The following permissions allow Lambda to create and store logs in Amazon CloudWatch Logs.

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

MSK cluster permissions

The following permissions allow Lambda to access your Amazon MSK cluster on your behalf:

- kafka:DescribeCluster
- kafka:DescribeClusterV2
- kafka:GetBootstrapBrokers

We recommend using kafka:DescribeClusterV2 instead of kafka:DescribeCluster. The v2 permission works with both provisioned and serverless Amazon MSK clusters. You only need one of these permissions in your policy.

VPC permissions

The following permissions allow Lambda to create and manage network interfaces when connecting to your Amazon MSK cluster:

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

Optional permissions

Your Lambda function might also need permissions to:

- Access cross-account Amazon MSK clusters. For cross-account event source mappings, you need kafka:DescribeVpcConnection in the execution role. An IAM principal creating a cross-account event source mapping needs kafka:ListVpcConnections.
- Access your SCRAM secret, if you're using SASL/SCRAM authentication. This lets your function use a username and password to connect to Kafka.
- Describe your Secrets Manager secret, if you're using SASL/SCRAM or mTLS authentication. This allows your function to retrieve the credentials or certificates needed for secure connections.
- Access your AWS KMS customer managed key, if your AWS Secrets Manager secret is encrypted with an AWS KMS customer managed key.
- Access your schema registry secrets, if you're using a schema registry with authentication:
  - For AWS Glue Schema Registry: Your function needs `glue:GetRegistry` and `glue:GetSchemaVersion` permissions. These allow your function to look up and use the message format rules stored in AWS Glue.
  - For Confluent Schema Registry with `BASIC_AUTH` or `CLIENT_CERTIFICATE_TLS_AUTH`: Your function needs `secretsmanager:GetSecretValue` permission for the secret containing the authentication credentials. This lets your function retrieve the username/password or certificates needed to access the Confluent Schema Registry.

  - For private CA certificates: Your function needs secretsmanager:GetSecretValue permission for the secret containing the certificate. This allows your function to verify the identity of schema registries that use custom certificates.
- Access Kafka cluster consumer groups and poll messages from the topic, if you're using IAM authentication for the event source mapping.

These correspond to the following required permissions:

- kafka:ListScramSecrets - Allows listing of SCRAM secrets for Kafka authentication
- secretsmanager:GetSecretValue - Enables retrieval of secrets from Secrets Manager
- kms:Decrypt - Permits decryption of encrypted data using AWS KMS
- glue:GetRegistry - Allows access to AWS Glue Schema Registry
- glue:GetSchemaVersion - Enables retrieval of specific schema versions from AWS Glue Schema Registry
- kafka-cluster:Connect - Grants permission to connect and authenticate to the cluster
- kafka-cluster:AlterGroup - Grants permission to join groups on a cluster, equivalent to Apache Kafka's READ GROUP ACL
- kafka-cluster:DescribeGroup - Grants permission to describe groups on a cluster, equivalent to Apache Kafka's DESCRIBE GROUP ACL
- kafka-cluster:DescribeTopic - Grants permission to describe topics on a cluster, equivalent to Apache Kafka's DESCRIBE TOPIC ACL
- kafka-cluster:ReadData - Grants permission to read data from topics on a cluster, equivalent to Apache Kafka's READ TOPIC ACL

Additionally, if you want to send records of failed invocations to an on-failure destination, you'll need the following permissions depending on the destination type:

- For Amazon SQS destinations: sqs:SendMessage - Allows sending messages to an Amazon SQS queue
- For Amazon SNS destinations: sns:Publish - Permits publishing messages to an Amazon SNS topic
- For Amazon S3 bucket destinations: s3:PutObject and s3:ListBucket - Enables writing and listing objects in an Amazon S3 bucket

For troubleshooting authentication and authorization errors, see the section called "Troubleshooting".

Configuring Amazon MSK event sources for Lambda

To use an Amazon MSK cluster as an event source for your Lambda function, you create an event source mapping that connects the two resources. This page describes how to create an event source mapping for Amazon MSK.

This page assumes that you've already properly configured your MSK cluster and the Amazon Virtual Private Cloud (VPC) it resides in. If you need to set up your cluster or VPC, see the section called "Cluster and network setup". To configure retry behavior for error handling, see the section called "Retry configurations".

Topics

- Using an Amazon MSK cluster as an event source
- Configuring Amazon MSK cluster authentication methods in Lambda
- Creating a Lambda event source mapping for an Amazon MSK event source
- Creating cross-account event source mappings in Lambda
- All Amazon MSK event source configuration parameters in Lambda

Using an Amazon MSK cluster as an event source

When you add your Apache Kafka or Amazon MSK cluster as a trigger for your Lambda function, the cluster is used as an event source.

Lambda reads event data from the Kafka topics that you specify as `Topics` in a CreateEventSourceMapping request, based on the starting position that you specify. After successful processing, your Kafka topic is committed to your Kafka cluster.

Lambda reads messages sequentially for each Kafka topic partition. A single Lambda payload can contain messages from multiple partitions. When more records are available, Lambda continues processing records in batches, based on the BatchSize value that you specify in a CreateEventSourceMapping request, until your function catches up with the topic.

After Lambda processes each batch, it commits the offsets of the messages in that batch. If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of

messages until processing succeeds or the messages expire. You can send records that fail all retry attempts to an on-failure destination for later processing.

> **Note**
>
> While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, Amazon DocumentDB, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes.

Configuring Amazon MSK cluster authentication methods in Lambda

Lambda needs permission to access your Amazon MSK cluster, retrieve records, and perform other tasks. Amazon MSK supports several ways to authenticate with your MSK cluster.

Cluster authentication methods

- Unauthenticated access
- SASL/SCRAM authentication
- Mutual TLS authentication
- IAM authentication
- How Lambda chooses a bootstrap broker

Unauthenticated access

If no clients access the cluster over the internet, you can use unauthenticated access.

SASL/SCRAM authentication

Lambda supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication, with the SHA-512 hash function and Transport Layer Security (TLS) encryption. For Lambda to connect to the cluster, store the authentication credentials (username and password) in a Secrets Manager secret, and reference this secret when configuring your event source mapping.

For more information about using Secrets Manager, see Sign-in credentials authentication with Secrets Manager in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

> **Note**
>
> Amazon MSK doesn't support SASL/PLAIN authentication.

Mutual TLS authentication

Mutual TLS (mTLS) provides two-way authentication between the client and the server. The client sends a certificate to the server for the server to verify the client. The server also sends a certificate to the client for the client to verify the server.

For Amazon MSK integrations with Lambda, your MSK cluster acts as the server, and Lambda acts as the client.

- For Lambda to verify your MSK cluster, you configure a client certificate as a secret in Secrets Manager, and reference this certificate in your event source mapping configuration. The client certificate must be signed by a certificate authority (CA) in the server's trust store.
- The MSK cluster also sends a server certificate to Lambda. The server certificate must be signed by a certificate authority (CA) in the AWS trust store.

Amazon MSK doesn't support self-signed server certificates. All brokers in Amazon MSK use public certificates signed by Amazon Trust Services CAs, which Lambda trusts by default.

Configuring the mTLS secret

The CLIENT_CERTIFICATE_TLS_AUTH secret requires a certificate field and a private key field. For an encrypted private key, the secret requires a private key password. Both the certificate and private key must be in PEM format.

> **Note**
>
> Lambda supports the PBES1 (but not PBES2) private key encryption algorithms.

The certificate field must contain a list of certificates, beginning with the client certificate, followed by any intermediate certificates, and ending with the root certificate. Each certificate must start on a new line with the following structure:

```
-----BEGIN CERTIFICATE-----
```

```
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager supports secrets up to 65,536 bytes, which is enough space for long certificate chains.

The private key must be in PKCS #8 format, with the following structure:

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

For an encrypted private key, use the following structure:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

The following example shows the contents of a secret for mTLS authentication using an encrypted private key. For an encrypted private key, you include the private key password in the secret.

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
```

```
}
```

For more information about mTLS for Amazon MSK, and instructions on how to generate a client certificate, see Mutual TLS client authentication for Amazon MSK in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

IAM authentication

You can use AWS Identity and Access Management (IAM) to authenticate the identity of clients that connect to the MSK cluster. With IAM auth, Lambda relies on the permissions in your function's execution role to connect to the cluster, retrieve records, and perform other required actions. For a sample policy that contains the necessary permissions, see Create authorization policies for the IAM role in the *Amazon Managed Streaming for Apache Kafka Developer Guide*.

If IAM auth is active on your MSK cluster, and you don't provide a secret, Lambda automatically defaults to using IAM auth.

For more information about IAM authentication in Amazon MSK, see IAM access control.

How Lambda chooses a bootstrap broker

Lambda chooses a bootstrap broker based on the authentication methods available on your cluster, and whether you provide a secret for authentication. If you provide a secret for mTLS or SASL/SCRAM, Lambda automatically chooses that auth method. If you don't provide a secret, Lambda selects the strongest auth method that's active on your cluster. The following is the order of priority in which Lambda selects a broker, from strongest to weakest auth:

- mTLS (secret provided for mTLS)
- SASL/SCRAM (secret provided for SASL/SCRAM)
- SASL IAM (no secret provided, and IAM auth active)
- Unauthenticated TLS (no secret provided, and IAM auth not active)
- Plaintext (no secret provided, and both IAM auth and unauthenticated TLS are not active)

> **Note**
>
> If Lambda can't connect to the most secure broker type, Lambda doesn't attempt to connect to a different (weaker) broker type. If you want Lambda to choose a weaker broker type, deactivate all stronger auth methods on your cluster.

Creating a Lambda event source mapping for an Amazon MSK event source

To create an event source mapping, you can use the Lambda console, the AWS Command Line Interface (CLI), or an AWS SDK.

> **Note**
>
> When you create the event source mapping, Lambda creates a hyperplane ENI in the private subnet that contains your MSK cluster, allowing Lambda to establish a secure connection. This hyperplane ENI allows uses the subnet and security group configuration of your MSK cluster, not your Lambda function.

The following console steps add an Amazon MSK cluster as a trigger for your Lambda function. Under the hood, this creates an event source mapping resource.

To add an Amazon MSK trigger to your Lambda function (console)

1. Open the Function page of the Lambda console.
2. Choose the name of the Lambda function you want to add an Amazon MSK trigger to.
3. Under **Function overview**, choose **Add trigger**.
4. Under **Trigger configuration**, choose **MSK**.
5. To specify your Kafka cluster details, do the following:
    1. For **MSK cluster**, select your cluster.
    2. For **Topic name**, enter the name of the Kafka topic to consume messages from.
    3. For **Consumer group ID**, enter the ID of a Kafka consumer group to join, if applicable. For more information, see the section called “Consumer group ID”.
6. For **Cluster authentication**, make the necessary configurations. For more information about cluster authentication, see the section called “Cluster authentication”.
    - Toggle on **Use authentication** if you want Lambda to perform authentication with your MSK cluster when establishing a connection. Authentication is recommended.
    - If you use authentication, for **Authentication method**, choose the authentication method to use.
    - If you use authentication, for **Secrets Manager key**, choose the Secrets Manager key that contains the authentication credentials needed to access your cluster.

7. Under **Event poller configuration**, make the necessary configurations.
   - Choose **Activate trigger** to enable the trigger immediately after creation.
   - Choose whether you want to **Configure provisioned mode** for your event source mapping. For more information, see the section called "Event poller scaling".
     - If you configure provisioned mode, enter a value for **Minimum event pollers**, a value for **Maximum event pollers**, and an optional value for PollerGroupName to specify grouping of multiple ESMs within the same event source VPC.
   - For **Starting position**, choose how you want Lambda to start reading from your stream. For more information, see the section called "Polling and stream positions".
8. Under **Batching**, make the necessary configurations. For more information about batching, see the section called "Batching behavior".
   1. For **Batch size**, enter the maximum number of messages to receive in a single batch.
   2. For **Batch window**, enter the maximum number of seconds that Lambda spends gathering records before invoking the function.
9. Under **Filtering**, make the necessary configurations. For more information about filtering, see the section called "Event filtering".
   - For **Filter criteria**, add filter criteria definitions to determine whether or not to process an event.
10. Under **Failure handling**, make the necessary configurations. For more information about failure handling, see the section called "Retain failed invocations".
    - For **On-failure destination**, specify the ARN of your on-failure destination.
11. For **Tags**, enter the tags to associate with this event source mapping.
12. To create the trigger, choose **Add**.

You can also create the event source mapping using the AWS CLI with the create-event-source-mapping command. The following example creates an event source mapping to map the Lambda function `my-msk-function` to the `AWSKafkaTopic` topic, starting from the `LATEST` message. This command also uses the SourceAccessConfiguration object to instruct Lambda to use SASL/SCRAM authentication when connecting to the cluster.

```
aws lambda create-event-source-mapping \
```

```
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

If the cluster uses mTLS authentication, include a SourceAccessConfiguration object that specifies CLIENT_CERTIFICATE_TLS_AUTH and a Secrets Manager key ARN. This is shown in the following command:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

When the cluster uses IAM authentication, you don't need a SourceAccessConfiguration object. This is shown in the following command:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Creating cross-account event source mappings in Lambda

You can use multi-VPC private connectivity to connect a Lambda function to a provisioned MSK cluster in a different AWS account. Multi-VPC connectivity uses AWS PrivateLink, which keeps all traffic within the AWS network.

> **Note**
>
> You can't create cross-account event source mappings for serverless MSK clusters.

To create a cross-account event source mapping, you must first configure multi-VPC connectivity for the MSK cluster. When you create the event source mapping, use the managed VPC connection ARN instead of the cluster ARN, as shown in the following examples. The CreateEventSourceMapping operation also differs depending on which authentication type the MSK cluster uses.

Example— Create cross-account event source mapping for cluster that uses IAM authentication

When the cluster uses IAM role-based authentication, you don't need a SourceAccessConfiguration object. Example:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example— Create cross-account event source mapping for cluster that uses SASL/SCRAM authentication

If the cluster uses SASL/SCRAM authentication, you must include a SourceAccessConfiguration object that specifies SASL_SCRAM_512_AUTH and a Secrets Manager secret ARN.

There are two ways to use secrets for cross-account Amazon MSK event source mappings with SASL/SCRAM authentication:

- Create a secret in the Lambda function account and sync it with the cluster secret. Create a rotation to keep the two secrets in sync. This option allows you to control the secret from the function account.
- Use the secret that's associated with the MSK cluster. This secret must allow cross-account access to the Lambda function account. For more information, see Permissions to AWS Secrets Manager secrets for users in a different account.

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
```

```
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Example— Create cross-account event source mapping for cluster that uses mTLS authentication

If the cluster uses mTLS authentication, you must include a SourceAccessConfiguration object that specifies `CLIENT_CERTIFICATE_TLS_AUTH` and a Secrets Manager secret ARN. The secret can be stored in the cluster account or the Lambda function account.

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

All Amazon MSK event source configuration parameters in Lambda

All Lambda event source types share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to Amazon MSK, as shown in the following table.

Parameter	Required	Default	Notes
AmazonManagedKafkaEventSourceConfig	N	Contains the ConsumerGroupId field, which defaults to a unique value.	Can set only on Create
BatchSize	N	100	Maximum: 10,000
DestinationConfig	N	N/A	the section called "Retain failed invocations"
Enabled	N	True	

Parameter	Required	Default	Notes
BisectBatchOnFunctionError	N	False	the section called "Retry configurations"
FunctionResponseTypes	N	N/A	the section called "Retry configurations"
MaximumRecordAgeInSeconds	N	-1 (infinite)	the section called "Retry configurations"
MaximumRetryAttempts	N	-1 (infinite)	the section called "Retry configurations"
EventSourceArn	Y	N/A	Can set only on Create
FilterCriteria	N	N/A	Control which events Lambda sends to your function
FunctionName	Y	N/A	
KMSKeyArn	N	N/A	the section called "Encryption of filter criteria"
MaximumBatchingWindowInSeconds	N	500 ms	Batching behavior

Parameter	Required	Default	Notes
ProvisionedPollersConfig	N	`MinimumPollers` : default value of 1 if not specified `MaximumPollers` : default value of 200 if not specified `PollerGroupName` : N/A	the section called "Provisioned mode"
SourceAccessConfigurations	N	No credentials	SASL/SCRAM or CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) authentication credentials for your event source
StartingPosition	Y	N/A	AT_TIMESTAMP, TRIM_HORIZON, or LATEST Can set only on Create
StartingPositionTimestamp	N	N/A	Required if StartingPosition is set to AT_TIMESTAMP
Tags	N	N/A	the section called "Event source mapping tags"

Parameter	Required	Default	Notes
Topics	Y	N/A	Kafka topic name Can set only on Create

Note

When you specify a `PollerGroupName`, multiple ESMs within the same Amazon VPC can share Event Poller Unit (EPU) capacity. You can use this option to optimize Provisioned mode costs for your ESMs. Requirements for ESM grouping:

- ESMs must be within the same Amazon VPC
- Maximum of 100 ESMs per poller group
- Aggregate maximum pollers across all ESMs in a group cannot exceed 2000

You can update the `PollerGroupName` to move an ESM to a different group, or remove an ESM from a group by setting `PollerGroupName` to an empty string ("").

Tutorial: Using an Amazon MSK event source mapping to invoke a Lambda function

In this tutorial, you will perform the following:

- Create a Lambda function in the same AWS account as an existing Amazon MSK cluster.
- Configure networking and authentication for Lambda to communicate with Amazon MSK.
- Set up a Lambda Amazon MSK event source mapping, which runs your Lambda function when events show up in the topic.

After you are finished with these steps, when events are sent to Amazon MSK, you will be able to set up a Lambda function to process those events automatically with your own custom Lambda code.

What can you do with this feature?

Example solution: Use an MSK event source mapping to deliver live scores to your customers.

Consider the following scenario: Your company hosts a web application where your customers can view information about live events, such as sports games. Information updates from the game are provided to your team through a Kafka topic on Amazon MSK. You want to design a solution that consumes updates from the MSK topic to provide an updated view of the live event to customers inside an application you develop. You have decided on the following design approach: Your client applications will communicate with a serverless backend hosted in AWS. Clients will connect over websocket sessions using the Amazon API Gateway WebSocket API.

In this solution, you need a component that reads MSK events, performs some custom logic to prepare those events for the application layer and then forwards that information to the API Gateway API. You can implement this component with AWS Lambda, by providing your custom logic in a Lambda function, then calling it with a AWS Lambda Amazon MSK event source mapping.

For more information about implementing solutions using the Amazon API Gateway WebSocket API, see WebSocket API tutorials in the API Gateway documentation.

Prerequisites

An AWS account with the following preconfigured resources:

To fulfill these prerequisites, we recommend following Getting started using Amazon MSK in the Amazon MSK documentation.

- An Amazon MSK cluster. See Create an Amazon MSK cluster in *Getting started using Amazon MSK*.
- The following configuration:
  - Ensure **IAM role-based authentication** is **Enabled** in your cluster security settings. This improves your security by limiting your Lambda function to only access the Amazon MSK resources needed. This is enabled by default on new Amazon MSK clusters.
  - Ensure **Public access** is off in your cluster networking settings. Restricting your Amazon MSK cluster's access to the internet improves your security by limiting how many intermediaries handle your data. This is enabled by default on new Amazon MSK clusters.
- A Kafka topic in your Amazon MSK cluster to use for this solution. See Create a topic in *Getting started using Amazon MSK*.
- A Kafka admin host set up to retrieve information from your Kafka cluster and send Kafka events to your topic for testing, such as an Amazon EC2 instance with the Kafka admin CLI and Amazon MSK IAM library installed. See Create a client machine in *Getting started using Amazon MSK*.

Once you have set up these resources, gather the following information from your AWS account to confirm that you are ready to continue.

- The name of your Amazon MSK cluster. You can find this information in the Amazon MSK console.
- The cluster UUID, part of the ARN for your Amazon MSK cluster, which you can find in the Amazon MSK console. Follow the procedures in Listing clusters in the Amazon MSK documentation to find this information.
- The security groups associated with your Amazon MSK cluster. You can find this information in the Amazon MSK console. In the following steps, refer to these as your *clusterSecurityGroups*.
- The id of the Amazon VPC containing your Amazon MSK cluster. You can find this information by identifying subnets associated with your Amazon MSK cluster in the Amazon MSK console, then identifying the Amazon VPC associated with the subnet in the Amazon VPC Console.
- The name of the Kafka topic used in your solution. You can find this information by calling your Amazon MSK cluster with the Kafka `topics` CLI from your Kafka admin host. For more information about the topics CLI, see Adding and removing topics in the Kafka documentation.
- The name of a consumer group for your Kafka topic, suitable for use by your Lambda function. This group can be created automatically by Lambda, so you don't need to create it with the Kafka CLI. If you do need to manage your consumer groups, to learn more about the consumer-groups CLI, see Managing Consumer Groups in the Kafka documentation.

The following permissions in your AWS account:

- Permission to create and manage a Lambda function.
- Permission to create IAM policies and associate them with your Lambda function.
- Permission to create Amazon VPC endpoints and alter networking configuration in the Amazon VPC hosting your Amazon MSK cluster.

Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

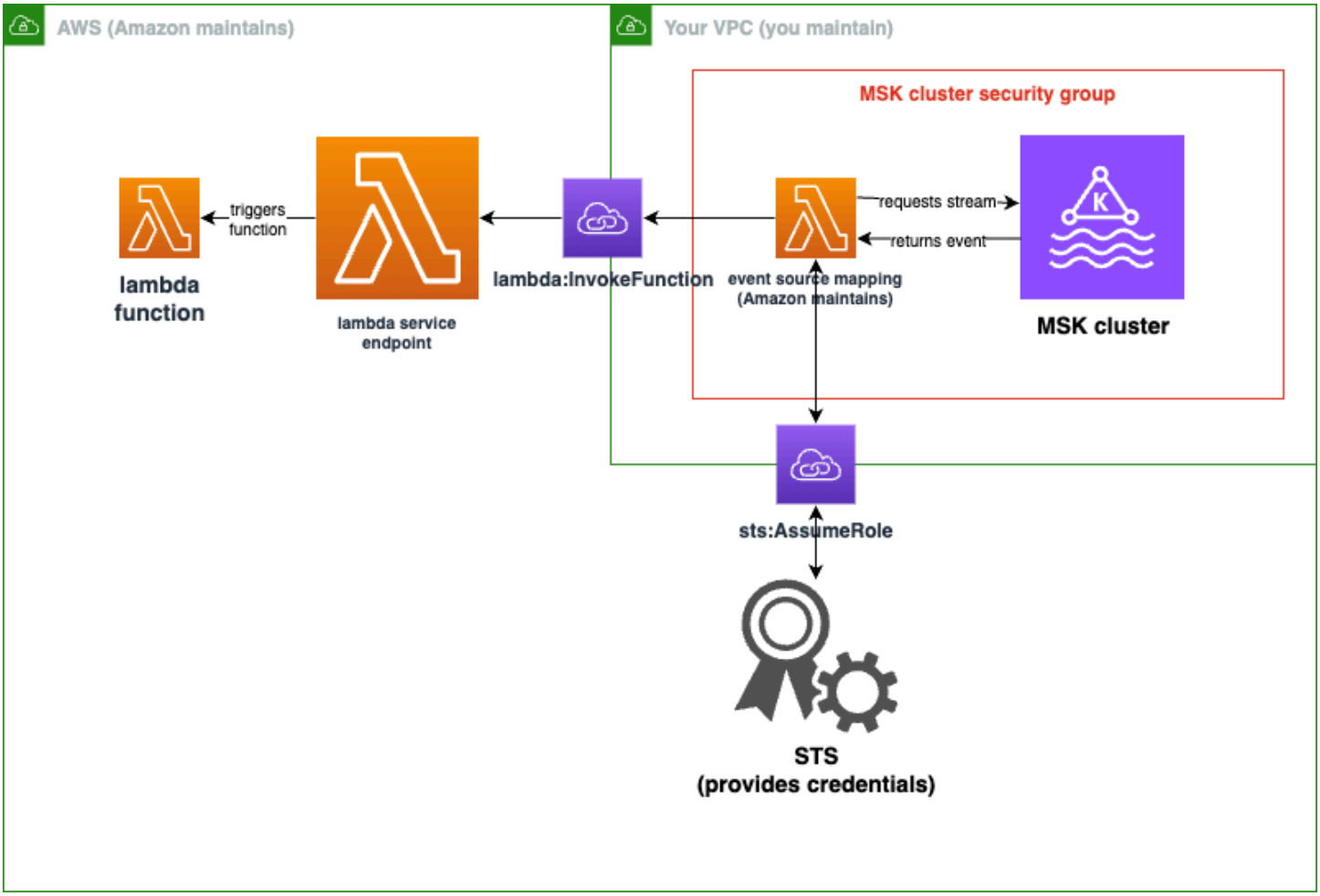
The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Configure network connectivity for Lambda to communicate with Amazon MSK

Use AWS PrivateLink to connect Lambda and Amazon MSK. You can do so by creating interface Amazon VPC endpoints in the Amazon VPC console. For more information about networking configuration, see the section called “Cluster and network setup”.

When a Amazon MSK event source mapping runs on the behalf of a Lambda function, it assumes the Lambda function’s execution role. This IAM role authorizes the mapping to access resources secured by IAM, such as your Amazon MSK cluster. Although the components share an execution role, the Amazon MSK mapping and your Lambda function have separate connectivity requirements for their respective tasks, as shown in the following diagram.



Your event source mapping belongs to your Amazon MSK cluster security group. In this networking step, create Amazon VPC endpoints from your Amazon MSK cluster VPC to connect the event source mapping to the Lambda and STS services. Secure these endpoints to accept traffic from your Amazon MSK cluster security group. Then, adjust the Amazon MSK cluster security groups to allow the event source mapping to communicate with the Amazon MSK cluster.

You can configure the following steps using the AWS Management Console.

To configure interface Amazon VPC endpoints to connect Lambda and Amazon MSK

1. Create a security group for your interface Amazon VPC endpoints, *`endpointSecurityGroup`*, that allows inbound TCP traffic on 443 from *`clusterSecurityGroups`*. Follow the procedure in Create a security group in the Amazon EC2 documentation to create a security group. Then, follow the procedure in Add rules to a security group in the Amazon EC2 documentation to add appropriate rules.

   Create a security group with the following information:

When adding your inbound rules, create a rule for each security group in *clusterSecurityGroups*. For each rule:

- For **Type**, select **HTTPS**.
- For **Source**, select one of *clusterSecurityGroups*.

2. Create an endpoint connecting the Lambda service to the Amazon VPC containing your Amazon MSK cluster. Follow the procedure in Create an interface endpoint.

   Create an interface endpoint with the following information:

   - For **Service name**, select `com.amazonaws.`*regionName*`.lambda`, where *regionName* hosts your Lambda function.
   - For **VPC**, select the Amazon VPC containing your Amazon MSK cluster.
   - For **Security groups**, select *endpointSecurityGroup*, which you created earlier.
   - For **Subnets**, select the subnets that host your Amazon MSK cluster.
   - For **Policy**, provide the following policy document, which secures the endpoint for use by the Lambda service principal for the `lambda:InvokeFunction` action.

   ```
   {
       "Statement": [
           {
               "Action": "lambda:InvokeFunction",
               "Effect": "Allow",
               "Principal": {
                   "Service": [
                       "lambda.amazonaws.com"
                   ]
               },
               "Resource": "*"
           }
       ]
   }
   ```

   - Ensure **Enable DNS name** remains set.

3. Create an endpoint connecting the AWS STS service to the Amazon VPC containing your Amazon MSK cluster. Follow the procedure in Create an interface endpoint.

   Create an interface endpoint with the following information:

- For **Service name**, select AWS STS.
- For **VPC**, select the Amazon VPC containing your Amazon MSK cluster.
- For **Security groups**, select *endpointSecurityGroup*.
- For **Subnets**, select the subnets that host your Amazon MSK cluster.
- For **Policy**, provide the following policy document, which secures the endpoint for use by the Lambda service principal for the `sts:AssumeRole` action.

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

- Ensure **Enable DNS name** remains set.

4. For each security group associated with your Amazon MSK cluster, that is, in *clusterSecurityGroups*, allow the following:

   - Allow all inbound and outbound TCP traffic on 9098 to all of *clusterSecurityGroups*, including within itself.
   - Allow all outbound TCP traffic on 443.

   Some of this traffic is allowed by default security group rules, so if your cluster is attached to a single security group, and that group has default rules, additional rules are not necessary. To adjust security group rules, follow the procedures in Add rules to a security group in the Amazon EC2 documentation.

   Add rules to your security groups with the following information:

   - For each inbound rule or outbound rule for port 9098, provide

- For **Type**, select **Custom TCP**.
- For **Port range**, provide 9098.
- For **Source**, provide one of *clusterSecurityGroups*.

- For each inbound rule for port 443, for **Type**, select **HTTPS**.

Create an IAM role for Lambda to read from your Amazon MSK topic

Identify the auth requirements for Lambda to read from your Amazon MSK topic, then define them in a policy. Create a role, *lambdaAuthRole*, that authorizes Lambda to use those permissions. Authorize actions on your Amazon MSK cluster using `kafka-cluster` IAM actions. Then, authorize Lambda to perform Amazon MSK `kafka` and Amazon EC2 actions needed to discover and connect to your Amazon MSK cluster, as well as CloudWatch actions so Lambda can log what it has done.

To describe the auth requirements for Lambda to read from Amazon MSK

1. Write an IAM policy document (a JSON document), *clusterAuthPolicy*, that allows Lambda to read from your Kafka topic in your Amazon MSK cluster using your Kafka consumer group. Lambda requires a Kafka consumer group to be set when reading.

   Alter the following template to align with your prerequisites:

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
```

```
                "arn:aws:kafka:us-
east-1:111122223333:cluster/mskClusterName/cluster-uuid",
                "arn:aws:kafka:us-
east-1:111122223333:topic/mskClusterName/cluster-uuid/mskTopicName",
                "arn:aws:kafka:us-
east-1:111122223333:group/mskClusterName/cluster-uuid/mskGroupName"
            ]
        }
    ]
}
```

For more information, consult the section called “Configure permissions”. When writing your policy:

- Replace *us-east-1* and *111122223333* with the AWS Region and AWS account of your Amazon MSK cluster.
- For *mskClusterName*, provide the name of your Amazon MSK cluster.
- For *cluster-uuid*, provide the UUID in the ARN for your Amazon MSK cluster.
- For *mskTopicName*, provide the name of your Kafka topic.
- For *mskGroupName*, provide the name of your Kafka consumer group.

2. Identify the Amazon MSK, Amazon EC2 and CloudWatch permissions required for Lambda to discover and connect your Amazon MSK cluster, and log those events.

   The `AWSLambdaMSKExecutionRole` managed policy permissively defines the required permissions. Use it in the following steps.

   In a production environment, assess `AWSLambdaMSKExecutionRole` to restrict your execution role policy based on the principle of least privilege, then write a policy for your role that replaces this managed policy.

For details about the IAM policy language, see the IAM documentation.

Now that you have written your policy document, create an IAM policy so you can attach it to your role. You can do this using the console with the following procedure.

To create an IAM policy from your policy document

1. Sign in to the AWS Management Console and open the IAM console at https://console.aws.amazon.com/iam/.
2. In the navigation pane on the left, choose **Policies**.
3. Choose **Create policy**.
4. In the **Policy editor** section, choose the **JSON** option.
5. Paste *clusterAuthPolicy*.
6. When you are finished adding permissions to the policy, choose **Next**.
7. On the **Review and create** page, type a **Policy Name** and a **Description** (optional) for the policy that you are creating. Review **Permissions defined in this policy** to see the permissions that are granted by your policy.
8. Choose **Create policy** to save your new policy.

For more information, see Creating IAM policies in the IAM documentation.

Now that you have appropriate IAM policies, create a role and attach them to it. You can do this using the console with the following procedure.

To create an execution role in the IAM console

1. Open the Roles page in the IAM console.
2. Choose **Create role**.
3. Under **Trusted entity type**, choose **AWS service**.
4. Under **Use case**, choose **Lambda**.
5. Choose **Next**.
6. Select the following policies:
   - *clusterAuthPolicy*
   - `AWSLambdaMSKExecutionRole`
7. Choose **Next**.
8. For **Role name**, enter *lambdaAuthRole* and then choose **Create role**.

For more information, see the section called “Execution role (permissions for functions to access other resources)”.

Create a Lambda function to read from your Amazon MSK topic

Create a Lambda function configured to use your IAM role. You can create your Lambda function using the console.

To create a Lambda function using your auth configuration

1. Open the Lambda console and select **Create function** from the header.
2. Select **Author from scratch**.
3. For **Function name**, provide an appropriate name of your choice.
4. For **Runtime**, choose the **Latest supported** version of `Node.js` to use the code provided in this tutorial.
5. Choose **Change default execution role**.
6. Select **Use an existing role**.
7. For **Existing role**, select *`lambdaAuthRole`*.

In a production environment, you usually need to add further policies to the execution role for your Lambda function to meaningfully process your Amazon MSK events. For more information on adding policies to your role, see Add or remove identity permissions in the IAM documentation.

Create an event source mapping to your Lambda function

Your Amazon MSK event source mapping provides the Lambda service the information necessary to invoke your Lambda when appropriate Amazon MSK events occur. You can create a Amazon MSK mapping using the console. Create a Lambda trigger, then the event source mapping is automatically set up.

To create a Lambda trigger (and event source mapping)

1. Navigate to your Lambda function's overview page.
2. In the function overview section, choose **Add trigger** on the bottom left.
3. In the **Select a source** dropdown, select **Amazon MSK**.
4. Don't set **authentication**.
5. For **MSK cluster**, select your cluster's name.
6. For **Batch size**, enter 1. This step makes this feature easier to test, and is not an ideal value in production.

7. For **Topic name**, provide the name of your Kafka topic.
8. For **Consumer group ID**, provide the id of your Kafka consumer group.

Update your Lambda function to read your streaming data

Lambda provides information about Kafka events through the event method parameter. For an example structure of a Amazon MSK event, see the section called “Example event”. After you understand how to interpret Lambda forwarded Amazon MSK events, you can alter your Lambda function code to use the information they provide.

Provide the following code to your Lambda function to log the contents of a Lambda Amazon MSK event for testing purposes:

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using .NET.

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{
```

```
    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);

                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Go.

```
package main

import (
 "encoding/base64"
```

```
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)

   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;
```

```
import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
                System.out.println("Record: " + record);

                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using JavaScript.

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
```

```
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

Consuming an Amazon MSK event with Lambda using TypeScript.

```
import { MSKEvent, Context } from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "msk-handler-sample",
});

export const handler = async (
  event: MSKEvent,
  context: Context
): Promise<void> => {
  for (const [topic, topicRecords] of Object.entries(event.records)) {
    logger.info(`Processing key: ${topic}`);

    // Process each record in the partition
    for (const record of topicRecords) {
      try {
        // Decode the message value from base64
        const decodedMessage = Buffer.from(record.value, 'base64').toString();

        logger.info({
          message: decodedMessage
        });
      }
      catch (error) {
        logger.error('Error processing event', { error });
        throw error;
      }
```

```
    };
  }
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using PHP.

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
```

```
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();

        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Python.

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Ruby.

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
```

```
    end
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Rust.

```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// Pre-Requisites:
/// 1. Install Cargo Lambda - see https://www.cargo-lambda.info/guide/getting-
started.html
/// 2. Add packages tracing, tracing-subscriber, serde_json, base64
///
/// This is the main body for the function.
/// Write your code inside it.
/// There are some code example in the following URLs:
/// - https://github.com/awslabs/aws-lambda-rust-runtime/tree/main/examples
/// - https://github.com/aws-samples/serverless-rust-demo/

async fn function_handler(event: LambdaEvent<KafkaEvent>) -> Result<Value, Error>
 {

    let payload = event.payload.records;

    for (_name, records) in payload.iter() {

        for record in records {
```

```
            let record_text = record.value.as_ref().ok_or("Value is None")?;
            info!("Record: {}", &record_text);

            // perform Base64 decoding
            let record_bytes = BASE64_STANDARD.decode(record_text)?;
            let message = std::str::from_utf8(&record_bytes)?;

            info!("Message: {}", message);
           }

    }

    Ok(().into())
}

#[tokio::main]
async fn main() -> Result<(), Error> {

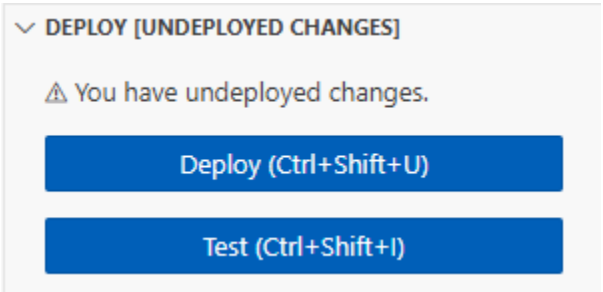
    // required to enable CloudWatch error logging by the runtime
    tracing::init_default_subscriber();
    info!("Setup CW subscriber!");

    run(service_fn(function_handler)).await
}
```

You can provide function code to your Lambda using the console.

To update function code using the console code editor

1. Open the Functions page of the Lambda console and select your function.
2. Select the **Code** tab.
3. In the **Code source** pane, select your source code file and edit it in the integrated code editor.
4. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Test your Lambda function to verify it is connected to your Amazon MSK topic

You can now verify whether or not your Lambda is being invoked by the event source by inspecting CloudWatch event logs.

To verify whether your Lambda function is being invoked

1. Use your Kafka admin host to generate Kafka events using the `kafka-console-producer` CLI. For more information, see Write some events into the topic in the Kafka documentation. Send enough events to fill up the batch defined by batch size for your event source mapping defined in the previous step, or Lambda will wait for more information to invoke.
2. If your function runs, Lambda writes what happened to CloudWatch. In the console, navigate to your Lambda function's detail page.
3. Select the **Configuration** tab.
4. From the sidebar, select **Monitoring and operations tools**.
5. Identify the **CloudWatch log group** under **Logging configuration**. The log group should start with `/aws/lambda`. Choose the link to the log group.
6. In the CloudWatch console, inspect the **Log events** for the log events Lambda has sent to the log stream. Identify if there are log events containing the message from your Kafka event, as in the following image. If there are, you have successfully connected a Lambda function to Amazon MSK with a Lambda event source mapping.



Using Lambda with self-managed Apache Kafka

This topic describes how to use Lambda with a self-managed Kafka cluster. In AWS terminology, a self-managed cluster includes non-AWS hosted Kafka clusters. For example, you can host your Kafka cluster with a cloud provider such as Confluent Cloud or Redpanda.

This chapter explains how to use a self-managed Apache Kafka cluster as an event source for your Lambda function. The general process for integrating self-managed Apache Kafka with Lambda involves the following steps:

1. **Cluster and network setup** – First, set up your self-managed Apache Kafka cluster with the correct networking configuration to allow Lambda to access your cluster.
2. **Event source mapping setup** – Then, create the event source mapping resource that Lambda needs to securely connect your Apache Kafka cluster to your function.
3. **Function and permissions setup** – Finally, ensure that your function is correctly set up, and has the necessary permissions in its execution role.

Apache Kafka as an event source operates similarly to using Amazon Simple Queue Service (Amazon SQS) or Amazon Kinesis. Lambda internally polls for new messages from the event source and then synchronously invokes the target Lambda function. Lambda reads the messages in batches and provides these to your function as an event payload. The maximum batch size is configurable (the default is 100 messages). For more information, see the section called "Batching behavior".

To optimize the throughput of your self-managed Apache Kafka event source mapping, configure provisioned mode. In provisioned mode, you can define the minimum and maximum number of event pollers allocated to your event source mapping. This can improve the ability of your event source mapping to handle unexpected message spikes. For more information, see provisioned mode.

> **Warning**
>
> Lambda event source mappings process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

For Kafka-based event sources, Lambda supports processing control parameters, such as batching windows and batch size. For more information, see Batching behavior.

For an example of how to use self-managed Kafka as an event source, see Using self-hosted Apache Kafka as an event source for AWS Lambda on the AWS Compute Blog.

Topics

- Example event
- Configuring your self-managed Apache Kafka cluster and network for Lambda
- Configuring Lambda execution role permissions
- Configuring self-managed Apache Kafka event sources for Lambda

Example event

Lambda sends the batch of messages in the event parameter when it invokes your Lambda function. The event payload contains an array of messages. Each array item contains details of the Kafka topic and Kafka partition identifier, together with a timestamp and a base64-encoded message.


```
{
   "eventSource": "SelfManagedKafka",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
```

```
                    100,
                    101,
                    114,
                    86,
                    97,
                    108,
                    117,
                    101
                ]
            }
        ]
    }
  ]
 }
}
```

Configuring your self-managed Apache Kafka cluster and network for Lambda

To connect your Lambda function to your self-managed Apache Kafka cluster, you need to correctly configure your cluster and the network it resides in. This page describes how to configure your cluster and network. If your cluster and network are already configured properly, see the section called "Configure event source" to configure the event source mapping.

Topics

- Self-managed Apache Kafka cluster setup
- Configure network security

Self-managed Apache Kafka cluster setup

You can host your self-managed Apache Kafka cluster with cloud providers such as Confluent Cloud or Redpanda, or run it on your own infrastructure. Ensure that your cluster is properly configured and accessible from the network where your Lambda event source mapping will connect.

Configure network security

To give Lambda full access to self-managed Apache Kafka through your event source mapping, either your cluster must use a public endpoint (public IP address), or you must provide access to the Amazon VPC you created the cluster in.

When you use self-managed Apache Kafka with Lambda, create AWS PrivateLink VPC endpoints that provide your function access to the resources in your Amazon VPC.

> **Note**
>
> AWS PrivateLink VPC endpoints are required for functions with event source mappings that use the default (on-demand) mode for event pollers. If your event source mapping uses provisioned mode, you don't need to configure AWS PrivateLink VPC endpoints.

Create an endpoint to provide access to the following resources:

- Lambda — Create an endpoint for the Lambda service principal.
- AWS STS — Create an endpoint for the AWS STS in order for a service principal to assume a role on your behalf.
- Secrets Manager — If your cluster uses Secrets Manager to store credentials, create an endpoint for Secrets Manager.

Alternatively, configure a NAT gateway on each public subnet in the Amazon VPC. For more information, see the section called “Internet access for VPC functions”.

When you create an event source mapping for self-managed Apache Kafka, Lambda checks whether Elastic Network Interfaces (ENIs) are already present for the subnets and security groups configured for your Amazon VPC. If Lambda finds existing ENIs, it attempts to re-use them. Otherwise, Lambda creates new ENIs to connect to the event source and invoke your function.

> **Note**
>
> Lambda functions always run inside VPCs owned by the Lambda service. Your function's VPC configuration does not affect the event source mapping. Only the networking configuration of the event source's determines how Lambda connects to your event source.

Configure the security groups for the Amazon VPC containing your cluster. By default, self-managed Apache Kafka uses the following ports: `9092`.

- Inbound rules – Allow all traffic on the default broker port for the security group associated with your event source. Alternatively, you can use a self-referencing security group rule to allow access from instances within the same security group.

- Outbound rules – Allow all traffic on port 443 for external destinations if your function needs to communicate with AWS services. Alternatively, you can also use a self-referencing security group rule to limit access to the broker if you don't need to communicate with other AWS services.
- Amazon VPC endpoint inbound rules — If you are using an Amazon VPC endpoint, the security group associated with your Amazon VPC endpoint must allow inbound traffic on port 443 from the cluster security group.

If your cluster uses authentication, you can also restrict the endpoint policy for the Secrets Manager endpoint. To call the Secrets Manager API, Lambda uses your function role, not the Lambda service principal.

Example VPC endpoint policy — Secrets Manager endpoint

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
              "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
          }
      ]
  }
```

When you use Amazon VPC endpoints, AWS routes your API calls to invoke your function using the endpoint's Elastic Network Interface (ENI). The Lambda service principal needs to call `lambda:InvokeFunction` on any roles and functions that use those ENIs.

By default, Amazon VPC endpoints have open IAM policies that allow broad access to resources. Best practice is to restrict these policies to perform the needed actions using that endpoint. To ensure that your event source mapping is able to invoke your Lambda function, the VPC endpoint policy must allow the Lambda service principal to call `sts:AssumeRole` and `lambda:InvokeFunction`. Restricting your VPC endpoint policies to allow only API calls

originating within your organization prevents the event source mapping from functioning properly, so `"Resource": "*"` is required in these policies.

The following example VPC endpoint policies show how to grant the required access to the Lambda service principal for the AWS STS and Lambda endpoints.

Example VPC Endpoint policy — AWS STS endpoint

```
{
      "Statement": [
          {
              "Action": "sts:AssumeRole",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
    }
```

Example VPC Endpoint policy — Lambda endpoint

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Configuring Lambda execution role permissions

In addition to accessing your self-managed Kafka cluster, your Lambda function needs permissions to perform various API actions. You add these permissions to the function's execution role. If your users need access to any API actions, add the required permissions to the identity policy for the AWS Identity and Access Management (IAM) user or role.

Topics

- Required Lambda function permissions
- Optional Lambda function permissions
- Adding permissions to your execution role
- Granting users access with an IAM policy

Required Lambda function permissions

To create and store logs in a log group in Amazon CloudWatch Logs, your Lambda function must have the following permissions in its execution role:

- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

Optional Lambda function permissions

Your Lambda function might also need permissions to:

- Describe your Secrets Manager secret.
- Access your AWS Key Management Service (AWS KMS) customer managed key.
- Access your Amazon VPC.
- Send records of failed invocations to a destination.

Secrets Manager and AWS KMS permissions

Depending on the type of access control that you're configuring for your Kafka brokers, your Lambda function might need permission to access your Secrets Manager secret or to decrypt your

AWS KMS customer managed key. To access these resources, your function's execution role must have the following permissions:

- secretsmanager:GetSecretValue
- kms:Decrypt

VPC permissions

If only users within a VPC can access your self-managed Apache Kafka cluster, your Lambda function must have permission to access your Amazon VPC resources. These resources include your VPC, subnets, security groups, and network interfaces. To access these resources, your function's execution role must have the following permissions:

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

Adding permissions to your execution role

To access other AWS services that your self-managed Apache Kafka cluster uses, Lambda uses the permissions policies that you define in your Lambda function's execution role.

By default, Lambda is not permitted to perform the required or optional actions for a self-managed Apache Kafka cluster. You must create and define these actions in an IAM trust policy for your execution role. This example shows how you might create a policy that allows Lambda to access your Amazon VPC resources.

JSON

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
```

```
            "Action":[
               "ec2:CreateNetworkInterface",
               "ec2:DescribeNetworkInterfaces",
               "ec2:DescribeVpcs",
               "ec2:DeleteNetworkInterface",
               "ec2:DescribeSubnets",
               "ec2:DescribeSecurityGroups"
            ],
            "Resource":"*"
         }
      ]
   }
```

Granting users access with an IAM policy

By default, users and roles don't have permission to perform event source API operations. To grant access to users in your organization or account, you create or update an identity-based policy. For more information, see Controlling access to AWS resources using policies in the *IAM User Guide*.

For troubleshooting authentication and authorization errors, see the section called "Troubleshooting".

Configuring self-managed Apache Kafka event sources for Lambda

To use a self-managed Apache Kafka cluster as an event source for your Lambda function, you create an event source mapping that connects the two resources. This page describes how to create an event source mapping for self-managed Apache Kafka.

This page assumes that you've already properly configured your Kafka cluster and the network it resides in. If you need to set up your cluster or network, see the section called "Cluster and network setup".

Topics

- Using a self-managed Apache Kafka cluster as an event source
- Configuring cluster authentication methods in Lambda
- Creating a Lambda event source mapping for a self-managed Apache Kafka event source
- All self-managed Apache Kafka event source configuration parameters in Lambda

Using a self-managed Apache Kafka cluster as an event source

When you add your Apache Kafka or Amazon MSK cluster as a trigger for your Lambda function, the cluster is used as an event source.

Lambda reads event data from the Kafka topics that you specify as `Topics` in a CreateEventSourceMapping request, based on the starting position that you specify. After successful processing, your Kafka topic is committed to your Kafka cluster.

Lambda reads messages sequentially for each Kafka topic partition. A single Lambda payload can contain messages from multiple partitions. When more records are available, Lambda continues processing records in batches, based on the BatchSize value that you specify in a CreateEventSourceMapping request, until your function catches up with the topic.

After Lambda processes each batch, it commits the offsets of the messages in that batch. If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of messages until processing succeeds or the messages expire. You can send records that fail all retry attempts to an on-failure destination for later processing.

> **Note**
>
> While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, Amazon DocumentDB, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes.

Configuring cluster authentication methods in Lambda

Lambda supports several methods to authenticate with your self-managed Apache Kafka cluster. Make sure that you configure the Kafka cluster to use one of these supported authentication methods. For more information about Kafka security, see the Security section of the Kafka documentation.

SASL/SCRAM authentication

Lambda supports Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) authentication with Transport Layer Security (TLS) encryption (SASL_SSL). Lambda sends the encrypted credentials to authenticate with the cluster.

Lambda doesn't support SASL/SCRAM with plaintext (SASL_PLAINTEXT). For more information about SASL/SCRAM authentication, see RFC 5802.

Lambda also supports SASL/PLAIN authentication. Because this mechanism uses clear text credentials, the connection to the server must use TLS encryption to ensure that the credentials are protected.

For SASL authentication, you store the sign-in credentials as a secret in AWS Secrets Manager. For more information about using Secrets Manager, see Create an AWS Secrets Manager secret in the *AWS Secrets Manager User Guide*.

> **Important**
>
> To use Secrets Manager for authentication, secrets must be stored in the same AWS region as your Lambda function.

Mutual TLS authentication

Mutual TLS (mTLS) provides two-way authentication between the client and server. The client sends a certificate to the server for the server to verify the client, and the server sends a certificate to the client for the client to verify the server.

In self-managed Apache Kafka, Lambda acts as the client. You configure a client certificate (as a secret in Secrets Manager) to authenticate Lambda with your Kafka brokers. The client certificate must be signed by a CA in the server's trust store.

The Kafka cluster sends a server certificate to Lambda to authenticate the Kafka brokers with Lambda. The server certificate can be a public CA certificate or a private CA/self-signed certificate. The public CA certificate must be signed by a certificate authority (CA) that's in the Lambda trust store. For a private CA/self-signed certificate, you configure the server root CA certificate (as a secret in Secrets Manager). Lambda uses the root certificate to verify the Kafka brokers.

For more information about mTLS, see Introducing mutual TLS authentication for Amazon MSK as an event source.

Configuring the client certificate secret

The CLIENT_CERTIFICATE_TLS_AUTH secret requires a certificate field and a private key field. For an encrypted private key, the secret requires a private key password. Both the certificate and private key must be in PEM format.

> **Note**
>
> Lambda supports the PBES1 (but not PBES2) private key encryption algorithms.

The certificate field must contain a list of certificates, beginning with the client certificate, followed by any intermediate certificates, and ending with the root certificate. Each certificate must start on a new line with the following structure:

```
-----BEGIN CERTIFICATE-----
            <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager supports secrets up to 65,536 bytes, which is enough space for long certificate chains.

The private key must be in PKCS #8 format, with the following structure:

```
-----BEGIN PRIVATE KEY-----
             <private key contents>
-----END PRIVATE KEY-----
```

For an encrypted private key, use the following structure:

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
              <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

The following example shows the contents of a secret for mTLS authentication using an encrypted private key. For an encrypted private key, include the private key password in the secret.

```
{"privateKeyPassword":"testpassword",
"certificate":"-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
```

```
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
"privateKey":"-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

Configuring the server root CA certificate secret

You create this secret if your Kafka brokers use TLS encryption with certificates signed by a private CA. You can use TLS encryption for VPC, SASL/SCRAM, SASL/PLAIN, or mTLS authentication.

The server root CA certificate secret requires a field that contains the Kafka broker's root CA certificate in PEM format. The following example shows the structure of the secret.

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
}
```

Creating a Lambda event source mapping for a self-managed Apache Kafka event source

To create an event source mapping, you can use the Lambda console, the AWS Command Line Interface (CLI), or an AWS SDK.

The following console steps add a self-managed Apache Kafka cluster as a trigger for your Lambda function. Under the hood, this creates an event source mapping resource.

Prerequisites

- A self-managed Apache Kafka cluster. Lambda supports Apache Kafka version 0.10.1.0 and later.
- An execution role with permission to access the AWS resources that your self-managed Kafka cluster uses.

Adding a self-managed Kafka cluster (console)

Follow these steps to add your self-managed Apache Kafka cluster and a Kafka topic as a trigger for your Lambda function.

To add an Apache Kafka trigger to your Lambda function (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of your Lambda function.
3. Under **Function overview**, choose **Add trigger**.
4. Under **Trigger configuration**, do the following:
    a. Choose the **Apache Kafka** trigger type.
    b. For **Bootstrap servers**, enter the host and port pair address of a Kafka broker in your cluster, and then choose **Add**. Repeat for each Kafka broker in the cluster.
    c. For **Topic name**, enter the name of the Kafka topic used to store records in the cluster.
    d. If you configure provisioned mode, enter a value for **Minimum event pollers**, a value for **Maximum event pollers**, and an optional value for PollerGroupName to specify grouping of multiple ESMs within the same event source VPC.
    e. (Optional) For **Batch size**, enter the maximum number of records to receive in a single batch.
    f. For **Batch window**, enter the maximum amount of seconds that Lambda spends gathering records before invoking the function.
    g. (Optional) For **Consumer group ID**, enter the ID of a Kafka consumer group to join.
    h. (Optional) For **Starting position**, choose **Latest** to start reading the stream from the latest record, **Trim horizon** to start at the earliest available record, or **At timestamp** to specify a timestamp to start reading from.
    i. (Optional) For **VPC**, choose the Amazon VPC for your Kafka cluster. Then, choose the **VPC subnets** and **VPC security groups**.

        This setting is required if only users within your VPC access your brokers.

    j. (Optional) For **Authentication**, choose **Add**, and then do the following:
        i. Choose the access or authentication protocol of the Kafka brokers in your cluster.
            - If your Kafka broker uses SASL/PLAIN authentication, choose **BASIC_AUTH**.

- If your broker uses SASL/SCRAM authentication, choose one of the **SASL_SCRAM** protocols.
- If you're configuring mTLS authentication, choose the **CLIENT_CERTIFICATE_TLS_AUTH** protocol.

ii. For SASL/SCRAM or mTLS authentication, choose the Secrets Manager secret key that contains the credentials for your Kafka cluster.

k. (Optional) For **Encryption**, choose the Secrets Manager secret containing the root CA certificate that your Kafka brokers use for TLS encryption, if your Kafka brokers use certificates signed by a private CA.

This setting applies to TLS encryption for SASL/SCRAM or SASL/PLAIN, and to mTLS authentication.

l. To create the trigger in a disabled state for testing (recommended), clear **Enable trigger**. Or, to enable the trigger immediately, select **Enable trigger**.

5. To create the trigger, choose **Add**.

Adding a self-managed Kafka cluster (AWS CLI)

Use the following example AWS CLI commands to create and view a self-managed Apache Kafka trigger for your Lambda function.

Using SASL/SCRAM

If Kafka users access your Kafka brokers over the internet, specify the Secrets Manager secret that you created for SASL/SCRAM authentication. The following example uses the create-event-source-mapping AWS CLI command to map a Lambda function named `my-kafka-function` to a Kafka topic named `AWSKafkaTopic`.

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-east-1:111122223333:secret:MyBrokerSecretName \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

Using a VPC

If only Kafka users within your VPC access your Kafka brokers, you must specify your VPC, subnets, and VPC security group. The following example uses the create-event-source-mapping AWS CLI command to map a Lambda function named `my-kafka-function` to a Kafka topic named `AWSKafkaTopic`.

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration '[{"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0011001100"}, {"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0022002200"}, {"Type": "VPC_SECURITY_GROUP", "URI":
 "security_group:sg-0123456789"}]' \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

Viewing the status using the AWS CLI

The following example uses the get-event-source-mapping AWS CLI command to describe the status of the event source mapping that you created.

```
aws lambda get-event-source-mapping
              --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

All self-managed Apache Kafka event source configuration parameters in Lambda

All Lambda event source types share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to self-managed Apache Kafka, as shown in the following table.

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
DestinationConfig	N	N/A	the section called “Retain failed invocations”
Enabled	N	True	

Parameter	Required	Default	Notes
FilterCriteria	N	N/A	Control which events Lambda sends to your function
FunctionName	Y	N/A	
KMSKeyArn	N	N/A	the section called "Encryption of filter criteria"
MaximumBatchingWindowInSeconds	N	500 ms	Batching behavior
ProvisionedPollerConfig	N	`MinimumPollers` : default value of 1 if not specified `MaximumPollers` : default value of 200 if not specified `PollerGroupName` : N/A	the section called "Provisioned mode"
SelfManagedEventSource	Y	N/A	List of Kafka Brokers. Can set only on Create
SelfManagedKafkaEventSourceConfig	N	Contains the ConsumerGroupId field which defaults to a unique value.	Can set only on Create

Parameter	Required	Default	Notes
SourceAccessConfigurations	N	No credentials	VPC information or authentication credentials for the cluster For SASL_PLAIN, set to BASIC_AUTH
StartingPosition	Y	N/A	AT_TIMESTAMP, TRIM_HORIZON, or LATEST Can set only on Create
StartingPositionTimestamp	N	N/A	Required if StartingPosition is set to AT_TIMESTAMP
Tags	N	N/A	the section called "Event source mapping tags"
Topics	Y	N/A	Topic name Can set only on Create

Note

When you specify a `PollerGroupName`, multiple ESMs within the same Amazon VPC can share Event Poller Unit (EPU) capacity. You can use this option to optimize Provisioned mode costs for your ESMs. Requirements for ESM grouping:

- ESMs must be within the same Amazon VPC
- Maximum of 100 ESMs per poller group

- Aggregate maximum pollers across all ESMs in a group cannot exceed 2000

You can update the `PollerGroupName` to move an ESM to a different group, or remove an ESM from a group by setting `PollerGroupName` to an empty string ("").

Apache Kafka event poller scaling modes in Lambda

You can choose between two modes of event poller scaling for Amazon MSK and self-managed Apache Kafka event source mappings:

- On-demand mode (default)
- Provisioned mode

On-demand mode (default)

When you initially create the Kafka event source, Lambda allocates a default number of event pollers to process all partitions in the Kafka topic. Lambda automatically scales up or down the number of event pollers based on message load.

In one-minute intervals, Lambda evaluates the offset lag of all the partitions in the topic. If the offset lag is too high, the partition is receiving messages faster than Lambda can process them. If necessary, Lambda adds or removes event pollers from the topic. This autoscaling process of adding or removing event pollers occurs within three minutes of evaluation.

If your target Lambda function is throttled, Lambda reduces the number of event pollers. This action reduces the workload on the function by reducing the number of messages that event pollers can retrieve and send to the function.

Provisioned mode

For workloads where you need to fine-tune the throughput of your event source mapping, you can use provisioned mode. In provisioned mode, you define minimum and maximum limits for the amount of provisioned event pollers. These provisioned event pollers are dedicated to your event source mapping, and can handle unexpected message spikes through responsive autoscaling. We recommend that you use provisioned mode for Kafka workloads that have strict performance requirements.

In Lambda, an event poller is a compute unit with throughput capabilities that vary by event source type. For Amazon MSK and self-managed Apache Kafka, each event poller can handle up to 5 MB/sec of throughput or up to 5 concurrent invocations. For example, if your event source produces an average payload of 1 MB and the average duration of your function is 1 second, a single Kafka event poller can support 5 MB/sec throughput and 5 concurrent Lambda invocations (assuming no payload transformation). For Amazon SQS, each event poller can handle up to 1 MB/sec of

throughput or up to 10 concurrent invocations. Using provisioned mode incurs additional costs based on your event poller usage. For pricing details, see AWS Lambda pricing.

> **Note**
>
> When using provisioned mode, you don't need to create AWS PrivateLink VPC endpoints or grant the associated permissions as part of your network configuration.

In provisioned mode, the range of accepted values for the minimum number of event pollers (`MinimumPollers`) is between 1 and 200, inclusive. The range of accepted values for the maximum number of event pollers (`MaximumPollers`) is between 1 and 2,000, inclusive. `MaximumPollers` must be greater than or equal to `MinimumPollers`. In addition, to maintain ordered processing within partitions, Lambda caps the `MaximumPollers` to the number of partitions in the topic.

For more details about choosing appropriate values for minimum and maximum event pollers, see Best practices.

You can configure provisioned mode for your Kafka event source mapping using the console or the Lambda API.

To configure provisioned mode for an existing event source mapping (console)

1. Open the Functions page of the Lambda console.
2. Choose the function with the event source mapping you want to configure provisioned mode for.
3. Choose **Configuration**, then choose **Triggers**.
4. Choose the event source mapping that you want to configure provisioned mode for, then choose **Edit**.
5. Under **Provisioned mode**, select **Configure**.
   - For **Minimum event pollers**, enter a value between 1 and 200. If you don't specify a value, Lambda chooses a default value of 1.
   - For **Maximum event pollers**, enter a value between 1 and 2,000. This value must be greater than or equal to your value for **Minimum event pollers**. If you don't specify a value, Lambda chooses a default value of 200.
6. Choose **Save**.

You can configure provisioned mode programmatically using the ProvisionedPollerConfig object in your EventSourceMappingConfiguration. For example, the following UpdateEventSourceMapping CLI command configures a `MinimumPollers` value of 5, and a `MaximumPollers` value of 100.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

After configuring provisioned mode, you can observe the usage of event pollers for your workload by monitoring the `ProvisionedPollers` metric. For more information, see the section called "Event source mapping metrics".

To disable provisioned mode and return to default (on-demand) mode, you can use the following UpdateEventSourceMapping CLI command:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

Advanced error handling and performance features

For Kafka event source mappings with provisioned mode enabled, you can configure additional features to improve error handling and performance:

- Retry configurations – Control how Lambda handles failed records with maximum retry attempts, record age limits, batch splitting, and partial batch responses.
- Kafka on-failure destinations – Send failed records to a Kafka topic for later processing or analysis.

Best practices and considerations when using provisioned mode

The optimal configuration of minimum and maximum event pollers for your event source mapping depends on your application's performance requirements. We recommend that you start with the default minimum event pollers to baseline the performance profile. Adjust your configuration based on observed message processing patterns and your desired performance profile.

For workloads with spiky traffic and strict performance needs, increase the minimum event pollers to handle sudden surges in messages. To determine the minimum event pollers required, consider

your workload's messages per second and average payload size, and use the throughput capacity of a single event poller (up to 5 MBps) as a reference.

To maintain ordered processing within a partition, Lambda limits the maximum event pollers to the number of partitions in the topic. Additionally, the maximum event pollers your event source mapping can scale to depends on the function's concurrency settings.

When activating provisioned mode, update your network settings to remove AWS PrivateLink VPC endpoints and associated permissions.

Cost Optimization for Provisioned mode

Provisioned mode pricing

Provisioned mode is charged based on the provisioned minimum event pollers, and the event pollers consumed during autoscaling. Charges are calculated using a billing unit called Event Poller Unit (EPU). You pay for the number and duration of EPUs used, measured in Event-Poller-Unit-hours. You can use Provisioned mode with either a single ESM for performance-sensitive applications or you can group multiple ESMs within the same VPC to share EPU capacity and costs. We will deep dive on two capabilities that help you optimize your Provisioned mode costs. For pricing details, see AWS Lambda pricing.

Enhanced EPU Utilization

Each EPU supports up to 20 MB/s throughput capacity for event polling and supports a default of 10 event pollers. When you create a Provisioned mode for Kafka ESM by setting minimum and maximum pollers, it uses minimum poller number to provision EPUs based on default of 10 event pollers per EPU. However, each event poller can independently scale to support up to 5 MB/s of throughput capacity, which may require lower density of event pollers on a specific EPU and can trigger scaling of EPUs. The number of event pollers allocated on an EPU depends on the compute capacity consumed by each event poller. This approach of enhanced EPU utilization allows event pollers with varying throughput requirements to utilize EPU capacity effectively, reducing costs for all ESMs.

ESM grouping

To optimize your Provisioned mode costs further, you can group multiple Kafka ESMs to share EPU capacity. With ESM grouping and enhanced EPU Utilization, you can reduce your Provisioned mode costs up to 90% for low-throughput workloads compared to running in single ESM mode. All ESMs that require less than 1 EPU capacity will benefit from ESM grouping. Such ESMs typically require few minimum event pollers to support their throughput needs. This capability will allow

you to adopt Provisioned mode for all your Kafka workloads, and benefit from features like schema validation, filtering of Avro/Protobuf events, low-latency invocations, and enhanced error handling that only available in Provisioned mode.

When you configure the `PollerGroupName` parameter with the same value for multiple ESMs within the same Amazon VPC, those ESMs share EPU resources instead of each requiring dedicated EPU capacity. You can group up to 100 ESMs per poller group and aggregate maximum pollers across all ESMs in a group cannot exceed 2000.

To configure ESM grouping (console)

1. Open the Functions page of the Lambda console.
2. Choose your function.
3. Choose **Configuration**, and then choose **Triggers**.
4. When creating a new Kafka event source mapping, or editing an existing one, select **Configure** under **Provisioned mode**.
5. For **Minimum event pollers**, enter a value between 1 and 200.
6. For **Maximum event pollers**, enter a value between 1 and 2,000.
7. For **Poller group name**, enter an identifier for the group. Use the same name for other ESMs you want to group together.
8. Choose **Save**.

To configure ESM grouping (AWS CLI)

The following example creates an ESM with a poller group named `production-app-group`:

```
aws lambda create-event-source-mapping \
  --function-name myFunction1 \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/MyCluster/abcd1234 \
  --topics topic1 \
  --starting-position LATEST \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "production-app-group"
  }'
```

To add another ESM to the same group (sharing EPU capacity), use the same PollerGroupName:

```
aws lambda create-event-source-mapping \
  --function-name myFunction2 \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/MyCluster/abcd1234 \
  --topics topic2 \
  --starting-position LATEST \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "production-app-group"
  }'
```

Note

You can update the `PollerGroupName` to move an ESM to a different group, or remove an ESM from a group by passing an empty string ("") for `PollerGroupName`:

```
# Move ESM to a different group
aws lambda update-event-source-mapping \
  --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "new-group-name"
  }'

# Remove ESM from group (use dedicated resources)
aws lambda update-event-source-mapping \
  --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": ""
  }'
```

Grouping strategy considerations

- **Application boundary** – Group ESMs that belong to the same applications or services for better cost allocation and management. Consider using naming conventions like `app-name-environment` (e.g., `order-processor-prod`).

- **Traffic pattern** – Avoid grouping ESMs with high throughput and spiky traffic pattern, as this may lead to resource contention.
- **Blast radius** – Consider the impact if the shared infrastructure experiences issues. All ESMs in the same group are affected by shared resource limitations. For mission-critical workloads, you may want to use separate groups or dedicated ESMs.

Cost optimization example

Consider a scenario where you have 10 ESMs, each configured with 1 event poller and throughput under 2 MB/s:

Without grouping:

- Each ESM requires its own EPU
- Total EPUs needed: 10
- Cost per EPU: $0.185/hour in US East (N. Virginia)
- Monthly EPU cost (720 hours): 10 × 720 × $0.185 = $1,332

With grouping:

- All 10 ESMs share EPU capacity
- 10 event pollers fit in 1 EPU (with new 10 poller per EPU support)
- Total EPUs needed: 1
- Monthly EPU cost (720 hours): 1 × 720 × $0.185 = $133.20
- **Cost savings: 90%** ($1,198.80 savings per month)

Apache Kafka polling and stream starting positions in Lambda

The StartingPosition parameter tells Lambda when to start reading messages from your Amazon MSK or self-managed Apache Kafka stream. There are three options to choose from:

- **Latest** – Lambda starts reading just after the most recent record in the Kafka topic.
- **Trim horizon** – Lambda starts reading from the last untrimmed record in the Kafka topic. This is also the oldest record in the topic.
- **At timestamp** – Lambda starts reading from a position defined by a timestamp, in Unix time seconds. Use the StartingPositionTimestamp parameter to specify the timestamp.

Stream polling during an event source mapping create or update is eventually consistent:

- During event source mapping creation, it may take several minutes to start polling events from the stream.
- During event source mapping updates, it may take up to 90 seconds to stop and restart polling events from the stream.

This behavior means that if you specify `LATEST` as the starting position for the stream, the event source mapping could miss events during a create or update. To ensure that no events are missed, specify either `TRIM_HORIZON` or `AT_TIMESTAMP`.

Customizable consumer group ID in Lambda

When setting up Amazon MSK or self-managed Apache Kafka as an event source, you can specify a consumer group ID. This consumer group ID is an existing identifier for the Kafka consumer group that you want your Lambda function to join. You can use this feature to seamlessly migrate any ongoing Kafka record processing setups from other consumers to Lambda.

Kafka distributes messages across all consumers in a consumer group. If you specify a consumer group ID that has other active consumers, Lambda receives only a portion of the messages from the Kafka topic. If you want Lambda to handle all messages in the topic, turn off any other consumers in that consumer group.

Additionally, if you specify a consumer group ID, and Kafka finds a valid existing consumer group with the same ID, Lambda ignores the StartingPosition for your event source mapping. Instead, Lambda begins processing records according to the committed offset of the consumer group. If you specify a consumer group ID, and Kafka cannot find an existing consumer group, then Lambda configures your event source with the specified `StartingPosition`.

The consumer group ID that you specify must be unique among all your Kafka event sources. After creating a Kafka event source mapping with the consumer group ID specified, you cannot update this value.

Filtering events from Amazon MSK and self-managed Apache Kafka event sources

You can use event filtering to control which records from a stream or queue Lambda sends to your function. For general information about how event filtering works, see the section called “Event filtering”.

> **Note**
>
> Amazon MSK and self-managed Apache Kafka event source mappings only support filtering on the `value` key.

Topics

- Kafka event filtering basics

Kafka event filtering basics

Suppose a producer is writing messages to a topic in your Kafka cluster, either in valid JSON format or as plain strings. An example record would look like the following, with the message converted to a Base64 encoded string in the `value` field.

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

Suppose your Apache Kafka producer is writing messages to your topic in the following JSON format.

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
    }
}
```

You can use the `value` key to filter records. Suppose you wanted to filter only those records where `device_ID` begins with the letters AB. The `FilterCriteria` object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

With Kafka, you can also filter records where the message is a plain string. Suppose you want to ignore those messages where the string is "error". The `FilterCriteria` object would look as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "value": [
```

```
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
```

```
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Kafka messages must be UTF-8 encoded strings, either plain strings or in JSON format. That's because Lambda decodes Kafka byte arrays into UTF-8 before applying filter criteria. If your messages use another encoding, such as UTF-16 or ASCII, or if the message format doesn't match the `FilterCriteria` format, Lambda processes metadata filters only. The following table summarizes the specific behavior:

Incoming message format	Filter pattern format for message properties	Resulting action
Plain string	Plain string	Lambda filters based on your filter criteria.
Plain string	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Plain string	Valid JSON	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Plain string	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.
Non-UTF-8 encoded string	JSON, plain string, or no pattern	Lambda filters (on the other metadata properties only) based on your filter criteria.

Using schema registries with Kafka event sources in Lambda

Schema registries help you define and manage data stream schemas. A schema defines the structure and format of a data record. In the context of Kafka event source mappings, you can configure a schema registry to validate the structure and format of Kafka messages against predefined schemas before they reach your Lambda function. This adds a layer of data governance to your application and allows you to efficiently manage data formats, ensure schema compliance, and optimize costs through event filtering.

This feature works with all programming languages, but consider these important points:

- Powertools for Lambda provides specific support for Java, Python, and TypeScript, maintaining consistency with existing Kafka development patterns and allowing direct access to business objects without custom deserialization code
- This feature is only available for event source mappings using provisioned mode. Schema registry doesn't support event source mappings in on-demand mode. If you're using provisioned mode and you have a schema registry configured, you can't change to on-demand mode unless you remove your schema registry configuration first. For more information, see the section called "Provisioned mode"
- You can configure only one schema registry per event source mapping (ESM). Using a schema registry with your Kafka event source may increase your Lambda Event Poller Unit (EPU) usage, which is a pricing dimension for Provisioned mode.

Topics

- Schema registry options
- How Lambda performs schema validation for Kafka messages
- Configuring a Kafka schema registry
- Filtering for Avro and Protobuf
- Payload formats and deserialization behavior
- Working with deserialized data in Lambda functions
- Authentication methods for your schema registry
- Error handling and troubleshooting for schema registry issues

Schema registry options

Lambda supports the following schema registry options:

- AWS Glue Schema Registry
- Confluent Cloud Schema Registry
- Self-managed Confluent Schema Registry

Your schema registry supports validating messages in the following data formats:

- Apache Avro
- Protocol Buffers (Protobuf)
- JSON Schema (JSON-SE)

To use a schema registry, first ensure that your event source mapping is in provisioned mode. When you use a schema registry, Lambda adds metadata about the schema to the payload. For more information, see Payload formats and deserialization behavior.

How Lambda performs schema validation for Kafka messages

When you configure a schema registry, Lambda performs the following steps for each Kafka message:

1. Lambda polls the Kafka record from your cluster.
2. Lambda validates selected message attributes in the record against a specific schema in your schema registry.
   - If the schema associated with the message is not found in the registry, Lambda sends the message to a DLQ with reason code `SCHEMA_NOT_FOUND`.
3. Lambda deserializes the message according to the schema registry configuration to validate the message. If event filtering is configured, Lambda then performs filtering based on the configured filter criteria.
   - If deserialization fails, Lambda sends the message to a DLQ with reason code `DESERIALIZATION_ERROR`. If no DLQ is configured, Lambda drops the message.
4. If the message is validated by the schema registry, and isn't filtered out by your filter criteria, Lambda invokes your function with the message.

This feature is intended to validate messages that are already produced using Kafka clients integrated with a schema registry. We recommend configuring your Kafka producers to work with your schema registry to create properly formatted messages.

Configuring a Kafka schema registry

The following console steps add a Kafka schema registry configuration to your event source mapping.

To add a Kafka schema registry configuration to your event source mapping (console)

1. Open the Function page of the Lambda console.
2. Choose **Configuration**.
3. Choose **Triggers**.
4. Select the Kafka event source mapping that you want to configure a schema registry for, and choose **Edit**.
5. Under **Event poller configuration**, choose **Configure schema registry**. Your event source mapping must be in provisioned mode to see this option.
6. For **Schema registry URI**, enter the ARN of your AWS Glue schema registry, or the HTTPS URL of your Confluent Cloud schema registry or Self-Managed Confluent Schema Registry.
7. The following configuration steps tell Lambda how to access your schema registry. For more information, see ???.
   - For **Access configuration type**, choose the type of authentication Lambda uses to access your schema registry.
   - For **Access configuration URI**, enter the ARN of the Secrets Manager secret to authenticate with your schema registry, if applicable. Ensure that your function's execution role contains the correct permissions.
8. The **Encryption** field applies only if your schema registry is signed by a private Certificate Authority (CA) or a certificate authority (CA) that's not in the Lambda trust store.. If applicable, provide the secret key containing the private CA certificate used by your schema registry for TLS encryption.
9. For **Event record format**, choose how you want Lambda to deliver the records your function after schema validation. For more information, see Payload format examples.

- If you choose **JSON**, Lambda delivers the attributes that you select in the Schema validation attribute below in standard JSON format. For the attributes that you don't select, Lambda delivers them as-is.
- If you choose **SOURCE**, Lambda delivers the attributes that you select in the Schema validation attribute below in its original source format.

10. For **Schema validation attribute**, select the message attributes that you want Lambda to validate and deserialize using your schema registry. You must select at least one of either **KEY** or **VALUE**. If you chose JSON for event record format, Lambda also deserializes the selected message attributes before sending them to your function. For more information, see Payload formats and deserialization behavior.
11. Choose **Save**.

You can also use the Lambda API to create or update your event source mapping with a schema registry configuration. The following examples demonstrate how to configure an AWS Glue or Confluent schema registry using the AWS CLI, which corresponds to the UpdateEventSourceMapping and CreateEventSourceMapping API operations in the *AWS Lambda API Reference*:

> **Important**
>
> If you are updating any schema registry configuration field using the AWS CLI or the `update-event-source-mapping` API, you must update all the fields of schema registry configuration.

Create Event Source Mapping

```
aws lambda create-event-source-mapping \
  --function-name my-schema-validator-function \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/
a1b2c3d4-5678-90ab-cdef-11111EXAMPLE \
  --topics my-kafka-topic \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --amazon-managed-kafka-event-source-mapping '{
      "SchemaRegistryConfig" : {
          "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
          "AccessConfigs": [{
```

```
              "Type": "BASIC_AUTH",
              "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName"
          }],
          "EventRecordFormat": "JSON",
          "SchemaValidationConfigs": [
          {
              "Attribute": "KEY"
          },
          {
              "Attribute": "VALUE"
          }]
      }
  }'
```

Update AWS Glue Schema Registry

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "arn:aws:glue:us-east-1:123456789012:registry/
registryName",
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Update Confluent Schema Registry with Authentication

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
            "AccessConfigs": [{
                "Type": "BASIC_AUTH",
```

```
                "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName"
            }],
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Update Confluent Schema Registry without Authentication

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Remove Schema Registry Configuration

To remove a schema registry configuration from your event source mapping, you can use the CLI command UpdateEventSourceMapping in the *AWS Lambda API Reference*.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {}
    }'
```

Filtering for Avro and Protobuf

When using Avro or Protobuf formats with a schema registry, you can apply event filtering to your Lambda function. The filter patterns are applied to the deserialized classic JSON representation of your data after schema validation. For example, with an Avro schema defining product details including price, you can filter messages based on the price value:

> **Note**
>
> When being deserialized, Avro is converted to standard JSON, which means it cannot be directly converted back to an Avro object. If you need to convert to an Avro object, use the SOURCE format instead.
> For Protobuf deserialization, field names in the resulting JSON match those defined in your schema, rather than being converted to camel case as Protobuf typically does. Keep this in mind when creating filtering patterns.

```
aws lambda create-event-source-mapping \
    --function-name myAvroFunction \
    --topics myAvroTopic \
    --starting-position TRIM_HORIZON \
    --kafka-bootstrap-servers '["broker1:9092", "broker2:9092"]' \
    --schema-registry-config '{
        "SchemaRegistryURI": "arn:aws:glue:us-east-1:123456789012:registry/
myAvroRegistry",
        "EventRecordFormat": "JSON",
        "SchemaValidationConfigs": [
            {
                "Attribute": "VALUE"
            }
        ]
    }' \
    --filter-criteria '{
        "Filters": [
            {
                "Pattern": "{ \"value\" : { \"field_1\" : [\"value1\"], \"field_2\" :
 [\"value2\"] } }"
            }
        ]
    }'
```

In this example, the filter pattern analyzes the `value` object, matching messages in `field_1` with `"value1"` and `field_2` with `"value2"`. The filter criteria are evaluated against the deserialized data, after Lambda converts the message from Avro format to JSON.

For more detailed information on event filtering, see Lambda event filtering.

Payload formats and deserialization behavior

When using a schema registry, Lambda delivers the final payload to your function in a format similar to the regular event payload, with some additional fields. The additional fields depend on the `SchemaValidationConfigs` parameter. For each attribute that you select for validation (key or value), Lambda adds corresponding schema metadata to the payload.

> **Note**
>
> You must update your aws-lambda-java-events to version 3.16.0 or above to use schema metadata fields.

For example, if you validate the `value` field, Lambda adds a field called `valueSchemaMetadata` to your payload. Similarly, for the `key` field, Lambda adds a field called `keySchemaMetadata`. This metadata contains information about the data format and the schema ID used for validation:

```
"valueSchemaMetadata": {
    "dataFormat": "AVRO",
    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
```

The `EventRecordFormat` parameter can be set to either `JSON` or `SOURCE`, which determines how Lambda handles schema-validated data before delivering it to your function. Each option provides different processing capabilities:

- `JSON` - Lambda deserializes the validated attributes into standard JSON format, making the data ready for direct use in languages with native JSON support. This format is ideal when you don't need to preserve the original binary format or work with generated classes.
- `SOURCE` - Lambda preserves the original binary format of the data as a Base64-encoded string, allowing direct conversion to Avro or Protobuf objects. This format is essential when working with strongly-typed languages or when you need to maintain the full capabilities of Avro or Protobuf schemas.

Based on these format characteristics and language-specific considerations, we recommend the following formats:

Recommended formats based on programming language

Language	Avro	Protobuf	JSON
Java	SOURCE	SOURCE	SOURCE
Python	JSON	JSON	JSON
NodeJS	JSON	JSON	JSON
.NET	SOURCE	SOURCE	SOURCE
Others	JSON	JSON	JSON

The following sections describe these formats in detail and provide example payloads for each format.

JSON format

If you choose `JSON` as the `EventRecordFormat`, Lambda validates and deserializes the message attributes that you've selected in the `SchemaValidationConfigs` field (the `key` and/or `value` attributes). Lambda delivers these selected attributes as base64-encoded strings of their standard JSON representation in your function.

> **Note**
>
> When being deserialized, Avro is converted to standard JSON, which means it cannot be directly converted back to an Avro object. If you need to convert to an Avro object, use the SOURCE format instead.
> For Protobuf deserialization, field names in the resulting JSON match those defined in your schema, rather than being converted to camel case as Protobuf typically does. Keep this in mind when creating filtering patterns.

The following shows an example payload, assuming you choose `JSON` as the `EventRecordFormat`, and both the `key` and `value` attributes as `SchemaValidationConfigs`:

```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111-1",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==", //Base64 encoded string of JSON
            "keySchemaMetadata": {
                "dataFormat": "AVRO",
                "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
            },
            "value":"abcDEFghiJKLmnoPQRstuVWXyz1234", //Base64 encoded string of JSON
            "valueSchemaMetadata": {
                "dataFormat": "AVRO",
                "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
            },
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
```

```
      ]
   }
}
```

In this example:

- Both `key` and `value` are base64-encoded strings of their JSON representation after deserialization.
- Lambda includes schema metadata for both attributes in `keySchemaMetadata` and `valueSchemaMetadata`.
- Your function can decode the `key` and `value` strings to access the deserialized JSON data.

The JSON format is recommended for languages that aren't strongly typed, such as Python or Node.js. These languages have native support for converting JSON into objects.

Source format

If you choose `SOURCE` as the `EventRecordFormat`, Lambda still validates the record against the schema registry, but delivers the original binary data to your function without deserialization. This binary data is delivered as a Base64 encoded string of the original byte data, with producer-appended metadata removed. As a result, you can directly convert the raw binary data into Avro and Protobuf objects within your function code. We recommend using Powertools for AWS Lambda, which will deserialize the raw binary data and give you Avro and Protobuf objects directly.

For example, if you configure Lambda to validate both the `key` and `value` attributes but use the `SOURCE` format, your function receives a payload like this:

```
{
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111-1",
    "bootstrapServers": "b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
    "records": {
        "mytopic-0": [
            {
                "topic": "mytopic",
                "partition": 0,
                "offset": 15,
```

```
                "timestamp": 1545084650987,
                "timestampType": "CREATE_TIME",
                "key": "abcDEFghiJKLmnoPQRstuVWXyz1234==", // Base64 encoded string of
 Original byte data, producer-appended metadata removed
                "keySchemaMetadata": {
                    "dataFormat": "AVRO",
                    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
                },
                "value": "abcDEFghiJKLmnoPQRstuVWXyz1234==", // Base64 encoded string
 of Original byte data, producer-appended metadata removed
                "valueSchemaMetadata": {
                    "dataFormat": "AVRO",
                    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
                },
                "headers": [
                    {
                        "headerKey": [
                            104,
                            101,
                            97,
                            100,
                            101,
                            114,
                            86,
                            97,
                            108,
                            117,
                            101
                        ]
                    }
                ]
            }
        ]
    }
}
```

In this example:

- Both `key` and `value` contain the original binary data as Base64 encoded strings.
- Your function needs to handle deserialization using the appropriate libraries.

Choosing SOURCE for `EventRecordFormat` is recommended if you're using Avro-generated or Protobuf-generated objects, especially with Java functions. This is because Java is strongly typed, and requires specific deserializers for Avro and Protobuf formats. In your function code, you can use your preferred Avro or Protobuf library to deserialize the data.

Working with deserialized data in Lambda functions

Powertools for AWS Lambda helps you deserialize Kafka records in your function code based on the format you use. This utility simplifies working with Kafka records by handling data conversion and providing ready-to-use objects.

To use Powertools for AWS Lambda in your function, you need to add Powertools for AWS Lambda either as a layer or include it as a dependency when building your Lambda function. For setup instructions and more information, see the Powertools for AWS Lambda documentation for your preferred language:

- Powertools for AWS Lambda (Java)
- Powertools for AWS Lambda (Python)
- Powertools for AWS Lambda (TypeScript)
- Powertools for AWS Lambda (.NET)

Note

When working with schema registry integration, you can choose SOURCE or JSON format. Each option supports different serialization formats as shown below:

Format	Supports
SOURCE	Avro and Protobuf (using Lambda Schema Registry integration)
JSON	JSON data

When using the SOURCE or JSON format, you can use Powertools for AWS to help deserialize the data in your function code. Here are examples of how to handle different data formats:

AVRO

Java example:

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.demo.kafka.avro.AvroProduct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class AvroDeserializationFunction implements
 RequestHandler<ConsumerRecords<String, AvroProduct>, String> {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(AvroDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_AVRO)
    public String handleRequest(ConsumerRecords<String, AvroProduct> records,
 Context context) {
        for (ConsumerRecord<String, AvroProduct> consumerRecord : records) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            AvroProduct product = consumerRecord.value();
            LOGGER.info("AvroProduct: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }
```

```
}
```

Python example:

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer
from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

logger = Logger(service="kafkaConsumerPowertools")

value_schema_str = open("customer_profile.avsc", "r").read()

schema_config = SchemaConfig(
value_schema_type="AVRO",
value_schema=value_schema_str)

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript example:

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';

import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
```

```
};

const schema = '{
    "type": "record",
    "name": "Product",
    "fields": [
        { "name": "id", "type": "int" },
        { "name": "name", "type": "string" },
        { "name": "price", "type": "double" }
    ]
}';

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
            // You can add more processing logic here
        }
    },
    {
        value: {
            type: 'avro',
            schema: schema,
        },
    }
);
```

.NET example:

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Avro;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaAvroSerializer))]

namespace ProtoBufClassLibrary;

public class Function
```

```
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

PROTOBUF

Java example:

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.demo.kafka.protobuf.ProtobufProduct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;
```

```
public class ProtobufDeserializationFunction
        implements RequestHandler<ConsumerRecords<String, ProtobufProduct>, String>
 {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(ProtobufDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_PROTOBUF)
    public String handleRequest(ConsumerRecords<String, ProtobufProduct> records,
 Context context) {
        for (ConsumerRecord<String, ProtobufProduct> consumerRecord : records) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            ProtobufProduct product = consumerRecord.value();
            LOGGER.info("ProtobufProduct: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }

}
```

Python example:

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer

from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

from user_pb2 import User # protobuf generated class

logger = Logger(service="kafkaConsumerPowertools")
```

```
schema_config = SchemaConfig(
value_schema_type="PROTOBUF",
value_schema=User)

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript example:

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';
import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';
import { Product } from './product.generated.js';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
        }
    },
    {
        value: {
            type: 'protobuf',
            schema: Product,
        },
    }
);
```

.NET example:

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Protobuf;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaProtobufSerializer))]

namespace ProtoBufClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

JSON

Java example:

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class JsonDeserializationFunction implements
 RequestHandler<ConsumerRecords<String, Product>, String> {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(JsonDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_JSON)
    public String handleRequest(ConsumerRecords<String, Product> consumerRecords,
 Context context) {
        for (ConsumerRecord<String, Product> consumerRecord : consumerRecords) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            Product product = consumerRecord.value();
            LOGGER.info("Product: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }
}
```

Python example:

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer
```

```
from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

logger = Logger(service="kafkaConsumerPowertools")

schema_config = SchemaConfig(value_schema_type="JSON")

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript example:

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';
import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
            // You can add more processing logic here
        }
    },
```

```
    {
        value: {
            type: 'json',
        },
    }
);
```

.NET example:

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Json;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaJsonSerializer))]

namespace JsonClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }
```

```
        return "Processed " + records.Count() + " records";
    }
}
```

Authentication methods for your schema registry

To use a schema registry, Lambda needs to be able to securely access it. If you're working with an AWS Glue schema registry, Lambda relies on IAM authentication. This means that your function's execution role must have the following permissions to access the AWS Glue registry:

- GetRegistry in the *AWS Glue Web API Reference*
- GetSchemaVersion in the *AWS Glue Web API Reference*

Example of the required IAM policy:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:GetRegistry",
                "glue:GetSchemaVersion"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Note

For AWS Glue schema registries, if you provide `AccessConfigs` for a AWS Glue registry, Lambda will return a validation exception.

If you're working with a Confluent schema registry, you can choose one of three supported authentication methods for the `Type` parameter of your KafkaSchemaRegistryAccessConfig object:

- **BASIC_AUTH** — Lambda uses username and password or API Key and API Secret authentication to access your registry. If you choose this option, provide the Secrets Manager ARN containing your credentials in the URI field.
- **CLIENT_CERTIFICATE_TLS_AUTH** — Lambda uses mutual TLS authentication with client certificates. To use this option, Lambda needs access to both the certificate and the private key. Provide the Secrets Manager ARN containing these credentials in the URI field.
- **NO_AUTH** — The public CA certificate must be signed by a certificate authority (CA) that's in the Lambda trust store. For a private CA/self-signed certificate, you configure the server root CA certificate. To use this option, omit the `AccessConfigs` parameter.

Additionally, if Lambda needs access to a private CA certificate to verify your schema registry's TLS certificate, choose `SERVER_ROOT_CA_CERT` as the `Type` and provide the Secrets Manager ARN to the certificate in the URI field.

> **Note**
>
> To configure the `SERVER_ROOT_CA_CERT` option in the console, provide the secret ARN containing the certificate in the **Encryption** field.

The authentication configuration for your schema registry is separate from any authentication you've configured for your Kafka cluster. You must configure both separately, even if they use similar authentication methods.

Error handling and troubleshooting for schema registry issues

When using a schema registry with your Amazon MSK event source, you may encounter various errors. This section provides guidance on common issues and how to resolve them.

Configuration errors

These errors occur when setting up your schema registry configuration.

Provisioned mode required

Error message: `SchemaRegistryConfig is only available for Provisioned Mode. To configure Schema Registry, please enable Provisioned Mode by specifying MinimumPollers in ProvisionedPollerConfig.`

Resolution: Enable provisioned mode for your event source mapping by configuring the `MinimumPollers` parameter in `ProvisionedPollerConfig`.

Invalid schema registry URL

Error message: `Malformed SchemaRegistryURI provided. Please provide a valid URI or ARN. For example, https://schema-registry.example.com:8081 or arn:aws:glue:us-east-1:123456789012:registry/ExampleRegistry.`

Resolution: Provide a valid HTTPS URL for Confluent Schema Registry or a valid ARN for AWS Glue Schema Registry.

Invalid or missing event record format

Error message: `EventRecordFormat is a required field for SchemaRegistryConfig. Please provide one of supported format types: SOURCE, JSON.`

Resolution: Specify either SOURCE or JSON as the EventRecordFormat in your schema registry configuration.

Duplicate validation attributes

Error message: `Duplicate KEY/VALUE Attribute in SchemaValidationConfigs. SchemaValidationConfigs must contain at most one KEY/VALUE Attribute.`

Resolution: Remove duplicate KEY or VALUE attributes from your SchemaValidationConfigs. Each attribute type can only appear once.

Missing validation configuration

Error message: `SchemaValidationConfigs is a required field for SchemaRegistryConfig.`

Resolution: Add SchemaValidationConfigs to your configuration, specifying at least one validation attribute (KEY or VALUE).

Access and permission errors

These errors occur when Lambda cannot access the schema registry due to permission or authentication issues.

AWS Glue Schema Registry access denied

> **Error message:** `Cannot access Glue Schema with provided role. Please ensure the provided role can perform the GetRegistry and GetSchemaVersion Actions on your schema.`
>
> **Resolution:** Add the required permissions (`glue:GetRegistry` and `glue:GetSchemaVersion`) to your function's execution role.

Confluent Schema Registry access denied

> **Error message:** `Cannot access Confluent Schema with the provided access configuration.`
>
> **Resolution:** Verify that your authentication credentials (stored in Secrets Manager) are correct and have the necessary permissions to access the schema registry.

Cross-account AWS Glue Schema Registry

> **Error message:** `Cross-account Glue Schema Registry ARN not supported.`
>
> **Resolution:** Use a AWS Glue Schema Registry that's in the same AWS account as your Lambda function.

Cross-region AWS Glue Schema Registry

> **Error message:** `Cross-region Glue Schema Registry ARN not supported.`
>
> **Resolution:** Use a AWS Glue Schema Registry that's in the same region as your Lambda function.

Secret access issues

> **Error message:** `Lambda received InvalidRequestException from Secrets Manager.`
>
> **Resolution:** Verify that your function's execution role has permission to access the secret and that the secret is not encrypted with a default AWS KMS key if accessing from a different account.

Connection errors

These errors occur when Lambda cannot establish a connection to the schema registry.

VPC connectivity issues

> **Error message:** `Cannot connect to your Schema Registry. Your Kafka cluster's VPC must be able to connect to the schema registry. You can provide access by configuring AWS PrivateLink or a NAT Gateway or VPC Peering between Kafka Cluster VPC and the schema registry VPC.`
>
> **Resolution:** Configure your VPC networking to allow connections to the schema registry using AWS PrivateLink, a NAT Gateway, or VPC peering.

TLS handshake failure

> **Error message:** `Unable to establish TLS handshake with the schema registry. Please provide correct CA-certificate or client certificate using Secrets Manager to access your schema registry.`
>
> **Resolution:** Verify that your CA certificates and client certificates (for mTLS) are correct and properly configured in Secrets Manager.

Throttling

> **Error message:** `Receiving throttling errors when accessing the schema registry. Please increase API TPS limits for your schema registry.`
>
> **Resolution:** Increase the API rate limits for your schema registry or reduce the rate of requests from your application.

Self-managed schema registry errors

> **Error message:** `Lambda received an internal server an unexpected error from the provided self-managed schema registry.`
>
> **Resolution:** Check the health and configuration of your self-managed schema registry server.

Low latency processing for Kafka event sources

AWS Lambda natively supports low latency event processing for applications that require consistent end-to-end latencies of less than 100 milliseconds. This page provides configuration details and recommendations to enable low latency workflows.

Enable low latency processing

To enable low latency processing on a Kafka event source mapping, the following basic configuration is required:

- Enable provisioned mode. For more information, see Provisioned mode.
- Set the event source mapping's `MaximumBatchingWindowInSeconds` parameter to 0. For more information, see Batching behavior.

Fine-tuning your low latency Kafka ESM

Consider the following recommendations to optimize your Kafka event source mapping for low latency:

Provisioned mode configuration

In provisioned mode for Kafka event source mapping, Lambda allows you to fine-tune the throughput of your event source mapping by configuring a minimum and maximum number of resources called **event pollers**. An event poller (or **a poller**) represents a compute resource that underpins an event source mapping in the provisioned mode, and allocates up to 5 MB/s throughput. Each event poller supports up to 5 concurrent Lambda invocations.

To determine the optimal poller configuration for your application, consider your peak ingestion rate and processing requirements. Let's look at a simplified example:

With a batch size of 20 records and an average target function duration of 50ms, each poller can handle 2,000 records per second subject to 5 MB/s limit. This is calculated as: (20 records × 1000ms/50ms) × 5 concurrent Lambda invocations. Therefore, if your desired peak ingestion rate is 20,000 records per second, you would need at least 10 event pollers.

> **Note**
>
> We recommend to provision additional event pollers as buffer to avoid consistently operating at maximum capacity.

Provisioned mode automatically scales your event pollers based on traffic patterns within configured minimum and maximum **event pollers** which can trigger rebalance, and therefore, introduce additional latency. You can disable auto-scaling by configuring same value for minimum and maximum **event poller**.

Additional considerations

Some of the additional considerations include:

- Cold starts from the invocation of your Lambda target function can potentially increase end-to-end latency. To reduce this risk, consider enabling provisioned concurrency or SnapStart on your event source mapping's target function. Additionally, optimize your function's memory allocation to ensure consistent and optimal executions.
- When `MaximumBatchingWindowInSeconds` is set to 0, Lambda will immediately process any available records without waiting to fill the complete batch size. For example, if your batch size is set to 1,000 records but only 100 records are available, Lambda will process those 100 records immediately rather than waiting for the full 1,000 records to accumulate.

> **Important**
>
> The optimal configuration for low latency processing varies significantly based on your specific workload. We strongly recommend testing different configurations with your actual workload to determine the best settings for your use case.

Configuring error handling controls for Kafka event sources

You can configure how Lambda handles errors and retries for your Kafka event source mappings. These configurations help you control how Lambda processes failed records and manages retry behavior.

Available retry configurations

The following retry configurations are available for both Amazon MSK and self-managed Kafka event sources:

- **Maximum retry attempts** – The maximum number of times Lambda retries when your function returns an error. This doesn't count the initial invocation attempt. The default is -1 (infinite). When you configure both infinite retries and an on-failure destination, Lambda automatically applies a maximum of 10 retry attempts.
- **Maximum record age** – The maximum age of a record that Lambda sends to your function. The default is -1 (infinite).
- **Split batch on error** – When your function returns an error, split the batch into two smaller batches and retry each separately. This helps isolate problematic records.
- **Partial batch response** – Allow your function to return information about which records in a batch failed processing, so Lambda can retry only the failed records.

Configuring error handling controls (console)

You can configure retry behavior when creating or updating a Kafka event source mapping in the Lambda console.

To configure retry behavior for a Kafka event source (console)

1. Open the Functions page of the Lambda console.
2. Choose your function name.
3. Do one of the following:
   - To add a new Kafka trigger, under **Function overview**, choose **Add trigger**.
   - To modify an existing Kafka trigger, choose the trigger and then choose **Edit**.
4. Under **Event poller configuration**, select provisioned mode to configure error handling controls:

a. For **Retry attempts**, enter the maximum number of retry attempts (0-10000, or -1 for infinite).
b. For **Maximum record age**, enter the maximum age in seconds (60-604800, or -1 for infinite).
c. To enable batch splitting when errors occur, select **Split batch on error**.
d. To enable partial batch response, select **ReportBatchItemFailures**.

5. Choose **Add** or **Save**.

Configuring retry behavior (AWS CLI)

Use the following AWS CLI commands to configure retry behavior for your Kafka event source mappings.

Creating an event source mapping with retry configurations

The following example creates a self-managed Kafka event source mapping with error handling controls:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics my-kafka-topic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:111122223333:secret:MyBrokerSecretName \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --maximum-retry-attempts 3 \
  --maximum-record-age-in-seconds 3600 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

For Amazon MSK event sources:

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSMSKKafkaTopic \
  --starting-position LATEST \
```

```
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]' \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --maximum-retry-attempts 3 \
  --maximum-record-age-in-seconds 3600 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

Updating retry configurations

Use the `update-event-source-mapping` command to modify retry configurations for an existing event source mapping:

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --maximum-retry-attempts 5 \
  --maximum-record-age-in-seconds 7200 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

PartialBatchResponse

Partial batch response, also known as ReportBatchItemFailures, is a key feature for error handling in Lambda's integration with Kafka sources. Without this feature, when an error occurs in one of the items in a batch, it results in reprocessing all messages in that batch. With partial batch response enabled and implemented, the handler returns identifiers only for the failed messages, allowing Lambda to retry just those specific items. This provides greater control over how batches containing failed messages are processed.

To report batch errors, you will use this JSON schema:

```
{
  "batchItemFailures": [
    {
      "itemIdentifier": {
        "partition": "topic-partition_number",
        "offset": 100
      }
    },
    ...
```

```
  ]
}
```

> **Important**
>
> If you return an empty valid JSON or null, the event source mapping will consider a batch as successfully processed. Any invalid topic-partition_number or offset returned that was not present in the invoked event will be treated as failure and entire batch will be retried.

The following code examples show how to implement partial batch response for Lambda functions that receive events from Kafka sources. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

Here is a Python Lambda handler implementation that shows this approach:

```
import base64
from typing import Any, Dict, List

def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, List[Dict[str,
 Dict[str, Any]]]]:
    failures: List[Dict[str, Dict[str, Any]]] = []
    records_dict = event.get("records", {})

    for topic_partition, records_list in records_dict.items():
        for record in records_list:
            topic = record.get("topic")
            partition = record.get("partition")
            offset = record.get("offset")
            value_b64 = record.get("value")

            try:
                data = base64.b64decode(value_b64).decode("utf-8")
                process_message(data)
            except Exception as exc:
                print(f"Failed to process record topic={topic} partition={partition}
 offset={offset}: {exc}")
                item_identifier: Dict[str, Any] = {
                    "partition": f"{topic}-{partition}",
                    "offset": int(offset) if offset is not None else None,
                }
                failures.append({"itemIdentifier": item_identifier})
```

```
    return {"batchItemFailures": failures}

def process_message(data: str) -> None:
    # Your business logic for a single message
    pass
```

Here is a Node.js version:

```
const { Buffer } = require("buffer");

const handler = async (event) => {
  const failures = [];

  for (let topicPartition in event.records) {
    const records = event.records[topicPartition];

    for (const record of records) {
      const topic = record.topic;
      const partition = record.partition;
      const offset = record.offset;
      const valueBase64 = record.value;
      const data = Buffer.from(valueBase64, "base64").toString("utf8");

      try {
        await processMessage(data);
      } catch (error) {
        console.error("Failed to process record", { topic, partition, offset, error });
        const itemIdentifier = {
          "partition": `${topic}-${partition}`,
          "offset": Number(offset),
        };
        failures.push({ itemIdentifier });
      }
    }
  }

  return { batchItemFailures: failures };
};

async function processMessage(payload) {
  // Your business logic for a single message
}
```

```
module.exports = { handler };
```

Capturing discarded batches for Amazon MSK and self-managed Apache Kafka event sources

To retain records of failed event source mapping invocations, add a destination to your function's event source mapping. Each record sent to the destination is a JSON document containing metadata about the failed invocation. For Amazon S3 destinations, Lambda also sends the entire invocation record along with the metadata. You can configure any Amazon SNS topic, Amazon SQS queue, Amazon S3 bucket, or Kafka as a destination.

With Amazon S3 destinations, you can use the Amazon S3 Event Notifications feature to receive notifications when objects are uploaded to your destination S3 bucket. You can also configure S3 Event Notifications to invoke another Lambda function to perform automated processing on failed batches.

Your execution role must have permissions for the destination:

- **For an SQS destination:** sqs:SendMessage
- **For an SNS destination:** sns:Publish
- **For an S3 destination:** s3:PutObject and s3:ListBucket
- **For a Kafka destination:** kafka-cluster:WriteData

You can configure a Kafka topic as an on-failure destination for your Kafka event source mappings. When Lambda can't process records after exhausting retry attempts or when records exceed the maximum age, Lambda sends the failed records to the specified Kafka topic for later processing. Refer to the section called “Kafka on-failure destination”.

You must deploy a VPC endpoint for your on-failure destination service inside your Kafka cluster VPC.

Additionally, if you configured a KMS key on your destination, Lambda needs the following permissions depending on the destination type:

- If you've enabled encryption with your own KMS key for an S3 destination, kms:GenerateDataKey is required. If the KMS key and S3 bucket destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:GenerateDataKey.
- If you've enabled encryption with your own KMS key for SQS destination, kms:Decrypt and kms:GenerateDataKey are required. If the KMS key and SQS queue destination are in a

different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:Decrypt, kms:GenerateDataKey, kms:DescribeKey, and kms:ReEncrypt.

- If you've enabled encryption with your own KMS key for SNS destination, kms:Decrypt and kms:GenerateDataKey are required. If the KMS key and SNS topic destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:Decrypt, kms:GenerateDataKey, kms:DescribeKey, and kms:ReEncrypt.

Configuring on-failure destinations for a Kafka event source mapping

To configure an on-failure destination using the console, follow these steps:

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add destination**.
4. For **Source**, choose **Event source mapping invocation**.
5. For **Event source mapping**, choose an event source that's configured for this function.
6. For **Condition**, select **On failure**. For event source mapping invocations, this is the only accepted condition.
7. For **Destination type**, choose the destination type that Lambda sends invocation records to.
8. For **Destination**, choose a resource.
9. Choose **Save**.

You can also configure an on-failure destination using the AWS CLI. For example, the following create-event-source-mapping command adds an event source mapping with an SQS on-failure destination to `MyFunction`:

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

The following update-event-source-mapping command adds an S3 on-failure destination to the event source associated with the input `uuid`:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

To remove a destination, supply an empty string as the argument to the `destination-config` parameter:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Security best practices for Amazon S3 destinations

Deleting an S3 bucket that's configured as a destination without removing the destination from your function's configuration can create a security risk. If another user knows your destination bucket's name, they can recreate the bucket in their AWS account. Records of failed invocations will be sent to their bucket, potentially exposing data from your function.

> **Warning**
>
> To ensure that invocation records from your function can't be sent to an S3 bucket in another AWS account, add a condition to your function's execution role that limits `s3:PutObject` permissions to buckets in your account.

The following example shows an IAM policy that limits your function's `s3:PutObject` permissions to buckets in your account. This policy also gives Lambda the `s3:ListBucket` permission it needs to use an S3 bucket as a destination.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
```

```
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

To add a permissions policy to your function's execution role using the AWS Management Console or AWS CLI, refer to the instructions in the following procedures:

Console

To add a permissions policy to a function's execution role (console)

1. Open the Functions page of the Lambda console.
2. Select the Lambda function whose execution role you want to modify.
3. In the **Configuration** tab, select **Permissions**.
4. In the **Execution role** tab, select your function's **Role name** to open the role's IAM console page.
5. Add a permissions policy to the role by doing the following:
    a. In the **Permissions policies** pane, choose **Add permissions** and select **Create inline policy**.
    b. In **Policy editor**, select **JSON**.
    c. Paste the policy you want to add into the editor (replacing the existing JSON), and then choose **Next**.
    d. Under **Policy details**, enter a **Policy name**.
    e. Choose **Create policy**.

AWS CLI

To add a permissions policy to a function's execution role (CLI)

1. Create a JSON policy document with the required permissions and save it in a local directory.
2. Use the IAM `put-role-policy` CLI command to add the permissions to your function's execution role. Run the following command from the directory you saved your JSON policy document in and replace the role name, policy name, and policy document with your own values.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

SNS and SQS example invocation record

The following example shows what Lambda sends to an SNS topic or SQS queue destination for a failed Kafka event source invocation. Each of the keys under `recordsInfo` contains both the Kafka topic and partition, separated by a hyphen. For example, for the key `"Topic-0"`, `Topic` is the Kafka topic, and `0` is the partition. For each topic and partition, you can use the offsets and timestamp data to find the original invocation records.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
```

```
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3 destination example invocation record

For S3 destinations, Lambda sends the entire invocation record along with the metadata to the destination. The following example shows that Lambda sends to an S3 bucket destination for a failed Kafka event source invocation. In addition to all of the fields from the previous example for SQS and SNS destinations, the `payload` field contains the original invocation record as an escaped JSON string.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
```

```
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

Tip

We recommend enabling S3 versioning on your destination bucket.

Using a Kafka topic as an on-failure destination

You can configure a Kafka topic as an on-failure destination for your Kafka event source mappings. When Lambda can't process records after exhausting retry attempts or when records exceed the maximum age, Lambda sends the failed records to the specified Kafka topic for later processing. When you configure both infinite retries and an on-failure destination, Lambda automatically applies a maximum of 10 retry attempts.

How a Kafka on-failure destination works

When you configure a Kafka topic as an on-failure destination, Lambda acts as a Kafka producer and writes failed records to the destination topic. This creates a dead letter topic (DLT) pattern within your Kafka infrastructure.

- **Same cluster requirement** – The destination topic must exist in the same Kafka cluster as your source topics.
- **Actual record content** – Kafka destinations receive the actual failed records along with failure metadata.
- **Recursion prevention** – Lambda prevents infinite loops by blocking configurations where the source and destination topics are the same.

Configuring a Kafka on-failure destination

You can configure a Kafka topic as an on-failure destination when creating or updating a Kafka event source mapping.

Configuring a Kafka destination (console)

To configure a Kafka topic as an on-failure destination (console)

1. Open the Functions page of the Lambda console.
2. Choose your function name.
3. Do one of the following:
   - To add a new Kafka trigger, under **Function overview**, choose **Add trigger**.
   - To modify an existing Kafka trigger, choose the trigger and then choose **Edit**.
4. Under **Additional settings**, for **On-failure destination**, choose **Kafka topic**.
5. For **Topic name**, enter the name of the Kafka topic where you want to send failed records.

6. Choose **Add** or **Save**.

Configuring a Kafka destination (AWS CLI)

Use the `kafka://` prefix to specify a Kafka topic as an on-failure destination.

Creating an event source mapping with Kafka destination

The following example creates a Amazon MSK event source mapping with a Kafka topic as the on-failure destination:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123 \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

For self-managed Kafka, use the same syntax:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Updating a Kafka destination

Use the `update-event-source-mapping` command to add or modify a Kafka destination:

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Record format for a Kafka destination

When Lambda sends failed records to a Kafka topic, each message contains both metadata about the failure and the actual record content.

Failure metadata

The metadata includes information about why the record failed and details about the original batch:


```
{
  "requestContext": {
    "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "condition": "RetriesExhausted",
    "approximateInvokeCount": 3
  },
  "responseContext": {
    "statusCode": 200,
    "executedVersion": "$LATEST",
    "functionError": "Unhandled"
  },
  "version": "1.0",
  "timestamp": "2019-11-14T18:16:05.568Z",
  "KafkaBatchInfo": {
    "batchSize": 1,
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123",
    "bootstrapServers": "b-1.mycluster.abc123.kafka.us-east-1.amazonaws.com:9098",
    "payloadSize": 1162,
    "recordInfo": {
      "offset": "4960118965842235937883629852182763847532018901230970472 2",
      "timestamp": "2019-11-14T18:16:04.835Z"
    }
  },
  "payload": {
    "bootstrapServers": "b-1.mycluster.abc123.kafka.us-east-1.amazonaws.com:9098",
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123",
    "records": {
      "my-topic-0": [
        {
          "headers": [],
          "key": "dGVzdC1rZXk=",
          "offset": 100,
          "partition": 0,
          "timestamp": 1749116692330,
          "timestampType": "CREATE_TIME",
          "topic": "my-topic",
```

```
          "value": "dGVzdC12YWx1ZQ=="
        }
      ]
    }
  }
}
```

Partition key behavior

Lambda uses the same partition key from the original record when producing to the destination topic. If the original record had no key, Lambda uses Kafka's default round-robin partitioning across all available partitions in the destination topic.

Requirements and limitations

- **Provisioned mode required** – A Kafka on-failure destination is only available for event source mappings with provisioned mode enabled.
- **Same cluster only** – The destination topic must exist in the same Kafka cluster as your source topics.
- **Topic permissions** – Your event source mapping must have write permissions to the destination topic. Example:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ClusterPermissions",
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeCluster",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:WriteData",
                "kafka-cluster:ReadData"
            ],
            "Resource": [
                "arn:aws:kafka:*:*:cluster/*"
            ]
        },
        {
            "Sid": "TopicPermissions",
```

```
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:WriteData",
                "kafka-cluster:ReadData"
            ],
            "Resource": [
                "arn:aws:kafka:*:*:topic/*/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "kafka:DescribeCluster",
                "kafka:GetBootstrapBrokers",
                "kafka:Produce"
            ],
            "Resource": "arn:aws:kafka:*:*:cluster/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        }
    ]
}
```

- **No recursion** – The destination topic name cannot be the same as any of your source topic names.

Kafka event source mapping logging

You can configure the system-level logging for your Kafka event source mappings to enable and filter the system logs that Lambda event pollers send to CloudWatch.

This feature is only available for Kafka event source mappings, and with Provisioned mode.

For event source mapping with logging config, you can also check the system logs from pre-built log queries in the **Monitor** tab from the page Console **Lambda** > **Additional resources** > **event source mappings** now.

How the logging works

When you set the logging config with log level in your event source mapping, the Lambda event poller sends out corresponding logs (event source mapping system logs).

The event source mapping reuses the same log destination with your Lambda function. Make sure the execution role of your Lambda function has the necessary logging permissions.

The event source mapping will have its own log stream, with date and event source mapping UUID as the log stream name, like `2020/01/01/12345678-1234-1234-1234-12345678901`.

For event source mapping system logs, you can choose between the following log levels.

Log level	Usage
DEBUG (most detail)	Detailed information for event source processing progress
INFO	Messages about the normal operation of your event source mapping
WARN (least detail)	Messages about potential warns and errors that may lead to unexpected behavior

When you select a log level, Lambda event poller sends logs at that level and lower. For example, if you set the event source mapping system log level to INFO, event poller doesn't send log outputs at the DEBUG level.

Configuring logging

You can set the logging configure when creating or updating a Kafka event source mapping.

Configuring logging (console)

To configure the logging (console)

1. Open the Functions page of the Lambda console.
2. Choose your function name.
3. Do one of the following:
   - To add a new Kafka trigger, under **Function overview**, choose **Add trigger**.
   - To modify an existing Kafka trigger, choose the trigger and then choose **Edit**.
4. Under **Event poller configuration**, for **Provisioned mode**, enable the **Configure** checkbox. And the **Log level** setting would show up.
5. Click **Log level** dropdown list and select a level for the event source mapping.
6. Choose **Add** or **Save** at the bottom to create or update the event source mapping.

Configuring logging (AWS CLI)

Creating an event source mapping with logging

The following example creates a Amazon MSK event source mapping with logging config:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123 \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --logging-config '{"SystemLogLevel":"DEBUG"}'
```

For self-managed Kafka, use the same syntax:

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
```

```
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --logging-config '{"SystemLogLevel":"DEBUG"}'
```

Updating logging config

Use the `update-event-source-mapping` command to add or modify logging config:

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --logging-config '{"SystemLogLevel":"WARN"}'
```

Record format for a Kafka event source mapping system log

When Lambda event poller sends the log, each log entry contains general event source mapping metadata and also event specific content.

WARN log record

WARN record contains errors or warnings from the event poller, and it's emitted when the event happened. For example:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1546347650000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Timeout expired while fetching topic metadata",
        "errorCode": "org.apache.kafka.common.errors.TimeoutException"
    }
}
```

INFO log record

INFO record contains Kafka consumer client configurations in each event poller, and it's emitted on the event of a consumer being built or changed. For example:

```
{
    "eventType": "POLLER_STATUS_EVENT",
    "timestamp": 1546347660000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "INFO",
    "kafkaEventSourceConnection": {
        "brokerEndpoints": "boot-abcd1234.c2.kafka-serverless.us-
east-1.amazonaws.com:9098",
        "consumerId": "12345678-1234-1234-1234-123456789012-0",
        "topics": [
            "test"
        ],
        "consumerGroupId": "12345678-1234-1234-1234-123456789012",
        "securityProtocol": "SASL_SSL",
        "saslMechanism": "AWS_MSK_IAM",
        "totalPartitionCount": 2,
        "assignedPartitionCount": 2,
        "partitionsAssignmentGeneration": 5,
        "assignedPartitions": [
            "test-0",
            "test-1"
        ],
        "networkConfig": {
            "ipAddresses": [
                "10.100.141.1"
            ],
            "subnetCidrBlock": "10.100.128.0/20",
            "securityGroups": [
                "sg-abcdefabcdefabcdef"
            ]
        }
    }
}
```

DEBUG log record

DEBUG log contains the Kafka offsets related info in event source mapping processing, and the offset info is emitted per minute. For example:

```
{
    "eventType": "KAFKA_STATUS_EVENT",
    "timestamp": 1546347670000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "DEBUG",
    "kafkaPartitionOffsets": {
        "partition": "test-1",
        "endOffset": 5004,
        "consumedOffset": 5003,
        "processedOffset": 5003,
        "committedOffset": 5004
    }
}
```

Troubleshooting Kafka event source mapping errors

The following topics provide troubleshooting advice for errors and issues that you might encounter when using Amazon MSK or self-managed Apache Kafka with Lambda.

For more help with troubleshooting, visit the AWS Knowledge Center.

Authentication and authorization errors

If any of the permissions required to consume data from the Kafka cluster are missing, Lambda displays one of the following error messages in the event source mapping under **LastProcessingResult**.

Error messages

- Cluster failed to authorize Lambda
- SASL authentication failed
- Server failed to authenticate Lambda
- Lambda failed to authenticate server
- Provided certificate or private key is invalid

Cluster failed to authorize Lambda

For SASL/SCRAM or mTLS, this error indicates that the provided user doesn't have all of the following required Kafka access control list (ACL) permissions:

- DescribeConfigs Cluster
- Describe Group
- Read Group
- Describe Topic
- Read Topic

When you create Kafka ACLs with the required `kafka-cluster` permissions, specify the topic and group as resources. The topic name must match the topic in the event source mapping. The group name must match the event source mapping's UUID.

After you add the required permissions to the execution role, it might take several minutes for the changes to take effect.

The following is an example ESM system-level log after enabling Logging Config for this issue:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567890123,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE11111-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Not authorized to access topics: [my-topic]",
        "errorCode": "org.apache.kafka.common.errors.TopicAuthorizationException"
    }
}
```

SASL authentication failed

For SASL/SCRAM or SASL/PLAIN, this error indicates that the provided sign-in credentials aren't valid.

For IAM access control, the execution role is missing the `kafka-cluster:Connect` permission for the cluster. Add this permission to the role and specify the cluster's Amazon Resource Name (ARN) as a resource.

You might see this error occurring intermittently. The cluster rejects connections after the number of TCP connections exceeds the service quota. Lambda backs off and retries until a connection is successful. After Lambda connects to the cluster and polls for records, the last processing result changes to OK.

The following is an example ESM system-level log after enabling Logging Config for this issue when using IAM authentication:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567890456,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE22222-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222/0",
```

```
    "logLevel": "WARN",
    "error": {
        "errorMessage": "[a1b2c3d4-5678-90ab-cdef-EXAMPLE22222]: Access denied",
        "errorCode": "org.apache.kafka.common.errors.SaslAuthenticationException"
    }
}
```

Server failed to authenticate Lambda

This error indicates that the Kafka broker failed to authenticate Lambda. This can occur for any of the following reasons:

- You didn't provide a client certificate for mTLS authentication.
- You provided a client certificate, but the Kafka brokers aren't configured to use mTLS authentication.
- A client certificate isn't trusted by the Kafka brokers.

Lambda failed to authenticate server

This error indicates that Lambda failed to authenticate the Kafka broker. This can occur for any of the following reasons:

- For self-managed Apache Kafka: The Kafka brokers use self-signed certificates or a private CA, but didn't provide the server root CA certificate.
- For self-managed Apache Kafka: The server root CA certificate doesn't match the root CA that signed the broker's certificate.
- Hostname validation failed because the broker's certificate doesn't contain the broker's DNS name or IP address as a subject alternative name.

Provided certificate or private key is invalid

This error indicates that the Kafka consumer couldn't use the provided certificate or private key. Make sure that the certificate and key use PEM format, and that the private key encryption uses a PBES1 algorithm.

The following is an example ESM system-level log after enabling Logging Config for this issue:

```
{
```

```
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567891234,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE44444",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE44444-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE44444/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Invalid PEM keystore configs",
        "errorCode": "org.apache.kafka.common.errors.InvalidConfigurationException"
    }
}
```

Network and connectivity errors

Network configuration issues can prevent Lambda from connecting to your Kafka cluster. The following topics describe common network-related errors.

Error messages

- Connection timeout due to security group configuration
- Kafka broker endpoints cannot be resolved

Connection timeout due to security group configuration

If the security group associated with your Kafka cluster doesn't allow inbound traffic from itself, Lambda can't connect to the cluster. Make sure that the security group's inbound rules allow traffic from the security group itself on the Kafka broker ports.

The following is an example ESM system-level log after enabling Logging Config for this issue:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567892345,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE55555",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE55555-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE55555/0",
    "logLevel": "WARN",
    "error": {
```

```
        "errorMessage": "Timeout expired while fetching topic metadata",
        "errorCode": "org.apache.kafka.common.errors.TimeoutException"
    }
}
```

You can also check the Kafka consumer INFO log to verify the connection and network configuration. The `brokerEndpoints` field shows the Kafka broker addresses, `securityProtocol` and `saslMechanism` (if applicable) show the authentication method, and the `networkConfig` field shows the IP addresses, subnet CIDR block, and security groups used by the event source mapping. Verify that the security groups listed allow the required inbound traffic:

```
{
    "eventType": "POLLER_STATUS_EVENT",
    "timestamp": 1734567892456,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-11111EXAMPLE",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-cluster/
a1b2c3d4-5678-90ab-cdef-11111EXAMPLE-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE/0",
    "logLevel": "INFO",
    "kafkaEventSourceConnection": {
        "brokerEndpoints": "boot-abcd1234.c2.kafka-serverless.us-
east-1.amazonaws.com:9098",
        "consumerId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE-0",
        "topics": [
            "my-topic"
        ],
        "consumerGroupId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE",
        "securityProtocol": "SASL_SSL",
        "saslMechanism": "AWS_MSK_IAM",
        "totalPartitionCount": 2,
        "assignedPartitionCount": 2,
        "partitionsAssignmentGeneration": 1,
        "assignedPartitions": [
            "my-topic-0",
            "my-topic-1"
        ],
        "networkConfig": {
            "ipAddresses": [
                "10.0.0.37"
            ],
            "subnetCidrBlock": "10.0.0.32/28",
            "securityGroups": [
```

```
                "sg-0123456789abcdef0"
            ]
        }
    }
}
```

Kafka broker endpoints cannot be resolved

This error indicates that the Kafka cluster doesn't exist or has been deleted. Verify that the cluster specified in the event source mapping exists and is in an active state.

The following is an example ESM system-level log after enabling Logging Config for this issue:

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567893456,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE66666",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE66666-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE66666/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "No resolvable bootstrap urls given in bootstrap.servers",
        "errorCode": "org.apache.kafka.common.config.ConfigException"
    }
}
```

Event source mapping errors

When you add your Apache Kafka cluster as an event source for your Lambda function, if your function encounters an error, your Kafka consumer stops processing records. Consumers of a topic partition are those that subscribe to, read, and process your records. Your other Kafka consumers can continue processing records, provided they don't encounter the same error.

To determine the cause of a stopped consumer, check the `StateTransitionReason` field in the response of `EventSourceMapping`. The following list describes the event source errors that you can receive:

`ESM_CONFIG_NOT_VALID`

The event source mapping configuration isn't valid.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda couldn't authenticate the event source.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda doesn't have the required permissions to access the event source.

`FUNCTION_CONFIG_NOT_VALID`

The function configuration isn't valid.

> **Note**
>
> If your Lambda event records exceed the allowed size limit of 6 MB, they can go unprocessed.

Invoking a Lambda function using an Amazon API Gateway endpoint

You can create a web API with an HTTP endpoint for your Lambda function by using Amazon API Gateway. API Gateway provides tools for creating and documenting web APIs that route HTTP requests to Lambda functions. You can secure access to your API with authentication and authorization controls. Your APIs can serve traffic over the internet or can be accessible only within your VPC.

> **Tip**
>
> Lambda offers two ways to invoke your function through an HTTP endpoint: API Gateway and Lambda function URLs. If you're not sure which is the best method for your use case, see the section called "API Gateway vs function URLs".

Resources in your API define one or more methods, such as GET or POST. Methods have an integration that routes requests to a Lambda function or another integration type. You can define each resource and method individually, or use special resource and method types to match all requests that fit a pattern. A proxy resource catches all paths beneath a resource. The `ANY` method catches all HTTP methods.

Sections

- Choosing an API type
- Adding an endpoint to your Lambda function
- Proxy integration
- Event format
- Response format
- Permissions
- Sample application
- The event handler from Powertools for AWS Lambda
- Tutorial: Using Lambda with API Gateway
- Handling Lambda errors with an API Gateway API
- Select a method to invoke your Lambda function using an HTTP request

Choosing an API type

API Gateway supports three types of APIs that invoke Lambda functions:

- HTTP API: A lightweight, low-latency RESTful API.
- REST API: A customizable, feature-rich RESTful API.
- WebSocket API: A web API that maintains persistent connections with clients for full-duplex communication.

HTTP APIs and REST APIs are both RESTful APIs that process HTTP requests and return responses. HTTP APIs are newer and are built with the API Gateway version 2 API. The following features are new for HTTP APIs:

HTTP API features

- **Automatic deployments** – When you modify routes or integrations, changes deploy automatically to stages that have automatic deployment enabled.
- **Default stage** – You can create a default stage (`$default`) to serve requests at the root path of your API's URL. For named stages, you must include the stage name at the beginning of the path.
- **CORS configuration** – You can configure your API to add CORS headers to outgoing responses, instead of adding them manually in your function code.

REST APIs are the classic RESTful APIs that API Gateway has supported since launch. REST APIs currently have more customization, integration, and management features.

REST API features

- **Integration types** – REST APIs support custom Lambda integrations. With a custom integration, you can send just the body of the request to the function, or apply a transform template to the request body before sending it to the function.
- **Access control** – REST APIs support more options for authentication and authorization.
- **Monitoring and tracing** – REST APIs support AWS X-Ray tracing and additional logging options.

For a detailed comparison, see Choose between HTTP APIs and REST APIs in the *API Gateway Developer Guide*.

WebSocket APIs also use the API Gateway version 2 API and support a similar feature set. Use a WebSocket API for applications that benefit from a persistent connection between the client and API. WebSocket APIs provide full-duplex communication, which means that both the client and the API can send messages continuously without waiting for a response.

HTTP APIs support a simplified event format (version 2.0). For an example of an event from an HTTP API, see Create AWS Lambda proxy integrations for HTTP APIs in API Gateway.

For more information, see Create AWS Lambda proxy integrations for HTTP APIs in API Gateway.

Adding an endpoint to your Lambda function

To add a public endpoint to your Lambda function

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add trigger**.
4. Select **API Gateway**.
5. Choose **Create an API** or **Use an existing API**.
    a. **New API:** For **API type**, choose **HTTP API**. For more information, see Choosing an API type.
    b. **Existing API:** Select the API from the dropdown list or enter the API ID (for example, r3pmxmplak).
6. For **Security**, choose **Open**.
7. Choose **Add**.

Proxy integration

API Gateway APIs are comprised of stages, resources, methods, and integrations. The stage and resource determine the path of the endpoint:

API path format

- `/prod/` – The `prod` stage and root resource.
- `/prod/user` – The `prod` stage and `user` resource.

- `/dev/{proxy+}` – Any route in the `dev` stage.
- `/` – (HTTP APIs) The default stage and root resource.

A Lambda integration maps a path and HTTP method combination to a Lambda function. You can configure API Gateway to pass the body of the HTTP request as-is (custom integration), or to encapsulate the request body in a document that includes all of the request information including headers, resource, path, and method.

For more information, see Lambda proxy integrations in API Gateway.

Event format

Amazon API Gateway invokes your function synchronously with an event that contains a JSON representation of the HTTP request. For a custom integration, the event is the body of the request. For a proxy integration, the event has a defined structure. For an example of a proxy event from an API Gateway REST API, see Input format of a Lambda function for proxy integration in the *API Gateway Developer Guide*.

Response format

API Gateway waits for a response from your function and relays the result to the caller. For a custom integration, you define an integration response and a method response to convert the output from the function to an HTTP response. For a proxy integration, the function must respond with a representation of the response in a specific format.

The following example shows a response object from a Node.js function. The response object represents a successful HTTP response that contains a JSON document.

Example index.mjs – Proxy integration response object (Node.js)

```
var response = {
      "statusCode": 200,
      "headers": {
        "Content-Type": "application/json"
      },
      "isBase64Encoded": false,
      "multiValueHeaders": {
        "X-Custom-Header": ["My value", "My other value"],
      },
```

```
    "body": "{\n  \"TotalCodeSize\": 104330022,\n  \"FunctionCount\": 26\n}"
  }
```

The Lambda runtime serializes the response object into JSON and sends it to the API. The API parses the response and uses it to create an HTTP response, which it then sends to the client that made the original request.

Example HTTP response

```
< HTTP/1.1 200 OK
  < Content-Type: application/json
  < Content-Length: 55
  < Connection: keep-alive
  < x-amzn-RequestId: 32998fea-xmpl-4268-8c72-16138d629356
  < X-Custom-Header: My value
  < X-Custom-Header: My other value
  < X-Amzn-Trace-Id: Root=1-5e6aa925-ccecxmplbae116148e52f036
  <
  {
    "TotalCodeSize": 104330022,
    "FunctionCount": 26
  }
```

Permissions

Amazon API Gateway gets permission to invoke your function from the function's resource-based policy. You can grant invoke permission to an entire API, or grant limited access to a stage, resource, or method.

When you add an API to your function by using the Lambda console, using the API Gateway console, or in an AWS SAM template, the function's resource-based policy is updated automatically. The following is an example function policy.

Example function policy

JSON

```
{
  "Version":"2012-10-17",
  "Id": "default",
```

```
  "Statement": [
    {
      "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLXE2F",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:111122223333:function:nodejs-apig-
function-1G3MXMPLXVXYI",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:execute-api:us-
east-2:111122223333:ktyvxmpls1/*/GET/"
        }
      }
    }
  ]
}
```

You can manage function policy permissions manually with the following API operations:

- AddPermission
- RemovePermission
- GetPolicy

To grant invocation permission to an existing API, use the `add-permission` command. Example:

```
aws lambda add-permission \
  --function-name my-function \
  --statement-id apigateway-get --action lambda:InvokeFunction \
  --principal apigateway.amazonaws.com \
  --source-arn "arn:aws:execute-api:us-east-2:123456789012:mnh1xmpli7/default/GET/"
```

You should see the following output:

```
{
```

```
    "Statement": "{\"Sid\":\"apigateway-test-2\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:my-function
\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-
east-2:123456789012:mnh1xmpli7/default/GET\"}}}"
}
```

> **Note**
>
> If your function and API are in different AWS Regions, the Region identifier in the source ARN must match the Region of the function, not the Region of the API. When API Gateway invokes a function, it uses a resource ARN that is based on the ARN of the API, but modified to match the function's Region.

The source ARN in this example grants permission to an integration on the GET method of the root resource in the default stage of an API, with ID `mnh1xmpli7`. You can use an asterisk in the source ARN to grant permissions to multiple stages, methods, or resources.

Resource patterns

- `mnh1xmpli7/*/GET/*` – GET method on all resources in all stages.
- `mnh1xmpli7/prod/ANY/user` – ANY method on the `user` resource in the `prod` stage.
- `mnh1xmpli7/*/*/*` – Any method on all resources in all stages.

For details on viewing the policy and removing statements, see Viewing resource-based IAM policies in Lambda.

Sample application

The API Gateway with Node.js sample app includes a function with an AWS SAM template that creates a REST API that has AWS X-Ray tracing enabled. It also includes scripts for deploying, invoking the function, testing the API, and cleanup.

The event handler from Powertools for AWS Lambda

The event handler from the Powertools for AWS Lambda toolkit provides routing, middleware, CORS configuration, OpenAPI spec generation, request validation, error handling, and other useful

features when writing Lambda functions invoked by an API Gateway endpoint (HTTP or REST). The event handler utility is available for Python and TypeScript/JavaScript. For more information, see Event Handler REST API in the *Powertools for AWS Lambda (Python) documentation* and Event Handler HTTP API in the *Powertools for AWS Lambda (TypeScript) documentation*.

Python

```
from aws_lambda_powertools import Logger
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools.utilities.typing.lambda_context import LambdaContext

app = APIGatewayRestResolver()
logger = Logger()

@app.get("/healthz")
def ping():
    return {"message": "health status ok"}

@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

Typescript

```
import { Router } from '@aws-lambda-powertools/event-handler/experimental-rest';
import { Logger } from '@aws-lambda-powertools/logger';
import {
  correlationPaths,
  search,
} from '@aws-lambda-powertools/logger/correlationId';
import type { Context } from 'aws-lambda/handler';

const logger = new Logger({
  correlationIdSearchFn: search,
});

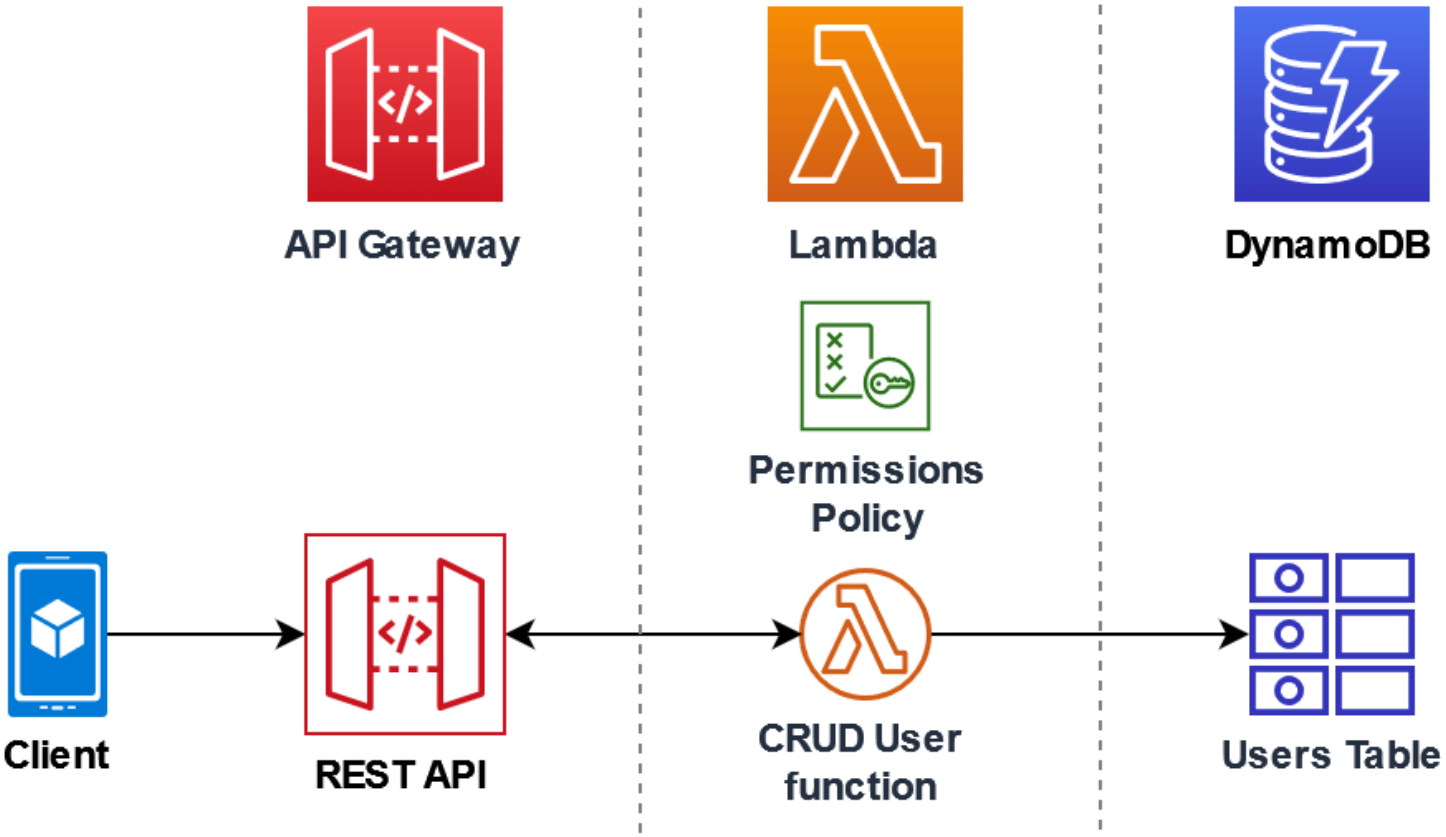
const app = new Router({ logger });

app.get("/healthz", async () => {
  return { message: "health status ok" };
});
```

```
export const handler = async (event: unknown, context: Context) => {
  // You can continue using other utilities just as before
  logger.addContext(context);
  logger.setCorrelationId(event, correlationPaths.API_GATEWAY_REST);
  return app.resolve(event, context);
};
```

Tutorial: Using Lambda with API Gateway

In this tutorial, you create a REST API through which you invoke a Lambda function using an HTTP request. Your Lambda function will perform create, read, update, and delete (CRUD) operations on a DynamoDB table. This function is provided here for demonstration, but you will learn to configure an API Gateway REST API that can invoke any Lambda function.



Using API Gateway provides users with a secure HTTP endpoint to invoke your Lambda function and can help manage large volumes of calls to your function by throttling traffic and automatically validating and authorizing API calls. API Gateway also provides flexible security controls using AWS Identity and Access Management (IAM) and Amazon Cognito. This is useful for use cases where advance authorization is required for calls to your application.

> **Tip**
>
> Lambda offers two ways to invoke your function through an HTTP endpoint: API Gateway and Lambda function URLs. If you're not sure which is the best method for your use case, see the section called "API Gateway vs function URLs".

To complete this tutorial, you will go through the following stages:

1. Create and configure a Lambda function in Python or Node.js to perform operations on a DynamoDB table.
2. Create a REST API in API Gateway to connect to your Lambda function.
3. Create a DynamoDB table and test it with your Lambda function in the console.
4. Deploy your API and test the full setup using curl in a terminal.

By completing these stages, you will learn how to use API Gateway to create an HTTP endpoint that can securely invoke a Lambda function at any scale. You will also learn how to deploy your API, and how to test it in the console and by sending an HTTP request using a terminal.

Create a permissions policy

Before you can create an execution role for your Lambda function, you first need to create a permissions policy to give your function permission to access the required AWS resources. For this tutorial, the policy allows Lambda to perform CRUD operations on a DynamoDB table and write to Amazon CloudWatch Logs.

To create the policy

1. Open the Policies page of the IAM console.
2. Choose **Create Policy**.
3. Choose the **JSON** tab, and then paste the following custom policy into the JSON editor.

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
```

```
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Effect": "Allow"
    }
  ]
}
```

4. Choose **Next: Tags**.
5. Choose **Next: Review**.
6. Under **Review policy**, for the policy **Name**, enter **`lambda-apigateway-policy`**.
7. Choose **Create policy**.

Create an execution role

An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and resources. To enable your function to perform operations on a DynamoDB table, you attach the permissions policy you created in the previous step.

To create an execution role and attach your custom permissions policy

1. Open the Roles page of the IAM console.

2. Choose **Create role**.
3. For the type of trusted entity, choose **AWS service**, then for the use case, choose **Lambda**.
4. Choose **Next**.
5. In the policy search box, enter **`lambda-apigateway-policy`**.
6. In the search results, select the policy that you created (`lambda-apigateway-policy`), and then choose **Next**.
7. Under **Role details**, for the **Role name**, enter **`lambda-apigateway-role`**, then choose **Create role**.

Create the Lambda function

1. Open the Functions page of the Lambda console and choose **Create Function**.
2. Choose **Author from scratch**.
3. For **Function name**, enter `LambdaFunctionOverHttps`.
4. For **Runtime**, choose the latest Node.js or Python runtime.
5. Under **Permissions**, expand **Change default execution role**.
6. Choose **Use an existing role**, and then select the **`lambda-apigateway-role`** role that you created earlier.
7. Choose **Create function**.
8. In the **Code source** pane, replace the default code with the following Node.js or Python code.

   Node.js

   The `region` setting must match the AWS Region where you deploy the function and create the DynamoDB table.

   Example index.mjs

```
import { DynamoDBDocumentClient, PutCommand, GetCommand,
         UpdateCommand, DeleteCommand} from "@aws-sdk/lib-dynamodb";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const ddbClient = new DynamoDBClient({ region: "us-east-2" });
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

// Define the name of the DDB table to perform the CRUD operations on
```

```
const tablename = "lambda-apigateway";

/**
 * Provide an event that contains the following keys:
 *
 *   - operation: one of 'create,' 'read,' 'update,' 'delete,' or 'echo'
 *   - payload: a JSON object containing the parameters for the table item
 *     to perform the operation on
 */
export const handler = async (event, context) => {

     const operation = event.operation;

     if (operation == 'echo'){
          return(event.payload);
     }

    else {
        event.payload.TableName = tablename;
        let response;

        switch (operation) {
          case 'create':
               response = await ddbDocClient.send(new
 PutCommand(event.payload));
               break;
          case 'read':
               response = await ddbDocClient.send(new
 GetCommand(event.payload));
               break;
          case 'update':
               response = ddbDocClient.send(new UpdateCommand(event.payload));
               break;
          case 'delete':
               response = ddbDocClient.send(new DeleteCommand(event.payload));
               break;
          default:
            response = 'Unknown operation: ${operation}';
          }
        console.log(response);
        return response;
    }
};
```

Python

Example lambda_function.py

```
import boto3

# Define the DynamoDB table that Lambda will connect to
table_name = "lambda-apigateway"

# Create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(table_name)

# Define some functions to perform the CRUD operations
def create(payload):
    return dynamo.put_item(Item=payload['Item'])

def read(payload):
    return dynamo.get_item(Key=payload['Key'])

def update(payload):
    return dynamo.update_item(**{k: payload[k] for k in ['Key',
 'UpdateExpression',
    'ExpressionAttributeNames', 'ExpressionAttributeValues'] if k in payload})

def delete(payload):
    return dynamo.delete_item(Key=payload['Key'])

def echo(payload):
    return payload

operations = {
    'create': create,
    'read': read,
    'update': update,
    'delete': delete,
    'echo': echo,
}

def lambda_handler(event, context):
    '''Provide an event that contains the following keys:
      - operation: one of the operations in the operations dict below
      - payload: a JSON object containing parameters to pass to the
        operation being performed
```

```
    '''

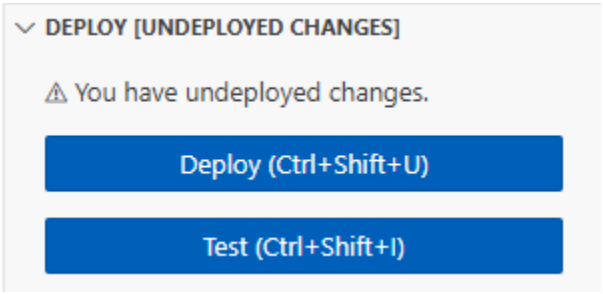
    operation = event['operation']
    payload = event['payload']

    if operation in operations:
        return operations[operation](payload)

    else:
        raise ValueError(f'Unrecognized operation "{operation}"')
```

> **Note**
>
> In this example, the name of the DynamoDB table is defined as a variable in your function code. In a real application, best practice is to pass this parameter as an environment variable and to avoid hardcoding the table name. For more information see Using AWS Lambda environment variables.

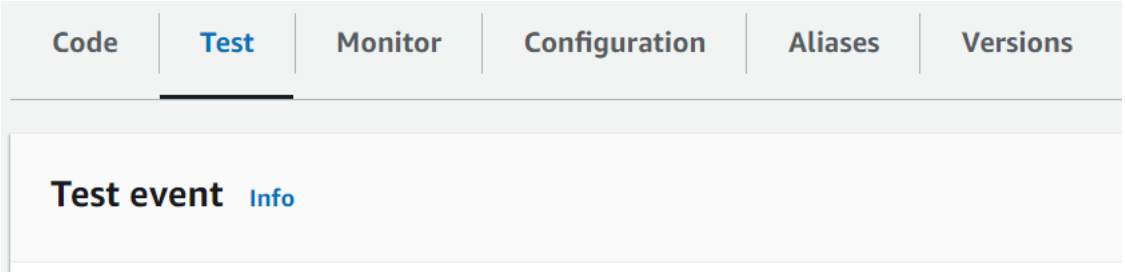
9. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Test the function

Before integrating your function with API Gateway, confirm that you have deployed the function successfully. Use the Lambda console to send a test event to your function.

1. On the Lambda console page for your function, choose the **Test** tab.



2. Scroll down to the **Event JSON** section and replace the default event with the following. This event matches the structure expected by the Lambda function.

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

3. Choose **Test**.
4. Under **Executing function: succeeded**, expand **Details**. You should see the following response:

```
{
  "somekey1": "somevalue1",
  "somekey2": "somevalue2"
}
```

Create a REST API using API Gateway

In this step, you create the API Gateway REST API you will use to invoke your Lambda function.

To create the API

1. Open the API Gateway console.
2. Choose **Create API**.
3. In the **REST API** box, choose **Build**.
4. Under **API details**, leave **New API** selected, and for **API Name**, enter **`DynamoDBOperations`**.
5. Choose **Create API**.

Create a resource on your REST API

To add an HTTP method to your API, you first need to create a resource for that method to operate on. Here you create the resource to manage your DynamoDB table.

To create the resource

1. In the API Gateway console, on the **Resources** page for your API, choose **Create resource**.
2. In **Resource details**, for **Resource name** enter **`DynamoDBManager`**.
3. Choose **Create Resource**.

Create an HTTP POST method

In this step, you create a method (`POST`) for your `DynamoDBManager` resource. You link this `POST` method to your Lambda function so that when the method receives an HTTP request, API Gateway invokes your Lambda function.

> **Note**
>
> For the purpose of this tutorial, one HTTP method (`POST`) is used to invoke a single Lambda function which carries out all of the operations on your DynamoDB table. In a real application, best practice is to use a different Lambda function and HTTP method for each operation. For more information, see The Lambda monolith in Serverless Land.

To create the POST method

1. On the **Resources** page for your API, ensure that the `/DynamoDBManager` resource is highlighted. Then, in the **Methods** pane, choose **Create method**.
2. For **Method type**, choose **POST**.
3. For **Integration type**, leave **Lambda function** selected.
4. For **Lambda function**, choose the Amazon Resource Name (ARN) for your function (`LambdaFunctionOverHttps`).
5. Choose **Create method**.

Create a DynamoDB table

Create an empty DynamoDB table that your Lambda function will perform CRUD operations on.

To create the DynamoDB table

1. Open the Tables page of the DynamoDB console.
2. Choose **Create table**.
3. Under **Table details**, do the following:
    1. For **Table name**, enter **`lambda-apigateway`**.
    2. For **Partition key**, enter **`id`**, and keep the data type set as **String**.
4. Under **Table settings**, keep the **Default settings**.
5. Choose **Create table**.

Test the integration of API Gateway, Lambda, and DynamoDB

You're now ready to test the integration of your API Gateway API method with your Lambda function and your DynamoDB table. Using the API Gateway console, you send requests directly to your POST method using the console's test function. In this step, you first use a `create` operation to add a new item to your DynamoDB table, then you use an `update` operation to modify the item.

Test 1: To create a new item in your DynamoDB table

1. In the API Gateway console, choose your API (`DynamoDBOperations`).
2. Choose the **POST** method under the `DynamoDBManager` resource.
3. Choose the **Test** tab. You might need to choose the right arrow button to show the tab.
4. Under **Test method**, leave **Query strings** and **Headers** empty. For **Request body**, paste the following JSON:

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
```

```
}
```

5. Choose **Test**.

   The results that are displayed when the test completes should show status `200`. This status code indicates that the `create` operation was successful.

   To confirm, check that your DynamoDB table now contains the new item.

6. Open the Tables page of the DynamoDB console and choose the `lambda-apigateway` table.
7. Chose **Explore table items**. In the **Items returned** pane, you should see one item with the **id** 1234ABCD and the **number** 5. Example:



Test 2: To update the item in your DynamoDB table

1. In the API Gateway console, return to your POST method's **Test** tab.
2. Under **Test method**, leave **Query strings** and **Headers** empty. For **Request body**, paste the following JSON:

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
        "UpdateExpression": "SET #num = :newNum",
        "ExpressionAttributeNames": {
            "#num": "number"
        },
        "ExpressionAttributeValues": {
            ":newNum": 10
        }
    }
```

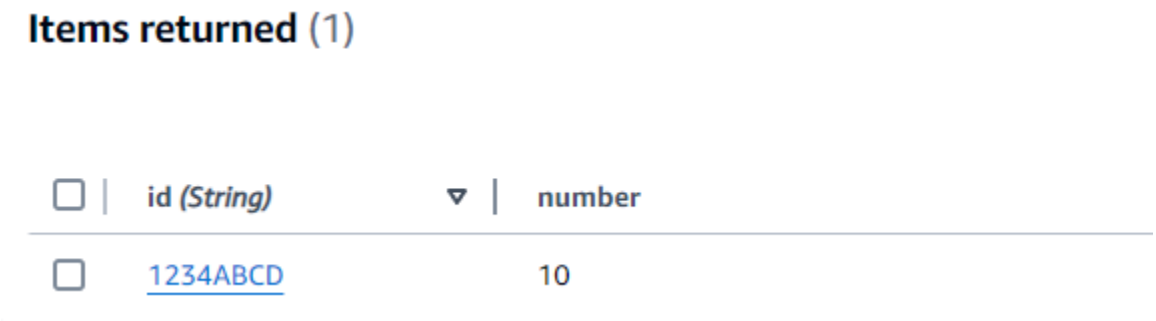
```
}
```

3. Choose **Test**.

   The results which are displayed when the test completes should show status `200`. This status code indicates that the `update` operation was successful.

   To confirm, check that the item in your DynamoDB table has been modified.

4. Open the Tables page of the DynamoDB console and choose the `lambda-apigateway` table.
5. Chose **Explore table items**. In the **Items returned** pane, you should see one item with the **id** `1234ABCD` and the **number** `10`.



Deploy the API

For a client to call the API, you must create a deployment and an associated stage. A stage represents a snapshot of your API including its methods and integrations.

To deploy the API

1. Open the **APIs** page of the API Gateway console and choose the `DynamoDBOperations` API.
2. On the **Resources** page for your API choose **Deploy API**.
3. For **Stage**, choose ***New stage***, then for **Stage name**, enter **`test`**.
4. Choose **Deploy**.
5. In the **Stage details** pane, copy the **Invoke URL**. You will use this in the next step to invoke your function using an HTTP request.

Use curl to invoke your function using HTTP requests

You can now invoke your Lambda function by issuing an HTTP request to your API. In this step, you will create a new item in your DynamoDB table and then perform read, update, and delete operations on that item.

To create an item in your DynamoDB table using curl

1. Open a terminal or command prompt on your local machine and run the following `curl` command using the invoke URL you copied in the previous step. This command uses the following options:

   - `-H`: Adds a custom header to the request. Here, it specifies the content type as JSON.
   - `-d`: Sends data in the request body. This option uses an HTTP POST method by default.

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number":
    15}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
    \"create\", \"payload\": {\"Item\": {\"id\": \"5678EFGH\", \"number\": 15}}}'
   ```

   If the operation was successful, you should see a response returned with an HTTP status code of 200.
2. You can also use the DynamoDB console to verify that the new item is in your table by doing the following:

   1. Open the Tables page of the DynamoDB console and choose the `lambda-apigateway` table.
   2. Choose **Explore table items**. In the **Items returned** pane, you should see an item with the **id** `5678EFGH` and the **number** 15.

To read the item in your DynamoDB table using curl

- In your terminal or command prompt, run the following `curl` command to read the value of the item you just created. Use your own invoke URL.

  Linux/macOS

  ```
  curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
   \
  -H "Content-Type: application/json" \
  -d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
  ```

  PowerShell

  ```
  curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
  DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\": \"read
  \", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
  ```

  You should see output like one of the following depending on whether you chose the Node.js or Python function code:

  Node.js

  ```
  {"$metadata":
  {"httpStatusCode":200,"requestId":"7BP3G5Q0C0O1E50FBQI9NS099JVV4KQNSO5AEMVJF66Q9ASUAAJG"
  "attempts":1,"totalRetryDelay":0},"Item":{"id":"5678EFGH","number":15}}
  ```

  Python

  ```
  {"Item":{"id":"5678EFGH","number":15},"ResponseMetadata":
  {"RequestId":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG",
  "HTTPStatusCode":200,"HTTPHeaders":{"server":"Server","date":"Wed, 31 Jul 2024
   00:37:01 GMT","content-type":"application/x-amz-json-1.0",
  "content-length":"52","connection":"keep-alive","x-amzn-
  requestid":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG","x-amz-
  crc32":"2589610852"},
  "RetryAttempts":0}}
  ```

To update the item in your DynamoDB table using curl

1. In your terminal or command prompt, run the following `curl` command to update the item you just created by changing the `number` value. Use your own invoke URL.

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "update", "payload": {"Key": {"id": "5678EFGH"},
    "UpdateExpression": "SET #num = :new_value", "ExpressionAttributeNames":
    {"#num": "number"}, "ExpressionAttributeValues": {":new_value": 42}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
    \"update\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}, \"UpdateExpression
   \": \"SET #num = :new_value\", \"ExpressionAttributeNames\": {\"#num\": \"number
   \"}, \"ExpressionAttributeValues\": {\":new_value\": 42}}}'
   ```

2. To confirm that the value of `number` for the item has been updated, run another read command:

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\": \"read
   \", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
   ```

To delete the item in your DynamoDB table using curl

1. In your terminal or command prompt, run the following `curl` command to delete the item you just created. Use your own invoke URL.

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
    \"delete\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
   ```

2. Confirm that the delete operation was successful. In the **Items returned** pane of the DynamoDB console **Explore items** page, verify that the item with **id** 5678EFGH is no longer in the table.

Clean up your resources (optional)

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the execution role

1. Open the Roles page of the IAM console.

2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the API

1. Open the APIs page of the API Gateway console.
2. Select the API you created.
3. Choose **Actions**, **Delete**.
4. Choose **Delete**.

To delete the DynamoDB table

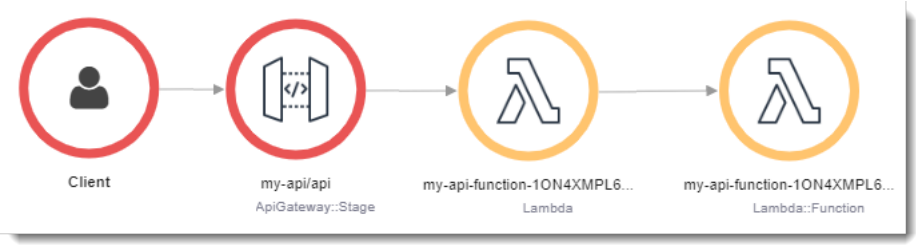
1. Open the Tables page of the DynamoDB console.
2. Select the table you created.
3. Choose **Delete**.
4. Enter **`delete`** in the text box.
5. Choose **Delete table**.

Handling Lambda errors with an API Gateway API

API Gateway treats all invocation and function errors as internal errors. If the Lambda API rejects the invocation request, API Gateway returns a 500 error code. If the function runs but returns an error, or returns a response in the wrong format, API Gateway returns a 502. In both cases, the body of the response from API Gateway is `{"message": "Internal server error"}`.

> **Note**
>
> API Gateway does not retry any Lambda invocations. If Lambda returns an error, API Gateway returns an error response to the client.

The following example shows an X-Ray trace map for a request that resulted in a function error and a 502 from API Gateway. The client receives the generic error message.

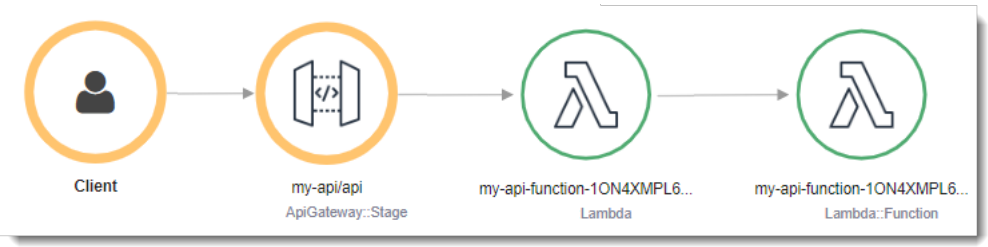


To customize the error response, you must catch errors in your code and format a response in the required format.

Example index.mjs – Error formatting

```
var formatError = function(error){
  var response = {
    "statusCode": error.statusCode,
    "headers": {
      "Content-Type": "text/plain",
      "x-amzn-ErrorType": error.code
    },
    "isBase64Encoded": false,
    "body": error.code + ": " + error.message
  }
  return response
}
```

API Gateway converts this response into an HTTP error with a custom status code and body. In the trace map, the function node is green because it handled the error.



Select a method to invoke your Lambda function using an HTTP request

Many common use cases for Lambda involve invoking your function using an HTTP request. For example, you might want a web application to invoke your function through a browser request.

Lambda functions can also be used to create full REST APIs, handle user interactions from mobile apps, process data from external services via HTTP calls, or create custom webhooks.

The following sections explain what your choices are for invoking Lambda through HTTP and provide information to help you make the right decision for your particular use case.

What are your choices when selecting an HTTP invoke method?

Lambda offers two main methods to invoke a function using an HTTP request - function URLs and API Gateway. The key differences between these two options are as follows:

- **Lambda function URLs** provide a simple, direct HTTP endpoint for a Lambda function. They are optimized for simplicity and cost-effectiveness and provide the fastest path to expose a Lambda function via HTTP.
- **API Gateway** is a more advanced service for building fully-featured APIs. API Gateway is optimized for building and managing productions APIs at scale and provides comprehensive tools for security, monitoring, and traffic management.

Recommendations if you already know your requirements

If you're already clear on your requirements, here are our basic recommendations:

We recommend **function URLs** for simple applications or prototyping where you only need basic authentication methods and request/response handling and where you want to keep costs and complexity to a minimum.

API Gateway is a better choice for production applications at scale or for cases where you need more advanced features like OpenAPI Description support, a choice of authentication options, custom domain names, or rich request/response handling including throttling, caching, and request/response transformation.

What to consider when selecting a method to invoke your Lambda function

When selecting between function URLs and API Gateway, you need to consider the following factors:

- Your authentication needs, such as whether you require OAuth or Amazon Cognito to authenticate users
- Your scaling requirements and the complexity of the API you want to implement

- Whether you need advanced features such as request validation and request/response formatting
- Your monitoring requirements
- Your cost goals

By understanding these factors, you can select the option that best balances your security, complexity, and cost requirements.

The following information summarizes the main differences between the two options.

Authentication

- **Function URLs** provide basic authentication options through AWS Identity and Access Management (IAM). You can configure your endpoints to be either public (no authentication) or to require IAM authentication. With IAM authentication, you can use standard AWS credentials or IAM roles to control access. While straightforward to set up, this approach provides limited options compared with other authenticaton methods.
- **API Gateway** provides access to a more comprehensive range of authentication options. As well as IAM authentication, you can use Lambda authorizers (custom authentication logic), Amazon Cognito user pools, and OAuth2.0 flows. This flexibility allows you to implement complex authentication schemes, including third-party authentication providers, token-based authentication, and multi-factor authentication.

Request/response handling

- **Function URLs** provide basic HTTP request and response handling. They support standard HTTP methods and include built-in cross-origin resource sharing (CORS) support. While they can handle JSON payloads and query parameters naturally, they don't offer request transformation or validation capabilities. Response handling is similarly straightforward – the client receives the response from your Lambda function exactly as Lambda returns it.
- **API Gateway** provides sophisticated request and response handling capabilities. You can define request validators, transform requests and responses using mapping templates, set up request/response headers, and implement response caching. API Gateway also supports binary payloads and custom domain names and can modify responses before they reach the client. You can set up models for request/response validation and transformation using JSON Schema.

Scaling

- **Function URLs** scale directly with your Lambda function's concurrency limits and handle traffic spikes by scaling your function up to its maximum configured concurrency limit. Once that limit is reached, Lambda responds to additional requests with HTTP 429 responses. There's no built-in queuing mechanism, so handling scaling is entirely dependent on your Lambda function's configuration. By default, Lambda functions have a limit of 1,000 concurrent executions per AWS Region.
- **API Gateway** provides additional scaling capabilities on top of Lambda's own scaling. It includes built-in request queuing and throttling controls, allowing you to manage traffic spikes more gracefully. API Gateway can handle up to 10,000 requests per second per region by default, with a burst capacity of 5,000 requests per second. It also provides tools to throttle requests at different levels (API, stage, or method) to protect your backend.

Monitoring

- **Function URLs** offer basic monitoring through Amazon CloudWatch metrics, including request count, latency, and error rates. You get access to standard Lambda metrics and logs, which show the raw requests coming into your function. While this provides essential operational visibility, the metrics are focused mainly on function execution.
- **API Gateway** provides comprehensive monitoring capabilities including detailed metrics, logging, and tracing options. You can monitor API calls, latency, error rates, and cache hit/miss rates through CloudWatch. API Gateway also integrates with AWS X-Ray for distributed tracing and provides customizable logging formats.

Cost

- **Function URLs** follow the standard Lambda pricing model – you only pay for function invocations and compute time. There are no additional charges for the URL endpoint itself. This makes it a cost-effective choice for simple APIs or low-traffic applications if you don't need the additional features of API Gateway.
- **API Gateway** offers a free tier that includes one million API calls received for REST APIs and one million API calls received for HTTP APIs. After this, API Gateway charges for API calls, data transfer, and caching (if enabled). Refer to the API Gateway pricing page to understand the costs for your own use case.

Other features

- **Function URLs** are designed for simplicity and direct Lambda integration. They support both HTTP and HTTPS endpoints, offer built-in CORS support, and provide dual-stack (IPv4 and IPv6) endpoints. While they lack advanced features, they excel in scenarios where you need a quick, straightforward way to expose Lambda functions via HTTP.
- **API Gateway** includes numerous additional features such as API versioning, stage management, API keys for usage plans, API documentation through Swagger/OpenAPI, WebSocket APIs, private APIs within a VPC, and WAF integration for additional security. It also supports canary deployments, mock integrations for testing, and integration with other AWS services beyond Lambda.

Select a method to invoke your Lambda function

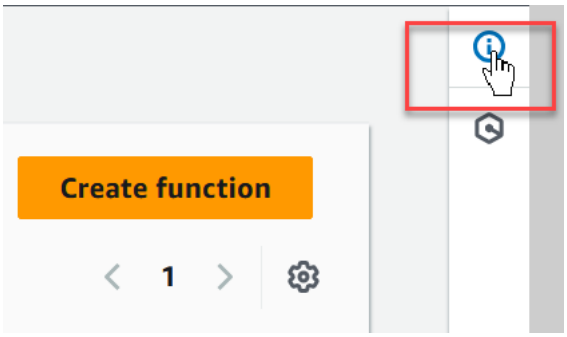
Now that you've read about the criteria for selecting between Lambda function URLs and API Gateway and the key differences between them, you can select the option that best meets your needs and use the following resources to help you get started using it.

Function URLs

Get started with function URLs with the following resources

- Follow the tutorial Creating a Lambda function with a function URL
- Learn more about function URLs in the the section called “Function URLs” chapter of this guide
- Try the in-console guided tutorial **Create a simple web app** by doing the following:

1. Open the functions page of the Lambda console.
2. Open the help panel by choosing the icon in the top right corner of the screen.



3. Select **Tutorials**.
4. In **Create a simple web app**, choose **Start tutorial**.

API Gateway

Get started with Lambda and API Gateway with the following resources

- Follow the tutorial Using Lambda with API Gateway to create a REST API integrated with a backend Lambda function.
- Learn more about the different kinds of API offered by API Gateway in the following sections of the *Amazon API Gateway Developer Guide*:
  - API Gateway REST APIs
  - API Gateway HTTP APIs
  - API Gateway WebSocket APIs
- Try one or more of the examples in the Tutorials and workshops section of the *Amazon API Gateway Developer Guide*.

Using AWS Lambda with AWS Infrastructure Composer

AWS Infrastructure Composer is a visual builder for desiging modern applications on AWS. You design your application architecture by dragging, grouping, and connecting AWS services in a visual canvas. Infrastructure Composer creates infrastructure as code (IaC) templates from your design that you can deploy using AWS SAM or CloudFormation.

Exporting a Lambda function to Infrastructure Composer

You can get started using Infrastructure Composer by creating a new project based on the configuration of an existing Lambda function using the Lambda console. To export your function's configuration and code to Infrastructure Composer to create a new project, do the following:

1. Open the Functions page of the Lambda console.
2. Select the function you want to use as a basis for your Infrastructure Composer project.
3. In the **Function overview** pane, choose **Export to Infrastructure Composer**.

   To export your function's configuration and code to Infrastructure Composer, Lambda creates an Amazon S3 bucket in your account to temporarily store this data.

4. In the dialog box, choose **Confirm and create project** to accept the default name for this bucket and export your function's configuration and code to Infrastructure Composer.
5. (Optional) To choose another name for the Amazon S3 bucket that Lambda creates, enter a new name and choose **Confirm and create project**. Amazon S3 bucket names must be globally unique and follow the bucket naming rules.
6. To save your project and function files in Infrastructure Composer, activate local sync mode.

> **Note**
>
> If you've used the **Export to Application Composer** feature before and created an Amazon S3 bucket using the default name, Lambda can re-use this bucket if it still exists. Accept the default bucket name in the dialog box to re-use the existing bucket.

Amazon S3 transfer bucket configuration

The Amazon S3 bucket that Lambda creates to transfer your function's configuration automatically encrypts objects using the AES 256 encryption standard. Lambda also configures the bucket to use the bucket owner condition to ensure that only your AWS account is able to add objects to the bucket.

Lambda configures the bucket to automatically delete objects 10 days after they are uploaded. However, Lambda doesn't automaticaly delete the bucket itself. To delete the bucket from your AWS account, follow the instructions in Deleting a bucket. The default bucket name uses the prefix `lambdasam`, a 10-digit alphanumeric string, and the AWS Region you created your function in:

```
lambdasam-06f22da95b-us-east-1
```

To avoid additional charges being added to your AWS account, we recommend that you delete the Amazon S3 bucket as soon as you have finished exporting your function to Infrastructure Composer.

Standard Amazon S3 pricing applies.

Required permissions

To use the Lambda integration with Infrastructure Composer feature, you need certain permissions to download an AWS SAM template and to write your function's configuration to Amazon S3.

To download an AWS SAM template, you must have permission to use the following API actions:

- GetPolicy
- iam:GetPolicyVersion
- iam:GetRole
- iam:GetRolePolicy
- iam:ListAttachedRolePolicies
- iam:ListRolePolicies
- iam:ListRoles

You can grant permission to use all of these actions by adding the `AWSLambda_ReadOnlyAccess` AWS managed policy to your IAM user role.

For Lambda to write your function's configuration to Amazon S3, you must have permission to use the following API actions:

- S3:PutObject
- S3:CreateBucket
- S3:PutBucketEncryption
- S3:PutBucketLifecycleConfiguration

If you are unable to export your function's configuration to Infrastructure Composer, check that your account has the required permissions for these operations. If you have the required permissions, but still cannot export your function's configuration, check for any resource-based policies that might limit access to Amazon S3.

Other resources

For a more detailed tutorial on how to design a serverless application in Infrastructure Composer based on an existing Lambda function, see the section called “Infrastructure as code (IaC)”.

To use Infrastructure Composer and AWS SAM to design and deploy a complete serverless application using Lambda, you can also follow the AWS Infrastructure Composer tutorial in the AWS Serverless Patterns Workshop.

Using AWS Lambda with CloudFormation

In an AWS CloudFormation template, you can specify a Lambda function as the target of a custom resource. Use custom resources to process parameters, retrieve configuration values, or call other AWS services during stack lifecycle events.

The following example invokes a function that's defined elsewhere in the template.

Example– Custom resource definition

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

The service token is the Amazon Resource Name (ARN) of the function that CloudFormation invokes when you create, update, or delete the stack. You can also include additional properties like `FunctionName`, which CloudFormation passes to your function as is.

CloudFormation invokes your Lambda function asynchronously with an event that includes a callback URL.

Example– CloudFormation message event

```
{
    "RequestType": "Create",
    "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
    "ResponseURL": "https://cloudformation-custom-resource-response-useast1.s3-us-
east-1.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-1%3A123456789012%3Astack/
lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke
%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM
%2F9Rf1o4%3D",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke",
    "ResourceType": "AWS::CloudFormation::CustomResource",
```

```
    "ResourceProperties": {
        "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
        "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
    }
}
```

The function is responsible for returning a response to the callback URL that indicates success or failure. For the full response syntax, see Custom resource response objects.

Example– CloudFormation custom resource response

```
{
    "Status": "SUCCESS",
    "PhysicalResourceId": "2019/04/18/[$LATEST]b3d1bfc65f19ec610654e4d9b9de47a0",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke"
}
```

CloudFormation provides a library called `cfn-response` that handles sending the response. If you define your function within a template, you can require the library by name. CloudFormation then adds the library to the deployment package that it creates for the function.

If your function that a Custom Resource uses has an Elastic Network Interface attached to it, add the following resources to the VPC policy where **`region`** is the Region the function is in without the dashes. For example, `us-east-1` is `useast1`. This will allow the Custom Resource to respond to the callback URL that sends a signal back to the CloudFormation stack.

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

The following example function invokes a second function. If the call succeeds, the function sends a success response to CloudFormation, and the stack update continues. The template uses the AWS::Serverless::Function resource type provided by AWS Serverless Application Model.

Example– Custom resource function

```
Transform: 'AWS::Serverless-2016-10-31'
```

```
Resources:
  primer:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs16.x
      InlineCode: |
        var aws = require('aws-sdk');
        var response = require('cfn-response');
        exports.handler = function(event, context) {
            // For Delete requests, immediately send a SUCCESS response.
            if (event.RequestType == "Delete") {
                response.send(event, context, "SUCCESS");
                return;
            }
            var responseStatus = "FAILED";
            var responseData = {};
            var functionName = event.ResourceProperties.FunctionName
            var lambda = new aws.Lambda();
            lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
                if (err) {
                    responseData = {Error: "Invoke call failed"};
                    console.log(responseData.Error + ":\n", err);
                }
                else responseStatus = "SUCCESS";
                response.send(event, context, responseStatus, responseData);
            });
        };
      Description: Invoke a function to create a log stream.
      MemorySize: 128
      Timeout: 8
      Role: !GetAtt role.Arn
      Tracing: Active
```

If the function that the custom resource invokes isn't defined in a template, you can get the source code for `cfn-response` from cfn-response module in the AWS CloudFormation User Guide.

For more information about custom resources, see Custom resources in the *AWS CloudFormation User Guide*.

Process Amazon DocumentDB events with Lambda

You can use a Lambda function to process events in an Amazon DocumentDB (with MongoDB compatibility) change stream by configuring an Amazon DocumentDB cluster as an event source. Then, you can automate event-driven workloads by invoking your Lambda function each time that data changes with your Amazon DocumentDB cluster.

> **Note**
>
> Lambda supports version 4.0 and 5.0 of Amazon DocumentDB only. Lambda doesn't support version 3.6.
> Also, for event source mappings, Lambda supports instance-based clusters and regional clusters only. Lambda doesn't support elastic clusters or global clusters. This limitation doesn't apply when using Lambda as a client to connect to Amazon DocumentDB. Lambda can connect to all cluster types to perform CRUD operations.

Lambda processes events from Amazon DocumentDB change streams sequentially in the order in which they arrive. Because of this, your function can handle only one concurrent invocation from Amazon DocumentDB at a time. To monitor your function, you can track its concurrency metrics.

> **Warning**
>
> Lambda event source mappings process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

Topics

- Example Amazon DocumentDB event
- Prerequisites and permissions
- Configure network security
- Creating an Amazon DocumentDB event source mapping (console)
- Creating an Amazon DocumentDB event source mapping (SDK or CLI)
- Polling and stream starting positions

- Monitoring your Amazon DocumentDB event source
- Tutorial: Using AWS Lambda with Amazon DocumentDB Streams

Example Amazon DocumentDB event


```
{
    "eventSourceArn": "arn:aws:rds:us-
east-1:123456789012:cluster:canaryclusterb2a659a2-qo5tcmqkcl03",
    "events": [
        {
            "event": {
                "_id": {
                    "_data": "0163eeb6e7000000090100000009000041e1"
                },
                "clusterTime": {
                    "$timestamp": {
                        "t": 1676588775,
                        "i": 9
                    }
                },
                "documentKey": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    }
                },
                "fullDocument": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    },
                    "anyField": "sampleValue"
                },
                "ns": {
                    "db": "test_database",
                    "coll": "test_collection"
                },
                "operationType": "insert"
            }
        }
    ],
    "eventSource": "aws:docdb"
}
```

For more information about the events in this example and their shapes, see Change Events on the MongoDB Documentation website.

Prerequisites and permissions

Before you can use Amazon DocumentDB as an event source for your Lambda function, note the following prerequisites. You must:

- **Have an existing Amazon DocumentDB cluster in the same AWS account and AWS Region as your function.** If you don't have an existing cluster, you can create one by following the steps in Get Started with Amazon DocumentDB in the *Amazon DocumentDB Developer Guide*. Alternatively, the first set of steps in Tutorial: Using AWS Lambda with Amazon DocumentDB Streams guide you through creating an Amazon DocumentDB cluster with all the necessary prerequisites.
- **Allow Lambda to access the Amazon Virtual Private Cloud (Amazon VPC) resources associated with your Amazon DocumentDB cluster.** For more information, see Configure network security.
- **Enable TLS on your Amazon DocumentDB cluster.** This is the default setting. If you disable TLS, then Lambda cannot communicate with your cluster.
- **Activate change streams on your Amazon DocumentDB cluster.** For more information, see Using Change Streams with Amazon DocumentDB in the *Amazon DocumentDB Developer Guide*.
- **Provide Lambda with credentials to access your Amazon DocumentDB cluster.** When setting up the event source, provide the AWS Secrets Manager key that contains the authentication details (username and password) required to access your cluster. To provide this key during setup, do either of the following:
  - If you're using the Lambda console for setup, then provide the key in the **Secrets manager key** field.
  - If you're using the AWS Command Line Interface (AWS CLI) for setup, then provide this key in the `source-access-configurations` option. You can include this option with either the `create-event-source-mapping` command or the `update-event-source-mapping` command. For example:

```
aws lambda create-event-source-mapping \
    ...
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
    ...
```

- **Grant Lambda permissions to manage resources related to your Amazon DocumentDB stream.** Manually add the following permissions to your function's execution role:
  - rds:DescribeDBClusters
  - rds:DescribeDBClusterParameters
  - rds:DescribeDBSubnetGroups
  - ec2:CreateNetworkInterface
  - ec2:DescribeNetworkInterfaces
  - ec2:DescribeVpcs
  - ec2:DeleteNetworkInterface
  - ec2:DescribeSubnets
  - ec2:DescribeSecurityGroups
  - kms:Decrypt
  - secretsmanager:GetSecretValue
- **Keep the size of Amazon DocumentDB change stream events that you send to Lambda under 6 MB.** Lambda supports payload sizes of up to 6 MB. If your change stream tries to send Lambda an event larger than 6 MB, then Lambda drops the message and emits the `OversizedRecordCount` metric. Lambda emits all metrics on a best-effort basis.

> **Note**
>
> While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, Amazon DocumentDB, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

Configure network security

To give Lambda full access to Amazon DocumentDB through your event source mapping, either your cluster must use a public endpoint (public IP address), or you must provide access to the Amazon VPC you created the cluster in.

When you use Amazon DocumentDB with Lambda, create AWS PrivateLink VPC endpoints that provide your function access to the resources in your Amazon VPC.

> **Note**
>
> AWS PrivateLink VPC endpoints are required for functions with event source mappings that use the default (on-demand) mode for event pollers. If your event source mapping uses provisioned mode, you don't need to configure AWS PrivateLink VPC endpoints.

Create an endpoint to provide access to the following resources:

- Lambda — Create an endpoint for the Lambda service principal.
- AWS STS — Create an endpoint for the AWS STS in order for a service principal to assume a role on your behalf.
- Secrets Manager — If your cluster uses Secrets Manager to store credentials, create an endpoint for Secrets Manager.

Alternatively, configure a NAT gateway on each public subnet in the Amazon VPC. For more information, see the section called “Internet access for VPC functions”.

When you create an event source mapping for Amazon DocumentDB, Lambda checks whether Elastic Network Interfaces (ENIs) are already present for the subnets and security groups configured for your Amazon VPC. If Lambda finds existing ENIs, it attempts to re-use them. Otherwise, Lambda creates new ENIs to connect to the event source and invoke your function.

> **Note**
>
> Lambda functions always run inside VPCs owned by the Lambda service. Your function's VPC configuration does not affect the event source mapping. Only the networking configuration of the event source's determines how Lambda connects to your event source.

Configure the security groups for the Amazon VPC containing your cluster. By default, Amazon DocumentDB uses the following ports: 27017.

- Inbound rules – Allow all traffic on the default broker port for the security group associated with your event source. Alternatively, you can use a self-referencing security group rule to allow access from instances within the same security group.
- Outbound rules – Allow all traffic on port 443 for external destinations if your function needs to communicate with AWS services. Alternatively, you can also use a self-referencing security group rule to limit access to the broker if you don't need to communicate with other AWS services.
- Amazon VPC endpoint inbound rules — If you are using an Amazon VPC endpoint, the security group associated with your Amazon VPC endpoint must allow inbound traffic on port 443 from the cluster security group.

If your cluster uses authentication, you can also restrict the endpoint policy for the Secrets Manager endpoint. To call the Secrets Manager API, Lambda uses your function role, not the Lambda service principal.

Example VPC endpoint policy — Secrets Manager endpoint

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
              "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
          }
      ]
  }
```

When you use Amazon VPC endpoints, AWS routes your API calls to invoke your function using the endpoint's Elastic Network Interface (ENI). The Lambda service principal needs to call `lambda:InvokeFunction` on any roles and functions that use those ENIs.

By default, Amazon VPC endpoints have open IAM policies that allow broad access to resources. Best practice is to restrict these policies to perform the needed actions using that endpoint. To ensure that your event source mapping is able to invoke your Lambda function, the VPC

endpoint policy must allow the Lambda service principal to call `sts:AssumeRole` and `lambda:InvokeFunction`. Restricting your VPC endpoint policies to allow only API calls originating within your organization prevents the event source mapping from functioning properly, so `"Resource": "*"` is required in these policies.

The following example VPC endpoint policies show how to grant the required access to the Lambda service principal for the AWS STS and Lambda endpoints.

Example VPC Endpoint policy — AWS STS endpoint

```
{
      "Statement": [
          {
              "Action": "sts:AssumeRole",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
    }
```

Example VPC Endpoint policy — Lambda endpoint

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Creating an Amazon DocumentDB event source mapping (console)

For a Lambda function to read from an Amazon DocumentDB cluster's change stream, create an event source mapping. This section describes how to do this from the Lambda console. For AWS SDK and AWS CLI instructions, see the section called "Creating an Amazon DocumentDB event source mapping (SDK or CLI)".

To create an Amazon DocumentDB event source mapping (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Under **Function overview**, choose **Add trigger**.
4. Under **Trigger configuration**, in the dropdown list, choose **DocumentDB**.
5. Configure the required options, and then choose **Add**.

Lambda supports the following options for Amazon DocumentDB event sources:

- **DocumentDB cluster** – Select an Amazon DocumentDB cluster.
- **Activate trigger** – Choose whether you want to activate the trigger immediately. If you select this check box, then your function immediately starts receiving traffic from the specified Amazon DocumentDB change stream upon creation of the event source mapping. We recommend that you clear the check box to create the event source mapping in a deactivated state for testing. After creation, you can activate the event source mapping at any time.
- **Database name** – Enter the name of a database within the cluster to consume.
- (Optional) **Collection name** – Enter the name of a collection within the database to consume. If you don't specify a collection, then Lambda listens to all events from each collection in the database.
- **Batch size** – Set the maximum number of messages to retrieve in a single batch, up to 10,000. The default batch size is 100.
- **Starting position** – Choose the position in the stream to start reading records from.
  - **Latest** – Process only new records that are added to the stream. Your function starts processing records only after Lambda finishes creating your event source. This means that some records may be dropped until your event source is created successfully.
  - **Trim horizon** – Process all records in the stream. Lambda uses the log retention duration of your cluster to determine where to start reading events from. Specifically, Lambda starts

reading from `current_time - log_retention_duration`. Your change stream must already be active before this timestamp for Lambda to read all events properly.

- **At timestamp** – Process records starting from a specific time. Your change stream must already be active before the specified timestamp for Lambda to read all events properly.

- **Authentication** – Choose the authentication method for accessing the brokers in your cluster.
  - **BASIC_AUTH** – With basic authentication, you must provide the Secrets Manager key that contains the credentials to access your cluster.
- **Secrets Manager key** – Choose the Secrets Manager key that contains the authentication details (username and password) required to access your Amazon DocumentDB cluster.
- (Optional) **Batch window** – Set the maximum amount of time in seconds to gather records before invoking your function, up to 300.
- (Optional) **Full document configuration** – For document update operations, choose what you want to send to the stream. The default value is `Default`, which means that for each change stream event, Amazon DocumentDB sends only a delta describing the changes made. For more information about this field, see FullDocument in the MongoDB Javadoc API documentation.
  - **Default** – Lambda sends only a partial document describing the changes made.
  - **UpdateLookup** – Lambda sends a delta describing the changes, along with a copy of the entire document.

Creating an Amazon DocumentDB event source mapping (SDK or CLI)

To create or manage an Amazon DocumentDB event source mapping with an AWS SDK, you can use the following API operations:

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

To create the event source mapping with the AWS CLI, use the `create-event-source-mapping` command. The following example uses this command to map a function named `my-function` to an Amazon DocumentDB change stream. The event source is specified by an Amazon Resource Name (ARN), with a batch size of 500, starting from the timestamp in Unix time. The command

also specifies the Secrets Manager key that Lambda uses to connect to Amazon DocumentDB. Additionally, it includes `document-db-event-source-config` parameters that specify the database and the collection to read from.

```
aws lambda create-event-source-mapping --function-name my-function \
    --event-source-arn arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy
    --batch-size 500 \
    --starting-position AT_TIMESTAMP \
    --starting-position-timestamp 1541139109 \
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocDBSecret-BAtjxi"}]' \
    --document-db-event-source-config '{"DatabaseName":"test_database",
 "CollectionName": "test_collection"}' \
```

You should see output that looks like this:

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541348195.412,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action"
}
```

After creation, you can use the `update-event-source-mapping` command to update the settings for your Amazon DocumentDB event source. The following example updates the batch size to 1,000 and the batch window to 10 seconds. For this command, you need the UUID of your event source mapping, which you can retrieve using the `list-event-source-mapping` command or the Lambda console.

```
aws lambda update-event-source-mapping --function-name my-function \
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
    --batch-size 1000 \
    --batch-window 10
```

You should see this output that looks like this:

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Updating",
    "StateTransitionReason": "User action"
}
```

Lambda updates settings asynchronously, so you may not see these changes in the output until the process completes. To view the current settings of your event source mapping, use the [get-event-source-mapping] command.

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

You should see this output that looks like this:

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "BatchSize": 1000,
```

```
    "MaximumBatchingWindowInSeconds": 10,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Enabled",
    "StateTransitionReason": "User action"
}
```

To delete your Amazon DocumentDB event source mapping, use the [delete-event-source-mapping](#) command.

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

Polling and stream starting positions

Be aware that stream polling during event source mapping creation and updates is eventually consistent.

- During event source mapping creation, it may take several minutes to start polling events from the stream.
- During event source mapping updates, it may take several minutes to stop and restart polling events from the stream.

This behavior means that if you specify `LATEST` as the starting position for the stream, the event source mapping could miss events during creation or updates. To ensure that no events are missed, specify the stream starting position as `TRIM_HORIZON` or `AT_TIMESTAMP`.

Monitoring your Amazon DocumentDB event source

To help you monitor your Amazon DocumentDB event source, Lambda emits the `IteratorAge` metric when your function finishes processing a batch of records. *Iterator age* is the difference between the timestamp of the most recent event and the current timestamp. Essentially, the `IteratorAge` metric indicates how old the last processed record in the batch is. If your function is currently processing new events, then you can use the iterator age to estimate the latency between when a record is added and when your function processes it. An increasing trend in `IteratorAge`

can indicate issues with your function. For more information, see Using CloudWatch metrics with Lambda.

Amazon DocumentDB change streams aren't optimized to handle large time gaps between events. If your Amazon DocumentDB event source doesn't receive any events for an extended period of time, Lambda may disable the event source mapping. The length of this time period can vary from a few weeks to a few months depending on cluster size and other workloads.

Lambda supports payloads of up to 6 MB. However, Amazon DocumentDB change stream events can be up to 16 MB in size. If your change stream tries to send Lambda a change stream event larger than 6 MB, then Lambda drops the message and emits the `OversizedRecordCount` metric. Lambda emits all metrics on a best-effort basis.

Tutorial: Using AWS Lambda with Amazon DocumentDB Streams

In this tutorial, you create a basic Lambda function that consumes events from an Amazon DocumentDB (with MongoDB compatibility) change stream. To complete this tutorial, you will go through the following stages:

- Set up your Amazon DocumentDB cluster, connect to it, and activate change streams on it.
- Create your Lambda function, and configure your Amazon DocumentDB cluster as an event source for your function.
- Test the setup by inserting items into your Amazon DocumentDB database.

Create the Amazon DocumentDB cluster

1. Open the Amazon DocumentDB console. Under **Clusters**, choose **Create**.
2. Create a cluster with the following configuration:
   - For **Cluster type**, choose **Instance-based cluster**. This is the default option.
   - Under **Cluster configuration**, make sure that **Engine version** 5.0.0 is selected. This is the default option.
   - Under **Instance configuration**:
     - For **DB instance class**, select **Memory optimized classes**. This is the default option.
     - For **Number of regular replica instances**, choose 1.
     - For **Instance class**, use the default selection.

- Under **Authentication**, enter a username for the primary user, and then choose **Self managed**. Enter a password, then confirm it.
- Keep all other default settings.

3. Choose **Create cluster**.

Create the secret in Secrets Manager

While Amazon DocumentDB is creating your cluster, create an AWS Secrets Manager secret to store your database credentials. You'll provide this secret when you create the Lambda event source mapping in a later step.

To create the secret in Secrets Manager

1. Open the Secrets Manager console and choose **Store a new secret**.
2. For **Choose secret type**, choose the following options:
   - Under **Basic details**:
     - **Secret type**: Credentials for your Amazon DocumentDB database
     - Under **Credentials**, enter the same username and password that you used to create your Amazon DocumentDB cluster.
     - **Database**: Choose your Amazon DocumentDB cluster.
     - Choose **Next**.
3. For **Configure secret**, choose the following options:
   - **Secret name**: `DocumentDBSecret`
   - Choose **Next**.
4. Choose **Next**.
5. Choose **Store**.
6. Refresh the console to verify that you successfully stored the `DocumentDBSecret` secret.

Note the **Secret ARN**. You'll need it in a later step.

Connect to the cluster

Connect to your Amazon DocumentDB cluster using AWS CloudShell

1. On the Amazon DocumentDB management console, under **Clusters**, locate the cluster you created. Choose your cluster by clicking the check box next to it.
2. Choose **Connect to cluster**. The CloudShell **Run command** screen appears.
3. In the **New environment name** field, enter a unique name, such as "test" and choose **Create and run**.
4. When prompted, enter your password. When the prompt becomes `rs0 [direct: primary] <env-name>>`, you are successfully connected to your Amazon DocumentDB cluster.

Activate change streams

For this tutorial, you'll track changes to the `products` collection of the `docdbdemo` database in your Amazon DocumentDB cluster. You do this by activating change streams.

To create a new database within your cluster

1. Run the following command to create a new database called `docdbdemo`:

```
use docdbdemo
```

2. In the terminal window, use the following command to insert a record into `docdbdemo`:

```
db.products.insertOne({"hello":"world"})
```

You should see an output like this:

```
{
  acknowledged: true,
  insertedId: ObjectId('67f85066ca526410fd531d59')
}
```

3. Next, activate change streams on the `products` collection of the `docdbdemo` database using the following command:

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
```

```
    collection: "products",
    enable: true});
```

You should see output that looks like this:

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

Create interface VPC endpoints

Next, create interface VPC endpoints to ensure that Lambda and Secrets Manager (used later to store our cluster access credentials) can connect to your default VPC.

To create interface VPC endpoints

1. Open the VPC console. In the left menu, under **Virtual private cloud**, choose **Endpoints**.
2. Choose **Create endpoint**. Create an endpoint with the following configuration:
   - For **Name tag**, enter `lambda-default-vpc`.
   - For **Service category**, choose AWS services.
   - For **Services**, enter `lambda` in the search box. Choose the service with format `com.amazonaws.<region>.lambda`.
   - For **VPC**, choose the VPC that your Amazon DocumentDB cluster is in. This is typically the default VPC.
   - For **Subnets**, check the boxes next to each availability zone. Choose the correct subnet ID for each availability zone.
   - For **IP address type**, select IPv4.
   - For **Security groups**, choose the security group that your Amazon DocumentDB cluster uses. This is typically the `default` security group.
   - Keep all other default settings.
   - Choose **Create endpoint**.
3. Again, choose **Create endpoint**. Create an endpoint with the following configuration:
   - For **Name tag**, enter `secretsmanager-default-vpc`.
   - For **Service category**, choose AWS services.

- For **Services**, enter `secretsmanager` in the search box. Choose the service with format `com.amazonaws.<region>.secretsmanager`.
- For **VPC**, choose the VPC that your Amazon DocumentDB cluster is in. This is typically the default VPC.
- For **Subnets**, check the boxes next to each availability zone. Choose the correct subnet ID for each availability zone.
- For **IP address type**, select IPv4.
- For **Security groups**, choose the security group that your Amazon DocumentDB cluster uses. This is typically the `default` security group.
- Keep all other default settings.
- Choose **Create endpoint**.

This completes the cluster setup portion of this tutorial.

Create the execution role

In the next set of steps, you'll create your Lambda function. First, you need to create the execution role that gives your function permission to access your cluster. You do this by creating an IAM policy first, then attaching this policy to an IAM role.

To create IAM policy

1. Open the Policies page in the IAM console and choose **Create policy**.
2. Choose the **JSON** tab. In the following policy, replace the Secrets Manager resource ARN in the final line of the statement with your secret ARN from earlier, and copy the policy into the editor.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
```

```
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups",
                "kms:Decrypt"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMAccess",
            "Effect": "Allow",
            "Action": [
                "rds:DescribeDBClusters",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBSubnetGroups"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMGetSecretValueAccess",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocumentDBSecret"
        }
    ]
}
```

3. Choose **Next: Tags**, then choose **Next: Review**.
4. For **Name**, enter `AWSDocumentDBLambdaPolicy`.
5. Choose **Create policy**.

To create the IAM role

1. Open the Roles page in the IAM console and choose **Create role**.
2. For **Select trusted entity**, choose the following options:
   - **Trusted entity type**: AWS service

- **Service or use case**: Lambda
- Choose **Next**.

3. For **Add permissions**, choose the `AWSDocumentDBLambdaPolicy` policy you just created, as well as the `AWSLambdaBasicExecutionRole` to give your function permissions to write to Amazon CloudWatch Logs.
4. Choose **Next**.
5. For **Role name**, enter `AWSDocumentDBLambdaExecutionRole`.
6. Choose **Create role**.

Create the Lambda function

This tutorial uses the Python 3.14 runtime, but we've also provided example code files for other runtimes. You can select the tab in the following box to see the code for the runtime you're interested in.

The code receives an Amazon DocumentDB event input and processes the message that it contains.

To create the Lambda function

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Choose **Author from scratch**
4. Under **Basic information**, do the following:
   a. For **Function name**, enter `ProcessDocumentDBRecords`
   b. For **Runtime**, choose **Python 3.14**.
   c. For **Architecture**, choose **x86_64**.
5. In the **Change default execution role** tab, do the following:
   a. Expand the tab, then choose **Use an existing role**.
   b. Select the `AWSDocumentDBLambdaExecutionRole` you created earlier.
6. Choose **Create function**.

To deploy the function code

1. Choose the **Python** tab in the following box and copy the code.

   .NET

   SDK for .NET

   > **Note**
   >
   > There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

   Consuming a Amazon DocumentDB event with Lambda using .NET.

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace LambdaDocDb;

public class Function
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
```

```
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument,
new JsonSerializerOptions { WriteIndented = true });

        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
        public string EventSource { get; set; }
    }

    public class DocumentDBEventRecord
    {
        [JsonPropertyName("event")]
        public EventData Event { get; set; }
    }

    public class EventData
```

```
    {
        [JsonPropertyName("_id")]
        public IdData Id { get; set; }

        [JsonPropertyName("clusterTime")]
        public ClusterTime ClusterTime { get; set; }

        [JsonPropertyName("documentKey")]
        public DocumentKey DocumentKey { get; set; }

        [JsonPropertyName("fullDocument")]
        public Dictionary<string, object> FullDocument { get; set; }

        [JsonPropertyName("ns")]
        public Namespace Ns { get; set; }

        [JsonPropertyName("operationType")]
        public string OperationType { get; set; }
    }

    public class IdData
    {
        [JsonPropertyName("_data")]
        public string Data { get; set; }
    }

    public class ClusterTime
    {
        [JsonPropertyName("$timestamp")]
        public Timestamp Timestamp { get; set; }
    }

    public class Timestamp
    {
        [JsonPropertyName("t")]
        public long T { get; set; }

        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
```

```
        public Id Id { get; set; }
    }

    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Go.

```
package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
```

```
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Java.

```
import java.util.List;
import java.util.Map;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Example implements RequestHandler<Map<String, Object>, String> {

    @SuppressWarnings("unchecked")
    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        List<Map<String, Object>> events = (List<Map<String, Object>>)
 event.get("events");
        for (Map<String, Object> record : events) {
            Map<String, Object> eventData = (Map<String, Object>)
 record.get("event");
            processEventData(eventData);
        }

        return "OK";
    }

    @SuppressWarnings("unchecked")
    private void processEventData(Map<String, Object> eventData) {
        String operationType = (String) eventData.get("operationType");
        System.out.println("operationType: %s".formatted(operationType));

        Map<String, Object> ns = (Map<String, Object>) eventData.get("ns");
```

```
        String db = (String) ns.get("db");
        System.out.println("db: %s".formatted(db));
        String coll = (String) ns.get("coll");
        System.out.println("coll: %s".formatted(coll));

        Map<String, Object> fullDocument = (Map<String, Object>)
 eventData.get("fullDocument");
        System.out.println("fullDocument: %s".formatted(fullDocument));
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using JavaScript.

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument,
 null, 2));
};
```

Consuming a Amazon DocumentDB event with Lambda using TypeScript

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from
 'aws-lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument,
 null, 2));
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using PHP.

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];

        // Log the event details

        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument,
 JSON_PRETTY_PRINT) . "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Python.

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Ruby.

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features
 = ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }
```

```
    for record in records{
        log_document_db_event(record);
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

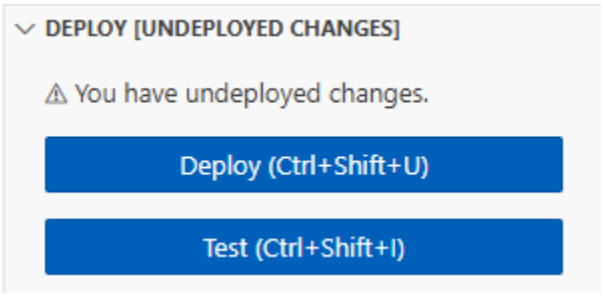
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

2. In the **Code source** pane on the Lambda console, paste the code into the code editor, replacing the code that Lambda created.
3. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Create the Lambda event source mapping

Create the event source mapping that associates your Amazon DocumentDB change stream with your Lambda function. After you create this event source mapping, AWS Lambda immediately starts polling the stream.

To create the event source mapping

1. Open the Functions page in the Lambda console.
2. Choose the `ProcessDocumentDBRecords` function you created earlier.
3. Choose the **Configuration**tab, then choose **Triggers** in the left menu.
4. Choose **Add trigger**.
5. Under **Trigger configuration**, for the source, select **Amazon DocumentDB**.
6. Create the event source mapping with the following configuration:
   - **Amazon DocumentDB cluster**: Choose the cluster you created earlier.
   - **Database name**: docdbdemo
   - **Collection name**: products
   - **Batch size**: 1
   - **Starting position**: Latest
   - **Authentication**: BASIC_AUTH
   - **Secrets Manager key**: Choose the secret for your Amazon DocumentDB cluster. It will be called something like `rds!cluster-12345678-a6f0-52c0-b290-db4aga89274f`.
   - **Batch window**: 1
   - **Full document configuration**: UpdateLookup
7. Choose **Add**. Creating your event source mapping can take a few minutes.

Test your function

Wait for the event source mapping to reach the **Enabled** state. This can take several minutes. Then, test the end-to-end setup by inserting, updating, and deleting database records. Before you begin:

1. Reconnect to your Amazon DocumentDB cluster in your CloudShell environment.
2. Run the following command to ensure that you're using the `docdbdemo` database:

```
use docdbdemo
```

Insert a record

Insert a record into the `products` collection of the `docdbdemo` database:

```
db.products.insertOne({"name":"Pencil", "price": 1.00})
```

Verify that your function successfully processed this event by checking CloudWatch Logs. You should see a log entry like this:

Timestamp	Message
2025-05-05T23:55:23.474Z	Operation type: insert
2025-05-05T23:55:23.474Z	db: docdbdemo
2025-05-05T23:55:23.474Z	collection: products
2025-05-05T23:55:23.474Z	Full document: {
2025-05-05T23:55:23.474Z	"_id": {
2025-05-05T23:55:23.474Z	"$oid": "68194fea08d12462ea531d58"
2025-05-05T23:55:23.474Z	},
2025-05-05T23:55:23.474Z	"name": "Pencil",
2025-05-05T23:55:23.474Z	"price": 1
2025-05-05T23:55:23.474Z	}

Update a record

Update the record you just inserted with the following command:

```
db.products.updateOne(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

Verify that your function successfully processed this event by checking CloudWatch Logs. You should see a log entry like this:

Timestamp	Message
2025-05-05T23:55:27.918Z	Operation type: update
2025-05-05T23:55:27.918Z	db: docdbdemo
2025-05-05T23:55:27.918Z	collection: products
2025-05-05T23:55:27.918Z	Full document: {
2025-05-05T23:55:27.918Z	"_id": {
2025-05-05T23:55:27.918Z	"$oid": "68194fea08d12462ea531d58"
2025-05-05T23:55:27.918Z	},
2025-05-05T23:55:27.918Z	"name": "Pencil",
2025-05-05T23:55:27.918Z	"price": 0.5
2025-05-05T23:55:27.918Z	}

Delete a record

Delete the record that you just updated with the following command:

```
db.products.deleteOne( { "name": "Pencil" } )
```

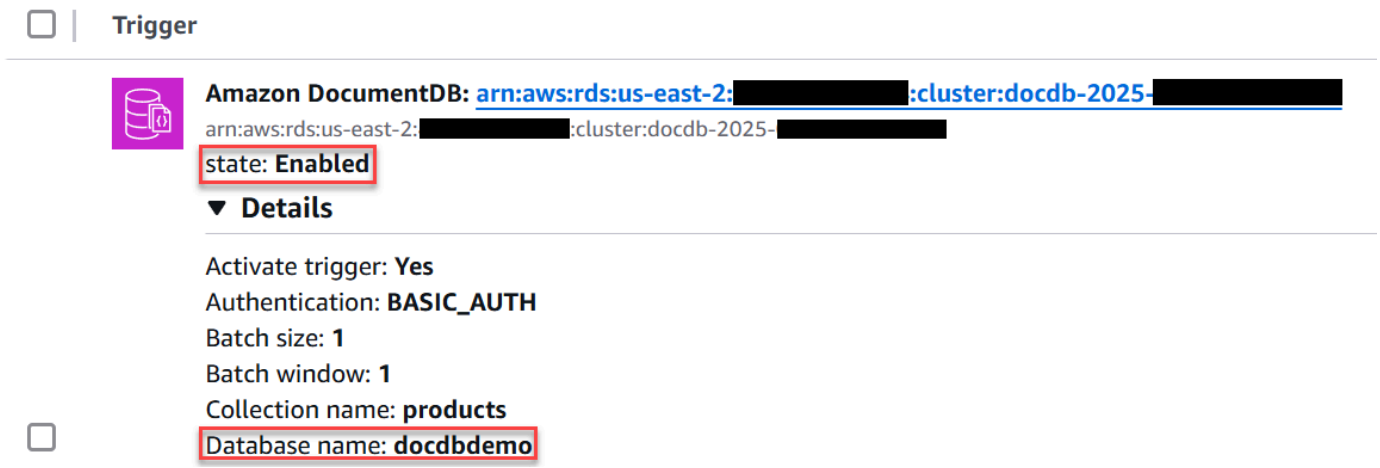
Verify that your function successfully processed this event by checking CloudWatch Logs. You should see a log entry like this:

Timestamp	Message
2025-05-05T23:55:32.362Z	Operation type: delete
2025-05-05T23:55:32.362Z	db: docdbdemo
2025-05-05T23:55:32.362Z	collection: products
2025-05-05T23:55:32.362Z	Full document: {}

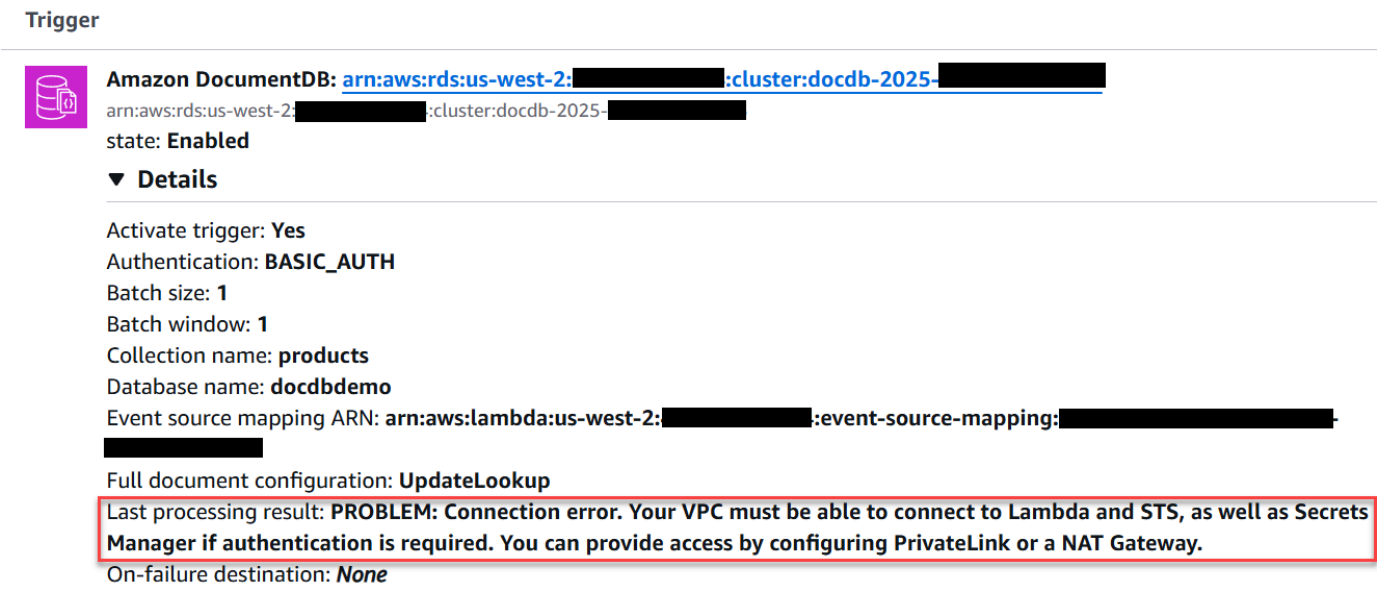
Troubleshooting

If you don't see any database events in your function's CloudWatch logs, check the following:

- Make sure that the Lambda event source mapping (also known as a trigger) is in the **Enabled** state. Event source mappings can take several minutes to create.
- If the event source mapping is **Enabled** but you still don't see database events in CloudWatch:
  - Make sure that the **Database name** in the event source mapping is set to `docdbdemo`.



  - Check the event source mapping **Last processing result** field for the following message "PROBLEM: Connection error. Your VPC must be able to connect to Lambda and STS, as well as Secrets Manager if authentication is required." If you see this error, make sure that you created the Lambda and Secrets Manager VPC interface endpoints, and that the endpoints use the same VPC and subnets that your Amazon DocumentDB cluster uses.



Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the VPC endpoints

1. Open the VPC console. In the left menu, under **Virtual private cloud**, choose **Endpoints**.
2. Select the endpoints you created.
3. Choose **Actions**, **Delete VPC endpoints**.
4. Enter **`delete`** in the text input field.
5. Choose **Delete**.

To delete the Amazon DocumentDB cluster

1. Open the Amazon DocumentDB console.
2. Choose the Amazon DocumentDB cluster you created for this tutorial, and disable deletion protection.
3. In the main **Clusters** page, choose your Amazon DocumentDB cluster again.
4. Choose **Actions**, **Delete**.
5. For **Create final cluster snapshot**, select **No**.
6. Enter **`delete`** in the text input field.
7. Choose **Delete**.

To delete the secret in Secrets Manager

1. Open the Secrets Manager console.
2. Choose the secret you created for this tutorial.
3. Choose **Actions**, **Delete secret**.
4. Choose **Schedule deletion**.

Using AWS Lambda with Amazon DynamoDB

> **Note**
>
> If you want to send data to a target other than a Lambda function or enrich the data before sending it, see Amazon EventBridge Pipes.

You can use an AWS Lambda function to process records in an Amazon DynamoDB stream. With DynamoDB Streams, you can trigger a Lambda function to perform additional work each time a DynamoDB table is updated.

When processing DynamoDB streams, you need to implement partial batch response logic to prevent successfully processed records from being retried when some records in a batch fail. The Batch Processor utility from Powertools for AWS Lambda is available in Python, TypeScript, .NET, and Java and simplifies this implementation by automatically handling partial batch response logic, reducing development time and improving reliability.

Topics

- Polling and batching streams
- Polling and stream starting positions
- Simultaneous readers of a shard in DynamoDB Streams
- Example event
- Process DynamoDB records with Lambda
- Configuring partial batch response with DynamoDB and Lambda
- Retain discarded records for a DynamoDB event source in Lambda
- Implementing stateful DynamoDB stream processing in Lambda
- Lambda parameters for Amazon DynamoDB event source mappings
- Using event filtering with a DynamoDB event source
- Tutorial: Using AWS Lambda with Amazon DynamoDB streams

Polling and batching streams

Lambda polls shards in your DynamoDB stream for records at a base rate of 4 times per second. When records are available, Lambda invokes your function and waits for the result. If processing succeeds, Lambda resumes polling until it receives more records.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a *batching window*. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see Batching behavior.

> **Warning**
>
> Lambda event source mappings process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

Lambda doesn't wait for any configured extensions to complete before sending the next batch for processing. In other words, your extensions may continue to run as Lambda processes the next batch of records. This can cause throttling issues if you breach any of your account's concurrency settings or limits. To detect whether this is a potential issue, monitor your functions and check whether you're seeing higher concurrency metrics than expected for your event source mapping. Due to short times in between invokes, Lambda may briefly report higher concurrency usage than the number of shards. This can be true even for Lambda functions without extensions.

Configure the ParallelizationFactor setting to process one shard of a DynamoDB stream with more than one Lambda invocation simultaneously. You can specify the number of concurrent batches that Lambda polls from a shard via a parallelization factor from 1 (default) to 10. For example, when you set `ParallelizationFactor` to 2, you can have 200 concurrent Lambda invocations at maximum to process 100 DynamoDB stream shards (though in practice, you may see different values for the `ConcurrentExecutions` metric). This helps scale up the processing throughput when the data volume is volatile and the IteratorAge is high. When you increase the number of

concurrent batches per shard, Lambda still ensures in-order processing at the item (partition and sort key) level.

Polling and stream starting positions

Be aware that stream polling during event source mapping creation and updates is eventually consistent.

- During event source mapping creation, it may take several minutes to start polling events from the stream.
- During event source mapping updates, it may take several minutes to stop and restart polling events from the stream.

This behavior means that if you specify `LATEST` as the starting position for the stream, the event source mapping could miss events during creation or updates. To ensure that no events are missed, specify the stream starting position as `TRIM_HORIZON`.

Simultaneous readers of a shard in DynamoDB Streams

For single-Region tables that are not global tables, you can design for up to two Lambda functions to read from the same DynamoDB Streams shard at the same time. Exceeding this limit can result in request throttling. For global tables, we recommend you limit the number of simultaneous functions to one to avoid request throttling.

Example event

Example

```
{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
```

```
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
```

```
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    }
  ]}
```

Process DynamoDB records with Lambda

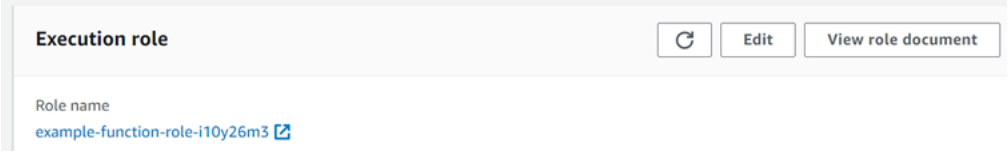
Create an event source mapping to tell Lambda to send records from your stream to a Lambda function. You can create multiple event source mappings to process the same data with multiple Lambda functions, or to process items from multiple streams with a single function.

You can configure event source mappings to process records from a stream in a different AWS account. To learn more, see the section called “Cross-account mappings”.

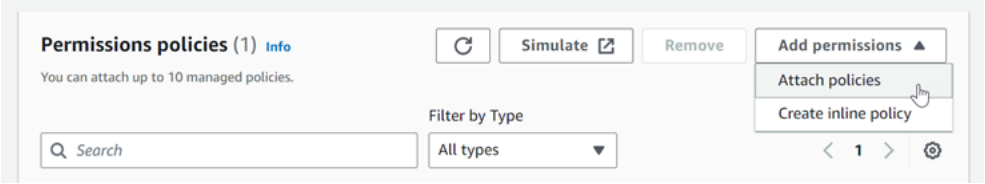
To configure your function to read from DynamoDB Streams, attach the AWSLambdaDynamoDBExecutionRole AWS managed policy to your execution role and then create a **DynamoDB** trigger.

To add permissions and create a trigger

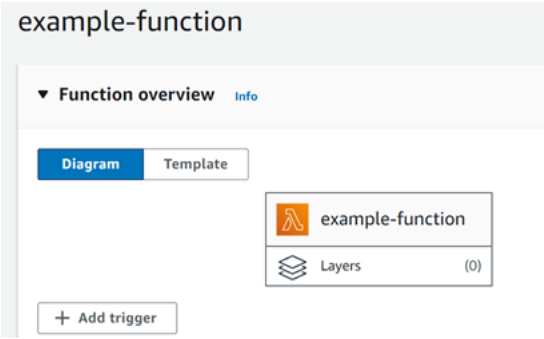
1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose the **Configuration** tab, and then choose **Permissions**.
4. Under **Role name**, choose the link to your execution role. This link opens the role in the IAM console.



5. Choose **Add permissions**, and then choose **Attach policies**.



6. In the search field, enter `AWSLambdaDynamoDBExecutionRole`. Add this policy to your execution role. This is an AWS managed policy that contains the permissions your function needs to read from the DynamoDB stream. For more information about this policy, see AWSLambdaDynamoDBExecutionRole in the *AWS Managed Policy Reference*.
7. Go back to your function in the Lambda console. Under **Function overview**, choose **Add trigger**.



8. Choose a trigger type.
9. Configure the required options, and then choose **Add**.

Lambda supports the following options for DynamoDB event sources:

Event source options

- **DynamoDB table** – The DynamoDB table to read records from.
- **Batch size** – The number of records to send to the function in each batch, up to 10,000. Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the payload limit for synchronous invocation (6 MB).
- **Batch window** – Specify the maximum amount of time to gather records before invoking the function, in seconds.
- **Starting position** – Process only new records, or all existing records.
  - **Latest** – Process new records that are added to the stream.
  - **Trim horizon** – Process all records in the stream.

  After processing any existing records, the function is caught up and continues to process new records.
- **On-failure destination** – A standard SQS queue or standard SNS topic for records that can't be processed. When Lambda discards a batch of records that's too old or has exhausted all retries, Lambda sends details about the batch to the queue or topic.

- **Retry attempts** – The maximum number of times that Lambda retries when the function returns an error. This doesn't apply to service errors or throttles where the batch didn't reach the function.
- **Maximum age of record** – The maximum age of a record that Lambda sends to your function.
- **Split batch on error** – When the function returns an error, split the batch into two before retrying. Your original batch size setting remains unchanged.
- **Concurrent batches per shard** – Concurrently process multiple batches from the same shard.
- **Enabled** – Set to true to enable the event source mapping. Set to false to stop processing records. Lambda keeps track of the last record processed and resumes processing from that point when the mapping is reenabled.

> **Note**
>
> You are not charged for GetRecords API calls invoked by Lambda as part of DynamoDB triggers.

To manage the event source configuration later, choose the trigger in the designer.

Creating a cross-account event source mapping

Amazon DynamoDB now supports resource-based policies. With this capability, you can process data from a DynamoDB stream in one AWS account with a Lambda function in another account.

To create an event source mapping for your Lambda function using a DynamoDB stream in a different AWS account, you must configure the stream using a resource-based policy to give your Lambda function permission to read records. To learn how to configure your stream for cross-account access, see Share access with cross-account Lambda functions in the *Amazon DynamoDB Developer Guide*.

Once you have configured your stream with a resource-based policy that gives your Lambda function the required permissions, create the event source mapping with your cross-account stream ARN. You can find the stream ARN under the table's **Exports and streams** tab in the cross-account DynamoDB console.

When using the Lambda console, paste the stream ARN directly into the DynamoDB table input field in the event source mapping creation page.

Note: Cross-region triggers are not supported.

Configuring partial batch response with DynamoDB and Lambda

When consuming and processing streaming data from an event source, by default Lambda checkpoints to the highest sequence number of a batch only when the batch is a complete success. Lambda treats all other results as a complete failure and retries processing the batch up to the retry limit. To allow for partial successes while processing batches from a stream, turn on `ReportBatchItemFailures`. Allowing partial successes can help to reduce the number of retries on a record, though it doesn't entirely prevent the possibility of retries in a successful record.

To turn on `ReportBatchItemFailures`, include the enum value **`ReportBatchItemFailures`** in the FunctionResponseTypes list. This list indicates which response types are enabled for your function. You can configure this list when you create or update an event source mapping.

> **Note**
>
> Even when your function code returns partial batch failure responses, these responses will not be processed by Lambda unless the `ReportBatchItemFailures` feature is explicitly turned on for your event source mapping.

Report syntax

When configuring reporting on batch item failures, the `StreamsEventResponse` class is returned with a list of batch item failures. You can use a `StreamsEventResponse` object to return the sequence number of the first failed record in the batch. You can also create your own custom class using the correct response syntax. The following JSON structure shows the required response syntax:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> If the `batchItemFailures` array contains multiple items, Lambda uses the record with the lowest sequence number as the checkpoint. Lambda then retries all records starting from that checkpoint.

Success and failure conditions

Lambda treats a batch as a complete success if you return any of the following:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

Lambda treats a batch as a complete failure if you return any of the following:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

Lambda retries failures based on your retry strategy.

Bisecting a batch

If your invocation fails and `BisectBatchOnFunctionError` is turned on, the batch is bisected regardless of your `ReportBatchItemFailures` setting.

When a partial batch success response is received and both `BisectBatchOnFunctionError` and `ReportBatchItemFailures` are turned on, the batch is bisected at the returned sequence number and Lambda retries only the remaining records.

To simplify the implementation of partial batch response logic, consider using the Batch Processor utility from Powertools for AWS Lambda, which automatically handles these complexities for you.

Here are some examples of function code that return the list of failed message IDs in the batch:

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
```

```
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)
```

```
type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using JavaScript.

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }
```

```
  return { batchItemFailures: [] };
};
```

Reporting DynamoDB batch item failures with Lambda using TypeScript.

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
```

```
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
```

```
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Ruby.

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Rust.

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
```

```
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Using Powertools for AWS Lambda batch processor

The batch processor utility from Powertools for AWS Lambda automatically handles partial batch response logic, reducing the complexity of implementing batch failure reporting. Here are examples using the batch processor:

Python

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing DynamoDB stream records with AWS Lambda batch processor.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import DynamoDBStreamEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.DynamoDBStreams)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing DynamoDB stream records with AWS Lambda batch processor.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { DynamoDBStreamEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.DynamoDBStreams);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: DynamoDBStreamEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Java

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing DynamoDB stream records with AWS Lambda batch processor.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
```

```
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import software.amazon.lambda.powertools.batch.BatchMessageHandlerBuilder;
import software.amazon.lambda.powertools.batch.handler.BatchMessageHandler;

public class DynamoDBStreamBatchHandler implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    private final BatchMessageHandler<DynamodbEvent, StreamsEventResponse> handler;

    public DynamoDBStreamBatchHandler() {
        handler = new BatchMessageHandlerBuilder()
                .withDynamoDbBatchHandler()
                .buildWithRawMessageHandler(this::processMessage);
    }

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent ddbEvent, Context
 context) {
        return handler.processBatch(ddbEvent, context);
    }

    private void processMessage(DynamodbEvent.DynamodbStreamRecord
 dynamodbStreamRecord, Context context) {
        // Process the change record
    }
}
```

.NET

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing DynamoDB stream records with AWS Lambda batch processor.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;
using Amazon.Lambda.Serialization.SystemTextJson;
using AWS.Lambda.Powertools.BatchProcessing;
```

```
[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]

namespace HelloWorld;

public class Customer
{
    public string? CustomerId { get; set; }
    public string? Name { get; set; }
    public string? Email { get; set; }
    public DateTime CreatedAt { get; set; }
}

internal class TypedDynamoDbRecordHandler : ITypedRecordHandler<Customer>
{
    public async Task<RecordHandlerResult> HandleAsync(Customer customer,
 CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(customer.Email))
        {
            throw new ArgumentException("Customer email is required");
        }

        return await Task.FromResult(RecordHandlerResult.None);
    }
}

public class Function
{
    [BatchProcessor(TypedRecordHandler = typeof(TypedDynamoDbRecordHandler))]
    public BatchItemFailuresResponse HandlerUsingTypedAttribute(DynamoDBEvent _)
    {
        return TypedDynamoDbStreamBatchProcessor.Result.BatchItemFailuresResponse;
    }
}
```

Retain discarded records for a DynamoDB event source in Lambda

Error handling for DynamoDB event source mappings depends on whether the error occurs before the function is invoked or during function invocation:

- **Before invocation:** If a Lambda event source mapping is unable to invoke the function due to throttling or other issues, it retries until the records expire or exceed the maximum age configured on the event source mapping (MaximumRecordAgeInSeconds).
- **During invocation:** If the function is invoked but returns an error, Lambda retries until the records expire, exceed the maximum age (MaximumRecordAgeInSeconds), or reach the configured retry quota (MaximumRetryAttempts). For function errors, you can also configure BisectBatchOnFunctionError, which splits a failed batch into two smaller batches, isolating bad records and avoiding timeouts. Splitting batches doesn't consume the retry quota.

If the error handling measures fail, Lambda discards the records and continues processing batches from the stream. With the default settings, this means that a bad record can block processing on the affected shard for up to one day. To avoid this, configure your function's event source mapping with a reasonable number of retries and a maximum record age that fits your use case.

Configuring destinations for failed invocations

To retain records of failed event source mapping invocations, add a destination to your function's event source mapping. Each record sent to the destination is a JSON document containing metadata about the failed invocation. For Amazon S3 destinations, Lambda also sends the entire invocation record along with the metadata. You can configure any Amazon SNS topic, Amazon SQS queue, Amazon S3 bucket, or Kafka as a destination.

With Amazon S3 destinations, you can use the Amazon S3 Event Notifications feature to receive notifications when objects are uploaded to your destination S3 bucket. You can also configure S3 Event Notifications to invoke another Lambda function to perform automated processing on failed batches.

Your execution role must have permissions for the destination:

- **For an SQS destination:** sqs:SendMessage
- **For an SNS destination:** sns:Publish
- **For an S3 destination:** s3:PutObject and s3:ListBucket
- **For a Kafka destination:** kafka-cluster:WriteData

You can configure a Kafka topic as an on-failure destination for your Kafka event source mappings. When Lambda can't process records after exhausting retry attempts or when records exceed the

maximum age, Lambda sends the failed records to the specified Kafka topic for later processing. Refer to the section called “Kafka on-failure destination”.

If you've enabled encryption with your own KMS key for an S3 destination, your function's execution role must also have permission to call kms:GenerateDataKey. If the KMS key and S3 bucket destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:GenerateDataKey.

To configure an on-failure destination using the console, follow these steps:

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add destination**.
4. For **Source**, choose **Event source mapping invocation**.
5. For **Event source mapping**, choose an event source that's configured for this function.
6. For **Condition**, select **On failure**. For event source mapping invocations, this is the only accepted condition.
7. For **Destination type**, choose the destination type that Lambda sends invocation records to.
8. For **Destination**, choose a resource.
9. Choose **Save**.

You can also configure an on-failure destination using the AWS Command Line Interface (AWS CLI). For example, the following create-event-source-mapping command adds an event source mapping with an SQS on-failure destination to `MyFunction`:

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

The following update-event-source-mapping command updates an event source mapping to send failed invocation records to an SNS destination after two retry attempts, or if the records are more than an hour old.

```
aws lambda update-event-source-mapping \
```

```
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

Updated settings are applied asynchronously and aren't reflected in the output until the process completes. Use the get-event-source-mapping command to view the current status.

To remove a destination, supply an empty string as the argument to the `destination-config` parameter:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Security best practices for Amazon S3 destinations

Deleting an S3 bucket that's configured as a destination without removing the destination from your function's configuration can create a security risk. If another user knows your destination bucket's name, they can recreate the bucket in their AWS account. Records of failed invocations will be sent to their bucket, potentially exposing data from your function.

> **Warning**
>
> To ensure that invocation records from your function can't be sent to an S3 bucket in another AWS account, add a condition to your function's execution role that limits `s3:PutObject` permissions to buckets in your account.

The following example shows an IAM policy that limits your function's `s3:PutObject` permissions to buckets in your account. This policy also gives Lambda the `s3:ListBucket` permission it needs to use an S3 bucket as a destination.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
```

```
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

To add a permissions policy to your function's execution role using the AWS Management Console or AWS CLI, refer to the instructions in the following procedures:

Console

To add a permissions policy to a function's execution role (console)

1. Open the Functions page of the Lambda console.
2. Select the Lambda function whose execution role you want to modify.
3. In the **Configuration** tab, select **Permissions**.
4. In the **Execution role** tab, select your function's **Role name** to open the role's IAM console page.
5. Add a permissions policy to the role by doing the following:
    a. In the **Permissions policies** pane, choose **Add permissions** and select **Create inline policy**.
    b. In **Policy editor**, select **JSON**.
    c. Paste the policy you want to add into the editor (replacing the existing JSON), and then choose **Next**.
    d. Under **Policy details**, enter a **Policy name**.
    e. Choose **Create policy**.

AWS CLI

To add a permissions policy to a function's execution role (CLI)

1. Create a JSON policy document with the required permissions and save it in a local directory.
2. Use the IAM `put-role-policy` CLI command to add the permissions to your function's execution role. Run the following command from the directory you saved your JSON policy document in and replace the role name, policy name, and policy document with your own values.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Example Amazon SNS and Amazon SQS invocation record

The following example shows an invocation record Lambda sends to an SQS or SNS destination for a DynamoDB stream.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
```

```
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    }
}
```

You can use this information to retrieve the affected records from the stream for troubleshooting. The actual records aren't included, so you must process this record and retrieve them from the stream before they expire and are lost.

Example Amazon S3 invocation record

The following example shows an invocation record Lambda sends to an S3 bucket for a DynamoDB stream. In addition to all of the fields from the previous example for SQS and SNS destinations, the `payload` field contains the original invocation record as an escaped JSON string.

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

The S3 object containing the invocation record uses the following naming convention:

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

Implementing stateful DynamoDB stream processing in Lambda

Lambda functions can run continuous stream processing applications. A stream represents unbounded data that flows continuously through your application. To analyze information from this continuously updating input, you can bound the included records using a window defined in terms of time.

Tumbling windows are distinct time windows that open and close at regular intervals. By default, Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with tumbling windows, you can maintain your state across invocations. This state contains the aggregate result of the messages previously processed for the current window. Your state can be a maximum of 1 MB per shard. If it exceeds that size, Lambda terminates the window early.

Each record in a stream belongs to a specific window. Lambda will process each record at least once, but doesn't guarantee that each record will be processed only once. In rare cases, such as error handling, some records might be processed more than once. Records are always processed in order the first time. If records are processed more than once, they might be processed out of order.

Aggregation and processing

Your user managed function is invoked both for aggregation and for processing the final results of that aggregation. Lambda aggregates all records received in the window. You can receive these records in multiple batches, each as a separate invocation. Each invocation receives a state. Thus, when using tumbling windows, your Lambda function response must contain a `state` property. If the response does not contain a `state` property, Lambda considers this a failed invocation. To satisfy this condition, your function can return a `TimeWindowEventResponse` object, which has the following JSON shape:

Example `TimeWindowEventResponse` values

```
{
    "state": {
        "1": 282,
        "2": 715
    },
```

```
    "batchItemFailures": []
}
```

> **Note**
>
> For Java functions, we recommend using a `Map<String, String>` to represent the state.

At the end of the window, the flag `isFinalInvokeForWindow` is set to `true` to indicate that this is the final state and that it's ready for processing. After processing, the window completes and your final invocation completes, and then the state is dropped.

At the end of your window, Lambda uses final processing for actions on the aggregation results. Your final processing is synchronously invoked. After successful invocation, your function checkpoints the sequence number and stream processing continues. If invocation is unsuccessful, your Lambda function suspends further processing until a successful invocation.

Example DynamodbTimeWindowEvent

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
```

```
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"2",
         "eventName":"MODIFY",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"222",
            "SizeBytes":59,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"3",
         "eventName":"REMOVE",
         "eventVersion":"1.0",
```

```
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ],
    "window": {
        "start": "2020-07-30T17:00:00Z",
        "end": "2020-07-30T17:05:00Z"
    },
    "state": {
        "1": "state1"
    },
    "shardId": "shard123456789",
    "eventSourceARN": "stream-ARN",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Configuration

You can configure tumbling windows when you create or update an event source mapping. To configure a tumbling window, specify the window in seconds (TumblingWindowInSeconds). The following example AWS Command Line Interface (AWS CLI) command creates a streaming event source mapping that has a tumbling window of 120 seconds. The Lambda function defined for aggregation and processing is named `tumbling-window-example-function`.

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda determines tumbling window boundaries based on the time when records were inserted into the stream. All records have an approximate timestamp available that Lambda uses in boundary determinations.

Tumbling window aggregations do not support resharding. When the shard ends, Lambda considers the window closed, and the child shards start their own window in a fresh state.

Tumbling windows fully support the existing retry policies `maxRetryAttempts` and `maxRecordAge`.

Example Handler.py – Aggregation and processing

The following Python function demonstrates how to aggregate and then process your final state:

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']
['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
```

```
    return {'state': state}
```

Lambda parameters for Amazon DynamoDB event source mappings

All Lambda event source types share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to DynamoDB Streams.

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
BisectBatchOnFunctionError	N	false	none
DestinationConfig	N	N/A	Standard Amazon SQS queue or standard Amazon SNS topic destination for discarded records
Enabled	N	true	none
EventSourceArn	Y	N/A	ARN of the data stream or a stream consumer
FilterCriteria	N	N/A	Control which events Lambda sends to your function
FunctionName	Y	N/A	none
FunctionResponseTypes	N	N/A	To let your function report specific failures in a batch, include the value `ReportBatchItemFailures`

Parameter	Required	Default	Notes
			in `FunctionResponseTypes`. For more information, see Configuring partial batch response with DynamoDB and Lambda.
MaximumBatchingWindowInSeconds	N	0	none
MaximumRecordAgeInSeconds	N	-1	-1 means infinite: failed records are retried until the record expires. The data retention limit for DynamoDB Streams is 24 hours. Minimum: -1 Maximum: 604,800
MaximumRetryAttempts	N	-1	-1 means infinite: failed records are retried until the record expires Minimum: 0 Maximum: 10,000
ParallelizationFactor	N	1	Maximum: 10

Parameter	Required	Default	Notes
StartingPosition	Y	N/A	TRIM_HORIZON or LATEST
TumblingWindowInSeconds	N	N/A	Minimum: 0 Maximum: 900

Using event filtering with a DynamoDB event source

You can use event filtering to control which records from a stream or queue Lambda sends to your function. For general information about how event filtering works, see the section called "Event filtering".

This section focuses on event filtering for DynamoDB event sources.

> **Note**
>
> DynamoDB event source mappings only support filtering on the `dynamodb` key.

Topics

- DynamoDB event
- Filtering with table attributes
- Filtering with Boolean expressions
- Using the Exists operator
- JSON format for DynamoDB filtering

DynamoDB event

Suppose you have a DynamoDB table with the primary key `CustomerName` and attributes `AccountManager` and `PaymentTerms`. The following shows an example record from your DynamoDB table's stream.

```
{
      "eventID": "1",
```

```
      "eventVersion": "1.0",
      "dynamodb": {
          "ApproximateCreationDateTime": "1678831218.0",
          "Keys": {
              "CustomerName": {
                  "S": "AnyCompany Industries"
              }
          },
          "NewImage": {
              "AccountManager": {
                  "S": "Pat Candella"
              },
              "PaymentTerms": {
                  "S": "60 days"
              },
              "CustomerName": {
                  "S": "AnyCompany Industries"
              }
          },
          "SequenceNumber": "111",
          "SizeBytes": 26,
          "StreamViewType": "NEW_IMAGE"
      }
  }
```

To filter based on the key and attribute values in your DynamoDB table, use the dynamodb key in the record. The following sections provide examples for different filter types.

Filtering with table keys

Suppose you want your function to process only those records where the primary key CustomerName is “AnyCompany Industries.” The FilterCriteria object would be as follows.

```
{
     "Filters": [
          {
              "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
          }
      ]
 }
```

For added clarity, here is the value of the filter's Pattern expanded in plain JSON.

```
{
     "dynamodb": {
          "Keys": {
              "CustomerName": {
                  "S": [ "AnyCompany Industries" ]
                  }
              }
          }
 }
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

Filtering with table attributes

With DynamoDB, you can also use the `NewImage` and `OldImage` keys to filter for attribute values. Suppose you want to filter records where the `AccountManager` attribute in the latest table image is “Pat Candella” or "Shirley Rodriguez." The `FilterCriteria` object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

Filtering with Boolean expressions

You can also create filters using Boolean AND expressions. These expressions can include both your table's key and attribute parameters. Suppose you want to filter records where the `NewImage` value of `AccountManager` is "Pat Candella" and the `OldImage` value is "Terry Whitlock". The `FilterCriteria` object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "dynamodb" : {
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> DynamoDB event filtering doesn't support the use of numeric operators (numeric equals and numeric range). Even if items in your table are stored as numbers, these parameters are converted to strings in the JSON record object.

Using the Exists operator

Because of the way that JSON event objects from DynamoDB are structured, using the Exists operator requires special care. The Exists operator only works on leaf nodes in the event JSON, so if your filter pattern uses Exists to test for an intermediate node, it won't work. Consider the following DynamoDB table item:

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
  }
}
```

You might want to create a filter pattern like the following that would test for events containing `"Organizations"`:

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

However, this filter pattern would never return a match because `"Organizations"` is not a leaf node. The following example shows how to properly use the Exists operator to construct the desired filter pattern:

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

JSON format for DynamoDB filtering

To properly filter events from DynamoDB sources, both the data field and your filter criteria for the data field (`dynamodb`) must be in valid JSON format. If either field isn't in a valid JSON format, Lambda drops the message or throws an exception. The following table summarizes the specific behavior:

Incoming data format	Filter pattern format for data properties	Resulting action
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Non-JSON	Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.
Non-JSON	Valid JSON	Lambda drops the record.
Non-JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Non-JSON	Non-JSON	Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.

Tutorial: Using AWS Lambda with Amazon DynamoDB streams

In this tutorial, you create a Lambda function to consume events from an Amazon DynamoDB stream.

Prerequisites

Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create the execution role

Create the execution role that gives your function permission to access AWS resources.

To create an execution role

1. Open the roles page in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
   - **Trusted entity** – Lambda.
   - **Permissions** – **AWSLambdaDynamoDBExecutionRole**.
   - **Role name** – **`lambda-dynamodb-role`**.

The **AWSLambdaDynamoDBExecutionRole** has the permissions that the function needs to read items from DynamoDB and write logs to CloudWatch Logs.

Create the function

Create a Lambda function that processes your DynamoDB events. The function code writes some of the incoming event data to CloudWatch Logs.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
```

```
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }
```

```
 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
```

```
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Consuming a DynamoDB event with Lambda using TypeScript.

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;
```

```
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Python.

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Ruby.

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
```

```
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())
```

```
}
```

To create the function

1. Copy the sample code into a file named `example.js`.
2. Create a deployment package.

```
zip function.zip example.js
```

3. Create a Lambda function with the `create-function` command.

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
    --zip-file fileb://function.zip --handler example.handler --runtime nodejs24.x
 \
    --role arn:aws:iam::111122223333:role/lambda-dynamodb-role
```

Test the Lambda function

In this step, you invoke your Lambda function manually using the `invoke` AWS Lambda CLI command and the following sample DynamoDB event. Copy the following into a file named `input.txt`.

Example input.txt

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
```

```
            "Message":{
               "S":"New item!"
            },
            "Id":{
               "N":"101"
            }
         },
         "SequenceNumber":"111",
         "SizeBytes":26,
         "StreamViewType":"NEW_AND_OLD_IMAGES"
      },
      "eventSourceARN":"stream-ARN"
   },
   {
      "eventID":"2",
      "eventName":"MODIFY",
      "eventVersion":"1.0",
      "eventSource":"aws:dynamodb",
      "awsRegion":"us-east-1",
      "dynamodb":{
         "Keys":{
            "Id":{
               "N":"101"
            }
         },
         "NewImage":{
            "Message":{
               "S":"This item has changed"
            },
            "Id":{
               "N":"101"
            }
         },
         "OldImage":{
            "Message":{
               "S":"New item!"
            },
            "Id":{
               "N":"101"
            }
         },
         "SequenceNumber":"222",
         "SizeBytes":59,
         "StreamViewType":"NEW_AND_OLD_IMAGES"
```

```
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"3",
         "eventName":"REMOVE",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ]
}
```

Run the following `invoke` command.

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

The function returns the string `message` in the response body.

Verify the output in the `outputfile.txt` file.

Create a DynamoDB table with a stream enabled

Create an Amazon DynamoDB table with a stream enabled.

To create a DynamoDB table

1. Open the DynamoDB console.
2. Choose **Create table**.
3. Create a table with the following settings.
   - **Table name** – **`lambda-dynamodb-stream`**
   - **Primary key** – **`id`** (string)
4. Choose **Create**.

To enable streams

1. Open the DynamoDB console.
2. Choose **Tables**.
3. Choose the **lambda-dynamodb-stream** table.
4. Under **Exports and streams**, choose **DynamoDB stream details**.
5. Choose **Turn on**.
6. For **View type**, choose **Key attributes only**.
7. Choose **Turn on stream**.

Write down the stream ARN. You need this in the next step when you associate the stream with your Lambda function. For more information on enabling streams, see Capturing table activity with DynamoDB Streams.

Add an event source in AWS Lambda

Create an event source mapping in AWS Lambda. This event source mapping associates the DynamoDB stream with your Lambda function. After you create this event source mapping, AWS Lambda starts polling the stream.

Run the following AWS CLI `create-event-source-mapping` command. After the command runs, note down the UUID. You'll need this UUID to refer to the event source mapping in any commands, for example, when deleting the event source mapping.

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

This creates a mapping between the specified DynamoDB stream and the Lambda function. You can associate a DynamoDB stream with multiple Lambda functions, and associate the same Lambda function with multiple streams. However, the Lambda functions will share the read throughput for the stream they share.

You can get the list of event source mappings by running the following command.

```
aws lambda list-event-source-mappings
```

The list returns all of the event source mappings you created, and for each mapping it shows the `LastProcessingResult`, among other things. This field is used to provide an informative message if there are any problems. Values such as `No records processed` (indicates that AWS Lambda has not started polling or that there are no records in the stream) and `OK` (indicates AWS Lambda successfully read records from the stream and invoked your Lambda function) indicate that there are no issues. If there are issues, you receive an error message.

If you have a lot of event source mappings, use the function name parameter to narrow down the results.

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

Test the setup

Test the end-to-end experience. As you perform table updates, DynamoDB writes event records to the stream. As AWS Lambda polls the stream, it detects new records in the stream and invokes your Lambda function on your behalf by passing events to the function.

1. In the DynamoDB console, add, update, and delete items to the table. DynamoDB writes records of these actions to the stream.
2. AWS Lambda polls the stream and when it detects updates to the stream, it invokes your Lambda function by passing in the event data it finds in the stream.

3. Your function runs and creates logs in Amazon CloudWatch. You can verify the logs reported in the Amazon CloudWatch console.

Next steps

This tutorial showed you the basics of processing DynamoDB stream events with Lambda. For production workloads, consider implementing partial batch response logic to handle individual record failures more efficiently. The batch processor utility from Powertools for AWS Lambda is available in Python, TypeScript, .NET, and Java and provides a robust solution for this, automatically handling the complexity of partial batch responses and reducing the number of retries for successfully processed records.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the DynamoDB table

1. Open the Tables page of the DynamoDB console.
2. Select the table you created.

3. Choose **Delete**.
4. Enter **`delete`** in the text box.
5. Choose **Delete table**.

Process Amazon EC2 lifecycle events with a Lambda function

You can use AWS Lambda to process lifecycle events from Amazon Elastic Compute Cloud and manage Amazon EC2 resources. Amazon EC2 sends events to Amazon EventBridge (CloudWatch Events) for lifecycle events such as when an instance changes state, when an Amazon Elastic Block Store volume snapshot completes, or when a spot instance is scheduled to be terminated. You configure EventBridge (CloudWatch Events) to forward those events to a Lambda function for processing.

EventBridge (CloudWatch Events) invokes your Lambda function asynchronously with the event document from Amazon EC2.

Example instance lifecycle event

```
{
    "version": "0",
    "id": "b6ba298a-7732-2226-xmpl-976312c1a050",
    "detail-type": "EC2 Instance State-change Notification",
    "source": "aws.ec2",
    "account": "111122223333",
    "time": "2019-10-02T17:59:30Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:ec2:us-east-1:111122223333:instance/i-0c314xmplcd5b8173"
    ],
    "detail": {
        "instance-id": "i-0c314xmplcd5b8173",
        "state": "running"
    }
}
```

For details on configuring events, see Invoke a Lambda function on a schedule. For an example function that processes Amazon EBS snapshot notifications, see EventBridge Scheduler for Amazon EBS.

You can also use the AWS SDK to manage instances and other resources with the Amazon EC2 API.

Granting permissions to EventBridge (CloudWatch Events)

To process lifecycle events from Amazon EC2, EventBridge (CloudWatch Events) needs permission to invoke your function. This permission comes from the function's resource-based policy. If

you use the EventBridge (CloudWatch Events) console to configure an event trigger, the console updates the resource-based policy on your behalf. Otherwise, add a statement like the following:

Example resource-based policy statement for Amazon EC2 lifecycle notifications

```
{
  "Sid": "ec2-events",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:us-east-1:12456789012:function:my-function",
  "Condition": {
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:events:us-east-1:12456789012:rule/*"
    }
  }
}
```

To add a statement, use the `add-permission` AWS CLI command.

```
aws lambda add-permission --action lambda:InvokeFunction --statement-id ec2-events \
--principal events.amazonaws.com --function-name my-function --source-arn
 'arn:aws:events:us-east-1:12456789012:rule/*'
```

If your function uses the AWS SDK to manage Amazon EC2 resources, add Amazon EC2 permissions to the function's execution role.

Process Application Load Balancer requests with Lambda

You can use a Lambda function to process requests from an Application Load Balancer. Elastic Load Balancing supports Lambda functions as a target for an Application Load Balancer. Use load balancer rules to route HTTP requests to a function, based on path or header values. Process the request and return an HTTP response from your Lambda function.

Elastic Load Balancing invokes your Lambda function synchronously with an event that contains the request body and metadata.

Example Application Load Balancer request event


```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-
east-1:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8",
        "accept-encoding": "gzip",
        "accept-language": "en-US,en;q=0.9",
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-1.elb.amazonaws.com",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
        "x-imforwards": "20"
    },
    "body": "",
    "isBase64Encoded": False
```

```
}
```

Your function processes the event and returns a response document to the load balancer in JSON. Elastic Load Balancing converts the document to an HTTP success or error response and returns it to the user.

Example response document format

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

To configure an Application Load Balancer as a function trigger, grant Elastic Load Balancing permission to run the function, create a target group that routes requests to the function, and add a rule to the load balancer that sends requests to the target group.

Use the `add-permission` command to add a permission statement to your function's resource-based policy.

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

You should see the following output:

```
{
    "Statement": "{\"Sid\":\"load-balancer\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"elasticloadbalancing.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-west-2:123456789012:function:alb-function\"}"
}
```

For instructions on configuring the Application Load Balancer listener and target group, see Lambda functions as a target in the *User Guide for Application Load Balancers*.

Event Handler from Powertools for AWS Lambda

The event handler from the Powertools for AWS Lambda toolkit provides routing, middleware, CORS configuration, OpenAPI spec generation, request validation, error handling, and other useful features when writing Lambda functions invoked by an Application Load Balancer. The Event Handler utility is available for Python. For more information, see Event Handler REST API in the *Powertools for AWS Lambda (Python) documentation*.

Python

```
import requests
from requests import Response

from aws_lambda_powertools import Logger, Tracer
from aws_lambda_powertools.event_handler import ALBResolver
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools.utilities.typing import LambdaContext

tracer = Tracer()
logger = Logger()
app = ALBResolver()


@app.get("/todos")
@tracer.capture_method
def get_todos():
    todos: Response = requests.get("https://jsonplaceholder.typicode.com/todos")
    todos.raise_for_status()

    # for brevity, we'll limit to the first 10 only
    return {"todos": todos.json()[:10]}


# You can continue to use other utilities just as before
@logger.inject_lambda_context(correlation_id_path=correlation_paths.APPLICATION_LOAD_BALANCER)
@tracer.capture_lambda_handler
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

Invoke a Lambda function on a schedule

Amazon EventBridge Scheduler is a serverless scheduler that allows you to create, run, and manage tasks from one central, managed service. With EventBridge Scheduler, you can create schedules using cron and rate expressions for recurring patterns, or configure one-time invocations. You can set up flexible time windows for delivery, define retry limits, and set the maximum retention time for unprocessed events.

When you set up EventBridge Scheduler with Lambda, EventBridge Scheduler invokes your Lambda function asynchronously. This page explains how to use EventBridge Scheduler to invoke a Lambda function on a schedule.

Set up the execution role

When you create a new schedule, EventBridge Scheduler must have permission to invoke its target API operation on your behalf. You grant these permissions to EventBridge Scheduler using an *execution role*. The permission policy you attach to your schedule's execution role defines the required permissions. These permissions depend on the target API you want EventBridge Scheduler to invoke.

When you use the EventBridge Scheduler console to create a schedule, as in the following procedure, EventBridge Scheduler automatically sets up an execution role based on your selected target. If you want to create a schedule using one of the EventBridge Scheduler SDKs, the AWS CLI, or CloudFormation, you must have an existing execution role that grants the permissions EventBridge Scheduler requires to invoke a target. For more information about manually setting up an execution role for your schedule, see Setting up an execution role in the *EventBridge Scheduler User Guide*.

Create a schedule

To create a schedule by using the console

1. Open the Amazon EventBridge Scheduler console at https://console.aws.amazon.com/scheduler/home.
2. On the **Schedules** page, choose **Create schedule**.
3. On the **Specify schedule detail** page, in the **Schedule name and description** section, do the following:

a. For **Schedule name**, enter a name for your schedule. For example, **`MyTestSchedule`**.

b. (Optional) For **Description**, enter a description for your schedule. For example, **`My first schedule`**.

c. For **Schedule group**, choose a schedule group from the dropdown list. If you don't have a group, choose **default**. To create a schedule group, choose **create your own schedule**.

You use schedule groups to add tags to groups of schedules.

4. • Choose your schedule options.

Occurrence	Do this...
One-time schedule A one-time schedule invokes a target only once at the date and time that you specify.	For **Date and time**, do the following: • Enter a valid date in `YYYY/MM/DD` format. • Enter a timestamp in 24-hour `hh:mm` format. • For **Timezone**, choose the timezone.
Recurring schedule A recurring schedule invokes a target at a rate that you specify using a **cron** expression or rate expression.	a. For **Schedule type**, do one of the following: • To use a cron expression to define the schedule, choose **Cron-based schedule** and enter the cron expression. • To use a rate expression to define the schedule, choose **Rate-based schedule** and enter the rate expression.

Occurrence	Do this...	
	For more informati on about cron and rate expressions, see Schedule types on EventBridge Scheduler in the *Amazon EventBridge Scheduler User Guide*. b. For **Flexible time window**, choose **Off** to turn off the option, or choose one of the pre-defined time windows. For example, if you choose **15 minutes** and you set a recurring schedule to invoke its target once every hour, the schedule runs within 15 minutes after the start of every hour.	

5. (Optional) If you chose **Recurring schedule** in the previous step, in the **Timeframe** section, do the following:

   a. For **Timezone**, choose a timezone.

   b. For **Start date and time**, enter a valid date in `YYYY/MM/DD` format, and then specify a timestamp in 24-hour `hh:mm` format.

   c. For **End date and time**, enter a valid date in `YYYY/MM/DD` format, and then specify a timestamp in 24-hour `hh:mm` format.

6. Choose **Next**.

7. On the **Select target** page, choose the AWS API operation that EventBridge Scheduler invokes:

   a. Choose **AWS Lambda Invoke**.

b. In the **Invoke** section, select a function or choose **Create new Lambda function**.

c. (Optional) Enter a JSON payload. If you don't enter a payload, EventBridge Scheduler uses an empty event to invoke the function.

8. Choose **Next**.

9. On the **Settings** page, do the following:

a. To turn on the schedule, under **Schedule state**, toggle **Enable schedule**.

b. To configure a retry policy for your schedule, under **Retry policy and dead-letter queue (DLQ)**, do the following:

- Toggle **Retry**.
- For **Maximum age of event**, enter the maximum **hour(s)** and **min(s)** that EventBridge Scheduler must keep an unprocessed event.
- The maximum time is 24 hours.
- For **Maximum retries**, enter the maximum number of times EventBridge Scheduler retries the schedule if the target returns an error.

  The maximum value is 185 retries.

With retry policies, if a schedule fails to invoke its target, EventBridge Scheduler re-runs the schedule. If configured, you must set the maximum retention time and retries for the schedule.

c. Choose where EventBridge Scheduler stores undelivered events.

Dead-letter queue (DLQ) option	Do this...
Don't store	Choose **None**.
Store the event in the same AWS account where you're creating the schedule	a. Choose **Select an Amazon SQS queue in my AWS account as a DLQ**.

Dead-letter queue (DLQ) option	Do this...	
	b. Choose the Amazon Resource Name (ARN) of the Amazon SQS queue.	
Store the event in a different AWS account from where you're creating the schedule	a. Choose **Specify an Amazon SQS queue in other AWS accounts as a DLQ**. b. Enter the Amazon Resource Name (ARN) of the Amazon SQS queue.	

d. To use a customer managed key to encrypt your target input, under **Encryption**, choose **Customize encryption settings (advanced)**.

   If you choose this option, enter an existing KMS key ARN or choose **Create an AWS KMS key** to navigate to the AWS KMS console. For more information about how EventBridge Scheduler encrypts your data at rest, see Encryption at rest in the *Amazon EventBridge Scheduler User Guide*.

e. To have EventBridge Scheduler create a new execution role for you, choose **Create new role for this schedule**. Then, enter a name for **Role name**. If you choose this option, EventBridge Scheduler attaches the required permissions necessary for your templated target to the role.

10. Choose **Next**.
11. In the **Review and create schedule** page, review the details of your schedule. In each section, choose **Edit** to go back to that step and edit its details.
12. Choose **Create schedule**.

    You can view a list of your new and existing schedules on the **Schedules** page. Under the **Status** column, verify that your new schedule is **Enabled**.

To confirm that EventBridge Scheduler invoked the function, check the function's Amazon CloudWatch logs.

Related resources

For more information about EventBridge Scheduler, see the following:

- EventBridge Scheduler User Guide
- EventBridge Scheduler API Reference
- EventBridge Scheduler Pricing

Using AWS Lambda with AWS IoT

AWS IoT provides secure communication between internet-connected devices (such as sensors) and the AWS Cloud. This makes it possible for you to collect, store, and analyze telemetry data from multiple devices.

You can create AWS IoT rules for your devices to interact with AWS services. The AWS IoT Rules Engine provides a SQL-based language to select data from message payloads and send the data to other services, such as Amazon S3, Amazon DynamoDB, and AWS Lambda. You define a rule to invoke a Lambda function when you want to invoke another AWS service or a third-party service.

When an incoming IoT message triggers the rule, AWS IoT invokes your Lambda function asynchronously and passes data from the IoT message to the function.

The following example shows a moisture reading from a greenhouse sensor. The **row** and **pos** values identify the location of the sensor. This example event is based on the greenhouse type in the AWS IoT Rules tutorials.

Example AWS IoT message event

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

For asynchronous invocation, Lambda queues the message and retries if your function returns an error. Configure your function with a destination to retain events that your function could not process.

You need to grant permission for the AWS IoT service to invoke your Lambda function. Use the `add-permission` command to add a permission statement to your function's resource-based policy.

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com
```

You should see the following output:

```
{
    "Statement": "{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":
\"arn:aws:lambda:us-east-1:123456789012:function:my-function\"}"
}
```

For more information about how to use Lambda with AWS IoT, see Creating an AWS Lambda rule.

Using Lambda to process records from Amazon Kinesis Data Streams

You can use a Lambda function to process records in an Amazon Kinesis data stream. You can map a Lambda function to a Kinesis Data Streams shared-throughput consumer (standard iterator), or to a dedicated-throughput consumer with enhanced fan-out. For standard iterators, Lambda polls each shard in your Kinesis stream for records using HTTP protocol. The event source mapping shares read throughput with other consumers of the shard.

For details about Kinesis data streams, see Reading Data from Amazon Kinesis Data Streams.

> **Note**
>
> Kinesis charges for each shard and, for enhanced fan-out, data read from the stream. For pricing details, see Amazon Kinesis pricing.

Polling and batching streams

Lambda reads records from the data stream and invokes your function synchronously with an event that contains stream records. Lambda reads records in batches and invokes your function to process records from the batch. Each batch contains records from a single shard/data stream.

Your Lambda function is a consumer application for your data stream. It processes one batch of records at a time from each shard. You can map a Lambda function to a shared-throughput consumer (standard iterator), or to a dedicated-throughput consumer with enhanced fan-out.

- **Standard iterator:** Lambda polls each shard in your Kinesis stream for records at a base rate of once per second. When more records are available, Lambda keeps processing batches until the function catches up with the stream. The event source mapping shares read throughput with other consumers of the shard.
- **Enhanced fan-out:** To minimize latency and maximize read throughput, create a data stream consumer with enhanced fan-out. Enhanced fan-out consumers get a dedicated connection to each shard that doesn't impact other applications reading from the stream. Stream consumers use HTTP/2 to reduce latency by pushing records to Lambda over a long-lived connection and by compressing request headers. You can create a stream consumer with the Kinesis RegisterStreamConsumer API.

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

You should see the following output:

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

To increase the speed at which your function processes records, add shards to your data stream. Lambda processes records in each shard in order. It stops processing additional records in a shard if your function returns an error. With more shards, there are more batches being processed at once, which lowers the impact of errors on concurrency.

If your function can't scale up to handle the total number of concurrent batches, request a quota increase or reserve concurrency for your function.

By default, Lambda invokes your function as soon as records are available. If the batch that Lambda reads from the event source has only one record in it, Lambda sends only one record to the function. To avoid invoking the function with a small number of records, you can tell the event source to buffer records for up to 5 minutes by configuring a *batching window*. Before invoking the function, Lambda continues to read records from the event source until it has gathered a full batch, the batching window expires, or the batch reaches the payload limit of 6 MB. For more information, see Batching behavior.

> **Warning**
>
> Lambda event source mappings process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

Lambda doesn't wait for any configured extensions to complete before sending the next batch for processing. In other words, your extensions may continue to run as Lambda processes the next batch of records. This can cause throttling issues if you breach any of your account's concurrency settings or limits. To detect whether this is a potential issue, monitor your functions and check whether you're seeing higher concurrency metrics than expected for your event source mapping. Due to short times in between invokes, Lambda may briefly report higher concurrency usage than the number of shards. This can be true even for Lambda functions without extensions.

Configure the ParallelizationFactor setting to process one shard of a Kinesis data stream with more than one Lambda invocation simultaneously. You can specify the number of concurrent batches that Lambda polls from a shard via a parallelization factor from 1 (default) to 10. For example, when you set `ParallelizationFactor` to 2, you can have 200 concurrent Lambda invocations at maximum to process 100 Kinesis data shards (though in practice, you may see different values for the `ConcurrentExecutions` metric). This helps scale up the processing throughput when the data volume is volatile and the `IteratorAge` is high. When you increase the number of concurrent batches per shard, Lambda still ensures in-order processing at the partition-key level.

You can also use `ParallelizationFactor` with Kinesis aggregation. The behavior of the event source mapping depends on whether you're using enhanced fan-out:

- **Without enhanced fan-out**: All of the events inside an aggregated event must have the same partition key. The partition key must also match that of the aggregated event. If the events inside the aggregated event have different partition keys, Lambda cannot guarantee in-order processing of the events by partition key.
- **With enhanced fan-out**: First, Lambda decodes the aggregated event into its individual events. The aggregated event can have a different partition key than events it contains. However, events that don't correspond to the partition key are dropped and lost. Lambda doesn't process these events, and doesn't send them to a configured failure destination.

Example event

Example

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
```

```
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        },
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692540925702759324208523137515618",
                "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
                "approximateArrivalTimestamp": 1545084711.166
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        }
    ]
}
```

Process Amazon Kinesis Data Streams records with Lambda

To process Amazon Kinesis Data Streams records with Lambda, create a Lambda event source mapping. You can map a Lambda function to a standard iterator or enhanced fan-out consumer. For more information, see Polling and batching streams.

Create an Kinesis event source mapping

To invoke your Lambda function with records from your data stream, create an event source mapping. You can create multiple event source mappings to process the same data with multiple Lambda functions, or to process items from multiple data streams with a single function. When processing items from multiple streams, each batch contains records from only a single shard or stream.

You can configure event source mappings to process records from a stream in a different AWS account. To learn more, see the section called “Cross-account mappings”.

Before you create an event source mapping, you need to give your Lambda function permission to read from a Kinesis data stream. Lambda needs the following permissions to manage resources related to your Kinesis data stream:

- kinesis:DescribeStream
- kinesis:DescribeStreamSummary
- kinesis:GetRecords
- kinesis:GetShardIterator
- kinesis:ListShards
- kinesis:SubscribeToShard

The AWS managed policy AWSLambdaKinesisExecutionRole includes these permissions. Add this managed policy to your function as described in the following procedure.

Note

- You don't need the `kinesis:ListStreams` permission to create and manage event source mappings for Kinesis. However, if you create an event source mapping in the console and you don't have this permission, you won't be able to select a Kinesis stream from a dropdown list and the console will display an error. To create the event source mapping, you'll need to manually enter the Amazon Resource Name (ARN) of your stream.
- Lambda makes `kinesis:GetRecords` and `kinesis:GetShardIterator` API calls when retrying failed invocations.

AWS Management Console

To add Kinesis permissions to your function

1. Open the Functions page of the Lambda console and select your function.
2. In the **Configuration** tab, select **Permissions**.
3. In the **Execution role** pane, under **Role name**, choose the link to your function's execution role. This link opens the page for that role in the IAM console.
4. In the **Permissions policies** pane, choose **Add permissions**, then select **Attach policies**.
5. In the search field, enter **`AWSLambdaKinesisExecutionRole`**.
6. Select the checkbox next to the policy and choose **Add permission**.

AWS CLI

To add Kinesis permissions to your function

- Run the following CLI command to add the `AWSLambdaKinesisExecutionRole` policy to your function's execution role:

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

To add Kinesis permissions to your function

- In your function's definition, add the `Policies` property as shown in the following example:

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
```

```
        - AWSLambdaKinesisExecutionRole
```

After configuring the required permissions, create the event source mapping.

AWS Management Console

To create the Kinesis event source mapping

1. Open the Functions page of the Lambda console and select your function.
2. In the **Function overview** pane, choose **Add trigger**.
3. Under **Trigger configuration**, for the source, select **Kinesis**.
4. Select the Kinesis stream you want to create the event source mapping for and, optionally, a consumer of your stream.
5. (Optional) edit the **Batch size**, **Starting position**, and **Batch window** for your event source mapping.
6. Choose **Add**.

When creating your event source mapping from the console, your IAM role must have the kinesis:ListStreams and kinesis:ListStreamConsumers permissions.

AWS CLI

To create the Kinesis event source mapping

- Run the following CLI command to create a Kinesis event source mapping. Choose your own batch size and starting position according to your use case.

```
aws lambda create-event-source-mapping \
--function-name MyFunction \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--starting-position LATEST \
--batch-size 100
```

To specify a batching window, add the `--maximum-batching-window-in-seconds` option. For more information about using this and other parameters, see create-event-source-mapping in the *AWS CLI Command Reference*.

AWS SAM

To create the Kinesis event source mapping

- In your function's definition, add the `KinesisEvent` property as shown in the following example:

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
        - AWSLambdaKinesisExecutionRole
      Events:
        KinesisEvent:
          Type: Kinesis
          Properties:
            Stream: !GetAtt MyKinesisStream.Arn
            StartingPosition: LATEST
            BatchSize: 100

  MyKinesisStream:
    Type: AWS::Kinesis::Stream
    Properties:
      ShardCount: 1
```

To learn more about creating an event source mapping for Kinesis Data Streams in AWS SAM, see Kinesis in the *AWS Serverless Application Model Developer Guide*.

Polling and stream starting position

Be aware that stream polling during event source mapping creation and updates is eventually consistent.

- During event source mapping creation, it may take several minutes to start polling events from the stream.

- During event source mapping updates, it may take several minutes to stop and restart polling events from the stream.

This behavior means that if you specify `LATEST` as the starting position for the stream, the event source mapping could miss events during creation or updates. To ensure that no events are missed, specify the stream starting position as `TRIM_HORIZON` or `AT_TIMESTAMP`.

Creating a cross-account event source mapping

Amazon Kinesis Data Streams supports resource-based policies. Because of this, you can process data ingested into a stream in one AWS account with a Lambda function in another account.

To create an event source mapping for your Lambda function using a Kinesis stream in a different AWS account, you must configure the stream using a resource-based policy to give your Lambda function permission to read items. To learn how to configure your stream to allow cross-account access, see Sharing access with cross-account AWS Lambda functions in the *Amazon Kinesis Streams Developer guide*.

Once you've configured your stream with a resource-based policy that gives your Lambda function the required permissions, create the event source mapping using any of the methods described in the previous section.

If you choose to create your event source mapping using the Lambda console, paste the ARN of your stream directly into the input field. If you want to specify a consumer for your stream, pasting the ARN of the consumer automatically populates the stream field.

Configuring partial batch response with Kinesis Data Streams and Lambda

When consuming and processing streaming data from an event source, by default Lambda checkpoints to the highest sequence number of a batch only when the batch is a complete success. Lambda treats all other results as a complete failure and retries processing the batch up to the retry limit. To allow for partial successes while processing batches from a stream, turn on `ReportBatchItemFailures`. Allowing partial successes can help to reduce the number of retries on a record, though it doesn't entirely prevent the possibility of retries in a successful record.

To turn on `ReportBatchItemFailures`, include the enum value **`ReportBatchItemFailures`** in the FunctionResponseTypes list. This list indicates which response types are enabled for your function. You can configure this list when you create or update an event source mapping.

> **Note**
>
> Even when your function code returns partial batch failure responses, these responses will not be processed by Lambda unless the `ReportBatchItemFailures` feature is explicitly turned on for your event source mapping.

Report syntax

When configuring reporting on batch item failures, the `StreamsEventResponse` class is returned with a list of batch item failures. You can use a `StreamsEventResponse` object to return the sequence number of the first failed record in the batch. You can also create your own custom class using the correct response syntax. The following JSON structure shows the required response syntax:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> If the `batchItemFailures` array contains multiple items, Lambda uses the record with the lowest sequence number as the checkpoint. Lambda then retries all records starting from that checkpoint.

Success and failure conditions

Lambda treats a batch as a complete success if you return any of the following:

- An empty `batchItemFailure` list
- A null `batchItemFailure` list
- An empty `EventResponse`
- A null `EventResponse`

Lambda treats a batch as a complete failure if you return any of the following:

- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name

Lambda retries failures based on your retry strategy.

Bisecting a batch

If your invocation fails and `BisectBatchOnFunctionError` is turned on, the batch is bisected regardless of your `ReportBatchItemFailures` setting.

When a partial batch success response is received and both `BisectBatchOnFunctionError` and `ReportBatchItemFailures` are turned on, the batch is bisected at the returned sequence number and Lambda retries only the remaining records.

To simplify the implementation of partial batch response logic, consider using the Batch Processor utility from Powertools for AWS Lambda, which automatically handles these complexities for you.

Here are some examples of function code that return the list of failed message IDs in the batch:

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;
```

```
// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
```

```
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Javascript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Reporting Kinesis batch item failures with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

Using Powertools for AWS Lambda batch processor

The batch processor utility from Powertools for AWS Lambda automatically handles partial batch response logic, reducing the complexity of implementing batch failure reporting. Here are examples using the batch processor:

Python

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing Kinesis Data Streams stream records with AWS Lambda batch processor.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import KinesisEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.KinesisDataStreams)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed
```

```
def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing Kinesis Data Streams stream records with AWS Lambda batch processor.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { KinesisEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.KinesisDataStreams);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: KinesisEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Java

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing Kinesis Data Streams stream records with AWS Lambda batch processor.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import software.amazon.lambda.powertools.batch.BatchMessageHandlerBuilder;
import software.amazon.lambda.powertools.batch.handler.BatchMessageHandler;

public class KinesisStreamBatchHandler implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    private final BatchMessageHandler<KinesisEvent, StreamsEventResponse> handler;

    public KinesisStreamBatchHandler() {
        handler = new BatchMessageHandlerBuilder()
                .withKinesisBatchHandler()
                .buildWithRawMessageHandler(this::processMessage);
    }

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent kinesisEvent, Context
 context) {
        return handler.processBatch(kinesisEvent, context);
    }

    private void processMessage(KinesisEvent.KinesisEventRecord kinesisEventRecord,
 Context context) {
        // Process the stream record
    }
}
```

.NET

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing Kinesis Data Streams stream records with AWS Lambda batch processor.

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using Amazon.Lambda.Serialization.SystemTextJson;
using AWS.Lambda.Powertools.BatchProcessing;

[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]

namespace HelloWorld;

public class OrderEvent
{
    public string? OrderId { get; set; }
    public string? CustomerId { get; set; }
    public decimal Amount { get; set; }
    public DateTime OrderDate { get; set; }
}

internal class TypedKinesisRecordHandler : ITypedRecordHandler<OrderEvent>
{
    public async Task<RecordHandlerResult> HandleAsync(OrderEvent orderEvent,
 CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(orderEvent.OrderId))
        {
            throw new ArgumentException("Order ID is required");
        }

        return await Task.FromResult(RecordHandlerResult.None);
    }
}
```

```
public class Function
{
    [BatchProcessor(TypedRecordHandler = typeof(TypedKinesisRecordHandler))]
    public BatchItemFailuresResponse HandlerUsingTypedAttribute(KinesisEvent _)
    {
        return TypedKinesisStreamBatchProcessor.Result.BatchItemFailuresResponse;
    }
}
```

Retain discarded batch records for a Kinesis Data Streams event source in Lambda

Error handling for Kinesis event source mappings depends on whether the error occurs before the function is invoked or during function invocation:

- **Before invocation:** If a Lambda event source mapping is unable to invoke the function due to throttling or other issues, it retries until the records expire or exceed the maximum age configured on the event source mapping (MaximumRecordAgeInSeconds).
- **During invocation:** If the function is invoked but returns an error, Lambda retries until the records expire, exceed the maximum age (MaximumRecordAgeInSeconds), or reach the configured retry quota (MaximumRetryAttempts). For function errors, you can also configure BisectBatchOnFunctionError, which splits a failed batch into two smaller batches, isolating bad records and avoiding timeouts. Splitting batches doesn't consume the retry quota.

If the error handling measures fail, Lambda discards the records and continues processing batches from the stream. With the default settings, this means that a bad record can block processing on the affected shard for up to one week. To avoid this, configure your function's event source mapping with a reasonable number of retries and a maximum record age that fits your use case.

Configuring destinations for failed invocations

To retain records of failed event source mapping invocations, add a destination to your function's event source mapping. Each record sent to the destination is a JSON document containing metadata about the failed invocation. For Amazon S3 destinations, Lambda also sends the entire invocation record along with the metadata. You can configure any Amazon SNS topic, Amazon SQS queue, Amazon S3 bucket, or Kafka as a destination.

With Amazon S3 destinations, you can use the Amazon S3 Event Notifications feature to receive notifications when objects are uploaded to your destination S3 bucket. You can also configure S3 Event Notifications to invoke another Lambda function to perform automated processing on failed batches.

Your execution role must have permissions for the destination:

- **For an SQS destination:** sqs:SendMessage
- **For an SNS destination:** sns:Publish
- **For an S3 destination:** s3:PutObject and s3:ListBucket
- **For a Kafka destination:** kafka-cluster:WriteData

You can configure a Kafka topic as an on-failure destination for your Kafka event source mappings. When Lambda can't process records after exhausting retry attempts or when records exceed the maximum age, Lambda sends the failed records to the specified Kafka topic for later processing. Refer to the section called “Kafka on-failure destination”.

If you've enabled encryption with your own KMS key for an S3 destination, your function's execution role must also have permission to call kms:GenerateDataKey. If the KMS key and S3 bucket destination are in a different account from your Lambda function and execution role, configure the KMS key to trust the execution role to allow kms:GenerateDataKey.

To configure an on-failure destination using the console, follow these steps:

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Under **Function overview**, choose **Add destination**.
4. For **Source**, choose **Event source mapping invocation**.
5. For **Event source mapping**, choose an event source that's configured for this function.
6. For **Condition**, select **On failure**. For event source mapping invocations, this is the only accepted condition.
7. For **Destination type**, choose the destination type that Lambda sends invocation records to.
8. For **Destination**, choose a resource.
9. Choose **Save**.

You can also configure an on-failure destination using the AWS Command Line Interface (AWS CLI). For example, the following create-event-source-mapping command adds an event source mapping with an SQS on-failure destination to `MyFunction`:

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

The following update-event-source-mapping command updates an event source mapping to send failed invocation records to an SNS destination after two retry attempts, or if the records are more than an hour old.

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

Updated settings are applied asynchronously and aren't reflected in the output until the process completes. Use the get-event-source-mapping command to view the current status.

To remove a destination, supply an empty string as the argument to the `destination-config` parameter:

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Security best practices for Amazon S3 destinations

Deleting an S3 bucket that's configured as a destination without removing the destination from your function's configuration can create a security risk. If another user knows your destination bucket's name, they can recreate the bucket in their AWS account. Records of failed invocations will be sent to their bucket, potentially exposing data from your function.

> **Warning**
>
> To ensure that invocation records from your function can't be sent to an S3 bucket in another AWS account, add a condition to your function's execution role that limits `s3:PutObject` permissions to buckets in your account.

The following example shows an IAM policy that limits your function's `s3:PutObject` permissions to buckets in your account. This policy also gives Lambda the `s3:ListBucket` permission it needs to use an S3 bucket as a destination.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

To add a permissions policy to your function's execution role using the AWS Management Console or AWS CLI, refer to the instructions in the following procedures:

Console

To add a permissions policy to a function's execution role (console)

1. Open the Functions page of the Lambda console.
2. Select the Lambda function whose execution role you want to modify.
3. In the **Configuration** tab, select **Permissions**.
4. In the **Execution role** tab, select your function's **Role name** to open the role's IAM console page.
5. Add a permissions policy to the role by doing the following:
   a. In the **Permissions policies** pane, choose **Add permissions** and select **Create inline policy**.
   b. In **Policy editor**, select **JSON**.
   c. Paste the policy you want to add into the editor (replacing the existing JSON), and then choose **Next**.
   d. Under **Policy details**, enter a **Policy name**.
   e. Choose **Create policy**.

AWS CLI

To add a permissions policy to a function's execution role (CLI)

1. Create a JSON policy document with the required permissions and save it in a local directory.
2. Use the IAM `put-role-policy` CLI command to add the permissions to your function's execution role. Run the following command from the directory you saved your JSON policy document in and replace the role name, policy name, and policy document with your own values.

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Example Amazon SNS and Amazon SQS invocation record

The following example shows what Lambda sends to an SQS queue or SNS topic for a failed Kinesis event source invocation. Because Lambda sends only the metadata for these destination types, use the `streamArn`, `shardId`, `startSequenceNumber`, and `endSequenceNumber` fields to obtain the full original record. All of the fields shown in the `KinesisBatchInfo` property will always be present.

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    }
}
```

You can use this information to retrieve the affected records from the stream for troubleshooting. The actual records aren't included, so you must process this record and retrieve them from the stream before they expire and are lost.

Example Amazon S3 invocation record

The following example shows what Lambda sends to an Amazon S3 bucket for a failed Kinesis event source invocation. In addition to all of the fields from the previous example for SQS and SNS destinations, the `payload` field contains the original invocation record as an escaped JSON string.

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

The S3 object containing the invocation record uses the following naming convention:

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

Implementing stateful Kinesis Data Streams processing in Lambda

Lambda functions can run continuous stream processing applications. A stream represents unbounded data that flows continuously through your application. To analyze information from

this continuously updating input, you can bound the included records using a window defined in terms of time.

Tumbling windows are distinct time windows that open and close at regular intervals. By default, Lambda invocations are stateless—you cannot use them for processing data across multiple continuous invocations without an external database. However, with tumbling windows, you can maintain your state across invocations. This state contains the aggregate result of the messages previously processed for the current window. Your state can be a maximum of 1 MB per shard. If it exceeds that size, Lambda terminates the window early.

Each record in a stream belongs to a specific window. Lambda will process each record at least once, but doesn't guarantee that each record will be processed only once. In rare cases, such as error handling, some records might be processed more than once. Records are always processed in order the first time. If records are processed more than once, they might be processed out of order.

Aggregation and processing

Your user managed function is invoked both for aggregation and for processing the final results of that aggregation. Lambda aggregates all records received in the window. You can receive these records in multiple batches, each as a separate invocation. Each invocation receives a state. Thus, when using tumbling windows, your Lambda function response must contain a `state` property. If the response does not contain a `state` property, Lambda considers this a failed invocation. To satisfy this condition, your function can return a `TimeWindowEventResponse` object, which has the following JSON shape:

Example `TimeWindowEventResponse` values

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> For Java functions, we recommend using a `Map<String, String>` to represent the state.

At the end of the window, the flag `isFinalInvokeForWindow` is set to `true` to indicate that this is the final state and that it's ready for processing. After processing, the window completes and your final invocation completes, and then the state is dropped.

At the end of your window, Lambda uses final processing for actions on the aggregation results. Your final processing is synchronously invoked. After successful invocation, your function checkpoints the sequence number and stream processing continues. If invocation is unsuccessful, your Lambda function suspends further processing until a successful invocation.

Example KinesisTimeWindowEvent

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1607497475.000
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
            "awsRegion": "us-east-1",
            "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-
stream"
        }
    ],
    "window": {
        "start": "2020-12-09T07:04:00Z",
        "end": "2020-12-09T07:06:00Z"
    },
    "state": {
        "1": 282,
        "2": 715
    },
    "shardId": "shardId-000000000006",
```

```
    "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Configuration

You can configure tumbling windows when you create or update an event source mapping. To configure a tumbling window, specify the window in seconds (TumblingWindowInSeconds). The following example AWS Command Line Interface (AWS CLI) command creates a streaming event source mapping that has a tumbling window of 120 seconds. The Lambda function defined for aggregation and processing is named `tumbling-window-example-function`.

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda determines tumbling window boundaries based on the time when records were inserted into the stream. All records have an approximate timestamp available that Lambda uses in boundary determinations.

Tumbling window aggregations do not support resharding. When a shard ends, Lambda considers the current window to be closed, and any child shards will start their own window in a fresh state. When no new records are being added to the current window, Lambda waits for up to 2 minutes before assuming that the window is over. This helps ensure that the function reads all records in the current window, even if the records are added intermittently.

Tumbling windows fully support the existing retry policies `maxRetryAttempts` and `maxRecordAge`.

Example Handler.py – Aggregation and processing

The following Python function demonstrates how to aggregate and then process your final state:

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])
```

```
#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['kinesis']['partitionKey']] = state.get(record['kinesis']
['partitionKey'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Lambda parameters for Amazon Kinesis Data Streams event source mappings

All Lambda event source mappings share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to Kinesis.

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
BisectBatchOnFunctionError	N	false	none
DestinationConfig	N	N/A	Amazon SQS queue or Amazon SNS topic destination for discarded records. For more informati on, see Configuring

Parameter	Required	Default	Notes
			destinations for failed invocations.
Enabled	N	true	none
EventSourceArn	Y	N/A	ARN of the data stream or a stream consumer
FunctionName	Y	N/A	none
FunctionResponseTypes	N	N/A	To let your function report specific failures in a batch, include the value `ReportBatchItemFailures` in `FunctionResponseTypes`. For more information, see Configuring partial batch response with Kinesis Data Streams and Lambda.
MaximumBatchingWindowInSeconds	N	0	none

Parameter	Required	Default	Notes
MaximumRecordAgeInSeconds	N	-1	-1 means infinite: Lambda doesn't discard records (Kinesis Data Streams data retention settings still apply) Minimum: -1 Maximum: 604,800
MaximumRetryAttempts	N	-1	-1 means infinite: failed records are retried until the record expires Minimum: -1 Maximum: 10,000
ParallelizationFactor	N	1	Maximum: 10
StartingPosition	Y	N/A	AT_TIMESTAMP, TRIM_HORIZON, or LATEST
StartingPositionTimestamp	N	N/A	Only valid if StartingPosition is set to AT_TIMESTAMP. The time from which to start reading, in Unix time seconds
TumblingWindowInSeconds	N	N/A	Minimum: 0 Maximum: 900

Using event filtering with a Kinesis event source

You can use event filtering to control which records from a stream or queue Lambda sends to your function. For general information about how event filtering works, see the section called "Event filtering".

This section focuses on event filtering for Kinesis event sources.

> **Note**
>
> Kinesis event source mappings only support filtering on the `data` key.

Topics

- Kinesis event filtering basics
- Filtering Kinesis aggregated records

Kinesis event filtering basics

Suppose a producer is putting JSON formatted data into your Kinesis data stream. An example record would look like the following, with the JSON data converted to a Base64 encoded string in the `data` field.

```
{
    "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
        "data":
 "eyJSZWNvcmROdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhbXAiOiAieXl5eS1tbS1kZFRoaDptbTpzcyIsICJSZXF1ZXN0Q
        "approximateArrivalTimestamp": 1545084650.987
        },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
```

```
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

As long as the data the producer puts into the stream is valid JSON, you can use event filtering to filter records using the `data` key. Suppose a producer is putting records into your Kinesis stream in the following JSON format.

```
{
    "record": 12345,
    "order": {
        "type": "buy",
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

To filter only those records where the order type is “buy,” the `FilterCriteria` object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
            }
      }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/my-stream \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

To properly filter events from Kinesis sources, both the data field and your filter criteria for the data field must be in valid JSON format. If either field isn't in a valid JSON format, Lambda drops the message or throws an exception. The following table summarizes the specific behavior:

Incoming data format	Filter pattern format for data properties	Resulting action
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Non-JSON	Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.
Non-JSON	Valid JSON	Lambda drops the record.
Non-JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Non-JSON	Non-JSON	Lambda throws an exception at the time of the event source mapping creation or update. The filter pattern for data properties must be in a valid JSON format.

Filtering Kinesis aggregated records

With Kinesis, you can aggregate multiple records into a single Kinesis Data Streams record to increase your data throughput. Lambda can only apply filter criteria to aggregated records when you use Kinesis enhanced fan-out. Filtering aggregated records with standard Kinesis isn't supported. When using enhanced fan-out, you configure a Kinesis dedicated-throughput consumer

to act as the trigger for your Lambda function. Lambda then filters the aggregated records and passes only those records that meet your filter criteria.

To learn more about Kinesis record aggregation, refer to the Aggregation section on the Kinesis Producer Library (KPL) Key Concepts page. To Learn more about using Lambda with Kinesis enhanced fan-out, see Increasing real-time stream processing performance with Amazon Kinesis Data Streams enhanced fan-out and AWS Lambda on the AWS compute blog.

Tutorial: Using Lambda with Kinesis Data Streams

In this tutorial, you create a Lambda function to consume events from a Amazon Kinesis data stream.

1. Custom app writes records to the stream.
2. AWS Lambda polls the stream and, when it detects new records in the stream, invokes your Lambda function.
3. AWS Lambda runs the Lambda function by assuming the execution role you specified at the time you created the Lambda function.

Prerequisites

Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create the execution role

Create the execution role that gives your function permission to access AWS resources.

To create an execution role

1. Open the roles page in the IAM console.
2. Choose **Create role**.
3. Create a role with the following properties.
   - **Trusted entity** – **AWS Lambda**.
   - **Permissions** – **AWSLambdaKinesisExecutionRole**.
   - **Role name** – **`lambda-kinesis-role`**.

The **AWSLambdaKinesisExecutionRole** policy has the permissions that the function needs to read items from Kinesis and write logs to CloudWatch Logs.

Create the function

Create a Lambda function that processes your Kinesis messages. The function code logs the event ID and event data of the Kinesis record to CloudWatch Logs.

This tutorial uses the Node.js 24 runtime, but we've also provided example code in other runtime languages. You can select the tab in the following box to see code for the runtime you're interested in. The JavaScript code you'll use in this step is in the first example shown in the **JavaScript** tab.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;
```

```
// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
```

```
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
```

```
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
```

```
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
```

```
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Consuming a Kinesis event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
```

```
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
```

```
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
```

```
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
```

```
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

To create the function

1. Create a directory for the project, and then switch to that directory.

```
mkdir kinesis-tutorial
cd kinesis-tutorial
```

2. Copy the sample JavaScript code into a new file named `index.js`.
3. Create a deployment package.

```
zip function.zip index.js
```

4. Create a Lambda function with the `create-function` command.

```
aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/lambda-kinesis-role
```

Test the Lambda function

Invoke your Lambda function manually using the `invoke` AWS Lambda CLI command and a sample Kinesis event.

To test the Lambda function

1. Copy the following JSON into a file and save it as `input.txt`.

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
```

```
            "invokeIdentityArn": "arn:aws:iam::111122223333:role/lambda-kinesis-
role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:111122223333:stream/
lambda-stream"
        }
    ]
}
```

2. Use the `invoke` command to send the event to the function.

```
aws lambda invoke --function-name ProcessKinesisRecords \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

The response is saved to `out.txt`.

Create a Kinesis stream

Use the `create-stream` command to create a stream.

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

Run the following `describe-stream` command to get the stream ARN.

```
aws kinesis describe-stream --stream-name lambda-stream
```

You should see the following output:

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
```

```
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49591073947768692513481539594623130411957558361251844610"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream",
        "StreamName": "lambda-stream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

You use the stream ARN in the next step to associate the stream with your Lambda function.

Add an event source in AWS Lambda

Run the following AWS CLI `add-event-source` command.

```
aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source  arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

Note the mapping ID for later use. You can get a list of event source mappings by running the `list-event-source-mappings` command.

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

In the response, you can verify the status value is `enabled`. Event source mappings can be disabled to pause polling temporarily without losing any records.

Test the setup

To test the event source mapping, add event records to your Kinesis stream. The `--data` value is a string that the CLI encodes to base64 prior to sending it to Kinesis. You can run the same command more than once to add multiple records to the stream.

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda uses the execution role to read records from the stream. Then it invokes your Lambda function, passing in batches of records. The function decodes data from each record and logs it, sending the output to CloudWatch Logs. View the logs in the CloudWatch console.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the Kinesis stream

1. Sign in to the AWS Management Console and open the Kinesis console at https://console.aws.amazon.com/kinesis.

2. Select the stream you created.
3. Choose **Actions**, **Delete**.
4. Enter **`delete`** in the text input field.
5. Choose **Delete**.

Using Lambda with Kubernetes

You can deploy and manage Lambda functions with the Kubernetes API using AWS Controllers for Kubernetes (ACK) or Crossplane.

AWS Controllers for Kubernetes (ACK)

You can use ACK to deploy and manage AWS resources from the Kubernetes API. Through ACK, AWS provides open-source custom controllers for AWS services such as Lambda, Amazon Elastic Container Registry (Amazon ECR), Amazon Simple Storage Service (Amazon S3), and Amazon SageMaker AI. Each supported AWS service has its own custom controller. In your Kubernetes cluster, install a controller for each AWS service that you want to use. Then, create a Custom Resource Definition (CRD) to define the AWS resources.

We recommend that you use Helm 3.8 or later to install ACK controllers. Every ACK controller comes with its own Helm chart, which installs the controller, CRDs, and Kubernetes RBAC rules. For more information, see Install an ACK Controller in the ACK documentation.

After you create the ACK custom resource, you can use it like any other built-in Kubernetes object. For example, you can deploy and manage Lambda functions with your preferred Kubernetes toolchains, including kubectl.

Here are some example use cases for provisioning Lambda functions through ACK:

- Your organization uses role-based access control (RBAC) and IAM roles for service accounts to create permissions boundaries. With ACK, you can reuse this security model for Lambda without having to create new users and policies.
- Your organization has a DevOps process to deploy resources into an Amazon Elastic Kubernetes Service (Amazon EKS) cluster using Kubernetes manifests. With ACK, you can use a manifest to provision Lambda functions without creating separate infrastructure as code templates.

For more information about using ACK, see the Lambda tutorial in the ACK documentation.

Crossplane

Crossplane is an open-source Cloud Native Computing Foundation (CNCF) project that uses Kubernetes to manage cloud infrastructure resources. With Crossplane, developers can request infrastructure without needing to understand its complexities. Platform teams retain control over how the infrastructure is provisioned and managed.

Using Crossplane, you can deploy and manage Lambda functions with your preferred Kubernetes toolchains such as kubectl, and any CI/CD pipeline that can deploy manifests to Kubernetes. Here are some example use cases for provisioning Lambda functions through Crossplane:

- Your organization wants to enforce compliance by ensuring that Lambda functions have the correct tags. Platform teams can use Crossplane Compositions to define this policy through API abstractions. Developers can then use these abstractions to deploy Lambda functions with tags.
- Your project uses GitOps with Kubernetes. In this model, Kubernetes continuously reconciles the git repository (desired state) with the resources running inside the cluster (current state). If there are differences, the GitOps process automatically makes changes to the cluster. You can use GitOps with Kubernetes for deploying and managing Lambda functions through Crossplane, using familiar Kubernetes tools and concepts such as CRDs and Controllers.

To learn more about using Crossplane with Lambda, see the following:

- AWS Blueprints for Crossplane: This repository includes examples of how to use Crossplane to deploy AWS resources, including Lambda functions.

  > **Note**
  >
  > AWS Blueprints for Crossplane are under active development and should not be used in production.

- Deploying Lambda with Amazon EKS and Crossplane: This video demonstrates an advanced example of deploying an AWS serverless architecture with Crossplane, exploring the design from both the developer and platform perspectives.

Using Lambda with Amazon MQ

> **Note**
>
> If you want to send data to a target other than a Lambda function or enrich the data before sending it, see Amazon EventBridge Pipes.

Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ. A *message broker* enables software applications and components to communicate using various programming languages, operating systems, and formal messaging protocols through either topic or queue event destinations.

Amazon MQ can also manage Amazon Elastic Compute Cloud (Amazon EC2) instances on your behalf by installing ActiveMQ or RabbitMQ brokers and by providing different network topologies and other infrastructure needs.

You can use a Lambda function to process records from your Amazon MQ message broker. Lambda invokes your function through an event source mapping, a Lambda resource that reads messages from your broker and invokes the function synchronously.

> **Warning**
>
> Lambda event source mappings process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

The Amazon MQ event source mapping has the following configuration restrictions:

- Concurrency – Lambda functions that use an Amazon MQ event source mapping have a default maximum concurrency setting. For ActiveMQ, the Lambda service limits the number of concurrent execution environments to five per Amazon MQ event source mapping. For RabbitMQ, the number of concurrent execution environments is limited to 1 per Amazon MQ event source mapping. Even if you change your function's reserved or provisioned concurrency settings, the Lambda service won't make more execution environments available. To request an increase in the default maximum concurrency for a single Amazon MQ event source mapping,

contact Support with the event source mapping UUID, as well as the region. Because increases are applied at the specific event source mapping level, not the account or region level, you need to manually request a scaling increase for each event source mapping.

- Cross account – Lambda does not support cross-account processing. You cannot use Lambda to process records from an Amazon MQ message broker that is in a different AWS account.
- Authentication – For ActiveMQ, only the ActiveMQ SimpleAuthenticationPlugin is supported. For RabbitMQ, only the PLAIN authentication mechanism is supported. Users must use AWS Secrets Manager to manage their credentials. For more information about ActiveMQ authentication, see Integrating ActiveMQ brokers with LDAP in the *Amazon MQ Developer Guide*.
- Connection quota – Brokers have a maximum number of allowed connections per wire-level protocol. This quota is based on the broker instance type. For more information, see the Brokers section of **Quotas in Amazon MQ** in the *Amazon MQ Developer Guide*.
- Connectivity – You can create brokers in a public or private virtual private cloud (VPC). For private VPCs, your Lambda function needs access to the VPC to receive messages. For more information, see the section called “Configure network security” later in this section.
- Event destinations – Only queue destinations are supported. However, you can use a virtual topic, which behaves as a topic internally while interacting with Lambda as a queue. For more information, see Virtual Destinations on the Apache ActiveMQ website, and Virtual Hosts on the RabbitMQ website.
- Network topology – For ActiveMQ, only one single-instance or standby broker is supported per event source mapping. For RabbitMQ, only one single-instance broker or cluster deployment is supported per event source mapping. Single-instance brokers require a failover endpoint. For more information about these broker deployment modes, see Active MQ Broker Architecture and Rabbit MQ Broker Architecturein the *Amazon MQ Developer Guide*.
- Protocols – Supported protocols depend on the type of Amazon MQ integration.
  - For ActiveMQ integrations, Lambda consumes messages using the OpenWire/Java Message Service (JMS) protocol. No other protocols are supported for consuming messages. Within the JMS protocol, only `TextMessage` and `BytesMessage` are supported. Lambda also supports JMS custom properties. For more information about the OpenWire protocol, see OpenWire on the Apache ActiveMQ website.
  - For RabbitMQ integrations, Lambda consumes messages using the AMQP 0-9-1 protocol. No other protocols are supported for consuming messages. For more information about RabbitMQ's implementation of the AMQP 0-9-1 protocol, see AMQP 0-9-1 Complete Reference Guide on the RabbitMQ website.

Lambda automatically supports the latest versions of ActiveMQ and RabbitMQ that Amazon MQ supports. For the latest supported versions, see Amazon MQ release notes in the *Amazon MQ Developer Guide*.

> **Note**
>
> By default, Amazon MQ has a weekly maintenance window for brokers. During that window of time, brokers are unavailable. For brokers without standby, Lambda cannot process any messages during that window.

Topics

- Understanding the Lambda consumer group for Amazon MQ
- Configuring Amazon MQ event source for Lambda
- Event source mapping parameters
- Filter events from an Amazon MQ event source
- Troubleshoot Amazon MQ event source mapping errors

Understanding the Lambda consumer group for Amazon MQ

To interact with Amazon MQ, Lambda creates a consumer group which can read from your Amazon MQ brokers. The consumer group is created with the same ID as the event source mapping UUID.

For Amazon MQ event sources, Lambda batches records together and sends them to your function in a single payload. To control behavior, you can configure the batching window and batch size. Lambda pulls messages until it processes the payload size maximum of 6 MB, the batching window expires, or the number of records reaches the full batch size. For more information, see Batching behavior.

The consumer group retrieves the messages as a BLOB of bytes, base64-encodes them into a single JSON payload, and then invokes your function. If your function returns an error for any of the messages in a batch, Lambda retries the whole batch of messages until processing succeeds or the messages expire.

> **Note**
>
> While Lambda functions typically have a maximum timeout limit of 15 minutes, event source mappings for Amazon MSK, self-managed Apache Kafka, Amazon DocumentDB, and Amazon MQ for ActiveMQ and RabbitMQ only support functions with maximum timeout limits of 14 minutes. This constraint ensures that the event source mapping can properly handle function errors and retries.

You can monitor a given function's concurrency usage using the `ConcurrentExecutions` metric in Amazon CloudWatch. For more information about concurrency, see the section called "Configuring reserved concurrency".

Example Amazon MQ record events

ActiveMQ

```
{
   "eventSource": "aws:mq",
   "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
   "messages": [
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/text-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 1,
        "correlationId": "myJMSCoID",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"QUJDOkFBQUE=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
```

```
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      },
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/bytes-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 2,
        "correlationId": "myJMSCoID1",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"LQaGQ82S48k=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      }
   ]
}
```

RabbitMQ

```
{
  "eventSource": "aws:rmq",
  "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:pizzaBroker:b-9bcfa592-423a-4942-879d-eb284b418fc8",
  "rmqMessagesByQueue": {
    "pizzaQueue::/": [
      {
        "basicProperties": {
          "contentType": "text/plain",
```

```
          "contentEncoding": null,
          "headers": {
            "header1": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                49
              ]
            },
            "header2": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                50
              ]
            },
            "numberInHeader": 10
          },
          "deliveryMode": 1,
          "priority": 34,
          "correlationId": null,
          "replyTo": null,
          "expiration": "60000",
          "messageId": null,
          "timestamp": "Jan 1, 1970, 12:33:41 AM",
          "type": null,
          "userId": "AIDACKCEVSQ6C2EXAMPLE",
          "appId": null,
          "clusterId": null,
          "bodySize": 80
        },
        "redelivered": false,
        "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
      }
    ]
  }
}
```

> **Note**
>
> In the RabbitMQ example, `pizzaQueue` is the name of the RabbitMQ queue, and `/` is the name of the virtual host. When receiving messages, the event source lists messages under `pizzaQueue::/`.

Configuring Amazon MQ event source for Lambda

Topics

- Configure network security
- Create the event source mapping

Configure network security

To give Lambda full access to Amazon MQ through your event source mapping, either your broker must use a public endpoint (public IP address), or you must provide access to the Amazon VPC you created the broker in.

When you use Amazon MQ with Lambda, create AWS PrivateLink VPC endpoints that provide your function access to the resources in your Amazon VPC.

> **Note**
>
> AWS PrivateLink VPC endpoints are required for functions with event source mappings that use the default (on-demand) mode for event pollers. If your event source mapping uses provisioned mode, you don't need to configure AWS PrivateLink VPC endpoints.

Create an endpoint to provide access to the following resources:

- Lambda — Create an endpoint for the Lambda service principal.
- AWS STS — Create an endpoint for the AWS STS in order for a service principal to assume a role on your behalf.
- Secrets Manager — If your broker uses Secrets Manager to store credentials, create an endpoint for Secrets Manager.

Alternatively, configure a NAT gateway on each public subnet in the Amazon VPC. For more information, see the section called “Internet access for VPC functions”.

When you create an event source mapping for Amazon MQ, Lambda checks whether Elastic Network Interfaces (ENIs) are already present for the subnets and security groups configured for your Amazon VPC. If Lambda finds existing ENIs, it attempts to re-use them. Otherwise, Lambda creates new ENIs to connect to the event source and invoke your function.

> **Note**
>
> Lambda functions always run inside VPCs owned by the Lambda service. Your function's VPC configuration does not affect the event source mapping. Only the networking configuration of the event source's determines how Lambda connects to your event source.

Configure the security groups for the Amazon VPC containing your broker. By default, Amazon MQ uses the following ports: 61617 (Amazon MQ for ActiveMQ), and 5671 (Amazon MQ for RabbitMQ).

- Inbound rules – Allow all traffic on the default broker port for the security group associated with your event source. Alternatively, you can use a self-referencing security group rule to allow access from instances within the same security group.
- Outbound rules – Allow all traffic on port 443 for external destinations if your function needs to communicate with AWS services. Alternatively, you can also use a self-referencing security group rule to limit access to the broker if you don't need to communicate with other AWS services.
- Amazon VPC endpoint inbound rules — If you are using an Amazon VPC endpoint, the security group associated with your Amazon VPC endpoint must allow inbound traffic on port 443 from the broker security group.

If your broker uses authentication, you can also restrict the endpoint policy for the Secrets Manager endpoint. To call the Secrets Manager API, Lambda uses your function role, not the Lambda service principal.

Example VPC endpoint policy — Secrets Manager endpoint

```
{
      "Statement": [
          {
```

```
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

When you use Amazon VPC endpoints, AWS routes your API calls to invoke your function using the endpoint's Elastic Network Interface (ENI). The Lambda service principal needs to call `lambda:InvokeFunction` on any roles and functions that use those ENIs.

By default, Amazon VPC endpoints have open IAM policies that allow broad access to resources. Best practice is to restrict these policies to perform the needed actions using that endpoint. To ensure that your event source mapping is able to invoke your Lambda function, the VPC endpoint policy must allow the Lambda service principal to call `sts:AssumeRole` and `lambda:InvokeFunction`. Restricting your VPC endpoint policies to allow only API calls originating within your organization prevents the event source mapping from functioning properly, so `"Resource": "*"` is required in these policies.

The following example VPC endpoint policies show how to grant the required access to the Lambda service principal for the AWS STS and Lambda endpoints.

Example VPC Endpoint policy — AWS STS endpoint

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
```

```
      ]
    }
```

Example VPC Endpoint policy — Lambda endpoint

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

Create the event source mapping

Create an event source mapping to tell Lambda to send records from an Amazon MQ broker to a Lambda function. You can create multiple event source mappings to process the same data with multiple functions, or to process items from multiple sources with a single function.

To configure your function to read from Amazon MQ, add the required permissions and create an **MQ** trigger in the Lambda console.

To read records from an Amazon MQ broker, your Lambda function needs the following permissions. You grant Lambda permission to interact with your Amazon MQ broker and its underlying resouces by adding permission statements to your function execution role:

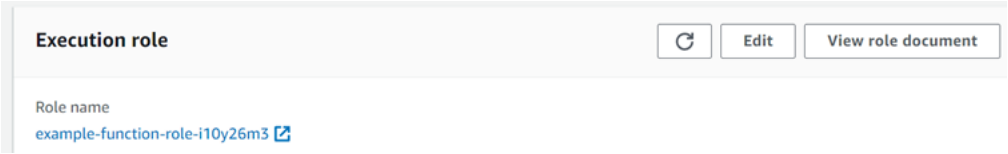
- mq:DescribeBroker
- secretsmanager:GetSecretValue
- ec2:CreateNetworkInterface
- ec2:DeleteNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets

- ec2:DescribeVpcs
- logs:CreateLogGroup
- logs:CreateLogStream
- logs:PutLogEvents

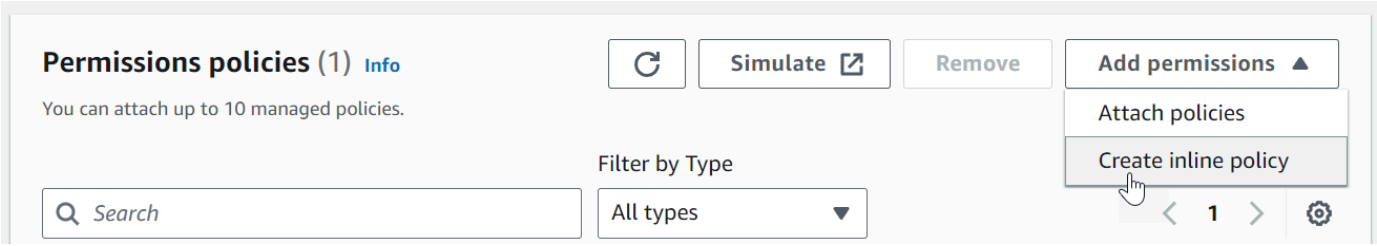
> **Note**
>
> When using an encrypted customer managed key, add the `kms:Decrypt` permission as well.

To add permissions and create a trigger

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose the **Configuration** tab, and then choose **Permissions**.
4. Under **Role name**, choose the link to your execution role. This link opens the role in the IAM console.



5. Choose **Add permissions**, and then choose **Create inline policy**.



6. In the **Policy editor**, choose **JSON**. Enter the following policy. Your function needs these permissions to read from an Amazon MQ broker.

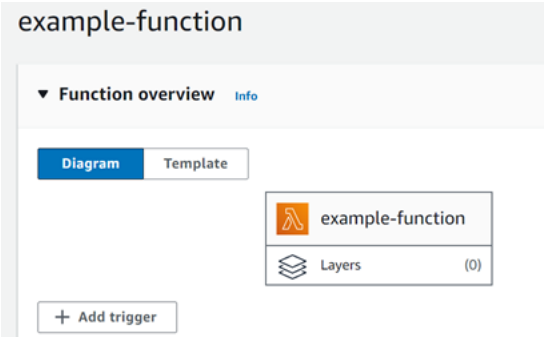
   JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "mq:DescribeBroker",
          "secretsmanager:GetSecretValue",
          "ec2:CreateNetworkInterface",
          "ec2:DeleteNetworkInterface",
          "ec2:DescribeNetworkInterfaces",
          "ec2:DescribeSecurityGroups",
          "ec2:DescribeSubnets",
          "ec2:DescribeVpcs",
          "logs:CreateLogGroup",
          "logs:CreateLogStream",
          "logs:PutLogEvents"
        ],
        "Resource": "*"
      }
    ]
  }
```

> **Note**
>
> When using an encrypted customer managed key, you must also add the `kms:Decrypt` permission.

7. Choose **Next**. Enter a policy name and then choose **Create policy**.
8. Go back to your function in the Lambda console. Under **Function overview**, choose **Add trigger**.



9. Choose the **MQ** trigger type.
10. Configure the required options, and then choose **Add**.

Lambda supports the following options for Amazon MQ event sources:

- **MQ broker** – Select an Amazon MQ broker.
- **Batch size** – Set the maximum number of messages to retrieve in a single batch.
- **Queue name** – Enter the Amazon MQ queue to consume.
- **Source access configuration** – Enter virtual host information and the Secrets Manager secret that stores your broker credentials.
- **Enable trigger** – Disable the trigger to stop processing records.

To enable or disable the trigger (or delete it), choose the **MQ** trigger in the designer. To reconfigure the trigger, use the event source mapping API operations.

Event source mapping parameters

All Lambda event source types share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to Amazon MQ and RabbitMQ.

Parameter	Required	Default	Notes
BatchSize	N	100	Maximum: 10,000
Enabled	N	true	none
FunctionName	Y	N/A	none
FilterCriteria	N	N/A	Control which events Lambda sends to your function
MaximumBatchingWindowInSeconds	N	500 ms	Batching behavior
Queues	N	N/A	The name of the Amazon MQ broker destination queue to consume.

Parameter	Required	Default	Notes
SourceAccessConfigurations	N	N/A	For ActiveMQ, BASIC_AUTH credentials. For RabbitMQ, can contain both BASIC_AUTH credentials and VIRTUAL_HOST information.

Filter events from an Amazon MQ event source

You can use event filtering to control which records from a stream or queue Lambda sends to your function. For general information about how event filtering works, see the section called “Event filtering”.

This section focuses on event filtering for Amazon MQ event sources.

> **Note**
>
> Amazon MQ event source mappings only support filtering on the `data` key.

Topics

- Amazon MQ event filtering basics

Amazon MQ event filtering basics

Suppose your Amazon MQ message queue contains messages either in valid JSON format or as plain strings. An example record would look like the following, with the data converted to a Base64 encoded string in the `data` field.

ActiveMQ

```
{
```

```
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/text-message",
    "deliveryMode": 1,
    "replyTo": null,
    "type": null,
    "expiration": "60000",
    "priority": 1,
    "correlationId": "myJMSCoID",
    "redelivered": false,
    "destination": {
      "physicalName": "testQueue"
    },
    "data":"QUJDOkFBQUE=",
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959,
    "properties": {
      "index": "1",
      "doAlarm": "false",
      "myCustomProperty": "value"
    }
}
```

RabbitMQ

```
{
    "basicProperties": {
        "contentType": "text/plain",
        "contentEncoding": null,
        "headers": {
            "header1": {
                "bytes": [
                  118,
                  97,
                  108,
                  117,
                  101,
                  49
                ]
            },
            "header2": {
                "bytes": [
```

```
                    118,
                    97,
                    108,
                    117,
                    101,
                    50
                ]
            },
            "numberInHeader": 10
        },
        "deliveryMode": 1,
        "priority": 34,
        "correlationId": null,
        "replyTo": null,
        "expiration": "60000",
        "messageId": null,
        "timestamp": "Jan 1, 1970, 12:33:41 AM",
        "type": null,
        "userId": "AIDACKCEVSQ6C2EXAMPLE",
        "appId": null,
        "clusterId": null,
        "bodySize": 80
        },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
}
```

For both Active MQ and Rabbit MQ brokers, you can use event filtering to filter records using the `data` key. Suppose your Amazon MQ queue contains messages in the following JSON format.

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

To filter only those records where the `timeout` field is greater than 0, the `FilterCriteria` object would be as follows.

```
{
    "Filters": [
        {
```

```
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
        }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

to add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

With Amazon MQ, you can also filter records where the message is a plain string. Suppose you want to process only records where the message begins with "Result: ". The `FilterCriteria` object would look as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "data": [
        {
        "prefix": "Result: "
        }
    ]
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Amazon MQ messages must be UTF-8 encoded strings, either plain strings or in JSON format. That's because Lambda decodes Amazon MQ byte arrays into UTF-8 before applying filter criteria. If your messages use another encoding, such as UTF-16 or ASCII, or if the message format doesn't match the `FilterCriteria` format, Lambda processes metadata filters only. The following table summarizes the specific behavior:

Incoming message format	Filter pattern format for message properties	Resulting action
Plain string	Plain string	Lambda filters based on your filter criteria.
Plain string	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Plain string	Valid JSON	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Plain string	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.
Non-UTF-8 encoded string	JSON, plain string, or no pattern	Lambda filters (on the other metadata properties only) based on your filter criteria.

Troubleshoot Amazon MQ event source mapping errors

When a Lambda function encounters an unrecoverable error, your Amazon MQ consumer stops processing records. Any other consumers can continue processing, provided that they do not encounter the same error. To determine the potential cause of a stopped consumer, check the `StateTransitionReason` field in the return details of your `EventSourceMapping` for one of the following codes:

`ESM_CONFIG_NOT_VALID`

The event source mapping configuration is not valid.

`EVENT_SOURCE_AUTHN_ERROR`

Lambda failed to authenticate the event source.

`EVENT_SOURCE_AUTHZ_ERROR`

Lambda does not have the required permissions to access the event source.

`FUNCTION_CONFIG_NOT_VALID`

The function's configuration is not valid.

Records also go unprocessed if Lambda drops them due to their size. The size limit for Lambda records is 6 MB. To redeliver messages upon function error, you can use a dead-letter queue (DLQ). For more information, see Message Redelivery and DLQ Handling on the Apache ActiveMQ website and Reliability Guide on the RabbitMQ website.

> **Note**
>
> Lambda does not support custom redelivery policies. Instead, Lambda uses a policy with the default values from the Redelivery Policy page on the Apache ActiveMQ website, with `maximumRedeliveries` set to 6.

Using AWS Lambda with Amazon RDS

You can connect a Lambda function to an Amazon Relational Database Service (Amazon RDS) database directly and through an Amazon RDS Proxy. Direct connections are useful in simple scenarios, and proxies are recommended for production. A database proxy manages a pool of shared database connections which enables your function to reach high concurrency levels without exhausting database connections.

We recommend using Amazon RDS Proxy for Lambda functions that make frequent short database connections, or open and close large numbers of database connections. For more information, see Automatically connecting a Lambda function and a DB instance in the Amazon Relational Database Service Developer Guide.

Tip

To quickly connect a Lambda function to an Amazon RDS database, you can use the in-console guided wizard. To open the wizard, do the following:

1. Open the Functions page of the Lambda console.
2. Select the function you want to connect a database to.
3. On the **Configuration** tab, select **RDS databases**.
4. Choose **Connect to RDS database**.

After you've connected your function to a database, you can create a proxy by choosing **Add proxy**.

Configuring your function to work with RDS resources

In the Lambda console, you can provision, and configure, Amazon RDS database instances and proxy resources. You can do this by navigating to **RDS databases** under the **Configuration** tab. Alternatively, you can also create and configure connections to Lambda functions in the Amazon RDS console. When configuring an RDS database instance to use with Lambda, note the following criteria:

- To connect to a database, your function must be in the same Amazon VPC where your database runs.

- You can use Amazon RDS databases with MySQL, MariaDB, PostgreSQL, or Microsoft SQL Server engines.
- You can also use Aurora DB clusters with MySQL or PostgreSQL engines.
- You need to provide a Secrets Manager secret for database authentication.
- An IAM role must provide permission to use the secret, and a trust policy must allow Amazon RDS to assume the role.
- The IAM principal that uses the console to configure the Amazon RDS resource, and connect it to your function must have the following permissions:

Example permissions policy

> **Note**
>
> You need the Amazon RDS Proxy permissions only if you configure an Amazon RDS Proxy to manage a pool of your database connections.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateSecurityGroup",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:AuthorizeSecurityGroupEgress",
        "ec2:RevokeSecurityGroupEgress",
        "ec2:CreateNetworkInterface",
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces"
      ],
      "Resource": "*"
    },
    {
```

```
      "Effect": "Allow",
      "Action": [
        "rds-db:connect",
        "rds:CreateDBProxy",
        "rds:CreateDBInstance",
        "rds:CreateDBSubnetGroup",
        "rds:DescribeDBClusters",
        "rds:DescribeDBInstances",
        "rds:DescribeDBSubnetGroups",
        "rds:DescribeDBProxies",
        "rds:DescribeDBProxyTargets",
        "rds:DescribeDBProxyTargetGroups",
        "rds:RegisterDBProxyTargets",
        "rds:ModifyDBInstance",
        "rds:ModifyDBProxy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:ListFunctions",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:AttachRolePolicy",
        "iam:CreateRole",
        "iam:CreatePolicy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetResourcePolicy",
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret",
        "secretsmanager:ListSecretVersionIds",
        "secretsmanager:CreateSecret"
```

```
      ],
      "Resource": "*"
    }
  ]
}
```

Amazon RDS charges an hourly rate for proxies based on the database instance size, see RDS Proxy pricing for details. For more information on proxy connections in general, see Using Amazon RDS Proxy in the Amazon RDS User Guide.

SSL/TLS requirements for Amazon RDS connections

To make secure SSL/TLS connections to an Amazon RDS database instance, your Lambda function must verify the database server's identity using a trusted certificate. Lambda handles these certificates differently depending on your deployment package type:

- .zip file archives: Certificate handling varies by runtime:
  - **Node.js 18 and earlier**: Lambda automatically includes CA certificates and RDS certificates.
  - **Node.js 20 and later**: Lambda no longer loads additional CA certificates by default. Set the `NODE_EXTRA_CA_CERTS` environment variable to `/var/runtime/ca-cert.pem`.

  It might take up to 4 weeks for Amazon RDS certificates for new AWS Regions to be added to the Lambda managed runtimes.
- Container images: AWS base images include only CA certificates. If your function connects to an Amazon RDS database instance, you must include the appropriate certificates in your container image. In your Dockerfile, download the certificate bundle that corresponds with the AWS Region where you host your database. Example:

```
RUN curl https://truststore.pki.rds.amazonaws.com/us-east-1/us-east-1-bundle.pem -o /
us-east-1-bundle.pem
```

This command downloads the Amazon RDS certificate bundle and saves it at the absolute path `/us-east-1-bundle.pem` in your container's root directory. When configuring the database connection in your function code, you must reference this exact path. Example:

Node.js

The `readFileSync` function is required because Node.js database clients need the actual certificate content in memory, not just the path to the certificate file. Without `readFileSync`, the client interprets the path string as certificate content, resulting in a "self-signed certificate in certificate chain" error.

Example Node.js connection config for OCI function

```
import { readFileSync } from 'fs';

// ...

let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: {
        ca: readFileSync('/us-east-1-bundle.pem') // Load RDS certificate content
 from file into memory
    }
};
```

Python

Example Python connection config for OCI function

```
connection = pymysql.connect(
    host=proxy_host_name,
    user=db_username,
    password=token,
    db=db_name,
    port=port,
    ssl={'ca': '/us-east-1-bundle.pem'}  #Path to the certificate in container
)
```

Java

For Java functions using JDBC connections, the connection string must include:

- `useSSL=true`

- `requireSSL=true`
- An `sslCA` parameter that points to the location of the Amazon RDS certificate in the container image

Example Java connection string for OCI function

```
// Define connection string
String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true&sslCA=/us-east-1-bundle.pem", // Path to the certificate
 in container
        System.getenv("ProxyHostName"),
        System.getenv("Port"),
        System.getenv("DBName"));
```

.NET

Example.NET connection string for MySQL connection in OCI function

```
/// Build the Connection String with the Token
string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +
                         $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

 $"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                         $"Pwd={authToken};" +
                         "SslMode=Required;" +
                         "SslCa=/us-east-1-bundle.pem";  // Path to the certificate
 in container
```

Go

For Go functions using MySQL connections, load the Amazon RDS certificate into a certificate pool and register it with the MySQL driver. The connection string must then reference this configuration using the `tls` parameter.

Example Go code for MySQL connection in OCI function

```
import (
    "crypto/tls"
    "crypto/x509"
    "os"
```

```
    "github.com/go-sql-driver/mysql"
)

...

// Create certificate pool and register TLS config
rootCertPool := x509.NewCertPool()
pem, err := os.ReadFile("/us-east-1-bundle.pem")  // Path to the certificate in
 container
if err != nil {
    panic("failed to read certificate file: " + err.Error())
}
if ok := rootCertPool.AppendCertsFromPEM(pem); !ok {
    panic("failed to append PEM")
}

mysql.RegisterTLSConfig("custom", &tls.Config{
    RootCAs: rootCertPool,
})

dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?allowCleartextPasswords=true&tls=custom",
    dbUser, authenticationToken, dbEndpoint, dbName,
)
```

Ruby

Example Ruby connection config for OCI function

```
conn = Mysql2::Client.new(
    host: endpoint,
    username: user,
    password: token,
    port: port,
    database: db_name,
    sslca: '/us-east-1-bundle.pem',  # Path to the certificate in container
    sslverify: true
)
```

Connecting to an Amazon RDS database in a Lambda function

The following code examples shows how to implement a Lambda function that connects to an Amazon RDS database. The function makes a simple database request and returns the result.

> **Note**
>
> These code examples are valid for .zip deployment packages only. If you're deploying your function using a container image, you must specify the Amazon RDS certificate file in your function code, as explained in the preceding section.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using .NET.

```
using System.Data;
using System.Text.Json;
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Core;
using MySql.Data.MySqlClient;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace aws_rds;

public class InputModel
{
    public string key1 { get; set; }
    public string key2 { get; set; }
}

public class Function
{
    /// <summary>
```

```
    // Handles the Lambda function execution for connecting to RDS using IAM
 authentication.
    /// </summary>
    /// <param name="input">The input event data passed to the Lambda function</
param>
    /// <param name="context">The Lambda execution context that provides runtime
 information</param>
    /// <returns>A response object containing the execution result</returns>

    public async Task<APIGatewayProxyResponse>
 FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
    {
        // Sample Input: {"body": "{\"key1\":\"20\", \"key2\":\"25\"}"}
        var input = JsonSerializer.Deserialize<InputModel>(request.Body);

        /// Obtain authentication token
        var authToken = RDSAuthTokenGenerator.GenerateAuthToken(
            Environment.GetEnvironmentVariable("RDS_ENDPOINT"),
            Convert.ToInt32(Environment.GetEnvironmentVariable("RDS_PORT")),
            Environment.GetEnvironmentVariable("RDS_USERNAME")
        );

        /// Build the Connection String with the Token
        string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +

 $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

 $"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                                  $"Pwd={authToken};";


        try
        {
            await using var connection = new MySqlConnection(connectionString);
            await connection.OpenAsync();

            const string sql = "SELECT @param1 + @param2 AS Sum";

            await using var command = new MySqlCommand(sql, connection);
            command.Parameters.AddWithValue("@param1", int.Parse(input.key1 ??
 "0"));
            command.Parameters.AddWithValue("@param2", int.Parse(input.key2 ??
 "0"));
```

```
            await using var reader = await command.ExecuteReaderAsync();
            if (await reader.ReadAsync())
            {
                int result = reader.GetInt32("Sum");

                //Sample Response: {"statusCode":200,"body":"{\"message\":\"The
 sum is: 45\"}","isBase64Encoded":false}
                return new APIGatewayProxyResponse
                {
                    StatusCode = 200,
                    Body = JsonSerializer.Serialize(new { message = $"The sum is:
 {result}" })
                };
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        return new APIGatewayProxyResponse
        {
            StatusCode = 500,
            Body = JsonSerializer.Serialize(new { error = "Internal server
 error" })
        };
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Go.

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }
```

```
 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();

        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
```

```
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }

        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
```

```
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
```

```
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Connecting to an Amazon RDS database in a Lambda function using TypeScript.

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
```

```
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using PHP.

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
            'region' => getenv('AWS_REGION'),
        ];

        // Create RDS AuthTokenGenerator object
        $generator = new
 AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();
```

```
        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

 "mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }

    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}
```

```
$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Python.

```
import json
import os
import boto3
import pymysql

# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
```

```
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
occurs
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Ruby.

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
```

```
port = ENV['Port']           # 3306
user = ENV['DBUser']
region = ENV['DBRegion']     # 'us-east-1'
db_name = ENV['DBName']

credentials = Aws::Credentials.new(
  ENV['AWS_ACCESS_KEY_ID'],
  ENV['AWS_SECRET_ACCESS_KEY'],
  ENV['AWS_SESSION_TOKEN']
)
rds_client = Aws::RDS::AuthTokenGenerator.new(
  region: region,
  credentials: credentials
)

token = rds_client.auth_token(
  endpoint: endpoint+ ':' + port,
  user_name: user,
  region: region
)

begin
  conn = Mysql2::Client.new(
    host: endpoint,
    username: user,
    password: token,
    port: port,
    database: db_name,
    sslca: '/var/task/global-bundle.pem',
    sslverify: true,
    enable_cleartext_plugin: true
  )
  a = 3
  b = 2
  result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
  puts result
  conn.close
  {
    statusCode: 200,
    body: result.to_json
  }
rescue => e
  puts "Database connection failed due to #{e}"
end
```

```
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Rust.

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
```

```
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
        db_user = db_username
    );

    let signable_request =
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}
```

```
async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

Processing event notifications from Amazon RDS

You can use Lambda to process event notifications from an Amazon RDS database. Amazon RDS sends notifications to an Amazon Simple Notification Service (Amazon SNS) topic, which you can configure to invoke a Lambda function. Amazon SNS wraps the message from Amazon RDS in its own event document and sends it to your function.

For more information about configuring an Amazon RDS database to send notifications, see Using Amazon RDS event notifications.

Example Amazon RDS message in an Amazon SNS event

```
{
        "Records": [
          {
            "EventVersion": "1.0",
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:rds-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
            "EventSource": "aws:sns",
            "Sns": {
              "SignatureVersion": "1",
              "Timestamp": "2023-01-02T12:45:07.000Z",
              "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi
+tE/1+82j...65r==",
              "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
              "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
              "Message": "{\"Event Source\":\"db-instance\",\"Event Time\":\"2023-01-02
 12:45:06.000\",\"Identifier Link\":\"https://console.aws.amazon.com/rds/home?
region=eu-west-1#dbinstance:id=dbinstanceid\",\"Source ID\":\"dbinstanceid\",\"Event ID
\":\"http://docs.amazonwebservices.com/AmazonRDS/latest/UserGuide/USER_Events.html#RDS-
EVENT-0002\",\"Event Message\":\"Finished DB Instance backup\"}",
              "MessageAttributes": {},
              "Type": "Notification",
              "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-2:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
              "TopicArn":"arn:aws:sns:us-east-2:123456789012:sns-lambda",
              "Subject": "RDS Notification Message"
            }
          }
        ]
```

```
    }
```

Complete Lambda and Amazon RDS tutorial

- Using a Lambda function to access an Amazon RDS database – From the Amazon RDS User Guide, learn how to use a Lambda function to write data to an Amazon RDS database through an Amazon RDS Proxy. Your Lambda function will read records from an Amazon SQS queue and write new items to a table in your database whenever a message is added.

Select a database service for your Lambda-based applications

Many serverless applications need to store and retrieve data. AWS offers multiple database options that work with Lambda functions. Two of the most popular choices are Amazon DynamoDB, a NoSQL database service, and Amazon RDS, a traditional relational database solution. The following sections explain the key differences between these services when using them with Lambda and help you select the right database service for your serverless application.

To learn more about the other database services offered by AWS, and to understand their use cases and tradeoffs more generally, see Choosing an AWS database service. All of the AWS database services are compatible with Lambda, but not all of them may be suited to your particular use case.

What are your choices when selecting a database service with Lambda?

AWS offers multiple database services. For serverless applications, two of the most popular choices are DynamoDB and Amazon RDS.

- **DynamoDB** is a fully managed NoSQL database service optimized for serverless applications. It provides seamless scaling and consistent single-digit millisecond performance at any scale.
- **Amazon RDS** is a managed relational database service that supports multiple database engines including MySQL and PostgreSQL. It provides familiar SQL capabilities with managed infrastructure.

Recommendations if you already know your requirements

If you're already clear on your requirements, here are our basic recommendations:

We recommend DynamoDB for serverless applications that need consistent low-latency performance, automatic scaling, and don't require complex joins or transactions. It's particularly well-suited for Lambda-based applications due to its serverless nature.

Amazon RDS is a better choice when you need complex SQL queries, joins, or have existing applications using relational databases. However, be aware that connecting Lambda functions to Amazon RDS requires additional configuration and can impact cold start times.

What to consider when selecting a database service

When selecting between DynamoDB and Amazon RDS for your Lambda applications, consider these factors:

- Connection management and cold starts
- Data access patterns
- Query complexity
- Data consistency requirements
- Scaling characteristics
- Cost model

By understanding these factors, you can select the option that best meets the needs of your particular use case.

Connection management and cold starts

- DynamoDB uses an HTTP API for all operations. Lambda functions can make immediate requests without maintaining connections, resulting in better cold start performance. Each request is authenticated using AWS credentials without connection overhead.
- Amazon RDS requires managing connection pools since it uses traditional database connections. This can impact cold starts as new Lambda instances need to establish connections. You'll need to implement connection pooling strategies and potentially use Amazon RDS Proxy to manage connections effectively. Note that using Amazon RDS Proxy incurs additional costs.

Data access patterns

- DynamoDB works best with known access patterns and single-table designs. It's ideal for Lambda applications that need consistent low-latency access to data based on primary keys or secondary indexes.
- Amazon RDS provides flexibility for complex queries and changing access patterns. It's better suited when your Lambda functions need to perform unique, tailored queries or complex joins across multiple tables.

Query complexity

- DynamoDB excels at simple, key-based operations and predefined access patterns. Complex queries must be designed around index structures, and joins must be handled in application code.
- Amazon RDS supports complex SQL queries with joins, subqueries, and aggregations. This can simplify your Lambda function code when complex data operations are needed.

Data consistency requirements

- DynamoDB offers both eventual and strong consistency options, with strong consistency available for single-item reads. Transactions are supported but with some limitations.
- Amazon RDS provides full atomicity, consistency, isolation, and durability (ACID) compliance and complex transaction support. If your Lambda functions require complex transactions or strong consistency across multiple records, Amazon RDS might be more suitable.

Scaling characteristics

- DynamoDB scales automatically with your workload. It can handle sudden spikes in traffic from Lambda functions without pre-provisioning. You can use on-demand capacity mode to pay only for what you use, perfectly matching Lambda's scaling model.
- Amazon RDS has fixed capacity based on the instance size you choose. If multiple Lambda functions try to connect simultaneously, you may exceed your connection quota. You need to carefully manage connection pools and potentially implement retry logic.

Cost model

- DynamoDB's pricing aligns well with serverless applications. With on-demand capacity, you pay only for the actual reads and writes performed by your Lambda functions. There are no charges for idle time.
- Amazon RDS charges for the running instance regardless of usage. This can be less cost-effective for sporadic workloads that can be typical in serverless applications. However, it might be more economical for high-throughput workloads with consistent usage.

Getting started with your chosen database service

Now that you've read about the criteria for selecting between DynamoDB and Amazon RDS and the key differences between them, you can select the option that best matches your needs and use the following resources to get started using it.

DynamoDB

Get started with DynamoDB with the following resources

- For an introduction to the DynamoDB service, read What is DynamoDB? in the *Amazon DynamoDB Developer Guide*.
- Follow the tutorial Using Lambda with API Gateway to see an example of using a Lambda function to perform CRUD operations on a DynamoDB table in response to an API request.
- Read Programming with DynamoDB and the AWS SDKs in the *Amazon DynamoDB Developer Guide* to learn more about how to access DynamoDB from within your Lambda function by using one of the AWS SDKs.

Amazon RDS

Get started with Amazon RDS with the following resources

- For an introduction to the Amazon RDS service, read What is Amazon Relational Database Service (Amazon RDS)? in the *Amazon Relational Database Service User Guide*.
- Follow the tutorial Using a Lambda function to access an Amazon RDS database in the *Amazon Relational Database Service User Guide*.
- Learn more about using Lambda with Amazon RDS by reading the section called “RDS”.

Process Amazon S3 event notifications with Lambda

You can use Lambda to process event notifications from Amazon Simple Storage Service. Amazon S3 can send an event to a Lambda function when an object is created or deleted. You configure notification settings on a bucket, and grant Amazon S3 permission to invoke a function on the function's resource-based permissions policy.

> **Warning**
>
> If your Lambda function uses the same bucket that triggers it, it could cause the function to run in a loop. For example, if the bucket triggers a function each time an object is uploaded, and the function uploads an object to the bucket, then the function indirectly triggers itself. To avoid this, use two buckets, or configure the trigger to only apply to a prefix used for incoming objects.

Amazon S3 invokes your function asynchronously with an event that contains details about the object. The following example shows an event that Amazon S3 sent when a deployment package was uploaded to Amazon S3.

Example Amazon S3 notification event

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-2",
      "eventTime": "2019-09-03T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "requestParameters": {
        "sourceIPAddress": "205.255.255.255"
      },
      "responseElements": {
        "x-amz-request-id": "D82B88E5F771F645",
        "x-amz-id-2":
 "vlR7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
      },
```

```
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMAI3E"
          },
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"
        },
        "object": {
          "key": "b21b84d653bb07b05b1e6b33684dc11b",
          "size": 1305107,
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",
          "sequencer": "0C0F6F405D6ED209E1"
        }
      }
    }
  ]
}
```

To invoke your function, Amazon S3 needs permission from the function's resource-based policy. When you configure an Amazon S3 trigger in the Lambda console, the console modifies the resource-based policy to allow Amazon S3 to invoke the function if the bucket name and account ID match. If you configure the notification in Amazon S3, you use the Lambda API to update the policy. You can also use the Lambda API to grant permission to another account, or restrict permission to a designated alias.

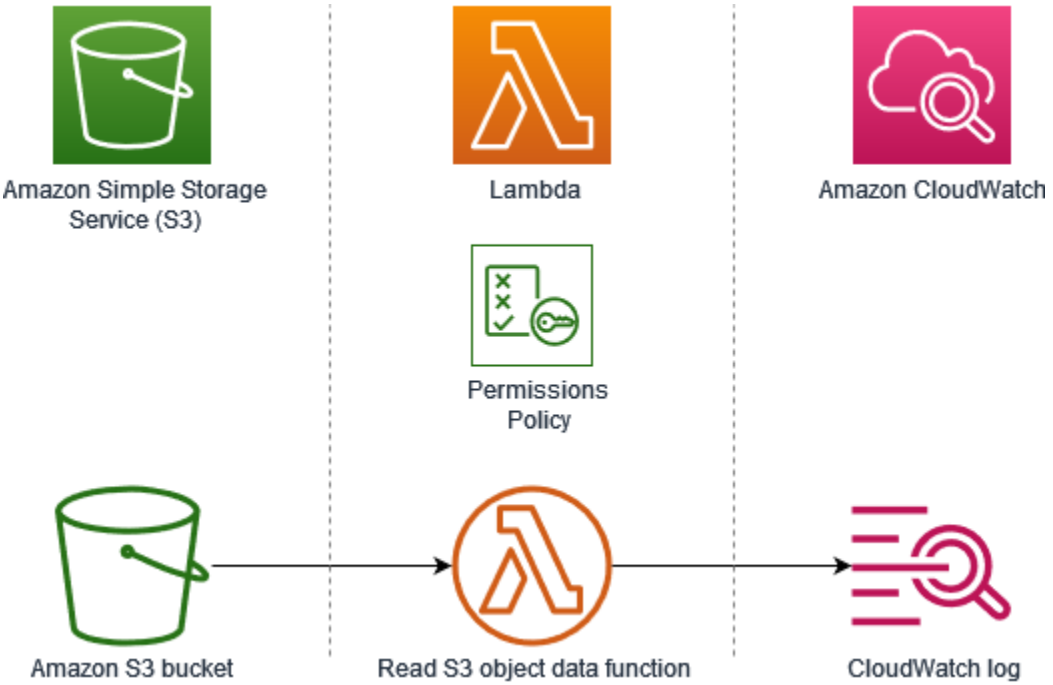
If your function uses the AWS SDK to manage Amazon S3 resources, it also needs Amazon S3 permissions in its execution role.

Topics

- Tutorial: Using an Amazon S3 trigger to invoke a Lambda function
- Tutorial: Using an Amazon S3 trigger to create thumbnail images

Tutorial: Using an Amazon S3 trigger to invoke a Lambda function

In this tutorial, you use the console to create a Lambda function and configure a trigger for an Amazon Simple Storage Service (Amazon S3) bucket. Every time that you add an object to your Amazon S3 bucket, your function runs and outputs the object type to Amazon CloudWatch Logs.

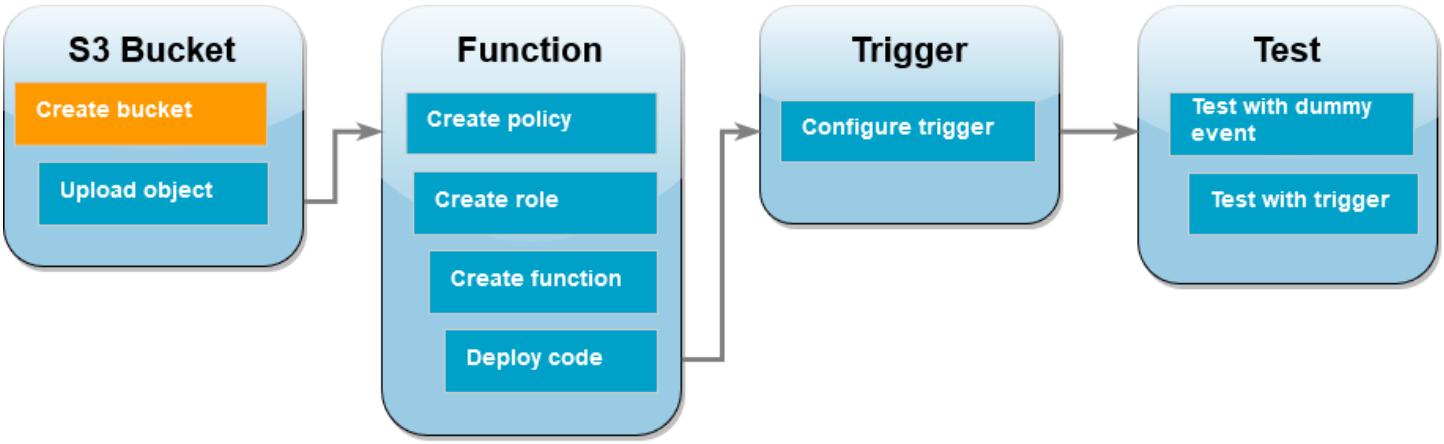


This tutorial demonstrates how to:

1. Create an Amazon S3 bucket.
2. Create a Lambda function that returns the object type of objects in an Amazon S3 bucket.
3. Configure a Lambda trigger that invokes your function when objects are uploaded to your bucket.
4. Test your function, first with a dummy event, and then using the trigger.

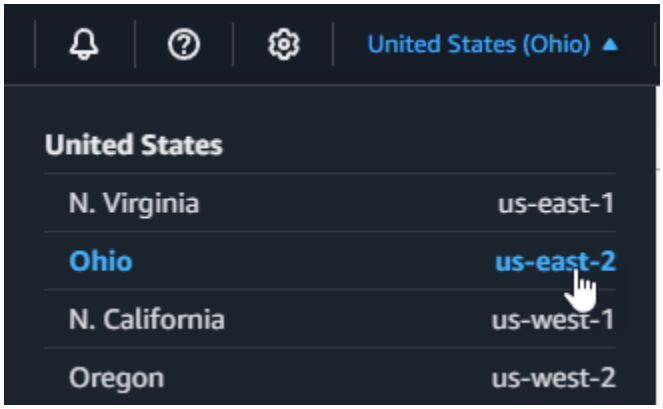
By completing these steps, you'll learn how to configure a Lambda function to run whenever objects are added to or deleted from an Amazon S3 bucket. You can complete this tutorial using only the AWS Management Console.

Create an Amazon S3 bucket



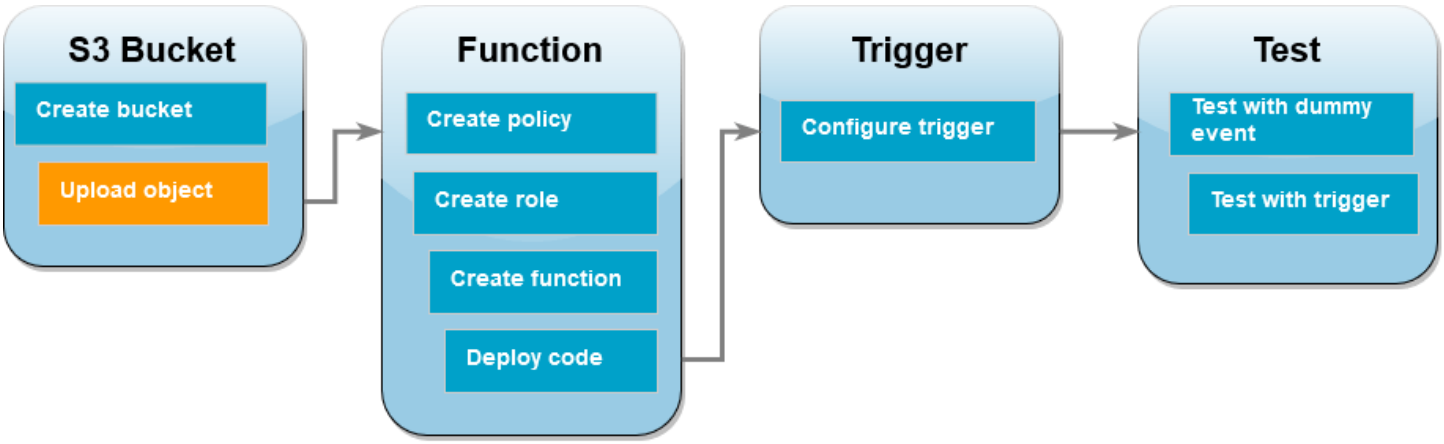
To create an Amazon S3 bucket

1. Open the Amazon S3 console and select the **General purpose buckets** page.
2. Select the AWS Region closest to your geographical location. You can change your region using the drop-down list at the top of the screen. Later in the tutorial, you must create your Lambda function in the same Region.



3. Choose **Create bucket**.
4. Under **General configuration**, do the following:
   a. For **Bucket type**, ensure **General purpose** is selected.
   b. For **Bucket name**, enter a globally unique name that meets the Amazon S3 Bucket naming rules. Bucket names can contain only lower case letters, numbers, dots (.), and hyphens (-).
5. Leave all other options set to their default values and choose **Create bucket**.

Upload a test object to your bucket

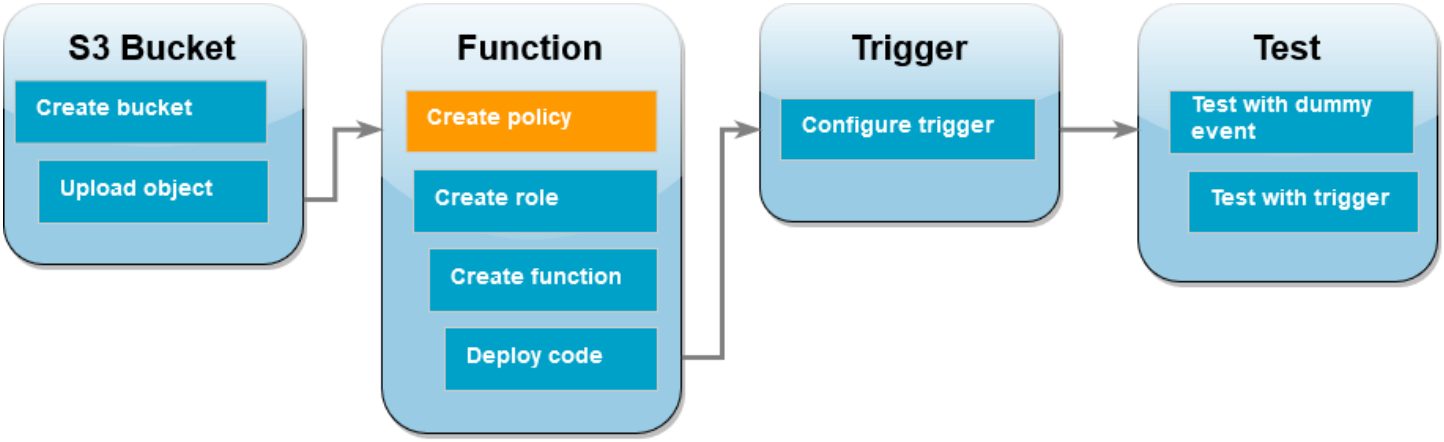


To upload a test object

1. Open the Buckets page of the Amazon S3 console and choose the bucket you created during the previous step.
2. Choose **Upload**.
3. Choose **Add files** and select the object that you want to upload. You can select any file (for example, `HappyFace.jpg`).
4. Choose **Open**, then choose **Upload**.

Later in the tutorial, you'll test your Lambda function using this object.

Create a permissions policy



Create a permissions policy that allows Lambda to get objects from an Amazon S3 bucket and to write to Amazon CloudWatch Logs.

To create the policy

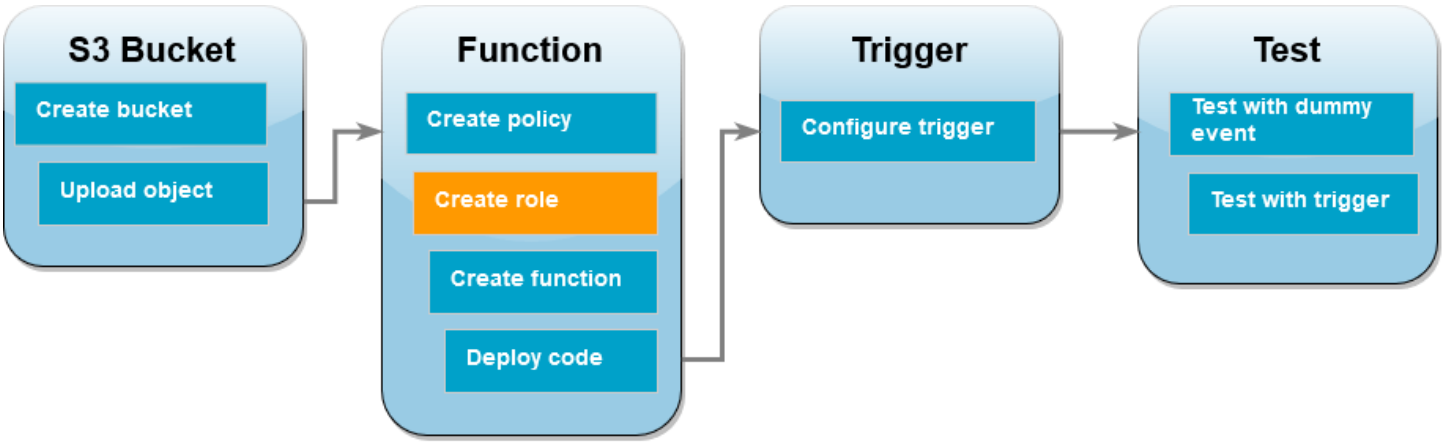
1. Open the Policies page of the IAM console.
2. Choose **Create Policy**.
3. Choose the **JSON** tab, and then paste the following custom policy into the JSON editor.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. Choose **Next: Tags**.
5. Choose **Next: Review**.
6. Under **Review policy**, for the policy **Name**, enter **`s3-trigger-tutorial`**.
7. Choose **Create policy**.

Create an execution role

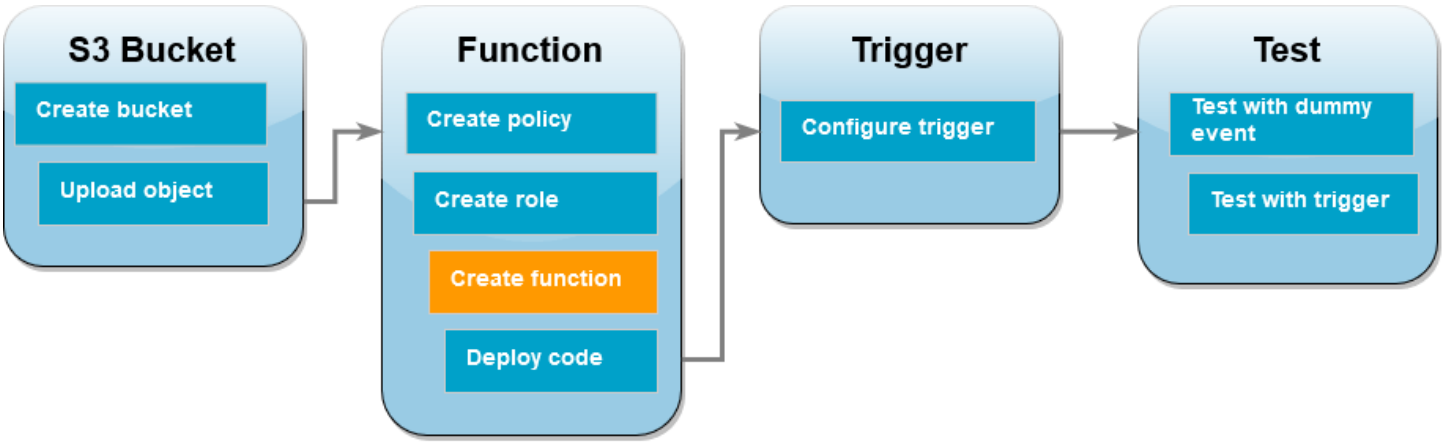


An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and resources. In this step, create an execution role using the permissions policy that you created in the previous step.

To create an execution role and attach your custom permissions policy

1. Open the Roles page of the IAM console.
2. Choose **Create role**.
3. For the type of trusted entity, choose **AWS service**, then for the use case, choose **Lambda**.
4. Choose **Next**.
5. In the policy search box, enter **`s3-trigger-tutorial`**.
6. In the search results, select the policy that you created (`s3-trigger-tutorial`), and then choose **Next**.
7. Under **Role details**, for the **Role name**, enter **`lambda-s3-trigger-role`**, then choose **Create role**.

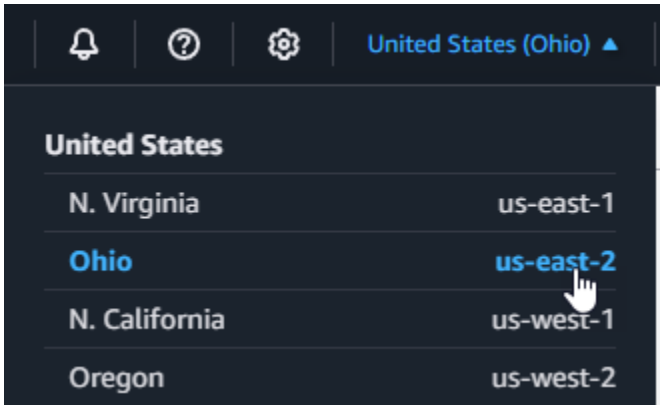
Create the Lambda function



Create a Lambda function in the console using the Python 3.14 runtime.

To create the Lambda function

1. Open the Functions page of the Lambda console.
2. Make sure you're working in the same AWS Region you created your Amazon S3 bucket in. You can change your Region using the drop-down list at the top of the screen.



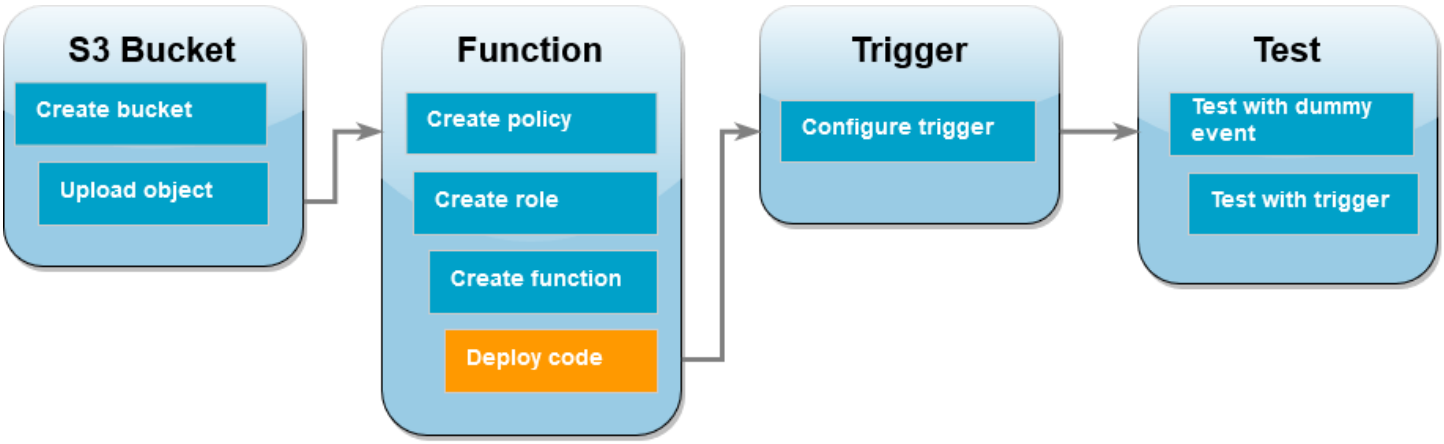
3. Choose **Create function**.
4. Choose **Author from scratch**
5. Under **Basic information**, do the following:
   a. For **Function name**, enter `s3-trigger-tutorial`
   b. For **Runtime**, choose **Python 3.14**.
   c. For **Architecture**, choose **x86_64**.
6. In the **Change default execution role** tab, do the following:

a. Expand the tab, then choose **Use an existing role**.

b. Select the `lambda-s3-trigger-role` you created earlier.

7. Choose **Create function**.

Deploy the function code



This tutorial uses the Python 3.14 runtime, but we've also provided example code files for other runtimes. You can select the tab in the following box to see the code for the runtime you're interested in.

The Lambda function retrieves the key name of the uploaded object and the name of the bucket from the `event` parameter it receives from Amazon S3. The function then uses the get_object method from the AWS SDK for Python (Boto3) to retrieve the object's metadata, including the content type (MIME type) of the uploaded object.

To deploy the function code

1. Choose the **Python** tab in the following box and copy the code.

   .NET

   SDK for .NET

   > **Note**
   >
   > There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext
 context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
```

```
                var key =
 HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket,
 key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
```

```
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNo

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger =
 LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
```

```
            HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

            logger.info("Successfully retrieved " + srcBucket + "/" + srcKey +
 " of type " + headObject.contentType());

            return "Ok";
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String
 bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using JavaScript.

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
```

```
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\
+/g, ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}.
 Make sure they exist and your bucket is in the same region as this
 function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

Consuming an S3 event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> =>
 {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
```

```
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using PHP.

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
```

```
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as
 this function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']
['key'], encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they
 exist and your bucket is in the same region as this function.'.format(key,
 bucket))
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Ruby.

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']
['key'], Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they
 exist and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
```

```
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request
 from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket =
 evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to
 exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object
 key to exist");

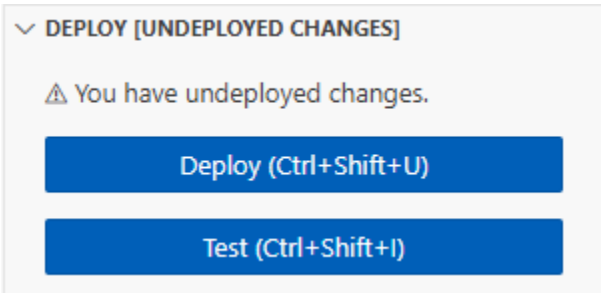
    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

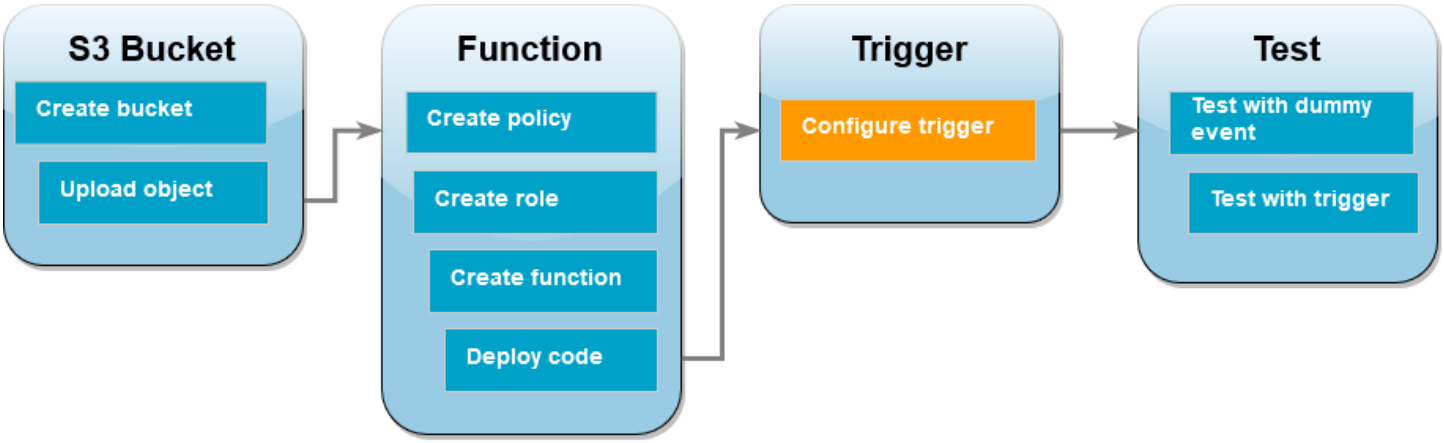
    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

2. In the **Code source** pane on the Lambda console, paste the code into the code editor, replacing the code that Lambda created.
3. In the **DEPLOY** section, choose **Deploy** to update your function's code:

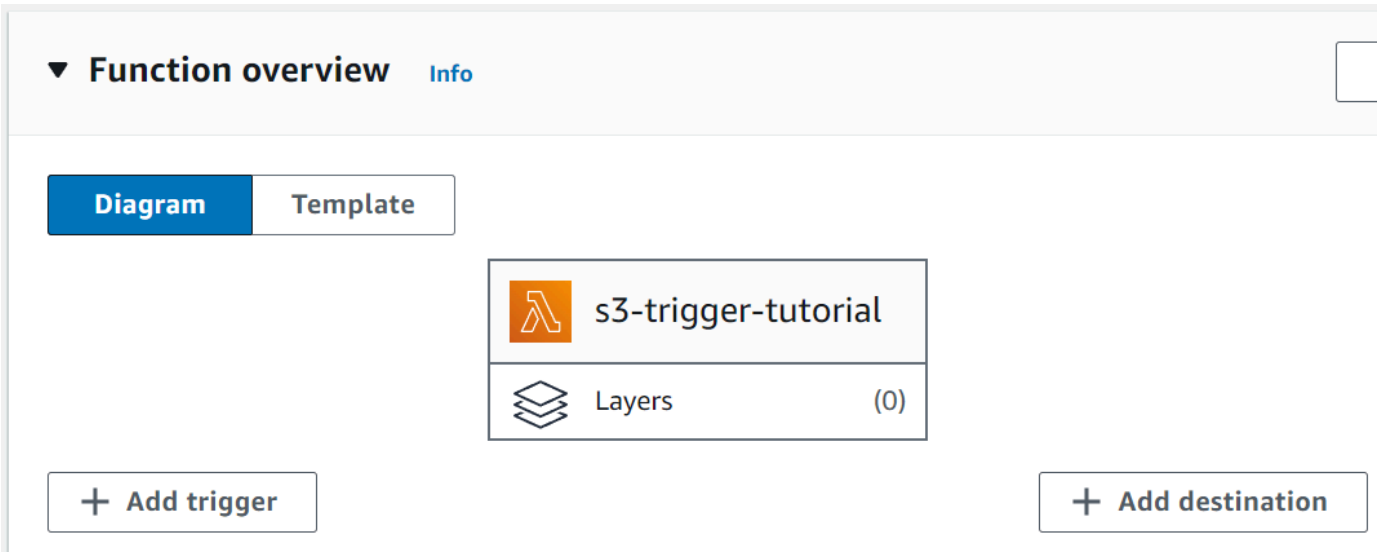


Create the Amazon S3 trigger



To create the Amazon S3 trigger

1. In the **Function overview** pane, choose **Add trigger**.

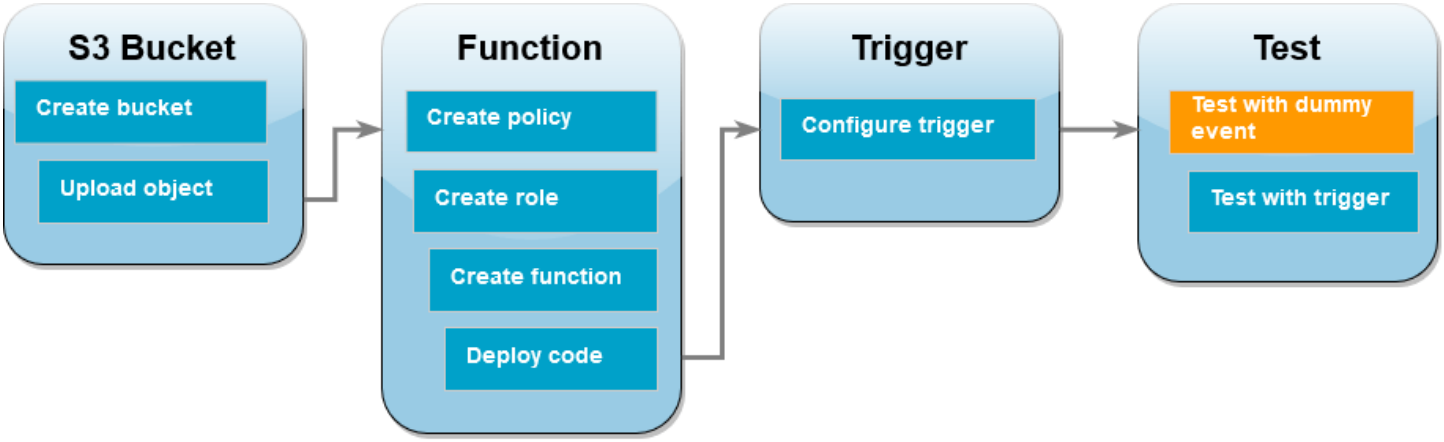


2. Select **S3**.
3. Under **Bucket**, select the bucket you created earlier in the tutorial.

4. Under **Event types**, be sure that **All object create events** is selected.
5. Under **Recursive invocation**, select the check box to acknowledge that using the same Amazon S3 bucket for input and output is not recommended.
6. Choose **Add**.

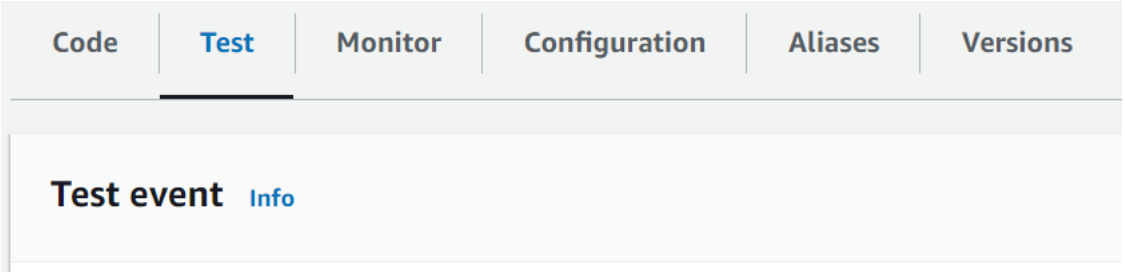
> **Note**
>
> When you create an Amazon S3 trigger for a Lambda function using the Lambda console, Amazon S3 configures an event notification on the bucket you specify. Before configuring this event notification, Amazon S3 performs a series of checks to confirm that the event destination exists and has the required IAM policies. Amazon S3 also performs these tests on any other event notifications configured for that bucket.
> Because of this check, if the bucket has previously configured event destinations for resources that no longer exist, or for resources that don't have the required permissions policies, Amazon S3 won't be able to create the new event notification. You'll see the following error message indicating that your trigger couldn't be created:
>
> ```
> An error occurred when creating the trigger: Unable to validate the following
>  destination configurations.
> ```
>
> You can see this error if you previously configured a trigger for another Lambda function using the same bucket, and you have since deleted the function or modified its permissions policies.

Test your Lambda function with a dummy event



To test the Lambda function with a dummy event

1. In the Lambda console page for your function, choose the **Test** tab.



2. For **Event name**, enter `MyTestEvent`.
3. In the **Event JSON**, paste the following test event. Be sure to replace these values:

   - Replace `us-east-1` with the region you created your Amazon S3 bucket in.
   - Replace both instances of `amzn-s3-demo-bucket` with the name of your own Amazon S3 bucket.
   - Replace `test%2FKey` with the name of the test object you uploaded to your bucket earlier (for example, `HappyFace.jpg`).

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
```

```
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

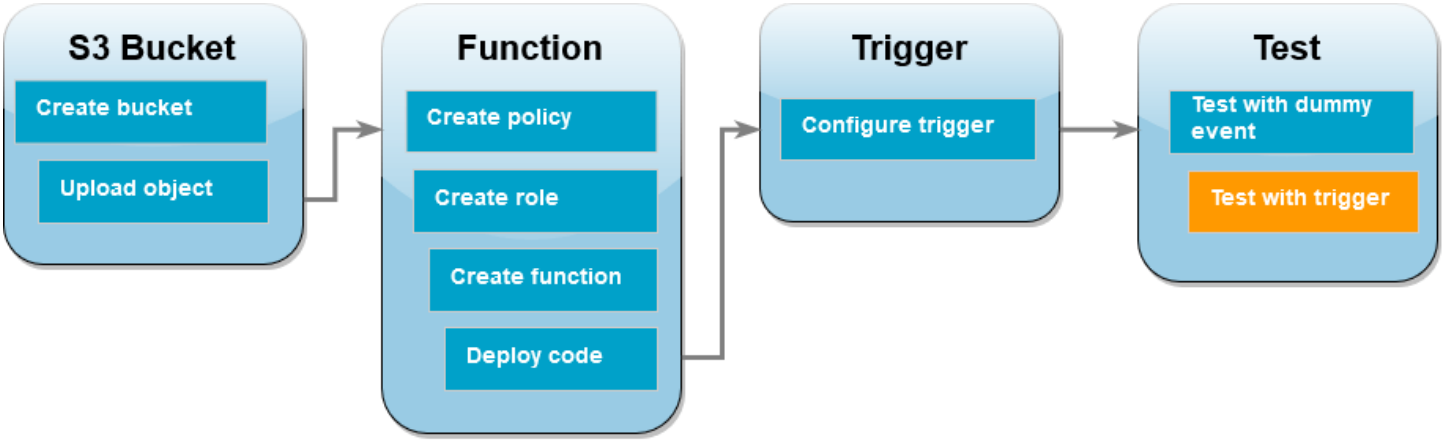
4. Choose **Save**.
5. Choose **Test**.
6. If your function runs successfully, you'll see output similar to the following in the **Execution results** tab.

```
Response
"image/jpeg"

Function Logs
START RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6 Version: $LATEST
2021-02-18T21:40:59.280Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    INPUT
 BUCKET AND KEY:  { Bucket: 'amzn-s3-demo-bucket', Key: 'HappyFace.jpg' }
2021-02-18T21:41:00.215Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    CONTENT
 TYPE: image/jpeg
END RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
REPORT RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6    Duration: 976.25 ms
 Billed Duration: 977 ms    Memory Size: 128 MB    Max Memory Used: 90 MB    Init
 Duration: 430.47 ms

Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

Test the Lambda function with the Amazon S3 trigger



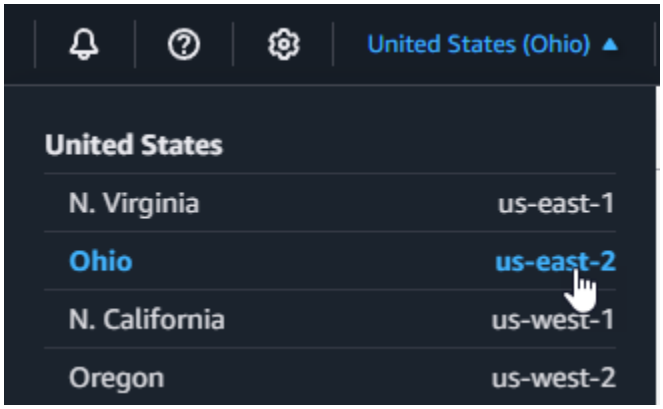
To test your function with the configured trigger, upload an object to your Amazon S3 bucket using the console. To verify that your Lambda function ran as expected, use CloudWatch Logs to view your function's output.

To upload an object to your Amazon S3 bucket

1. Open the Buckets page of the Amazon S3 console and choose the bucket that you created earlier.
2. Choose **Upload**.
3. Choose **Add files** and use the file selector to choose an object you want to upload. This object can be any file you choose.
4. Choose **Open**, then choose **Upload**.

To verify the function invocation using CloudWatch Logs

1. Open the CloudWatch console.
2. Make sure you're working in the same AWS Region you created your Lambda function in. You can change your Region using the drop-down list at the top of the screen.



3. Choose **Logs**, then choose **Log groups**.
4. Choose the log group for your function (`/aws/lambda/s3-trigger-tutorial`).
5. Under **Log streams**, choose the most recent log stream.
6. If your function was invoked correctly in response to your Amazon S3 trigger, you'll see output similar to the following. The `CONTENT TYPE` you see depends on the type of file you uploaded to your bucket.

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the execution role

1. Open the Roles page of the IAM console.

2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the S3 bucket

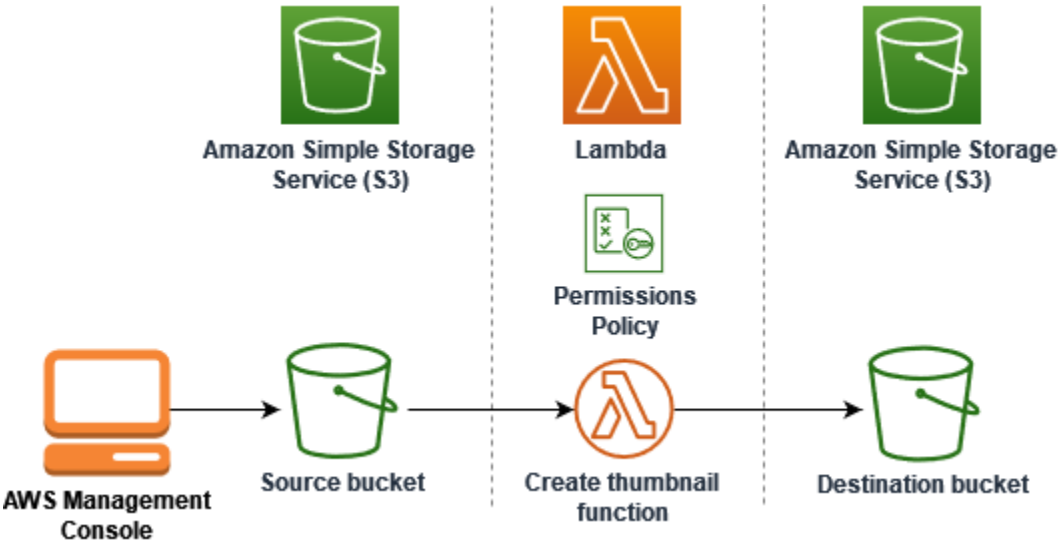
1. Open the Amazon S3 console.
2. Select the bucket you created.
3. Choose **Delete**.
4. Enter the name of the bucket in the text input field.
5. Choose **Delete bucket**.

Next steps

In Tutorial: Using an Amazon S3 trigger to create thumbnail images, the Amazon S3 trigger invokes a function that creates a thumbnail image for each image file that is uploaded to a bucket. This tutorial requires a moderate level of AWS and Lambda domain knowledge. It demonstrates how to create resources using the AWS Command Line Interface (AWS CLI) and how to create a .zip file archive deployment package for the function and its dependencies.

Tutorial: Using an Amazon S3 trigger to create thumbnail images

In this tutorial, you create and configure a Lambda function that resizes images added to an Amazon Simple Storage Service (Amazon S3) bucket. When you add an image file to your bucket, Amazon S3 invokes your Lambda function. The function then creates a thumbnail version of the image and outputs it to a different Amazon S3 bucket.



To complete this tutorial, you carry out the following steps:

1. Create source and destination Amazon S3 buckets and upload a sample image.
2. Create a Lambda function that resizes an image and outputs a thumbnail to an Amazon S3 bucket.
3. Configure a Lambda trigger that invokes your function when objects are uploaded to your source bucket.
4. Test your function, first with a dummy event, and then by uploading an image to your source bucket.

By completing these steps, you'll learn how to use Lambda to carry out a file processing task on objects added to an Amazon S3 bucket. You can complete this tutorial using the AWS Command Line Interface (AWS CLI) or the AWS Management Console.

If you're looking for a simpler example to learn how to configure an Amazon S3 trigger for Lambda, you can try Tutorial: Using an Amazon S3 trigger to invoke a Lambda function.

Topics

- Prerequisites
- Create two Amazon S3 buckets
- Upload a test image to your source bucket
- Create a permissions policy
- Create an execution role

- Create the function deployment package
- Create the Lambda function
- Configure Amazon S3 to invoke the function
- Test your Lambda function with a dummy event
- Test your function using the Amazon S3 trigger
- Clean up your resources

Prerequisites

If you want to use the AWS CLI to complete the tutorial, install the latest version of the AWS Command Line Interface.

For your Lambda function code, you can use Python or Node.js. Install the language support tools and a package manager for the language that you want to use.

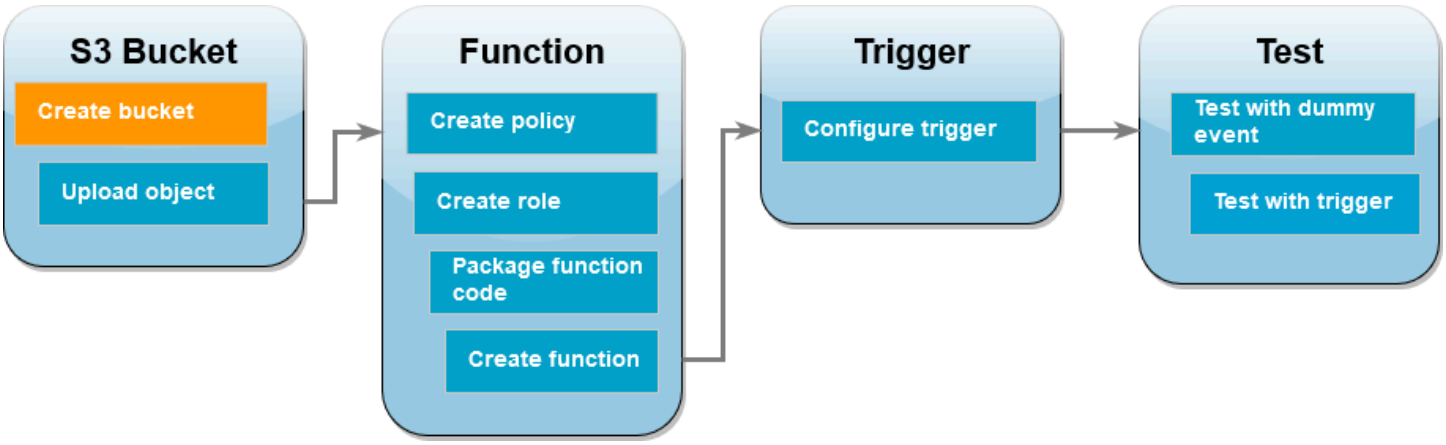
Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create two Amazon S3 buckets

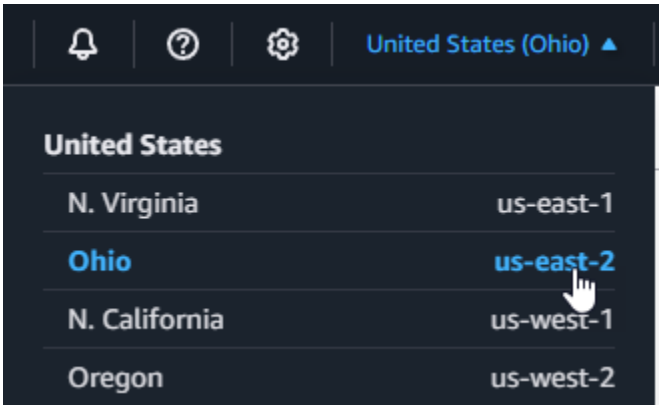


First create two Amazon S3 buckets. The first bucket is the source bucket you will upload your images to. The second bucket is used by Lambda to save the resized thumbnail when you invoke your function.

AWS Management Console

To create the Amazon S3 buckets (console)

1. Open the Amazon S3 console and select the **General purpose buckets** page.
2. Select the AWS Region closest to your geographical location. You can change your region using the drop-down list at the top of the screen. Later in the tutorial, you must create your Lambda function in the same Region.



3. Choose **Create bucket**.
4. Under **General configuration**, do the following:

   a. For **Bucket type**, ensure **General purpose** is selected.

b. For **Bucket name**, enter a globally unique name that meets the Amazon S3 Bucket naming rules. Bucket names can contain only lower case letters, numbers, dots (.), and hyphens (-).

5. Leave all other options set to their default values and choose **Create bucket**.
6. Repeat steps 1 to 5 to create your destination bucket. For **Bucket name**, enter **`amzn-s3-demo-source-bucket-resized`**, where **`amzn-s3-demo-source-bucket`** is the name of the source bucket you just created.

AWS CLI

To create the Amazon S3 buckets (AWS CLI)

1. Run the following CLI command to create your source bucket. The name you choose for your bucket must be globally unique and follow the Amazon S3 Bucket naming rules. Names can only contain lower case letters, numbers, dots (.), and hyphens (-). For `region` and `LocationConstraint`, choose the AWS Region closest to your geographical location.

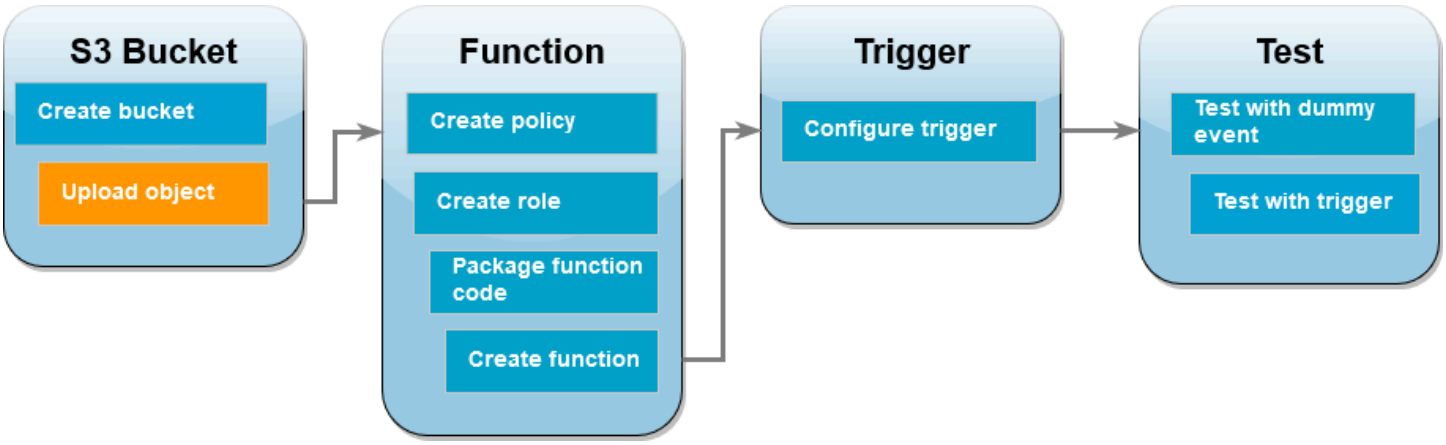
```
aws s3api create-bucket --bucket amzn-s3-demo-source-bucket --region us-east-1 \
--create-bucket-configuration LocationConstraint=us-east-1
```

   Later in the tutorial, you must create your Lambda function in the same AWS Region as your source bucket, so make a note of the region you chose.

2. Run the following command to create your destination bucket. For the bucket name, you must use **`amzn-s3-demo-source-bucket-resized`**, where **`amzn-s3-demo-source-bucket`** is the name of the source bucket you created in step 1. For `region` and `LocationConstraint`, choose the same AWS Region you used to create your source bucket.

```
aws s3api create-bucket --bucket amzn-s3-demo-source-bucket-resized --region us-
east-1 \
--create-bucket-configuration LocationConstraint=us-east-1
```

Upload a test image to your source bucket



Later in the tutorial, you'll test your Lambda function by invoking it using the AWS CLI or the Lambda console. To confirm that your function is operating correctly, your source bucket needs to contain a test image. This image can be any JPG or PNG file you choose.

AWS Management Console

To upload a test image to your source bucket (console)

1. Open the Buckets page of the Amazon S3 console.
2. Select the source bucket you created in the previous step.
3. Choose **Upload**.
4. Choose **Add files** and use the file selector to choose the object you want to upload.
5. Choose **Open**, then choose **Upload**.

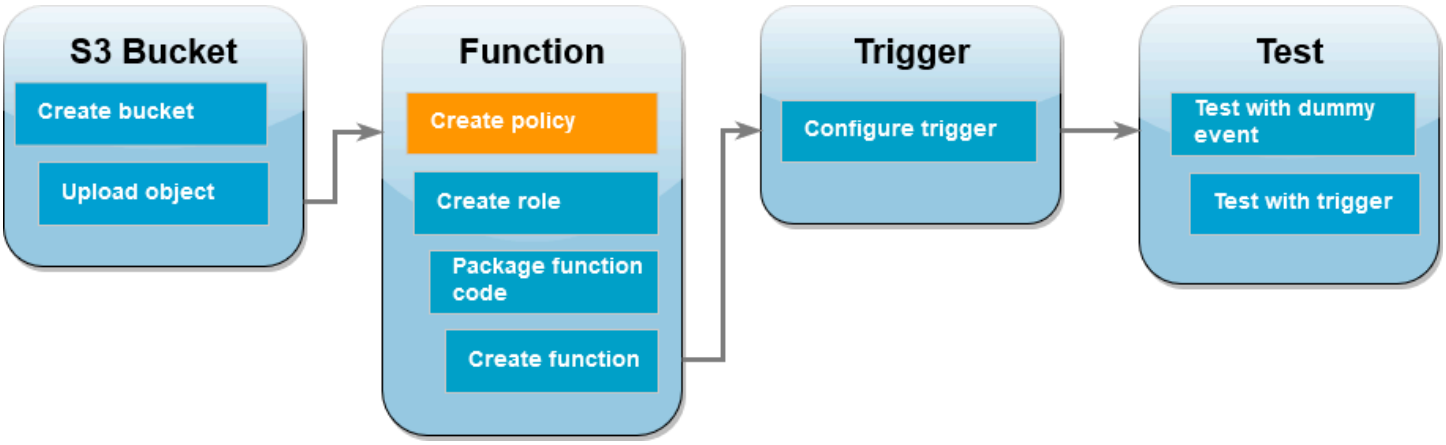
AWS CLI

To upload a test image to your source bucket (AWS CLI)

- From the directory containing the image you want to upload, run the following CLI command. Replace the `--bucket` parameter with the name of your source bucket. For the `--key` and `--body` parameters, use the filename of your test image.

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key HappyFace.jpg --
body ./HappyFace.jpg
```

Create a permissions policy



The first step in creating your Lambda function is to create a permissions policy. This policy gives your function the permissions it needs to access other AWS resources. For this tutorial, the policy gives Lambda read and write permissions for Amazon S3 buckets and allows it to write to Amazon CloudWatch Logs.

AWS Management Console

To create the policy (console)

1. Open the Policies page of the AWS Identity and Access Management (IAM) console.
2. Choose **Create policy**.
3. Choose the **JSON** tab, and then paste the following custom policy into the JSON editor.

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
```

```
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. Choose **Next**.
5. Under **Policy details**, for **Policy name**, enter **`LambdaS3Policy`**.
6. Choose **Create policy**.

AWS CLI

To create the policy (AWS CLI)

1. Save the following JSON in a file named `policy.json`.

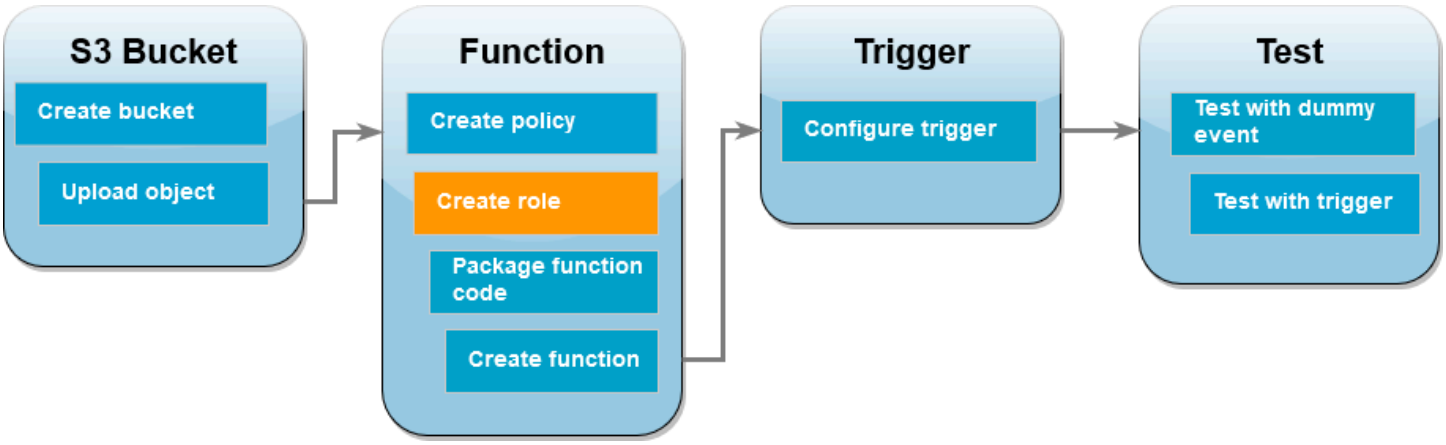
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
```

```
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. From the directory you saved the JSON policy document in, run the following CLI command.

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

Create an execution role



An execution role is an IAM role that grants a Lambda function permission to access AWS services and resources. To give your function read and write access to an Amazon S3 bucket, you attach the permissions policy you created in the previous step.

AWS Management Console

To create an execution role and attach your permissions policy (console)

1. Open the Roles page of the (IAM) console.
2. Choose **Create role**.

3. For **Trusted entity type**, select **AWS service**, and for **Use case**, select **Lambda**.
4. Choose **Next**.
5. Add the permissions policy you created in the previous step by doing the following:
   a. In the policy search box, enter **`LambdaS3Policy`**.
   b. In the search results, select the check box for `LambdaS3Policy`.
   c. Choose **Next**.
6. Under **Role details**, for the **Role name** enter **`LambdaS3Role`**.
7. Choose **Create role**.

AWS CLI

To create an execution role and attach your permissions policy (AWS CLI)

1. Save the following JSON in a file named `trust-policy.json`. This trust policy allows Lambda to use the role's permissions by giving the service principal `lambda.amazonaws.com` permission to call the AWS Security Token Service (AWS STS) `AssumeRole` action.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

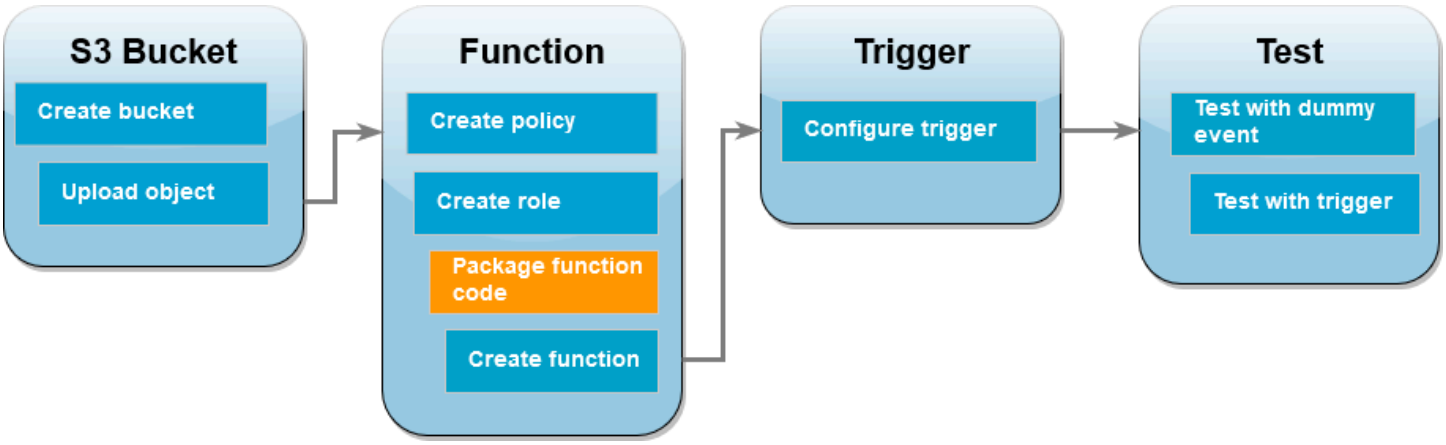
2. From the directory you saved the JSON trust policy document in, run the following CLI command to create the execution role.

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. To attach the permissions policy you created in the previous step, run the following CLI command. Replace the AWS account number in the policy's ARN with your own account number.

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

Create the function deployment package



To create your function, you create a *deployment package* containing your function code and its dependencies. For this `CreateThumbnail` function, your function code uses a separate library for the image resizing. Follow the instructions for your chosen language to create a deployment package containing the required library.

Node.js

To create the deployment package (Node.js)

1. Create a directory named `lambda-s3` for your function code and dependencies and navigate into it.

```
mkdir lambda-s3
cd lambda-s3
```

2. Create a new Node.js project with `npm`. To accept the default options provided in the interactive experience, press `Enter`.

```
npm init
```

3. Save the following function code in a file named `index.mjs`. Make sure to replace `us-east-1` with the AWS Region in which you created your own source and destination buckets.

```
// dependencies
import { S3Client, GetObjectCommand, PutObjectCommand } from '@aws-sdk/client-
s3';

import { Readable } from 'stream';

import sharp from 'sharp';
import util from 'util';


// create S3 client
const s3 = new S3Client({region: 'us-east-1'});

// define the handler function
export const handler = async (event, context) => {

// Read options from the event parameter and get the source bucket
console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
  const srcBucket = event.Records[0].s3.bucket.name;

// Object key may have spaces or unicode non-ASCII characters
const srcKey    = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, " "));
const dstBucket = srcBucket + "-resized";
const dstKey    = "resized-" + srcKey;

// Infer the image type from the file suffix
const typeMatch = srcKey.match(/\.([^.]*)$/);
if (!typeMatch) {
  console.log("Could not determine the image type.");
  return;
}

// Check that the image type is supported
const imageType = typeMatch[1].toLowerCase();
if (imageType != "jpg" && imageType != "png") {
  console.log(`Unsupported image type: ${imageType}`);
  return;
}
```

```
// Get the image from the source bucket. GetObjectCommand returns a stream.
try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var response = await s3.send(new GetObjectCommand(params));
  var stream = response.Body;

// Convert stream to buffer to pass to sharp resize function.
  if (stream instanceof Readable) {
    var content_buffer = Buffer.concat(await stream.toArray());

  } else {
    throw new Error('Unknown object stream type');
  }


} catch (error) {
  console.log(error);
  return;
}


// set thumbnail width. Resize will set the height automatically to maintain
 aspect ratio.
const width  = 200;

// Use the sharp module to resize the image and save in a buffer.
try {
  var output_buffer = await sharp(content_buffer).resize(width).toBuffer();

} catch (error) {
  console.log(error);
  return;
}

// Upload the thumbnail image to the destination bucket
try {
  const destparams = {
    Bucket: dstBucket,
    Key: dstKey,
    Body: output_buffer,
```

```
    ContentType: "image"
  };

  const putResult = await s3.send(new PutObjectCommand(destparams));

  } catch (error) {
    console.log(error);
    return;
  }

  console.log('Successfully resized ' + srcBucket + '/' + srcKey +
    ' and uploaded to ' + dstBucket + '/' + dstKey);
  };
```

4. In your `lambda-s3` directory, install the sharp library using npm. Note that the latest version of sharp (0.33) isn't compatible with Lambda. Install version 0.32.6 to complete this tutorial.

```
npm install sharp@0.32.6
```

The npm `install` command creates a `node_modules` directory for your modules. After this step, your directory structure should look like the following.

```
lambda-s3
|- index.mjs
|- node_modules
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

5. Create a .zip deployment package containing your function code and its dependencies. In MacOS and Linux, run the following command.

```
zip -r function.zip .
```

In Windows, use your preferred zip utility to create a .zip file. Ensure that your `index.mjs`, `package.json`, and `package-lock.json` files and your `node_modules` directory are all at the root of your .zip file.

Python

To create the deployment package (Python)

1. Save the example code as a file named `lambda_function.py`.

```
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
  with Image.open(image_path) as image:
    image.thumbnail(tuple(x / 2 for x in image.size))
    image.save(resized_path)

def lambda_handler(event, context):
  for record in event['Records']:
    bucket = record['s3']['bucket']['name']
    key = unquote_plus(record['s3']['object']['key'])
    tmpkey = key.replace('/', '')
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
    upload_path = '/tmp/resized-{}'.format(tmpkey)
    s3_client.download_file(bucket, key, download_path)
    resize_image(download_path, upload_path)
    s3_client.upload_file(upload_path, '{}-resized'.format(bucket), 'resized-
{}'.format(key))
```

2. In the same directory in which you created your `lambda_function.py` file, create a new directory named `package` and install the Pillow (PIL) library and the AWS SDK for Python (Boto3). Although the Lambda Python runtime includes a version of the Boto3 SDK, we recommend that you add all of your function's dependencies to your deployment package, even if they are included in the runtime. For more information, see Runtime dependencies in Python.

```
mkdir package
pip install \
```

```
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
--python-version 3.12 \
--only-binary=:all: --upgrade \
pillow boto3
```

The Pillow library contains C/C++ code. By using the `--platform manylinux_2014_x86_64` and `--only-binary=:all:` options, pip will download and install a version of Pillow that contains pre-compiled binaries compatible with the Amazon Linux 2 operating system. This ensures that your deployment package will work in the Lambda execution environment, regardless of the operating system and architecture of your local build machine.

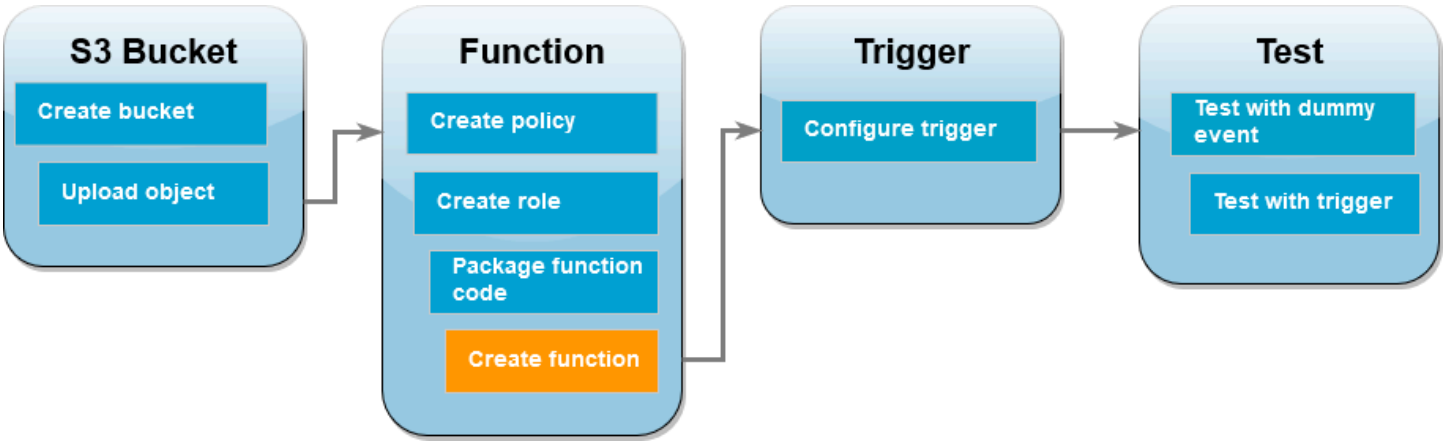
3. Create a .zip file containing your application code and the Pillow and Boto3 libraries. In Linux or MacOS, run the following commands from your command line interface.

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

In Windows, use your preferred zip tool to create the `lambda_function.zip` file. Make sure that your `lambda_function.py` file and the folders containing your dependencies are all at the root of the .zip file.

You can also create your deployment package using a Python virtual environment. See Working with .zip file archives for Python Lambda functions

Create the Lambda function



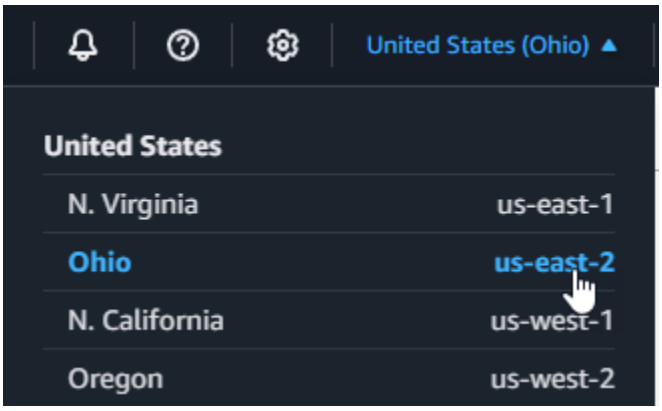
You can create your Lambda function using either the AWS CLI or the Lambda console. Follow the instructions for your chosen language to create the function.

AWS Management Console

To create the function (console)

To create your Lambda function using the console, you first create a basic function containing some 'Hello world' code. You then replace this code with your own function code by uploading the.zip or JAR file you created in the previous step.

1. Open the Functions page of the Lambda console.
2. Make sure you're working in the same AWS Region you created your Amazon S3 bucket in. You can change your region using the drop-down list at the top of the screen.



3. Choose **Create function**.
4. Choose **Author from scratch**.
5. Under **Basic information**, do the following:

a. For **Function name**, enter **`CreateThumbnail`**.

b. For **Runtime**, choose either **Node.js 22.x** or **Python 3.12** according to the language you chose for your function.

c. For **Architecture**, choose **x86_64**.

6. In the **Change default execution role** tab, do the following:

a. Expand the tab, then choose **Use an existing role**.

b. Select the `LambdaS3Role` you created earlier.

7. Choose **Create function**.

To upload the function code (console)

1. In the **Code source** pane, choose **Upload from**.
2. Choose **.zip file**.
3. Choose **Upload**.
4. In the file selector, select your .zip file and choose **Open**.
5. Choose **Save**.

AWS CLI

To create the function (AWS CLI)

- Run the CLI command for the language you chose. For the `role` parameter, make sure to replace `123456789012` with your own AWS account ID. For the `region` parameter, replace `us-east-1` with the region you created your Amazon S3 buckets in.

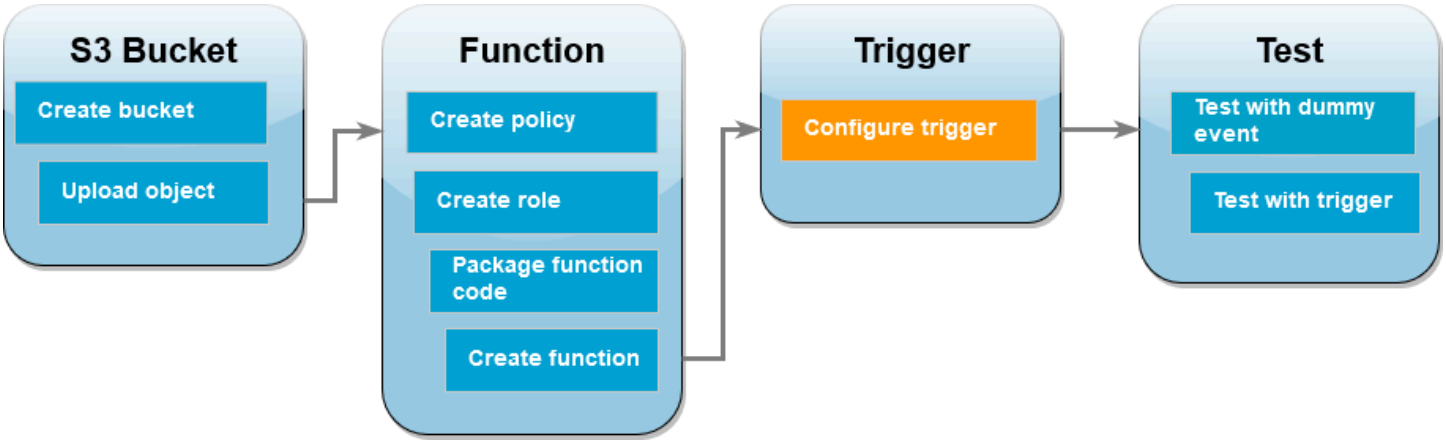
  - For **Node.js**, run the following command from the directory containing your `function.zip` file.

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
```

  - For **Python**, run the following command from the directory containing your `lambda_function.zip` file.

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://lambda_function.zip --handler
 lambda_function.lambda_handler \
--runtime python3.14 --timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
```

Configure Amazon S3 to invoke the function



For your Lambda function to run when you upload an image to your source bucket, you need to configure a trigger for your function. You can configure the Amazon S3 trigger using either the console or the AWS CLI.

> **Important**
>
> This procedure configures the Amazon S3 bucket to invoke your function every time that an object is created in the bucket. Be sure to configure this only on the source bucket. If your Lambda function creates objects in the same bucket that invokes it, your function can be invoked continuously in a loop. This can result in un expected charges being billed to your AWS account.

AWS Management Console

To configure the Amazon S3 trigger (console)

1. Open the Functions page of the Lambda console and choose your function (`CreateThumbnail`).

2. Choose **Add trigger**.
3. Select **S3**.
4. Under **Bucket**, select your source bucket.
5. Under **Event types**, select **All object create events**.
6. Under **Recursive invocation**, select the check box to acknowledge that using the same Amazon S3 bucket for input and output is not recommended. You can learn more about recursive invocation patterns in Lambda by reading Recursive patterns that cause run-away Lambda functions in Serverless Land.
7. Choose **Add**.

   When you create a trigger using the Lambda console, Lambda automatically creates a resource based policy to give the service you select permission to invoke your function.

AWS CLI

To configure the Amazon S3 trigger (AWS CLI)

1. For your Amazon S3 source bucket to invoke your function when you add an image file, you first need to configure permissions for your function using a resource based policy. A resource-based policy statement gives other AWS services permission to invoke your function. To give Amazon S3 permission to invoke your function, run the following CLI command. Be sure to replace the `source-account` parameter with your own AWS account ID and to use your own source bucket name.

```
aws lambda add-permission --function-name CreateThumbnail \
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::amzn-s3-demo-source-bucket \
--source-account 123456789012
```

   The policy you define with this command allows Amazon S3 to invoke your function only when an action takes place on your source bucket.

   > **Note**
   >
   > Although Amazon S3 bucket names are globally unique, when using resource-based policies it is best practice to specify that the bucket must belong to your account.

This is because if you delete a bucket, it is possible for another AWS account to create a bucket with the same Amazon Resource Name (ARN).

2. Save the following JSON in a file named `notification.json`. When applied to your source bucket, this JSON configures the bucket to send a notification to your Lambda function every time a new object is added. Replace the AWS account number and AWS Region in the Lambda function ARN with your own account number and region.

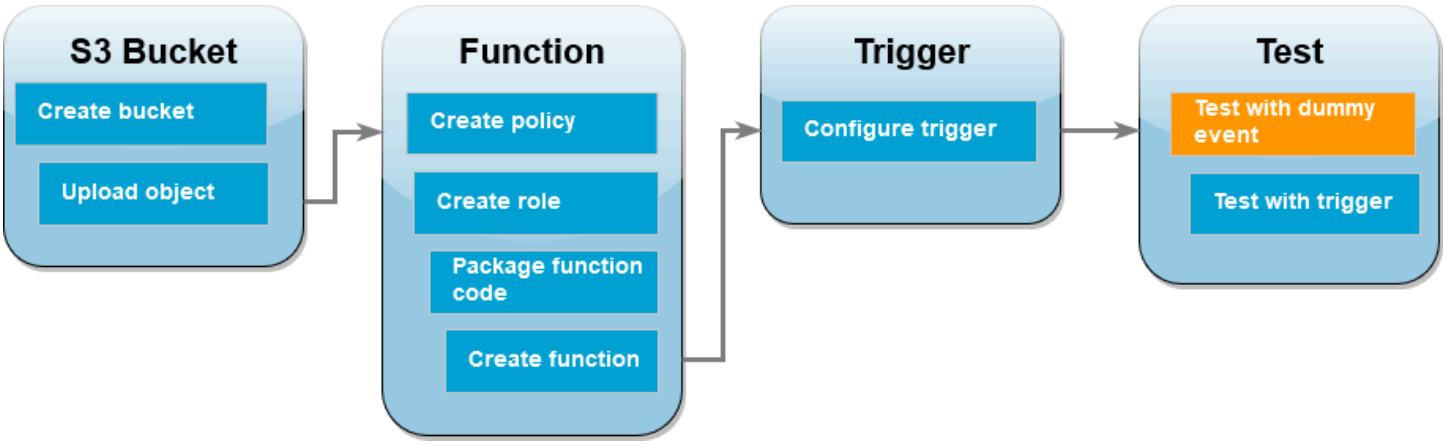
```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. Run the following CLI command to apply the notification settings in the JSON file you created to your source bucket. Replace `amzn-s3-demo-source-bucket` with the name of your own source bucket.

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-source-
bucket \
--notification-configuration file://notification.json
```

To learn more about the `put-bucket-notification-configuration` command and the `notification-configuration` option, see put-bucket-notification-configuration in the *AWS CLI Command Reference*.

Test your Lambda function with a dummy event



Before you test your whole setup by adding an image file to your Amazon S3 source bucket, you test that your Lambda function is working correctly by invoking it with a dummy event. An event in Lambda is a JSON-formatted document that contains data for your function to process. When your function is invoked by Amazon S3, the event sent to your function contains information such as the bucket name, bucket ARN, and object key.

AWS Management Console

To test your Lambda function with a dummy event (console)

1. Open the Functions page of the Lambda console and choose your function (`CreateThumbnail`).
2. Choose the **Test** tab.
3. To create your test event, in the **Test event** pane, do the following:
   a. Under **Test event action**, select **Create new event**.
   b. For **Event name**, enter **`myTestEvent`**.
   c. For **Template**, select **S3 Put**.
   d. Replace the values for the following parameters with your own values.
      - For `awsRegion`, replace `us-east-1` with the AWS Region you created your Amazon S3 buckets in.
      - For `name`, replace `amzn-s3-demo-bucket` with the name of your own Amazon S3 source bucket.

- For `key`, replace `test%2Fkey` with the filename of the test object you uploaded to your source bucket in the step [Upload a test image to your source bucket](#).

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

e. Choose **Save**.

4. In the **Test event** pane, choose **Test**.
5. To check the your function has created a resized verison of your image and stored it in your target Amazon S3 bucket, do the following:

   a. Open the Buckets page of the Amazon S3 console.

   b. Choose your target bucket and confirm that your resized file is listed in the **Objects** pane.

AWS CLI

To test your Lambda function with a dummy event (AWS CLI)

1. Save the following JSON in a file named `dummyS3Event.json`. Replace the values for the following parameters with your own values:

   - For `awsRegion`, replace `us-east-1` with the AWS Region you created your Amazon S3 buckets in.
   - For `name`, replace `amzn-s3-demo-bucket` with the name of your own Amazon S3 source bucket.
   - For `key`, replace `test%2Fkey` with the filename of the test object you uploaded to your source bucket in the step Upload a test image to your source bucket.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
```

```
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

2. From the directory you saved your `dummyS3Event.json` file in, invoke the function by running the following CLI command. This command invokes your Lambda function synchronously by specifying `RequestResponse` as the value of the invocation-type parameter. To learn more about synchronous and asynchronous invocation, see Invoking Lambda functions.

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

   The cli-binary-format option is required if you are using version 2 of the AWS CLI. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options.

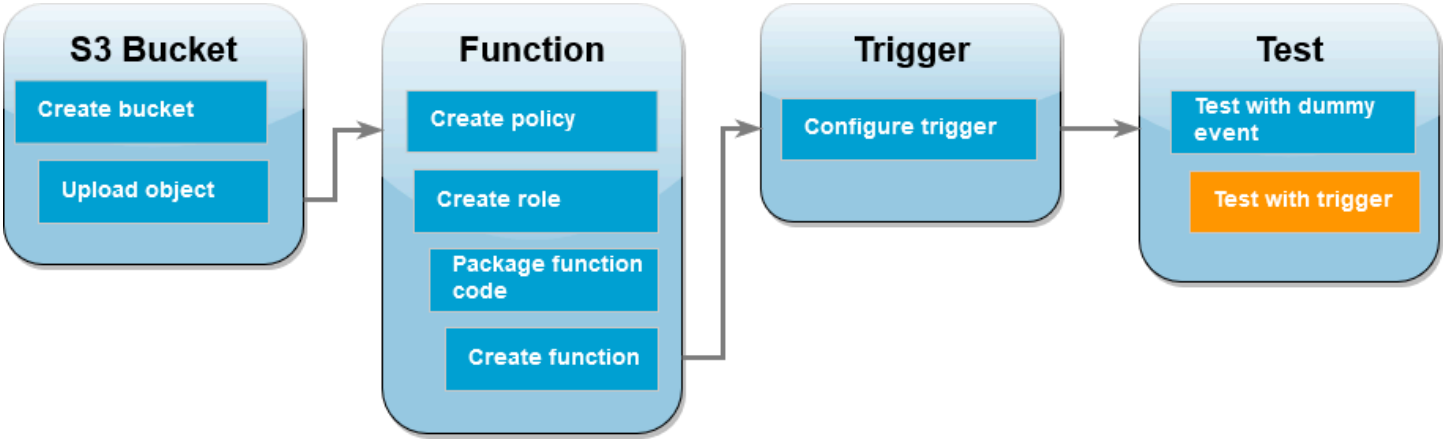
3. Verify that your function has created a thumbnail version of your image and saved it to your target Amazon S3 bucket. Run the following CLI command, replacing `amzn-s3-demo-source-bucket-resized` with the name of your own destination bucket.

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

You should see output similar to the following. The Key parameter shows the filename of your resized image file.

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
            "Size": 2633,
            "StorageClass": "STANDARD"
        }
    ]
}
```

Test your function using the Amazon S3 trigger



Now that you've confirmed your Lambda function is operating correctly, you're ready to test your complete setup by adding an image file to your Amazon S3 source bucket. When you add your image to the source bucket, your Lambda function should be automatically invoked. Your function creates a resized version of the file and stores it in your target bucket.

AWS Management Console

To test your Lambda function using the Amazon S3 trigger (console)

1. To upload an image to your Amazon S3 bucket, do the following:
   a. Open the Buckets page of the Amazon S3 console and choose your source bucket.
   b. Choose **Upload**.
   c. Choose **Add files** and use the file selector to choose the image file you want to upload. Your image object can be any .jpg or .png file.
   d. Choose **Open**, then choose **Upload**.
2. Verify that Lambda has saved a resized version of your image file in your target bucket by doing the following:
   a. Navigate back to the Buckets page of the Amazon S3 console and choose your destination bucket.
   b. In the **Objects** pane, you should now see two resized image files, one from each test of your Lambda function. To download your resized image, select the file, then choose **Download**.

AWS CLI

To test your Lambda function using the Amazon S3 trigger (AWS CLI)

1. From the directory containing the image you want to upload, run the following CLI command. Replace the `--bucket` parameter with the name of your source bucket. For the `--key` and `--body` parameters, use the filename of your test image. Your test image can be any .jpg or .png file.

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key SmileyFace.jpg --
body ./SmileyFace.jpg
```

2. Verify that your function has created a thumbnail version of your image and saved it to your target Amazon S3 bucket. Run the following CLI command, replacing `amzn-s3-demo-source-bucket-resized` with the name of your own destination bucket.

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

If your function runs successfully, you'll see output similar to the following. Your target bucket should now contain two resized files.

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        },
        {
            "Key": "resized-SmileyFace.jpg",
            "LastModified": "2023-06-07T00:13:18+00:00",
            "ETag": "\"ca536e5a1b9e32b22cd549e18792cdbc\"",
            "Size": 1245,
            "StorageClass": "STANDARD"
        }
    ]
}
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the policy that you created

1. Open the Policies page of the IAM console.

2. Select the policy that you created (**AWSLambdaS3Policy**).
3. Choose **Policy actions**, **Delete**.
4. Choose **Delete**.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the S3 bucket

1. Open the Amazon S3 console.
2. Select the bucket you created.
3. Choose **Delete**.
4. Enter the name of the bucket in the text input field.
5. Choose **Delete bucket**.

Use Secrets Manager secrets in Lambda functions

AWS Secrets Manager helps you manage credentials, API keys, and other secrets that your Lambda functions need. You have two main approaches for retrieving secrets in your Lambda functions, both offering better performance and lower costs compared to retrieving secrets directly using the AWS SDK:

- **AWS parameters and secrets Lambda extension** - A runtime-agnostic solution that provides a simple HTTP interface for retrieving secrets
- **Powertools for AWS Lambda parameters utility** - A code-integrated solution that supports multiple providers (Secrets Manager, Parameter Store, AppConfig) with built-in transformations

Both approaches maintain local caches of secrets, eliminating the need for your function to call Secrets Manager for every invocation. When your function requests a secret, the cache is checked first. If the secret is available and hasn't expired, it's returned immediately. Otherwise, it's retrieved from Secrets Manager, cached, and returned. This caching mechanism results in faster response times and reduced costs by minimizing API calls.

Choosing an approach

Consider these factors when choosing between the extension and PowerTools:

Use the AWS parameters and secrets Lambda extension when:

- You want a runtime-agnostic solution that works with any Lambda runtime
- You prefer not to add code dependencies to your function
- You only need to retrieve secrets from Secrets Manager or Parameter Store

Use Powertools for AWS Lambda parameters utility when:

- You want an integrated development experience with your application code
- You need support for multiple providers (Secrets Manager, Parameter Store, AppConfig)
- You want built-in data transformations (JSON parsing, base64 decoding)
- You're using Python, TypeScript, Java, or .NET runtimes

When to use Secrets Manager with Lambda

Common scenarios for using Secrets Manager with Lambda include:

- Storing database credentials that your function uses to connect to Amazon RDS or other databases
- Managing API keys for external services your function calls
- Storing encryption keys or other sensitive configuration data
- Rotating credentials automatically without needing to update your function code

Using the AWS parameters and secrets Lambda extension

The AWS parameters and secrets Lambda extension uses a simple HTTP interface compatible with any Lambda runtime. By default, it caches secrets for 300 seconds (5 minutes) and can hold up to 1,000 secrets. You can customize these settings with environment variables.

Use Secrets Manager in a Lambda function

This section assumes that you already have a Secrets Manager secret. To create a secret, see Create an AWS Secrets Manager secret.

Create the deployment package

Choose your preferred runtime and follow the steps to create a function that retrieves secrets from Secrets Manager. The example function retrieves a secret from Secrets Manager and can be used to access database credentials, API keys, or other sensitive configuration data in your applications.

Python

To create a Python function

1. Create and navigate to a new project directory. Example:

```
mkdir my_function
cd my_function
```

2. Create a file named `lambda_function.py` with the following code. For `secret_name`, use the name or Amazon Resource Name (ARN) of your secret.

```
import json
import os
import requests

def lambda_handler(event, context):
```

```
    try:
        # Replace with the name or ARN of your secret
        secret_name = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"

        secrets_extension_endpoint = f"http://localhost:2773/secretsmanager/get?
secretId={secret_name}"
        headers = {"X-Aws-Parameters-Secrets-Token":
 os.environ.get('AWS_SESSION_TOKEN')}

        response = requests.get(secrets_extension_endpoint, headers=headers)
        print(f"Response status code: {response.status_code}")

        secret = json.loads(response.text)["SecretString"]
        print(f"Retrieved secret: {secret}")

        return {
            'statusCode': response.status_code,
            'body': json.dumps({
                'message': 'Successfully retrieved secret',
                'secretRetrieved': True
            })
        }

    except Exception as e:
        print(f"Error: {str(e)}")
        return {
            'statusCode': 500,
            'body': json.dumps({
                'message': 'Error retrieving secret',
                'error': str(e)
            })
        }
```

3. Create a file named `requirements.txt` with this content:

```
requests
```

4. Install the dependencies:

```
pip install -r requirements.txt -t .
```

5. Create a .zip file containing all files:

```
zip -r function.zip .
```

Node.js

To create a Node.js function

1. Create and navigate to a new project directory. Example:

```
mkdir my_function
cd my_function
```

2. Create a file named `index.mjs` with the following code. For `secret_name`, use the name or Amazon Resource Name (ARN) of your secret.

```
import http from 'http';

export const handler = async (event) => {
    try {
        // Replace with the name or ARN of your secret
        const secretName = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME";
        const options = {
            hostname: 'localhost',
            port: 2773,
            path: `/secretsmanager/get?secretId=${secretName}`,
            headers: {
                'X-Aws-Parameters-Secrets-Token': process.env.AWS_SESSION_TOKEN
            }
        };

        const response = await new Promise((resolve, reject) => {
            http.get(options, (res) => {
                let data = '';
                res.on('data', (chunk) => { data += chunk; });
                res.on('end', () => {
                    resolve({
                        statusCode: res.statusCode,
                        body: data
                    });
                });
            }).on('error', reject);
```

```
        });

        const secret = JSON.parse(response.body).SecretString;
        console.log('Retrieved secret:', secret);

        return {
            statusCode: response.statusCode,
            body: JSON.stringify({
                message: 'Successfully retrieved secret',
                secretRetrieved: true
            })
        };
    } catch (error) {
        console.error('Error:', error);
        return {
            statusCode: 500,
            body: JSON.stringify({
                message: 'Error retrieving secret',
                error: error.message
            })
        };
    }
};
```

3. Create a .zip file containing the `index.mjs` file:

```
zip -r function.zip index.mjs
```

Java

To create a Java function

1. Create a Maven project:

```
mvn archetype:generate \
    -DgroupId=example \
    -DartifactId=lambda-secrets-demo \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.4 \
    -DinteractiveMode=false
```

2. Navigate to the project directory:

```
cd lambda-secrets-demo
```

3. Open the pom.xml and replace the contents with the following:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>example</groupId>
    <artifactId>lambda-secrets-demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.1</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
```

```
                            <finalName>function</finalName>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

4. Rename the `/lambda-secrets-demo/src/main/java/example/App.java` to `Hello.java` to match Lambda's default Java handler name (`example.Hello::handleRequest`):

```
mv src/main/java/example/App.java src/main/java/example/Hello.java
```

5. Open the `Hello.java` file and replace its contents with the following. For `secretName`, use the name or Amazon Resource Name (ARN) of your secret.

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class Hello implements RequestHandler<Object, String> {
    private final HttpClient client = HttpClient.newHttpClient();

    @Override
    public String handleRequest(Object input, Context context) {
        try {
            // Replace with the name or ARN of your secret
            String secretName = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME";
            String endpoint = "http://localhost:2773/secretsmanager/get?
secretId=" + secretName;

            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(endpoint))
                .header("X-Aws-Parameters-Secrets-Token",
 System.getenv("AWS_SESSION_TOKEN"))
```

```
                .GET()
                .build();

            HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());

            String secret = response.body();
            secret = secret.substring(secret.indexOf("SecretString") + 15);
            secret = secret.substring(0, secret.indexOf("\""));

            System.out.println("Retrieved secret: " + secret);
            return String.format(
                "{\"statusCode\": %d, \"body\": \"%s\"}",
                response.statusCode(), "Successfully retrieved secret"
            );

        } catch (Exception e) {
            e.printStackTrace();
            return String.format(
                "{\"body\": \"Error retrieving secret: %s\"}",
                e.getMessage()
            );
        }
    }
}
```

6. Remove the test directory. Maven creates this by default, but we don't need it for this example.

```
rm -rf src/test
```

7. Build the project:

```
mvn package
```

8. Download the JAR file (`target/function.jar`) for later use.

Create the function

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Select **Author from scratch**.

4. For **Function name**, enter **`secret-retrieval-demo`**.
5. Choose your preferred **Runtime**.
6. Choose **Create function**.

To upload the deployment package

1. In the function's **Code** tab, choose **Upload from** and select **.zip file** (for Python and Node.js) or **.jar file** (for Java).
2. Upload the deployment package you created earlier.
3. Choose **Save**.

Add the extension

To add the AWS Parameters and Secrets Lambda extension as a layer

1. In the function's **Code** tab, scroll down to **Layers**.
2. Choose **Add a layer**.
3. Select **AWS layers**.
4. Choose **AWS-Parameters-and-Secrets-Lambda-Extension**.
5. Select the latest version.
6. Choose **Add**.

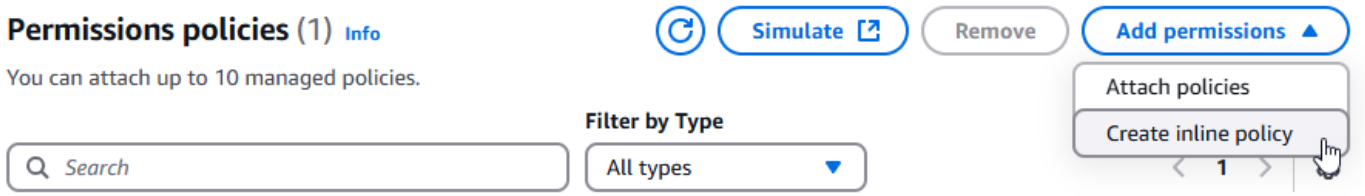
Add permissions

To add Secrets Manager permissions to your execution role

1. Choose the **Configuration** tab, and then choose **Permissions**.
2. Under **Role name**, choose the link to your execution role. This link opens the role in the IAM console.



3. Choose **Add permissions**, and then choose **Create inline policy**.



4. Choose the **JSON** tab and add the following policy. For `Resource`, enter the ARN of your secret.

   JSON

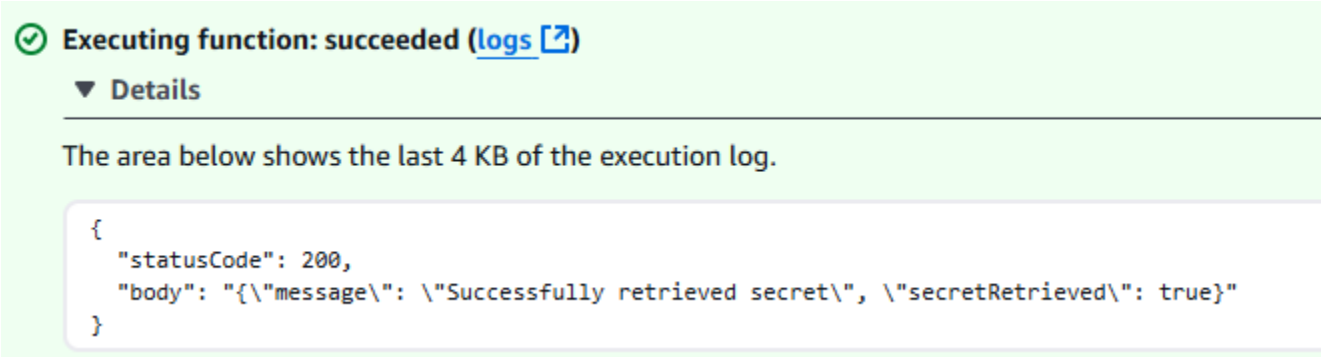
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"
        }
    ]
}
```

5. Choose **Next**.
6. Enter a name for the policy.
7. Choose **Create policy**.

Test the function

To test the function

1. Return to the Lambda console.
2. Select the **Test** tab.
3. Choose **Test**. You should see the following response:



Environment variables

The AWS Parameters and Secrets Lambda extension uses the following default settings. You can override these settings by creating the corresponding environment variables. To view the current settings for a function, set `PARAMETERS_SECRETS_EXTENSION_LOG_LEVEL` to `DEBUG`. The extension will log its configuration information to CloudWatch Logs at the start of each function invocation.

Setting	Default value	Valid values	Environment variable	Details
HTTP port	2773	1 - 65535	PARAMETERS_SECRETS_EXTENSION_HTTP_PORT	Port for the local HTTP server
Cache enabled	TRUE	TRUE | FALSE	PARAMETERS_SECRETS_EXTENSION_CACHE_ENABLED	Enable or disable the cache
Cache size	1000	0 - 1000	PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE	Set to 0 to disable caching
Secrets Manager TTL	300 seconds	0 - 300 seconds	SECRETS_MANAGER_TTL	Time-to-live for cached secrets. Set to 0 to disable caching. This variable is ignored if the value for PARAMETER

Setting	Default value	Valid values	Environment variable	Details
				S_SECRETS_EXTENSIO N_CACHE_SIZE is 0.
Parameter Store TTL	300 seconds	0 - 300 seconds	SSM_PARAMETER_STOR E_TTL	Time-to-live for cached parameters. Set to 0 to disable caching. This variable is ignored if the value for PARAMETER S_SECRETS_EXTENSIO N_CACHE_SIZE is 0.
Log level	INFO	DEBUG | INFO | WARN | ERROR | NONE	PARAMETERS_SECRETS _EXTENSION_LOG_LEV EL	The level of detail reported in logs for the extension
Max connectio ns	3	1 or greater	PARAMETERS_SECRETS _EXTENSION_MAX_CON NECTIONS	Maximum number of HTTP connections for requests to Parameter Store or Secrets Manager
Secrets Manager timeout	0 (no timeout)	All whole numbers	SECRETS_MANAGER_TI MEOUT_MILLIS	Timeout for requests to Secrets Manager (in milliseconds)
Parameter Store timeout	0 (no timeout)	All whole numbers	SSM_PARAMETER_STOR E_TIMEOUT_MILLIS	Timeout for requests to Parameter Store (in milliseconds)

Working with secret rotation

If you rotate secrets frequently, the default 300-second cache duration might cause your function to use outdated secrets. You have two options to ensure your function uses the latest secret value:

- Reduce the cache TTL by setting the `SECRETS_MANAGER_TTL` environment variable to a lower value (in seconds). For example, setting it to `60` ensures your function will never use a secret that's more than one minute old.
- Use the `AWSCURRENT` or `AWSPREVIOUS` staging labels in your secret request to ensure you get the specific version you want:

```
secretsmanager/get?secretId=YOUR_SECRET_NAME&versionStage=AWSCURRENT
```

Choose the approach that best balances your needs for performance and freshness. A lower TTL means more frequent calls to Secrets Manager but ensures you're working with the most recent secret values.

Using the parameters utility from Powertools for AWS Lambda

The parameters utility from Powertools for AWS Lambda provides a unified interface for retrieving secrets from multiple providers including Secrets Manager, parameter store, and AppConfig. It handles caching, transformations, and provides a more integrated development experience compared to the extension approach.

Benefits of the parameters utility

- **Multiple providers** - Retrieve parameters from Secrets Manager, Parameter Store, and AppConfig using the same interface
- **Built-in transformations** - Automatic JSON parsing, base64 decoding, and other data transformations
- **Integrated caching** - Configurable caching with TTL support to reduce API calls
- **Type safety** - Strong typing support in TypeScript and other supported runtimes
- **Error handling** - Built-in retry logic and error handling

Code examples

The following examples show how to retrieve secrets using the Parameters utility in different runtimes:

Python

> **Note**
>
> For complete examples and setup instructions, see the Parameters utility documentation.

Retrieving secrets from Secrets Manager with Powertools for AWS Lambda Parameters utility.

```
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities import parameters

logger = Logger()

def lambda_handler(event, context):
    try:
        # Get secret with caching (default TTL: 5 seconds)
        secret_value = parameters.get_secret("my-secret-name")

        # Get secret with custom TTL
        secret_with_ttl = parameters.get_secret("my-secret-name", max_age=300)

        # Get secret and transform JSON
        secret_json = parameters.get_secret("my-json-secret", transform="json")

        logger.info("Successfully retrieved secrets")

        return {
            'statusCode': 200,
            'body': 'Successfully retrieved secrets'
        }

    except Exception as e:
        logger.error(f"Error retrieving secret: {str(e)}")
        return {
            'statusCode': 500,
            'body': f'Error: {str(e)}'
        }
```

TypeScript

> **Note**
>
> For complete examples and setup instructions, see the Parameters utility documentation.

Retrieving secrets from Secrets Manager with Powertools for AWS Lambda Parameters utility.

```
import { Logger } from '@aws-lambda-powertools/logger';
import { getSecret } from '@aws-lambda-powertools/parameters/secrets';
import type { Context } from 'aws-lambda';

const logger = new Logger();

export const handler = async (event: any, context: Context) => {
    try {
        // Get secret with caching (default TTL: 5 seconds)
        const secretValue = await getSecret('my-secret-name');

        // Get secret with custom TTL
        const secretWithTtl = await getSecret('my-secret-name', { maxAge: 300 });

        // Get secret and transform JSON
        const secretJson = await getSecret('my-json-secret', { transform: 'json' });

        logger.info('Successfully retrieved secrets');

        return {
            statusCode: 200,
            body: 'Successfully retrieved secrets'
        };

    } catch (error) {
        logger.error('Error retrieving secret', { error });
        return {
            statusCode: 500,
            body: `Error: ${error}`
        };
    }
};
```

Java

> **Note**
>
> For complete examples and setup instructions, see the Parameters utility documentation.

Retrieving secrets from Secrets Manager with Powertools for AWS Lambda Parameters utility.

```
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.parameters.SecretsProvider;
import software.amazon.lambda.powertools.parameters.ParamManager;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class SecretHandler implements RequestHandler<Object, String> {

    private final SecretsProvider secretsProvider =
 ParamManager.getSecretsProvider();

    @Logging
    @Override
    public String handleRequest(Object input, Context context) {
        try {
            // Get secret with caching (default TTL: 5 seconds)
            String secretValue = secretsProvider.get("my-secret-name");

            // Get secret with custom TTL (300 seconds)
            String secretWithTtl = secretsProvider.withMaxAge(300).get("my-secret-
name");

            // Get secret and transform JSON
            MySecret secretJson = secretsProvider.get("my-json-secret",
 MySecret.class);

            return "Successfully retrieved secrets";

        } catch (Exception e) {
            return "Error retrieving secret: " + e.getMessage();
        }
    }
```

```
    public static class MySecret {
        // Define your secret structure here
    }
}
```

.NET

> **Note**
>
> For complete examples and setup instructions, see the Parameters utility documentation.

Retrieving secrets from Secrets Manager with Powertools for AWS Lambda Parameters utility.

```
using AWS.Lambda.Powertools.Logging;
using AWS.Lambda.Powertools.Parameters;
using Amazon.Lambda.Core;

[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

public class Function
{
    private readonly ISecretsProvider _secretsProvider;

    public Function()
    {
        _secretsProvider = ParametersManager.SecretsProvider;
    }

    [Logging]
    public async Task<string> FunctionHandler(object input, ILambdaContext context)
    {
        try
        {
            // Get secret with caching (default TTL: 5 seconds)
            var secretValue = await _secretsProvider.GetAsync("my-secret-name");

            // Get secret with custom TTL
            var secretWithTtl = await
 _secretsProvider.WithMaxAge(TimeSpan.FromMinutes(5))
                .GetAsync("my-secret-name");
```

```
            // Get secret and transform JSON
            var secretJson = await _secretsProvider.GetAsync<MySecret>("my-json-
secret");

            return "Successfully retrieved secrets";
        }
        catch (Exception e)
        {
            return $"Error retrieving secret: {e.Message}";
        }
    }

    public class MySecret
    {
        // Define your secret structure here
    }
}
```

Setup and permissions

To use the Parameters utility, you need to:

1. Install Powertools for AWS Lambda for your runtime. For details, see Powertools for AWS Lambda.
2. Add the necessary IAM permissions to your function's execution role. Refer to Managing permissions in AWS Lambda for details.
3. Configure any optional settings through environment variables.

The required IAM permissions are the same as for the extension approach. The utility will automatically handle caching and API calls to Secrets Manager based on your configuration.

Using Lambda with Amazon SQS

> **Note**
>
> If you want to send data to a target other than a Lambda function or enrich the data before sending it, see Amazon EventBridge Pipes.

You can use a Lambda function to process messages in an Amazon Simple Queue Service (Amazon SQS) queue. Lambda supports both standard queues and first-in, first-out (FIFO) queues for event source mappings. You can also use provisioned mode to allocate dedicated polling resources for your Amazon SQS event source mappings. The Lambda function and the Amazon SQS queue must be in the same AWS Region, although they can be in different AWS accounts.

When processing Amazon SQS messages, you need to implement partial batch response logic to prevent successfully processed messages from being retried when some messages in a batch fail. The Batch Processor utility from Powertools for AWS Lambda simplifies this implementation by automatically handling partial batch response logic, reducing development time and improving reliability.

Topics

- Understanding polling and batching behavior for Amazon SQS event source mappings
- Using provisioned mode with Amazon SQS event source mappings
- Configuring provisioned mode for Amazon SQS event source mapping
- Example standard queue message event
- Example FIFO queue message event
- Creating and configuring an Amazon SQS event source mapping
- Configuring scaling behavior for SQS event source mappings
- Handling errors for an SQS event source in Lambda
- Lambda parameters for Amazon SQS event source mappings
- Using event filtering with an Amazon SQS event source
- Tutorial: Using Lambda with Amazon SQS
- Tutorial: Using a cross-account Amazon SQS queue as an event source

Understanding polling and batching behavior for Amazon SQS event source mappings

With Amazon SQS event source mappings, Lambda polls the queue and invokes your function synchronously with an event. Each event can contain a batch of multiple messages from the queue. Lambda receives these events one batch at a time, and invokes your function once for each batch. When your function successfully processes a batch, Lambda deletes its messages from the queue.

When Lambda receives a batch, the messages stay in the queue but are hidden for the length of the queue's visibility timeout. If your function successfully processes all messages in the batch, Lambda deletes the messages from the queue. By default, if your function encounters an error while processing a batch, all messages in that batch become visible in the queue again after the visibility timeout expires. For this reason, your function code must be able to process the same message multiple times without unintended side effects.

> **Warning**
>
> Lambda event source mappings process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

To prevent Lambda from processing a message multiple times, you can either configure your event source mapping to include batch item failures in your function response, or you can use the DeleteMessage API to remove messages from the queue as your Lambda function successfully processes them.

For more information about configuration parameters that Lambda supports for SQS event source mappings, see the section called “Creating an SQS event source mapping”.

Using provisioned mode with Amazon SQS event source mappings

For workloads where you need to fine-tune the throughput of your event source mapping, you can use provisioned mode. In provisioned mode, you define minimum and maximum limits for the amount of provisioned event pollers. These provisioned event pollers are dedicated to your event source mapping, and can handle unexpected message spikes through responsive autoscaling. Amazon SQS event source mapping configured with Provisioned Mode scales 3x faster (up to 1,000

concurrent invokes per minute) and supports 16x higher concurrency (up to 20,000 concurrent invokes) than default Amazon SQS event source mapping capability. We recommend that you use provisioned mode for Amazon SQS event- driven workloads that have strict performance requirements, such as financial services firms processing market data feeds, e-commerce platforms providing real-time personalized recommendations, and gaming companies managing live player interactions. Using provisioned mode incurs additional costs. For detailed pricing, see AWS Lambda pricing.

Each event poller in provisioned mode can handle up to 1 MB/s of throughput, up to 10 concurrent invokes, or up to 10 Amazon SQS polling API calls per second. The range of accepted values for the minimum number of event pollers (MinimumPollers) is between 2 and 200, with default of 2. The range of accepted values for the maximum number of event pollers (MaximumPollers) is between 2 and 2,000, with default of 200. MaximumPollers must be greater than or equal to MinimumPollers.

Determining required event pollers

To estimate the number of event pollers required to ensure optimal message processing performance when using provisioned mode for SQS ESM, gather the following metrics for your application: peak SQS events per second requiring low-latency processing, average SQS event payload size, average Lambda function duration, and configured batch size.

First you can estimate the number of SQS events per second (EPS) supported by an event poller for your workload using the following formula:

```
EPS per event poller =
        minimum(
            ceiling(1024 / average event size in KB),
            ceiling(10 / average function duration in seconds) * batch size,
            min(100, 10 * batch size)
                )
```

Then, you can calculate the number of minimum pollers required using below formula. This calculation ensures you provision sufficient capacity to handle your peak traffic requirements.

```
Required event pollers = (Peak number of events per second in Queue) / EPS per event
 poller
```

Consider a workload with a default batch size of 10, average event size of 3 KB, average function duration of 100 ms, and a requirement to handle 1,000 events per second. In this scenario, each

event poller will support approximately 100 events per second (EPS). Therefore, you should set minimum pollers to 10 to adequately handle your peak traffic requirements. If your workload has the same characteristics but with average function duration of 1 second, each poller will support only 10 EPS, requiring you to configure 100 minimum pollers to support 1,000 events per second at low latency.

We recommend using default batch size of 10 or higher to maximize the efficiency of provisioned mode event pollers. Higher batch sizes allow each poller to process more events per invocation, for improved throughput and cost efficiency. When planning your event poller capacity, account for potential traffic spikes and consider setting your minimumPollers value slightly higher than the calculated minimum to provide a buffer. Additionally, monitor your workload characteristics over time, as changes in message size, function duration, or traffic patterns may necessitate adjustments to your event poller configuration to maintain optimal performance and cost efficiency. For precise capacity planning, we recommend testing your specific workload to determine the actual EPS each event poller can drive.

Configuring provisioned mode for Amazon SQS event source mapping

You can configure provisioned mode for your Amazon SQS event source mapping using the console or the Lambda API.

To configure provisioned mode for an existing Amazon SQS event source mapping (console)

1. Open the Functions page of the Lambda console.
2. Choose the function with the Amazon SQS event source mapping that you want to configure provisioned mode for.
3. Choose **Configuration**, then choose **Triggers**.
4. Choose the Amazon SQS event source mapping that you want to configure provisioned mode for, then choose **Edit**.
5. Under **Event source mapping configuration**, choose **Configure provisioned mode**.
   - For **Minimum event pollers**, enter a value between 2 and 200. If you don't specify a value, Lambda chooses a default value of 2.
   - For **Maximum event pollers**, enter a value between 2 and 2,000. This value must be greater than or equal to your value for **Minimum event pollers**. If you don't specify a value, Lambda chooses a default value of 200.
6. Choose **Save**.

You can configure provisioned mode programmatically using the `ProvisionedPollerConfig` object in your `EventSourceMappingConfiguration`. For example, the following `UpdateEventSourceMapping` CLI command configures a `MinimumPollers` value of 5, and a `MaximumPollers` value of 100.

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

After configuring provisioned mode, you can observe the usage of event pollers for your workload by monitoring the `ProvisionedPollers` metric. For more information, see Event source mapping metrics.

To disable provisioned mode and return to default (on-demand) mode, you can use the following `UpdateEventSourceMapping` CLI command:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

> **Note**
>
> Provisioned mode cannot be used in conjunction with the maximum concurrency setting. When using provisioned mode, you control maximum concurrency through the maximum number of event pollers.

For more information on configuring provisioned mode, see the section called “Create mapping”.

Example standard queue message event

Example Amazon SQS message event (standard queue)

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
```

```
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {
                "myAttribute": {
                    "stringValue": "myValue",
                    "stringListValues": [],
                    "binaryListValues": [],
                    "dataType": "String"
                }
            },
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        },
        {
            "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",
            "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082650636",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082650649"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        }
    ]
}
```

By default, Lambda polls up to 10 messages in your queue at once and sends that batch to your function. To avoid invoking the function with a small number of records, you can configure the event source to buffer records for up to 5 minutes by configuring a batch window. Before invoking the function, Lambda continues to poll messages from the standard queue until the batch window

expires, the invocation payload size quota is reached, or the configured maximum batch size is reached.

If you're using a batch window and your SQS queue contains very low traffic, Lambda might wait for up to 20 seconds before invoking your function. This is true even if you set a batch window lower than 20 seconds.

> **Note**
>
> In Java, you might experience null pointer errors when deserializing JSON. This could be due to how case of "Records" and "eventSourceARN" is converted by the JSON object mapper.

Example FIFO queue message event

For FIFO queues, records contain additional attributes that are related to deduplication and sequencing.

Example Amazon SQS message event (FIFO queue)

```
{
    "Records": [
        {
            "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
            "receiptHandle": "AQEBBX8nesZEXmkhsmZeyIE8iQAMig7qw...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1573251510774",
                "SequenceNumber": "18849496460467696128",
                "MessageGroupId": "1",
                "SenderId": "AIDAIO23YVJENQZJOL4VO",
                "MessageDeduplicationId": "1",
                "ApproximateFirstReceiveTimestamp": "1573251510774"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo.fifo",
            "awsRegion": "us-east-2"
```

```
        }
    ]
}
```

Creating and configuring an Amazon SQS event source mapping

To process Amazon SQS messages with Lambda, configure your queue with the appropriate settings, then create a Lambda event source mapping.

Topics

- Configuring a queue to use with Lambda
- Setting up Lambda execution role permissions
- Creating an SQS event source mapping

Configuring a queue to use with Lambda

If you don't already have an existing Amazon SQS queue, create one to serve as an event source for your Lambda function. The Lambda function and the Amazon SQS queue must be in the same AWS Region, although they can be in different AWS accounts.

To allow your function time to process each batch of records, set the source queue's visibility timeout to at least six times the configuration timeout on your function. The extra time allows Lambda to retry if your function is throttled while processing a previous batch.

> **Note**
>
> Your function's timeout must be less than or equal to the queue's visibility timeout. Lambda validates this requirement when you create or update an event source mapping and will return an error if the function timeout exceeds the queue's visibility timeout.

By default, if Lambda encounters an error at any point while processing a batch, all messages in that batch return to the queue. After the visibility timeout, the messages become visible to Lambda again. You can configure your event source mapping to use partial batch responses to return only the failed messages back to the queue. In addition, if your function fails to process a message multiple times, Amazon SQS can send it to a dead-letter queue. We recommend setting the `maxReceiveCount` on your source queue's redrive policy to at least 5. This gives Lambda a few chances to retry before sending failed messages directly to the dead-letter queue.

Setting up Lambda execution role permissions

The AWSLambdaSQSQueueExecutionRole AWS managed policy includes the permissions that Lambda needs to read from your Amazon SQS queue. You can add this managed policy to your function's execution role.

Optionally, if you're using an encrypted queue, you also need to add the following permission to your execution role:

- kms:Decrypt

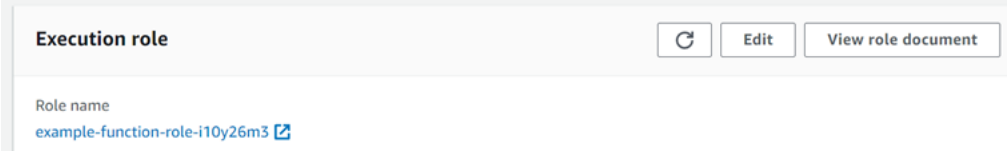
Creating an SQS event source mapping

Create an event source mapping to tell Lambda to send items from your queue to a Lambda function. You can create multiple event source mappings to process items from multiple queues with a single function. When Lambda invokes the target function, the event can contain multiple items, up to a configurable maximum *batch size*.

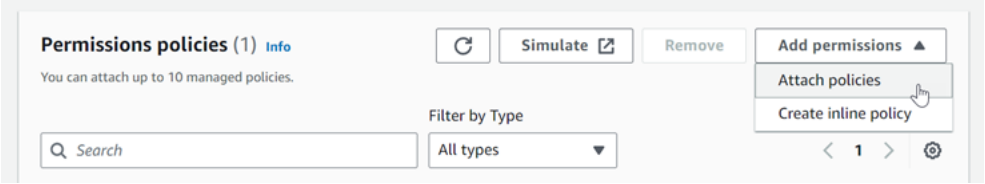
To configure your function to read from Amazon SQS, attach the AWSLambdaSQSQueueExecutionRole AWS managed policy to your execution role. Then, create an **SQS** event source mapping from the console using the following steps.

To add permissions and create a trigger

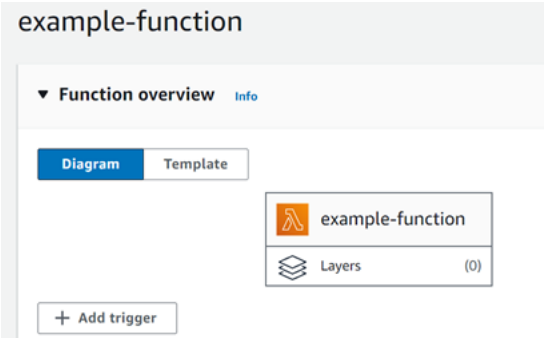
1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose the **Configuration** tab, and then choose **Permissions**.
4. Under **Role name**, choose the link to your execution role. This link opens the role in the IAM console.



5. Choose **Add permissions**, and then choose **Attach policies**.



6. In the search field, enter `AWSLambdaSQSQueueExecutionRole`. Add this policy to your execution role. This is an AWS managed policy that contains the permissions your function needs to read from an Amazon SQS queue. For more information about this policy, see AWSLambdaSQSQueueExecutionRole in the *AWS Managed Policy Reference*.
7. Go back to your function in the Lambda console. Under **Function overview**, choose **Add trigger**.



8. Choose a trigger type.
9. Configure the required options, and then choose **Add**.

Lambda supports the following configuration options for Amazon SQS event sources:

SQS queue

The Amazon SQS queue to read records from. The Lambda function and the Amazon SQS queue must be in the same AWS Region, although they can be in different AWS accounts.

Enable trigger

The status of the event source mapping. **Enable trigger** is selected by default.

Batch size

The maximum number of records to send to the function in each batch. For a standard queue, this can be up to 10,000 records. For a FIFO queue, the maximum is 10. For a batch size over 10, you must also set the batch window (`MaximumBatchingWindowInSeconds`) to at least 1 second.

Configure your function timeout to allow enough time to process an entire batch of items. If items take a long time to process, choose a smaller batch size. A large batch size can improve efficiency for workloads that are very fast or have a lot of overhead. If you configure reserved concurrency on your function, set a minimum of five concurrent executions to reduce the chance of throttling errors when Lambda invokes your function.

Lambda passes all of the records in the batch to the function in a single call, as long as the total size of the events doesn't exceed the invocation payload size quota for synchronous invocation (6 MB). Both Lambda and Amazon SQS generate metadata for each record. This additional metadata is counted towards the total payload size and can cause the total number of records sent in a batch to be lower than your configured batch size. The metadata fields that Amazon SQS sends can be variable in length. For more information about the Amazon SQS metadata fields, see the ReceiveMessage API operation documentation in the *Amazon Simple Queue Service API Reference*.

Batch window

The maximum amount of time to gather records before invoking the function, in seconds. This applies only to standard queues.

If you're using a batch window greater than 0 seconds, you must account for the increased processing time in your queue's visibility timeout. We recommend setting your queue's visibility timeout to six times your function timeout, plus the value of `MaximumBatchingWindowInSeconds`. This allows time for your Lambda function to process each batch of events and to retry in the event of a throttling error.

When messages become available, Lambda starts processing messages in batches. Lambda starts processing five batches at a time with five concurrent invocations of your function. If messages are still available, Lambda adds up to 300 concurrent invokes of your function a minute, up to a maximum of 1,250 concurrent invokes. When using provisioned mode, each event poller can handle up to 1 MB/s of throughput, up to 10 concurrent invokes, or up to 10 Amazon SQS polling API calls per second. Lambda scales the number of event pollers between your configured minimum and maximum, quickly adding up to 1,000 concurrent invokes per minute to provide low-latency processing of your Amazon SQS events. You control scaling and concurrency through these minimum and maximum event poller settings. To learn more about function scaling and concurrency, see Understanding Lambda function scaling.

To process more messages, you can optimize your Lambda function for higher throughput. For more information, see Understanding how AWS Lambda scales with Amazon SQS standard queues.

Filter criteria

Add filter criteria to control which events Lambda sends to your function for processing. For more information, see Control which events Lambda sends to your function.

Maximum concurrency

The maximum number of concurrent functions that the event source can invoke. Cannot be used with Provisioned Mode enabled. For more information, see Configuring maximum concurrency for Amazon SQS event sources.

Provisioned Mode

When enabled, allocates dedicated polling resources for your event source mapping. You can configure the minimum (2-200) and maximum (2-2000) number of event pollers. Each event poller can handle up to 1 MB/sec of throughput, up to 10 concurrent invokes, or up to 10 Amazon SQS polling API calls per second.

> **Note**
>
> Note: You cannot use Provisioned Mode and Maximum concurrency together. When Provisioned Mode is enabled, use the maximum pollers setting to control concurrency.

Configuring scaling behavior for SQS event source mappings

You can control the scaling behavior of your Amazon SQS event source mappings either through maximum concurrency settings or by enabling provisioned mode. These are mutually exclusive options.

By default, Lambda automatically scales event pollers based on message volume. When you enable provisioned mode, you allocate a minimum and maximum number of dedicated polling resources that remain ready to handle expected traffic patterns. This allows you to optimize your event source mapping's performance in two ways:

- Standard mode (Default): Lambda automatically manages scaling, starting with a small number of pollers and scaling up or down based on workload.
- Provisioned mode: You configure dedicated polling resources with minimum and maximum limits, enabling 3 times faster scaling and up to 16 times higher processing capacity.

For standard queues, Lambda uses long polling to poll a queue until it becomes active. When messages are available, Lambda starts processing five batches at a time with five concurrent invocations of your function. If messages are still available, Lambda increases the number of processes that are reading batches by up to 300 more concurrent invokes per minute. The

maximum number of invokes that an event source mapping can process simultaneously is 1,250. When traffic is low, Lambda scales back the processing to five concurrent invokes, and can optimize to as few as 2 concurrent invokes to reduce the Amazon SQS calls and corresponding costs. However, this optimization is not available when you enable the maximum concurrency setting.

For FIFO queues, Lambda sends messages to your function in the order that it receives them. When you send a message to a FIFO queue, you specify a message group ID. Amazon SQS ensures that messages in the same group are delivered to Lambda in order. When Lambda reads your messages into batches, each batch may contain messages from more than one message group, but the order of the messages is maintained. If your function returns an error, the function attempts all retries on the affected messages before Lambda receives additional messages from the same group.

When using provisioned mode, each event poller can handle up to 1 MB/sec of throughput, up to 10 concurrent invokes, or up to 10 Amazon SQS polling API calls per second. Lambda scales the number of event pollers between your configured minimum and maximum, quickly adding up to 1,000 concurrency per minute to provide consistent, low-latency processing of your Amazon SQS events. Using provisioned mode incurs additional costs. For detailed pricing, see AWS Lambda pricing. Each event poller uses long polling to your SQS queue with up to 10 polls per second, which incur SQS API requests cost. See Amazon SQS pricing for details. You control scaling and concurrency through these minimum and maximum event poller settings, rather than using the maximum concurrency setting, as these options cannot be used together.

> **Note**
>
> You cannot use the maximum concurrency setting and provisioned mode at the same time. When provisioned mode is enabled, you control the scaling and concurrency of your Amazon SQS event source mapping through the minimum and maximum number of event pollers.

Configuring maximum concurrency for Amazon SQS event sources

You can use the maximum concurrency setting to control scaling behavior for your SQS event sources. Note that maximum concurrency cannot be used with provisioned mode enabled. The maximum concurrency setting limits the number of concurrent instances of the function that an Amazon SQS event source can invoke. Maximum concurrency is an event source-level setting. If you have multiple Amazon SQS event sources mapped to one function, each event source can have a separate maximum concurrency setting. You can use maximum concurrency to prevent one

queue from using all of the function's reserved concurrency or the rest of the account's concurrency quota. There is no charge for configuring maximum concurrency on an Amazon SQS event source.

Importantly, maximum concurrency and reserved concurrency are two independent settings. Don't set maximum concurrency higher than the function's reserved concurrency. If you configure maximum concurrency, make sure that your function's reserved concurrency is greater than or equal to the total maximum concurrency for all Amazon SQS event sources on the function. Otherwise, Lambda may throttle your messages.

When your account's concurrency quota is set to the default value of 1,000, an Amazon SQS event source mapping can scale to invoke function instances up to this value, unless you specify a maximum concurrency.

If you receive an increase to your account's default concurrency quota, Lambda may not be able to invoke concurrent functions instances up to your new quota. By default, Lambda can scale to invoke up to 1,250 concurrent function instances for an Amazon SQS event source mapping. If this is insufficient for your use case, contact AWS support to discuss an increase to your account's Amazon SQS event source mapping concurrency.

> **Note**
>
> For FIFO queues, concurrent invocations are capped either by the number of message group IDs (`messageGroupId`) or the maximum concurrency setting—whichever is lower. For example, if you have six message group IDs and maximum concurrency is set to 10, your function can have a maximum of six concurrent invocations.

You can configure maximum concurrency on new and existing Amazon SQS event source mappings.

Configure maximum concurrency using the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Under **Function overview**, choose **SQS**. This opens the **Configuration** tab.
4. Select the Amazon SQS trigger and choose **Edit**.
5. For **Maximum concurrency**, enter a number between 2 and 1,000. To turn off maximum concurrency, leave the box empty.
6. Choose **Save**.

Configure maximum concurrency using the AWS Command Line Interface (AWS CLI)

Use the update-event-source-mapping command with the `--scaling-config` option. Example:

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config '{"MaximumConcurrency":5}'
```

To turn off maximum concurrency, enter an empty value for `--scaling-config`:

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

Configure maximum concurrency using the Lambda API

Use the CreateEventSourceMapping or UpdateEventSourceMapping action with a ScalingConfig object.

Handling errors for an SQS event source in Lambda

To handle errors related to an SQS event source, Lambda automatically uses a retry strategy with a backoff strategy. You can also customize error handling behavior by configuring your SQS event source mapping to return partial batch responses.

Backoff strategy for failed invocations

When an invocation fails, Lambda attempts to retry the invocation while implementing a backoff strategy. The backoff strategy differs slightly depending on whether Lambda encountered the failure due to an error in your function code, or due to throttling.

- If your **function code** caused the error, Lambda will stop processing and retrying the invocation. In the meantime, Lambda gradually backs off, reducing the amount of concurrency allocated to your Amazon SQS event source mapping. After your queue's visibility timeout runs out, the message will again reappear in the queue.
- If the invocation fails due to **throttling**, Lambda gradually backs off retries by reducing the amount of concurrency allocated to your Amazon SQS event source mapping. Lambda continues to retry the message until the message's timestamp exceeds your queue's visibility timeout, at which point Lambda drops the message.

Implementing partial batch responses

When your Lambda function encounters an error while processing a batch, all messages in that batch become visible in the queue again by default, including messages that Lambda processed successfully. As a result, your function can end up processing the same message several times.

To avoid reprocessing successfully processed messages in a failed batch, you can configure your event source mapping to make only the failed messages visible again. This is called a partial batch response. To turn on partial batch responses, specify `ReportBatchItemFailures` for the FunctionResponseTypes action when configuring your event source mapping. This lets your function return a partial success, which can help reduce the number of unnecessary retries on records.

> **Note**
>
> The Batch Processor utility from Powertools for AWS Lambda handles all of the partial batch response logic automatically. This utility simplifies implementing batch processing patterns and reduces the custom code needed to handle batch item failures correctly. It is available for Python, Java, Typescript, and .NET.

When `ReportBatchItemFailures` is activated, Lambda doesn't scale down message polling when function invocations fail. If you expect some messages to fail—and you don't want those failures to impact the message processing rate—use `ReportBatchItemFailures`.

> **Note**
>
> Keep the following in mind when using partial batch responses:
>
> - If your function throws an exception, the entire batch is considered a complete failure.
> - If you're using this feature with a FIFO queue, your function should stop processing messages after the first failure and return all failed and unprocessed messages in `batchItemFailures`. This helps preserve the ordering of messages in your queue.

To activate partial batch reporting

1. Review the Best practices for implementing partial batch responses.

2. Run the following command to activate `ReportBatchItemFailures` for your function. To retrieve your event source mapping's UUID, run the list-event-source-mappings AWS CLI command.

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--function-response-types "ReportBatchItemFailures"
```

3. Update your function code to catch all exceptions and return failed messages in a `batchItemFailures` JSON response. The `batchItemFailures` response must include a list of message IDs, as `itemIdentifier` JSON values.

   For example, suppose you have a batch of five messages, with message IDs `id1`, `id2`, `id3`, `id4`, and `id5`. Your function successfully processes `id1`, `id3`, and `id5`. To make messages `id2` and `id4` visible again in your queue, your function should return the following response:

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
}
```

   Here are some examples of function code that return the list of failed message IDs in the batch:

   .NET

   SDK for .NET

   > **Note**
   >
   > There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
```

```
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {
  if len(message.Body) > 0 {
   // Your message processing condition here
   fmt.Printf("Successfully processed message: %s\n", message.Body)
  } else {
   // Message processing failed
```

```
   fmt.Printf("Failed to process message %s\n", message.MessageId)
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
```

```
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context)
{
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
ArrayList<SQSBatchResponse.BatchItemFailure>();

        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures
list
                batchItemFailures.add(new
SQSBatchResponse.BatchItemFailure(message.getMessageId()));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using JavaScript.

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
```

```
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
    console.log(`Processed message: ${record.body}`);
}
```

Reporting SQS batch item failures with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure,
 SQSRecord } from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
```

```
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS
 records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}
```

```
        for record in event["Records"]:
            try:
                print(f"Processed message: {record['body']}")
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
```

```
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
```

```
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

If the failed events do not return to the queue, see How do I troubleshoot Lambda function SQS ReportBatchItemFailures? in the AWS Knowledge Center.

Success and failure conditions

Lambda treats a batch as a complete success if your function returns any of the following:

- An empty `batchItemFailures` list
- A null `batchItemFailures` list
- An empty `EventResponse`
- A null `EventResponse`

Lambda treats a batch as a complete failure if your function returns any of the following:

- An invalid JSON response
- An empty string `itemIdentifier`
- A null `itemIdentifier`
- An `itemIdentifier` with a bad key name
- An `itemIdentifier` value with a message ID that doesn't exist

CloudWatch metrics

To determine whether your function is correctly reporting batch item failures, you can monitor the `NumberOfMessagesDeleted` and `ApproximateAgeOfOldestMessage` Amazon SQS metrics in Amazon CloudWatch.

- `NumberOfMessagesDeleted` tracks the number of messages removed from your queue. If this drops to 0, this is a sign that your function response is not correctly returning failed messages.
- `ApproximateAgeOfOldestMessage` tracks how long the oldest message has stayed in your queue. A sharp increase in this metric can indicate that your function is not correctly returning failed messages.

Using Powertools for AWS Lambda batch processor

The batch processor utility from Powertools for AWS Lambda automatically handles partial batch response logic, reducing the complexity of implementing batch failure reporting. Here are examples using the batch processor:

Python

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing Amazon SQS messages with AWS Lambda batch processor.

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import SQSEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.SQS)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
```

```
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> For complete examples and setup instructions, see the batch processor documentation.

Processing Amazon SQS messages with AWS Lambda batch processor.

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { SQSEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.SQS);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: SQSEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Lambda parameters for Amazon SQS event source mappings

All Lambda event source types share the same CreateEventSourceMapping and UpdateEventSourceMapping API operations. However, only some of the parameters apply to Amazon SQS.

Parameter	Required	Default	Notes
BatchSize	N	10	For standard queues, the maximum is 10,000. For FIFO queues, the maximum is 10.
Enabled	N	true	none
EventSourceArn	Y	N/A	The ARN of the data stream or a stream consumer
FunctionName	Y	N/A	none
FilterCriteria	N	N/A	Control which events Lambda sends to your function
FunctionResponseTypes	N	N/A	To let your function report specific failures in a batch, include the value `ReportBatchItemFailures` in `FunctionResponseTypes`. For more information, see Implementing partial batch responses.
MaximumBatchingWindowInSeconds	N	0	Batching window is not supported for FIFO queues

Parameter	Required	Default	Notes
ProvisionedPollerConfig	N	N/A	Configures the minimum (2-200) and maximum (2-2000) number of dedicated event pollers for the SQS event source mapping. Each poller can handle up to 1 MB/sec of throughput and 10 concurrent invokes.
ScalingConfig	N	N/A	Configuring maximum concurrency for Amazon SQS event sources

Using event filtering with an Amazon SQS event source

You can use event filtering to control which records from a stream or queue Lambda sends to your function. For general information about how event filtering works, see the section called “Event filtering”.

This section focuses on event filtering for Amazon SQS event sources.

> **Note**
>
> Amazon SQS event source mappings only support filtering on the body key.

Topics

- Amazon SQS event filtering basics

Amazon SQS event filtering basics

Suppose your Amazon SQS queue contains messages in the following JSON format.

```
{
    "RecordNumber": 1234,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

An example record for this queue would look as follows.

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "RecordNumber": 1234,\n "TimeStamp": "yyyy-mm-ddThh:mm:ss",\n
 "RequestCode": "AAAA"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
        },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
    "awsRegion": "us-west-2"
}
```

To filter based on the contents of your Amazon SQS messages, use the `body` key in the Amazon SQS message record. Suppose you want to process only those records where the `RequestCode` in your Amazon SQS message is “BBBB.” The `FilterCriteria` object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
```

```
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

Suppose you want your function to process only those records where `RecordNumber` is greater than 9999. The `FilterCriteria` object would be as follows.

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

For added clarity, here is the value of the filter's `Pattern` expanded in plain JSON.

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

You can add your filter using the console, AWS CLI or an AWS SAM template.

Console

To add this filter using the console, follow the instructions in Attaching filter criteria to an event source mapping (console) and enter the following string for the **Filter criteria**.

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

To create a new event source mapping with these filter criteria using the AWS Command Line Interface (AWS CLI), run the following command.

```
aws lambda create-event-source-mapping \
```

```
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

To add these filter criteria to an existing event source mapping, run the following command.

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

To add this filter using AWS SAM, add the following snippet to the YAML template for your event source.

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

For Amazon SQS, the message body can be any string. However, this can be problematic if your `FilterCriteria` expect body to be in a valid JSON format. The reverse scenario is also true—if the incoming message body is in JSON format but your filter criteria expects body to be a plain string, this can lead to unintended behavior.

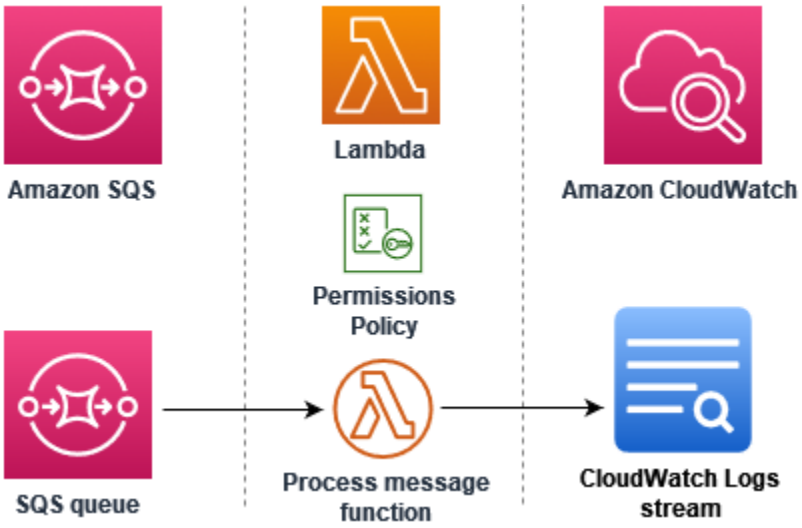
To avoid this issue, ensure that the format of body in your `FilterCriteria` matches the expected format of body in messages that you receive from your queue. Before filtering your messages, Lambda automatically evaluates the format of the incoming message body and of your filter pattern for body. If there is a mismatch, Lambda drops the message. The following table summarizes this evaluation:

Incoming message body format	Filter pattern body format	Resulting action
Plain string	Plain string	Lambda filters based on your filter criteria.

Incoming message body format	Filter pattern body format	Resulting action
Plain string	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Plain string	Valid JSON	Lambda drops the message.
Valid JSON	Plain string	Lambda drops the message.
Valid JSON	No filter pattern for data properties	Lambda filters (on the other metadata properties only) based on your filter criteria.
Valid JSON	Valid JSON	Lambda filters based on your filter criteria.

Tutorial: Using Lambda with Amazon SQS

In this tutorial, you create a Lambda function that consumes messages from an Amazon Simple Queue Service (Amazon SQS) queue. The Lambda function runs whenever a new message is added to the queue. The function writes the messages to an Amazon CloudWatch Logs stream. The following diagram shows the AWS resources you use to complete the tutorial.



To complete this tutorial, you carry out the following steps:

1. Create a Lambda function that writes messages to CloudWatch Logs.
2. Create an Amazon SQS queue.
3. Create a Lambda event source mapping. The event source mapping reads the Amazon SQS queue and invokes your Lambda function when a new message is added.
4. Test the setup by adding messages to your queue and monitoring the results in CloudWatch Logs.

Prerequisites

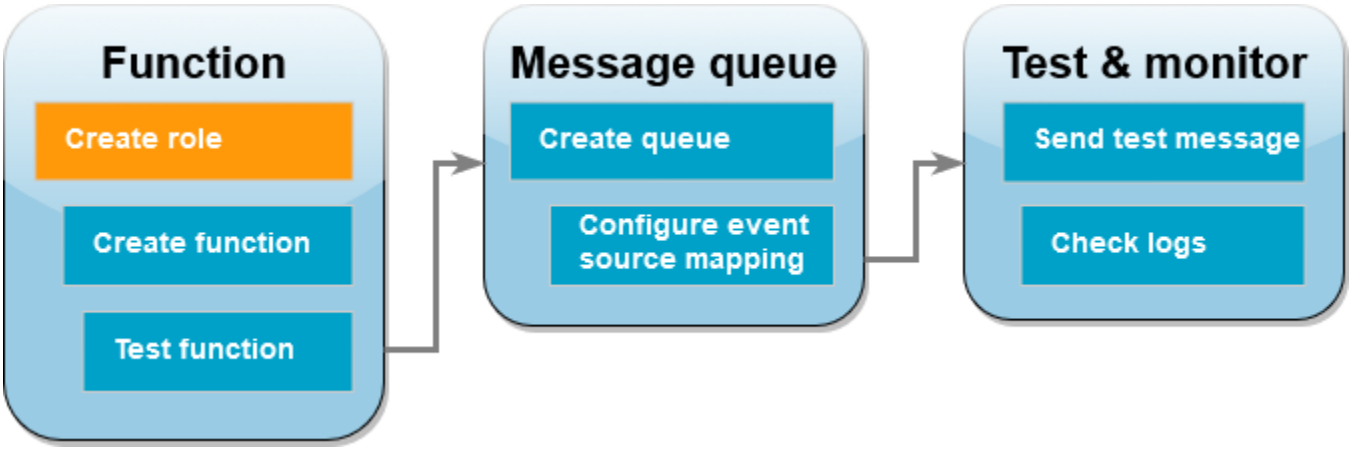
Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create the execution role



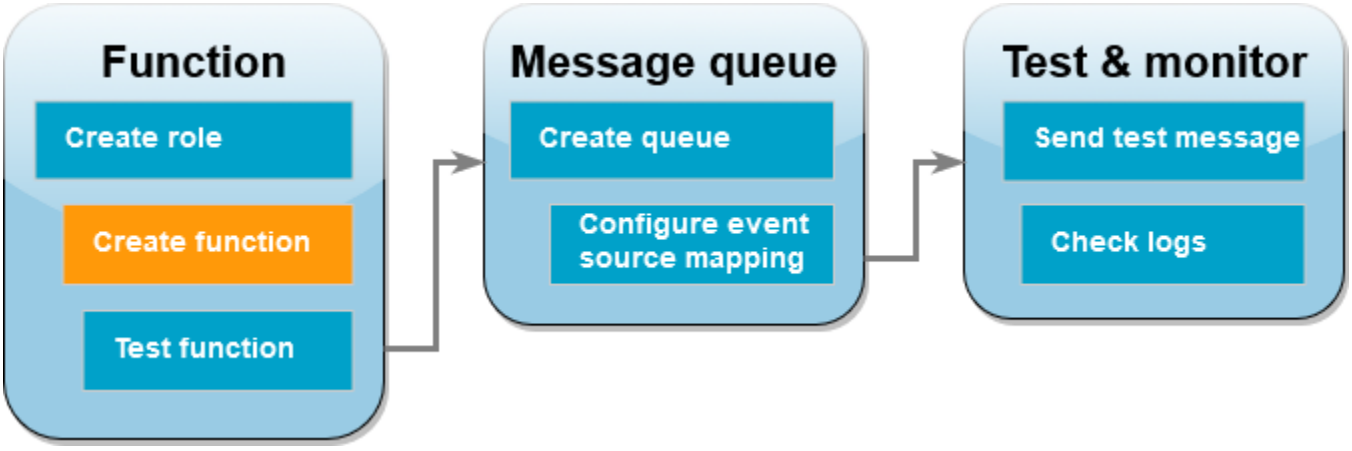
An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and resources. To allow your function to read items from Amazon SQS, attach the **AWSLambdaSQSQueueExecutionRole** permissions policy.

To create an execution role and attach an Amazon SQS permissions policy

1. Open the Roles page of the IAM console.
2. Choose **Create role**.
3. For **Trusted entity type**, choose **AWS service**.
4. For **Use case**, choose **Lambda**.
5. Choose **Next**.
6. In the **Permissions policies** search box, enter **`AWSLambdaSQSQueueExecutionRole`**.
7. Select the **AWSLambdaSQSQueueExecutionRole** policy, and then choose **Next**.
8. Under **Role details**, for **Role name**, enter **`lambda-sqs-role`**, then choose **Create role**.

After role creation, note down the Amazon Resource Name (ARN) of your execution role. You'll need it in later steps.

Create the function



Create a Lambda function that processes your Amazon SQS messages. The function code logs the body of the Amazon SQS message to CloudWatch Logs.

This tutorial uses the Node.js 24 runtime, but we've also provided example code in other runtime languages. You can select the tab in the following box to see code for the runtime you're interested in. The JavaScript code you'll use in this step is in the first example shown in the **JavaScript** tab.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
```

```
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
```

```
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
```

```
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }

    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
```

```
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SQS event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";

export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

To create a Node.js Lambda function

1. Create a directory for the project, and then switch to that directory.

```
mkdir sqs-tutorial
cd sqs-tutorial
```

2. Copy the sample JavaScript code into a new file named `index.js`.
3. Create a deployment package using the following `zip` command.

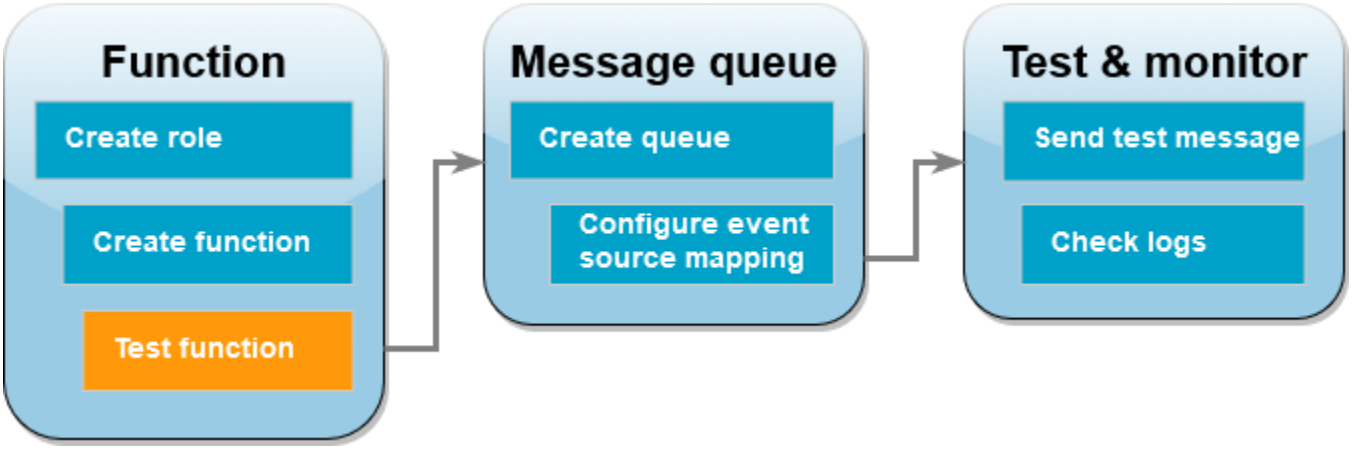
```
zip function.zip index.js
```

4. Create a Lambda function using the create-function AWS CLI command. For the `role` parameter, enter the ARN of the execution role that you created earlier.

> **Note**
>
> The Lambda function and the Amazon SQS queue must be in the same AWS Region.

```
aws lambda create-function --function-name ProcessSQSRecord \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/lambda-sqs-role
```

Test the function



Invoke your Lambda function manually using the `invoke` AWS CLI command and a sample Amazon SQS event.

To invoke the Lambda function with a sample event

1. Save the following JSON as a file named `input.json`. This JSON simulates an event that Amazon SQS might send to your Lambda function, where `"body"` contains the actual message from the queue. In this example, the message is `"test"`.

Example Amazon SQS event

This is a test event—you don't need to change the message or the account number.

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:my-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

2. Run the following invoke AWS CLI command. This command returns CloudWatch logs in the response. For more information about retrieving logs, see Access logs with the AWS CLI.

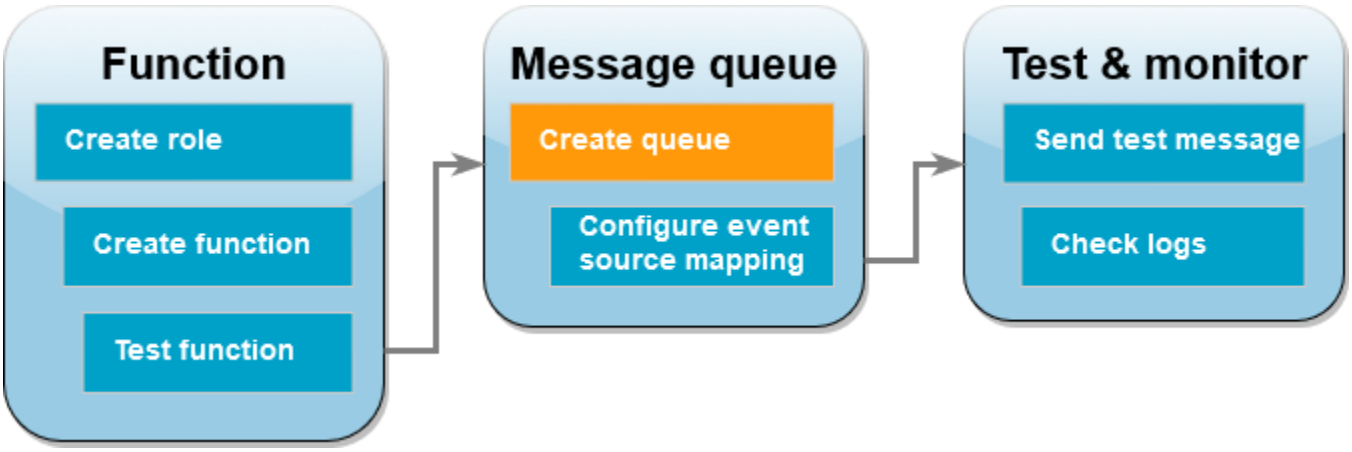
```
aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out
 --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

3. Find the `INFO` log in the response. This is where the Lambda function logs the message body. You should see logs that look like this:

```
2023-09-11T22:45:04.271Z 348529ce-2211-4222-9099-59d07d837b60 INFO Processed
 message test
2023-09-11T22:45:04.288Z 348529ce-2211-4222-9099-59d07d837b60 INFO done
```

Create an Amazon SQS queue



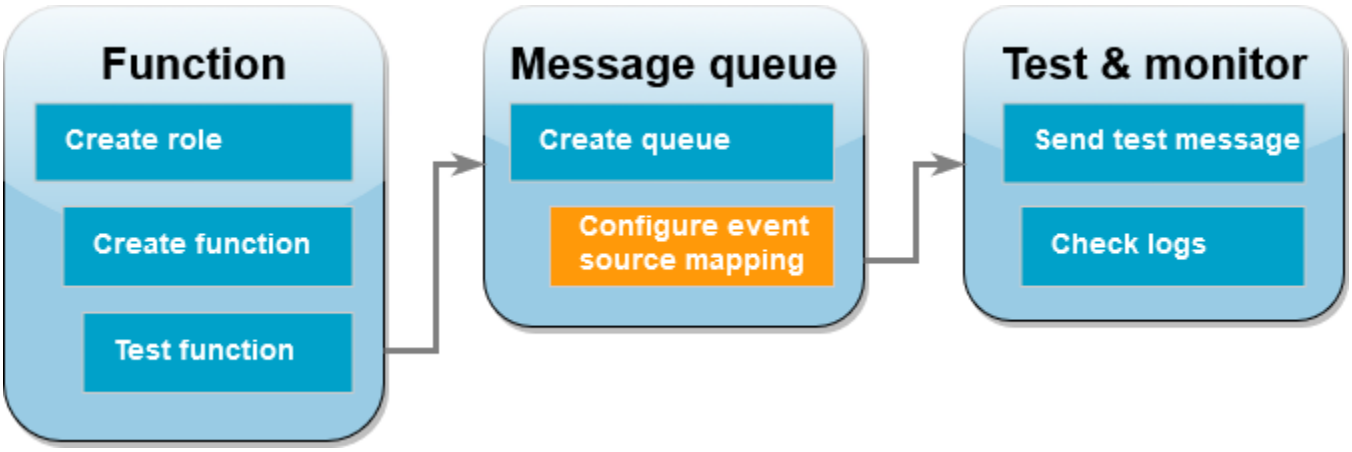
Create an Amazon SQS queue that the Lambda function can use as an event source. The Lambda function and the Amazon SQS queue must be in the same AWS Region.

To create a queue

1. Open the Amazon SQS console.
2. Choose **Create queue**.
3. Enter a name for the queue. Leave all other options at the default settings.
4. Choose **Create queue**.

After creating the queue, note down its ARN. You need this in the next step when you associate the queue with your Lambda function.

Configure the event source



Connect the Amazon SQS queue to your Lambda function by creating an event source mapping. The event source mapping reads the Amazon SQS queue and invokes your Lambda function when a new message is added.

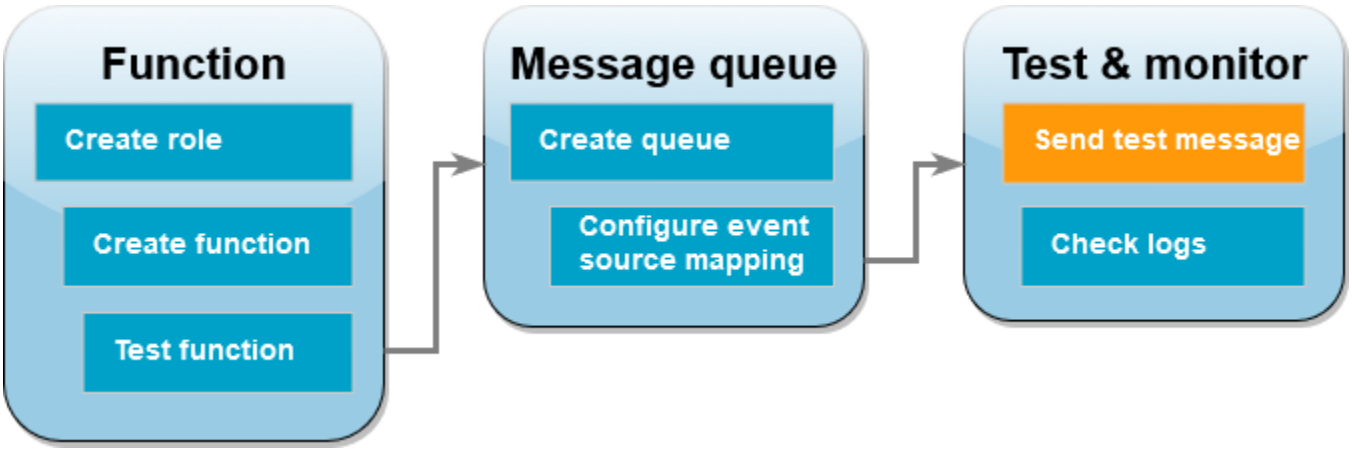
To create a mapping between your Amazon SQS queue and your Lambda function, use the create-event-source-mapping AWS CLI command. Example:

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

To get a list of your event source mappings, use the list-event-source-mappings command. Example:

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

Send a test message

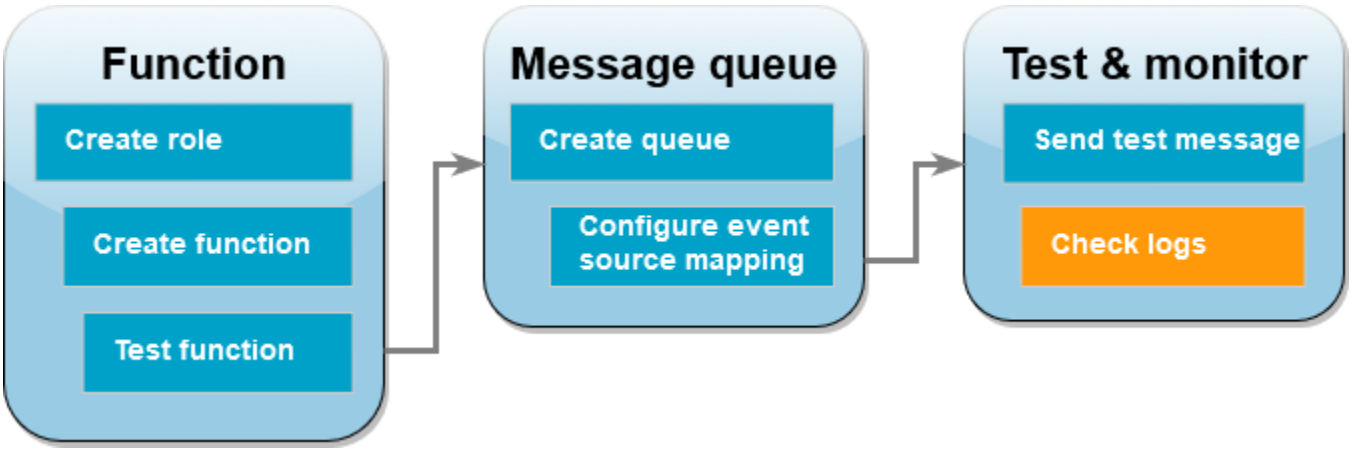


To send an Amazon SQS message to the Lambda function

1. Open the Amazon SQS console.
2. Choose the queue that you created earlier.
3. Choose **Send and receive messages**.
4. Under **Message body**, enter a test message, such as "this is a test message."
5. Choose **Send message**.

Lambda polls the queue for updates. When there is a new message, Lambda invokes your function with this new event data from the queue. If the function handler returns without exceptions, Lambda considers the message successfully processed and begins reading new messages in the queue. After successfully processing a message, Lambda automatically deletes it from the queue. If the handler throws an exception, Lambda considers the batch of messages not successfully processed, and Lambda invokes the function with the same batch of messages.

Check the CloudWatch logs



To confirm that the function processed the message

1. Open the Functions page of the Lambda console.
2. Choose the **ProcessSQSRecord** function.
3. Choose **Monitor**.
4. Choose **View CloudWatch logs**.
5. In the CloudWatch console, choose the **Log stream** for the function.
6. Find the `INFO` log. This is where the Lambda function logs the message body. You should see the message that you sent from the Amazon SQS queue. Example:

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.

4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the Amazon SQS queue

1. Sign in to the AWS Management Console and open the Amazon SQS console at https://console.aws.amazon.com/sqs/.
2. Select the queue you created.
3. Choose **Delete**.
4. Enter **`confirm`** in the text input field.
5. Choose **Delete**.

Tutorial: Using a cross-account Amazon SQS queue as an event source

In this tutorial, you create a Lambda function that consumes messages from an Amazon Simple Queue Service (Amazon SQS) queue in a different AWS account. This tutorial involves two AWS accounts: **Account A** refers to the account that contains your Lambda function, and **Account B** refers to the account that contains the Amazon SQS queue.

Prerequisites

Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create the execution role (Account A)

In **Account A**, create an execution role that gives your function permission to access the required AWS resources.

To create an execution role

1. Open the Roles page in the AWS Identity and Access Management (IAM) console.
2. Choose **Create role**.
3. Create a role with the following properties.
   - **Trusted entity** – **AWS Lambda**
   - **Permissions** – **AWSLambdaSQSQueueExecutionRole**
   - **Role name** – **`cross-account-lambda-sqs-role`**

The **AWSLambdaSQSQueueExecutionRole** policy has the permissions that the function needs to read items from Amazon SQS and to write logs to Amazon CloudWatch Logs.

Create the function (Account A)

In **Account A**, create a Lambda function that processes your Amazon SQS messages. The Lambda function and the Amazon SQS queue must be in the same AWS Region.

The following Node.js code example writes each message to a log in CloudWatch Logs.

Example index.mjs

```
export const handler = async function(event, context) {
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
```

```
}
```

To create the function

> **Note**
>
> Following these steps creates a Node.js function. For other languages, the steps are similar, but some details are different.

1. Save the code example as a file named `index.mjs`.
2. Create a deployment package.

```
zip function.zip index.mjs
```

3. Create the function using the `create-function` AWS Command Line Interface (AWS CLI) command. Replace `arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role` with the ARN of the execution role that you created earlier.

```
aws lambda create-function --function-name CrossAccountSQSExample \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role
```

Test the function (Account A)

In **Account A**, test your Lambda function manually using the `invoke` AWS CLI command and a sample Amazon SQS event.

If the handler returns normally without exceptions, Lambda considers the message to be successfully processed and begins reading new messages in the queue. After successfully processing a message, Lambda automatically deletes it from the queue. If the handler throws an exception, Lambda considers the batch of messages not successfully processed, and Lambda invokes the function with the same batch of messages.

1. Save the following JSON as a file named `input.txt`.

```
{
    "Records": [
        {
```

```
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:example-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

The preceding JSON simulates an event that Amazon SQS might send to your Lambda function, where `"body"` contains the actual message from the queue.

2. Run the following `invoke` AWS CLI command.

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

3. Verify the output in the file `outputfile.txt`.

Create an Amazon SQS queue (Account B)

In **Account B**, create an Amazon SQS queue that the Lambda function in **Account A** can use as an event source. The Lambda function and the Amazon SQS queue must be in the same AWS Region.

To create a queue

1. Open the Amazon SQS console.

2. Choose **Create queue**.
3. Create a queue with the following properties.

   - **Type** – **Standard**
   - **Name** – **LambdaCrossAccountQueue**
   - **Configuration** – Keep the default settings.
   - **Access policy** – Choose **Advanced**. Paste in the following JSON policy. Replace the following values:
     - 111122223333: AWS account ID for **Account A**
     - 444455556666: AWS account ID for **Account B**

     JSON

```
{
    "Version":"2012-10-17",
    "Id": "Queue1_Policy_UUID",
    "Statement": [
        {
            "Sid": "Queue1_AllActions",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::111122223333:role/cross-account-lambda-
sqs-role"
                ]
            },
            "Action": "sqs:*",
            "Resource": "arn:aws:sqs:us-
east-1:444455556666:LambdaCrossAccountQueue"
        }
    ]
}
```

     This policy grants the Lambda execution role in **Account A** permissions to consume messages from this Amazon SQS queue.
4. After creating the queue, record its Amazon Resource Name (ARN). You need this in the next step when you associate the queue with your Lambda function.

Configure the event source (Account A)

In **Account A**, create an event source mapping between the Amazon SQS queue in **Account B** and your Lambda function by running the following `create-event-source-mapping` AWS CLI command. Replace `arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue` with the ARN of the Amazon SQS queue that you created in the previous step.

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-
size 10 \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

To get a list of your event source mappings, run the following command.

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

Test the setup

You can now test the setup as follows:

1. In **Account B**, open the Amazon SQS console.
2. Choose **LambdaCrossAccountQueue**, which you created earlier.
3. Choose **Send and receive messages**.
4. Under **Message body**, enter a test message.
5. Choose **Send message**.

Your Lambda function in **Account A** should receive the message. Lambda will continue to poll the queue for updates. When there is a new message, Lambda invokes your function with this new event data from the queue. Your function runs and creates logs in Amazon CloudWatch. You can view the logs in the CloudWatch console.

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

In **Account A**, clean up your execution role and Lambda function.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

In **Account B**, clean up the Amazon SQS queue.

To delete the Amazon SQS queue

1. Sign in to the AWS Management Console and open the Amazon SQS console at https://console.aws.amazon.com/sqs/.
2. Select the queue you created.
3. Choose **Delete**.
4. Enter **`confirm`** in the text input field.
5. Choose **Delete**.

Orchestrating Lambda functions with Step Functions

AWS Step Functions provides visual workflow orchestration for coordinating Lambda functions with other AWS services. With native integrations to 220+ AWS services and fully managed, zero-maintenance infrastructure, Step Functions is ideal when you need visual workflow design and fully-managed service integrations.

For orchestration using standard programming languages within Lambda where workflow logic lives alongside business logic, consider Lambda durable functions. For help choosing between these options, see Durable functions or Step Functions.

For example, processing an order might require validating the order details, checking inventory levels, processing payment, and generating an invoice. Write separate Lambda functions for each task and use Step Functions to manage the workflow. Step Functions coordinates the flow of data between your functions and handles errors at each step. This separation makes your workflows easier to visualize, modify, and maintain as they grow more complex.

When to use Step Functions with Lambda

The following scenarios are good examples of when Step Functions is a particularly good fit for orchestrating Lambda-based applications.

- Sequential processing
- Complex error handling
- Conditional workflows and human approvals
- Parallel processing

Sequential processing

Sequential processing is when one task must complete before the next task can begin. For example, in an order processing system, payment processing can't begin until order validation is complete, and invoice generation must wait for payment confirmation. Write separate Lambda functions for each task and use Step Functions to manage the sequence and handle data flow between functions.

Anti-pattern example

A single Lambda function manages the entire order processing workflow by:

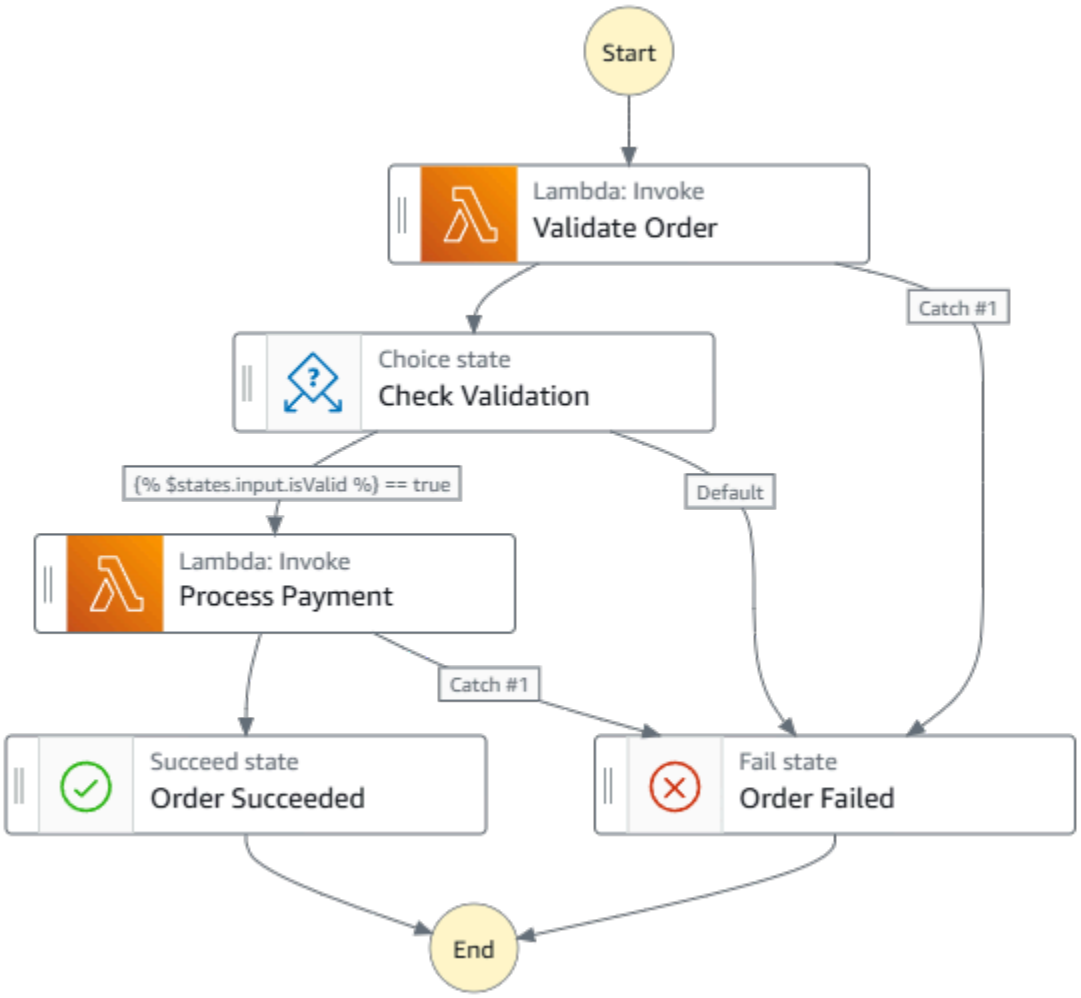
- Invoking other Lambda functions in sequence
- Parsing and validating responses from each function
- Implementing error handling and recovery logic
- Managing the flow of data between functions

Recommended approach

Use two Lambda functions: one to validate the order and one to process the payment. Step Functions coordinates these functions by:

- Running tasks in the correct sequence
- Passing data between functions
- Implementing error handling at each step
- Using Choice states to ensure only valid orders proceed to payment

Example workflow graph



> **Note**
>
> **Code-first alternative:** For sequential processing with code-based checkpointing and retry, see Lambda durable functions steps.

Complex error handling

While Lambda provides retry capabilities for asynchronous invocations and event source mappings, Step Functions offers more sophisticated error handling for complex workflows. You can configure automatic retries with exponential backoff and set different retry policies for different types of errors. When retries are exhausted, use `Catch` to route errors to a fallback state. This is particularly

useful when you need workflow-level error handling that coordinates multiple functions and services.

To learn more about handling Lambda function errors in a state machine, see Handling errors in *The AWS Step Functions Workshop*.

Anti-pattern example

A single Lambda function handles all of the following:

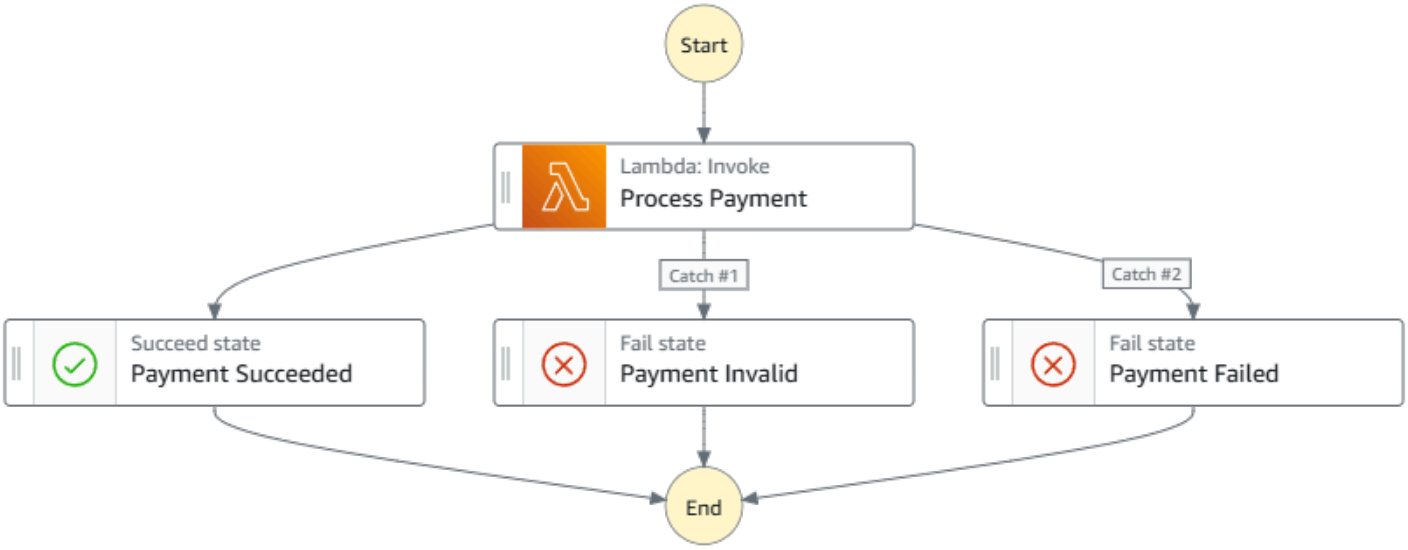
- Attempts to call a payment processing service
- If the payment service is unavailable, the function waits and tries again later.
- Implements a custom exponential backoff for the wait time
- After all attempts fail, catch the error and choose another flow

Recommended approach

Use a single Lambda function focused solely on payment processing. Step Functions manages error handling by:

- Automatically retrying failed tasks with configurable backoff periods
- Applying different retry policies based on error types
- Routing different types of errors to appropriate fallback states
- Maintaining error handling state and history

Example workflow graph



> **Note**
>
> **Code-first alternative:** Durable functions provide try-catch error handling with configurable retry strategies. See Error handling in durable functions.

Conditional workflows and human approvals

Use the Step Functions Choice state to route workflows based on function output and the waitForTaskToken suffix to pause workflows for human decisions. For example, to process a credit limit increase request, use a Lambda function to evaluate risk factors. Then, use Step Functions to route high-risk requests to manual approval and low-risk requests to automatic approval.

To deploy an example workflow that uses a callback task token integration pattern, see Callback with Task Token in *The AWS Step Functions Workshop*.

Anti-pattern example

A single Lambda function manages a complex approval workflow by:

- Implementing nested conditional logic to evaluate credit requests
- Invoking different approval functions based on request amounts
- Managing multiple approval paths and decision points

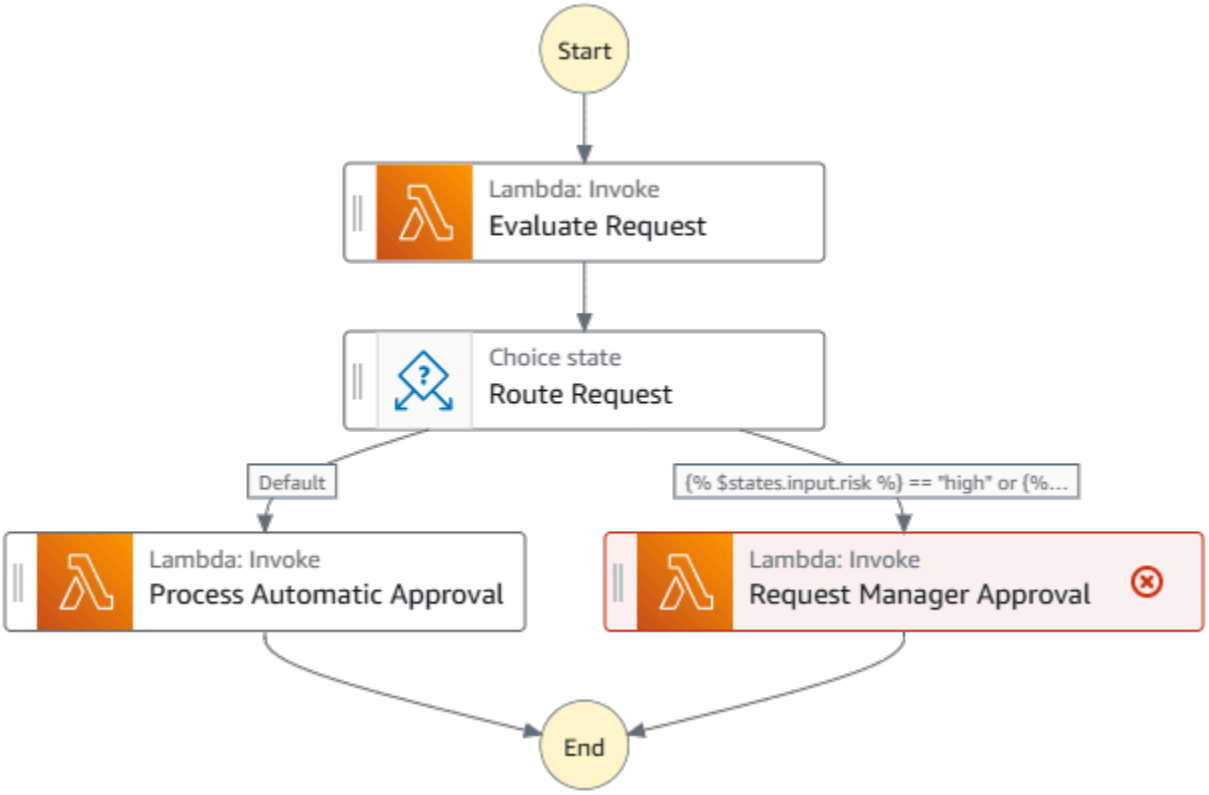
- Tracking the state of pending approvals
- Implementing timeout and notification logic for approvals

Recommended approach

Use three Lambda functions: one to evaluate the risk of each request, one to approve low-risk requests, and one to route high-risk requests to a manager for review. Step Functions manages the workflow by:

- Using Choice states to route requests based on amount and risk level
- Pausing execution while waiting for human approval
- Managing timeouts for pending approvals
- Providing visibility into the current state of each request

Example workflow graph



> **Note**
>
> **Code-first alternative:** Durable functions support callbacks for human-in-the-loop workflows. See Callbacks in durable functions.

Parallel processing

Step Functions provides three ways to handle parallel processing:

- The Parallel state executes multiple branches of your workflow simultaneously. Use this when you need to run different functions in parallel, such as generating thumbnails while extracting image metadata.
- The Inline Map state processes arrays of data with up to 40 concurrent iterations. Use this for small to medium datasets where you need to perform the same operation on each item.
- The Distributed Map state handles large-scale parallel processing with up to 10,000 concurrent executions, supporting both JSON arrays and Amazon Simple Storage Service (Amazon S3) data sources. Use this when processing large datasets or when you need higher concurrency.

Anti-pattern example

A single Lambda function attempts to manage parallel processing by:

- Simultaneously invoking multiple image processing functions
- Implementing custom parallel execution logic
- Managing timeouts and error handling for each parallel task
- Collecting and aggregating results from all functions

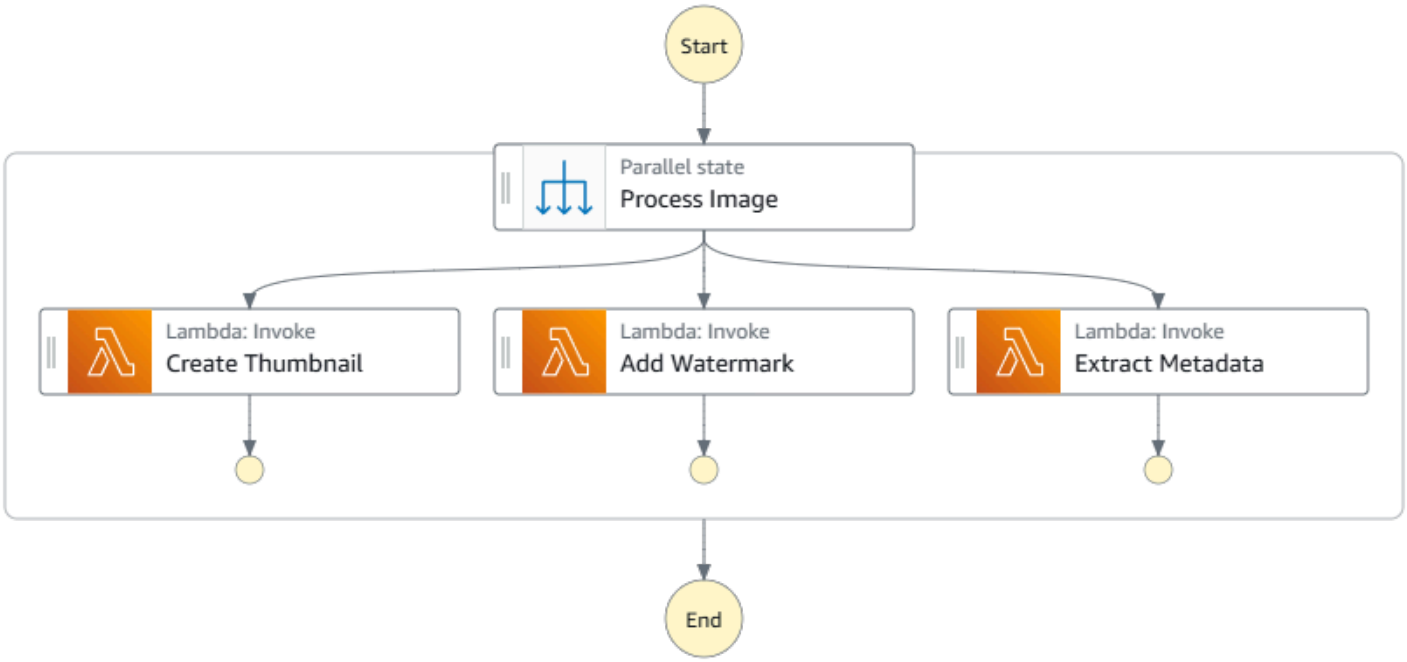
Recommended approach

Use three Lambda functions: one to create a thumbnail image, one to add a watermark, and one to extract the metadata. Step Functions manages these functions by:

- Running all functions simultaneously using the Parallel state
- Collecting results from each function into an ordered array
- Managing timeouts and error handling across all parallel executions

- Proceeding only when all parallel branches complete

Example workflow graph



> **Note**
>
> **Code-first alternative:** Durable functions provide `parallel()` and `map()` operations. See Parallel execution.

When not to use Step Functions with Lambda

Not all Lambda-based applications benefit from using Step Functions. Consider these scenarios when choosing your application architecture.

- Simple applications
- Complex data processing
- CPU-intensive workloads

Simple applications

> **Note**
>
> For workflows that don't require visual design or extensive service integrations, Lambda durable functions may be a simpler alternative that keeps workflow logic in code within Lambda.

For applications that don't require complex orchestration, using Step Functions might add unnecessary complexity. For example, if you're simply processing messages from an Amazon SQS queue or responding to Amazon EventBridge events, you can configure these services to invoke your Lambda functions directly. Similarly, if your application consists of only one or two Lambda functions with straightforward error handling, direct Lambda invocation or event-driven architectures might be simpler to deploy and maintain.

Complex data processing

You can use the Step Functions Distributed Map state to concurrently process large Amazon S3 datasets with Lambda functions. This is effective for many large-scale parallel workloads, including processing semi-structured data like JSON or CSV files. However, for more complex data transformations or advanced analytics, consider these alternatives:

- **Data transformation pipelines**: Use AWS Glue for ETL jobs that process structured or semi-structured data from multiple sources. AWS Glue is particularly useful when you need built-in data catalog and schema management capabilities.
- **Data analytics:** Use Amazon EMR for petabyte-scale data analytics, especially when you need Apache Hadoop ecosystem tools or for machine learning workloads that exceed Lambda's memory limits.

CPU-intensive workloads

While Step Functions can orchestrate CPU-intensive tasks, Lambda functions may not be suitable for these workloads due to their limited CPU resources. For computationally intensive operations within your workflows, consider these alternatives:

- **Container orchestration:** Use Step Functions to manage Amazon Elastic Container Service (Amazon ECS) tasks for more consistent and scalable compute resources.

- **Batch processing:** Integrate AWS Batch with Step Functions for managing compute-intensive batch jobs that require sustained CPU usage.

Invoke a Lambda function with Amazon S3 batch events

You can use Amazon S3 batch operations to invoke a Lambda function on a large set of Amazon S3 objects. Amazon S3 tracks the progress of batch operations, sends notifications, and stores a completion report that shows the status of each action.

To run a batch operation, you create an Amazon S3 batch operations job. When you create the job, you provide a manifest (the list of objects) and configure the action to perform on those objects.

When the batch job starts, Amazon S3 invokes the Lambda function synchronously for each object in the manifest. The event parameter includes the names of the bucket and the object.

The following example shows the event that Amazon S3 sends to the Lambda function for an object that is named **customerImage1.jpg** in the **amzn-s3-demo-bucket** bucket.

Example Amazon S3 batch request event

```
{
"invocationSchemaVersion": "1.0",
    "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
    "job": {
        "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"
    },
    "tasks": [
        {
            "taskId": "dGFza2lkZ29lc2hlcmUK",
            "s3Key": "customerImage1.jpg",
            "s3VersionId": "1",
            "s3BucketArn": "arn:aws:s3:::amzn-s3-demo-bucket"
        }
    ]
}
```

Your Lambda function must return a JSON object with the fields as shown in the following example. You can copy the `invocationId` and `taskId` from the event parameter. You can return a string in the `resultString`. Amazon S3 saves the `resultString` values in the completion report.

Example Amazon S3 batch request response

```
{
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

Invoking Lambda functions from Amazon S3 batch operations

You can invoke the Lambda function with an unqualified or qualified function ARN. If you want to use the same function version for the entire batch job, configure a specific function version in the `FunctionARN` parameter when you create your job. If you configure an alias or the $LATEST qualifier, the batch job immediately starts calling the new version of the function if the alias or $LATEST is updated during the job execution.

Note that you can't reuse an existing Amazon S3 event-based function for batch operations. This is because the Amazon S3 batch operation passes a different event parameter to the Lambda function and expects a return message with a specific JSON structure.

In the resource-based policy that you create for the Amazon S3 batch job, ensure that you set permission for the job to invoke your Lambda function.

In the execution role for the function, set a trust policy for Amazon S3 to assume the role when it runs your function.

If your function uses the AWS SDK to manage Amazon S3 resources, you need to add Amazon S3 permissions in the execution role.

When the job runs, Amazon S3 starts multiple function instances to process the Amazon S3 objects in parallel, up to the concurrency limit of the function. Amazon S3 limits the initial ramp-up of instances to avoid excess cost for smaller jobs.

If the Lambda function returns a `TemporaryFailure` response code, Amazon S3 retries the operation.

For more information about Amazon S3 batch operations, see Performing batch operations in the *Amazon S3 Developer Guide*.

For an example of how to use a Lambda function in Amazon S3 batch operations, see Invoking a Lambda function from Amazon S3 batch operations in the *Amazon S3 Developer Guide*.

Invoking Lambda functions with Amazon SNS notifications

You can use a Lambda function to process Amazon Simple Notification Service (Amazon SNS) notifications. Amazon SNS supports Lambda functions as a target for messages sent to a topic. You can subscribe your function to topics in the same account or in other AWS accounts. For a detailed walkthrough, see the section called "Tutorial".

Lambda supports SNS triggers for standard SNS topics only. FIFO topics aren't supported.

Lambda processes SNS messages asynchronously by queuing the messages and handling retries. If Amazon SNS can't reach Lambda or the message is rejected, Amazon SNS retries at increasing intervals over several hours. For details, see Reliability in the Amazon SNS FAQs.

> **Warning**
>
> Lambda asynchronous invocations process each event at least once, and duplicate processing of records can occur. To avoid potential issues related to duplicate events, we strongly recommend that you make your function code idempotent. To learn more, see How do I make my Lambda function idempotent in the AWS Knowledge Center.

Idempotency utility from Powertools for AWS Lambda

The idempotency utility from Powertools for AWS Lambda makes your Lambda functions idempotent. It is available for Python, TypeScript, Java, and .NET. For more information, see Idempotency utility in the *Powertools for AWS Lambda (Python) documentation*, Idempotency Utility in the *Powertools for AWS Lambda (TypeScript) documentation*, Idempotency Utility in the *Powertools for AWS Lambda (Java) documentation*, and Idempotency Utility in the *Powertools for AWS Lambda (.NET) documentation*.

Topics

- Adding an Amazon SNS topic trigger for a Lambda function using the console
- Manually adding an Amazon SNS topic trigger for a Lambda function
- Sample SNS event shape
- Tutorial: Using AWS Lambda with Amazon Simple Notification Service

Adding an Amazon SNS topic trigger for a Lambda function using the console

To add an SNS topic as a trigger for a Lambda function, the easiest way is to use the Lambda console. When you add the trigger via the console, Lambda automatically sets up the necessary permissions and subscriptions to start receiving events from the SNS topic.

To add an SNS topic as a trigger for a Lambda function (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of a function you want to add the trigger for.
3. Choose **Configuration**, and then choose **Triggers**.
4. Choose **Add trigger**.
5. Under **Trigger configuration**, in the dropdown menu, choose **SNS**.
6. For **SNS topic**, choose the SNS topic to subscribe to.

Manually adding an Amazon SNS topic trigger for a Lambda function

To set up an SNS trigger for a Lambda function manually, you need to complete the following steps:

- Define a resource-based policy for your function to allow SNS to invoke it.
- Subscribe your Lambda function to the Amazon SNS topic.

> **Note**
>
> If your SNS topic and your Lambda function are in different AWS accounts, you also need to grant extra permissions to allow cross-account subscriptions to the SNS topic. For more information, see Grant cross-account permission for Amazon SNS subscription.

You can use the AWS Command Line Interface (AWS CLI) to complete both of these steps. First, to define a resource-based policy for a Lambda function that allows SNS invocations, use the following AWS CLI command. Be sure to replace the value of `--function-name` with your Lambda function name, and the value of `--source-arn` with your SNS topic ARN.

```
aws lambda add-permission --function-name example-function \
    --source-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com
```

To subscribe your function to the SNS topic, use the following AWS CLI command. Replace the value of `--topic-arn` with your SNS topic ARN, and the value of `--notification-endpoint` with your Lambda function ARN.

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-east-1:123456789012:function:example-
function
```

Sample SNS event shape

Amazon SNS invokes your function asynchronously with an event that contains a message and metadata.

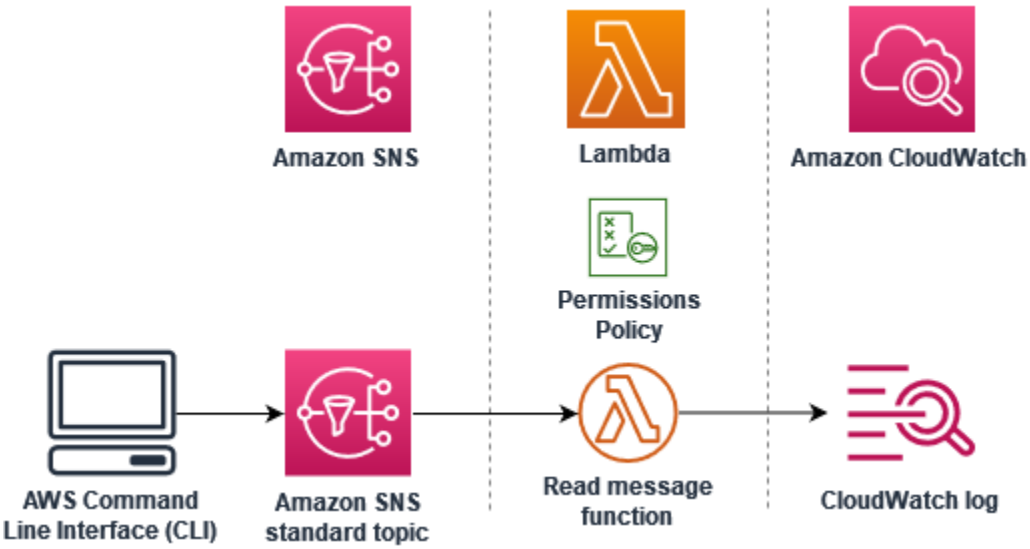
Example Amazon SNS message event

```
{
  "Records": [
    {
      "EventVersion": "1.0",
      "EventSubscriptionArn": "arn:aws:sns:us-east-1:123456789012:sns-lambda:21be56ed-
a058-49f5-8c98-aedd2564c486",
      "EventSource": "aws:sns",
      "Sns": {
        "SignatureVersion": "1",
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "SigningCertURL": "https://sns.us-east-1.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        "MessageAttributes": {
          "Test": {
            "Type": "String",
            "Value": "TestString"
```

```
          },
          "TestBinary": {
            "Type": "Binary",
            "Value": "TestBinary"
          }
        },
        "Type": "Notification",
        "UnsubscribeUrl": "https://sns.us-east-1.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-1:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
        "TopicArn":"arn:aws:sns:us-east-1:123456789012:sns-lambda",
        "Subject": "TestInvoke"
      }
    }
  ]
}
```

Tutorial: Using AWS Lambda with Amazon Simple Notification Service

In this tutorial, you use a Lambda function in one AWS account to subscribe to an Amazon Simple Notification Service (Amazon SNS) topic in a separate AWS account. When you publish messages to your Amazon SNS topic, your Lambda function reads the contents of the message and outputs it to Amazon CloudWatch Logs. To complete this tutorial, you use the AWS Command Line Interface (AWS CLI).



To complete this tutorial, you perform the following steps:

- In **account A**, create an Amazon SNS topic.

- In **account B**, create a Lambda function that will read messages from the topic.
- In **account B**, create a subscription to the topic.
- Publish messages to the Amazon SNS topic in **account A** and confirm that the Lambda function in **account B** outputs them to CloudWatch Logs.

By completing these steps, you will learn how to configure an Amazon SNS topic to invoke a Lambda function. You will also learn how to create an AWS Identity and Access Management (IAM) policy that gives permission for a resource in another AWS account to invoke Lambda.

In the tutorial, you use two separate AWS accounts. The AWS CLI commands illustrate this by using two named profiles called `accountA` and `accountB`, each configured for use with a different AWS account. To learn how to configure the AWS CLI to use different profiles, see Configuration and credential file settings in the *AWS Command Line Interface User Guide for Version 2*. Be sure to configure the same default AWS Region for both profiles.

If the AWS CLI profiles you create for the two AWS accounts use different names, or if you use the default profile and one named profile, modify the AWS CLI commands in the following steps as needed.

Prerequisites

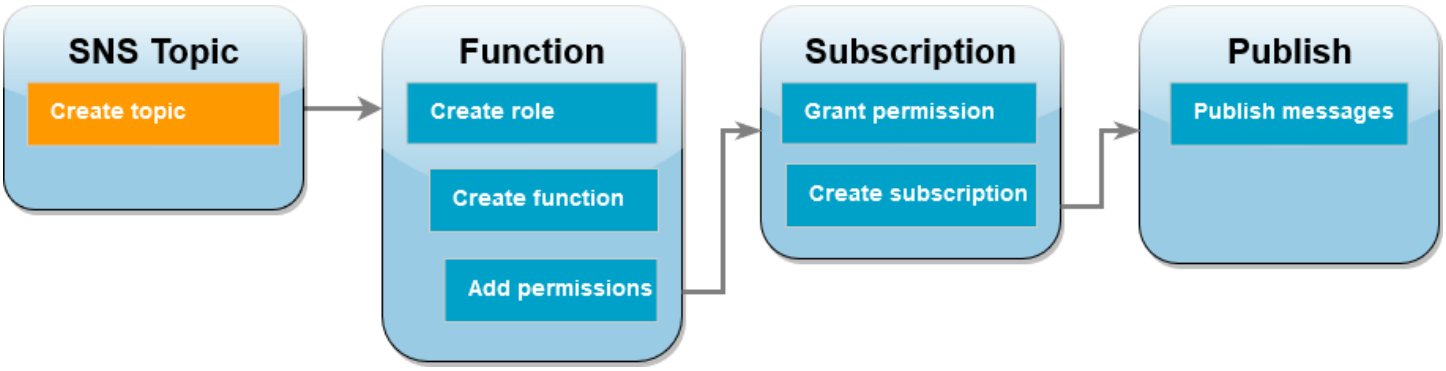
Install the AWS Command Line Interface

If you have not yet installed the AWS Command Line Interface, follow the steps at Installing or updating the latest version of the AWS CLI to install it.

The tutorial requires a command line terminal or shell to run commands. In Linux and macOS, use your preferred shell and package manager.

> **Note**
>
> In Windows, some Bash CLI commands that you commonly use with Lambda (such as `zip`) are not supported by the operating system's built-in terminals. To get a Windows-integrated version of Ubuntu and Bash, install the Windows Subsystem for Linux.

Create an Amazon SNS topic (account A)



To create the topic

- In **account A**, create an Amazon SNS standard topic using the following AWS CLI command.

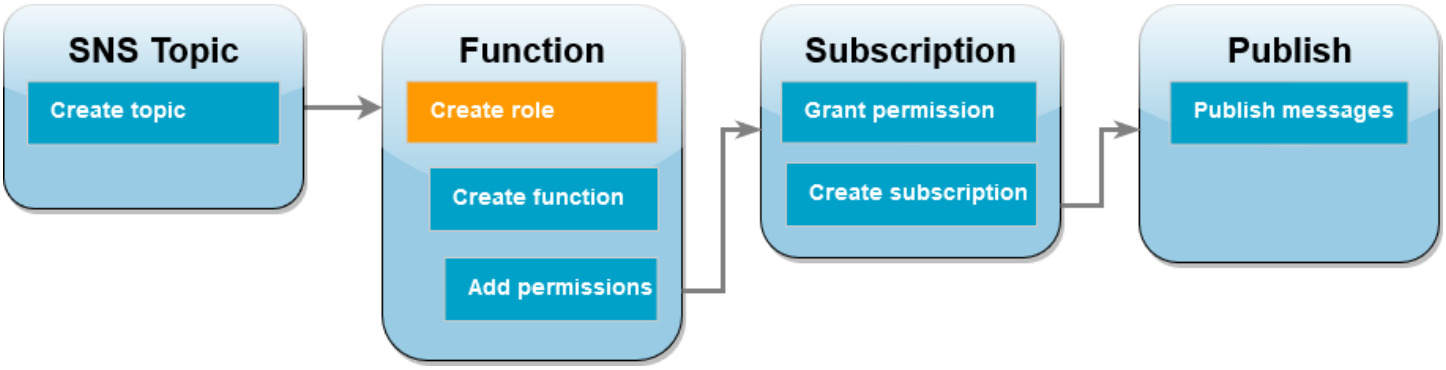
  ```
  aws sns create-topic --name sns-topic-for-lambda --profile accountA
  ```

  You should see output similar to the following.

  ```
  {
      "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
  }
  ```

  Make a note of the Amazon Resource Name (ARN) of your topic. You'll need it later in the tutorial when you add permissions to your Lambda function to subscribe to the topic.

Create a function execution role (account B)



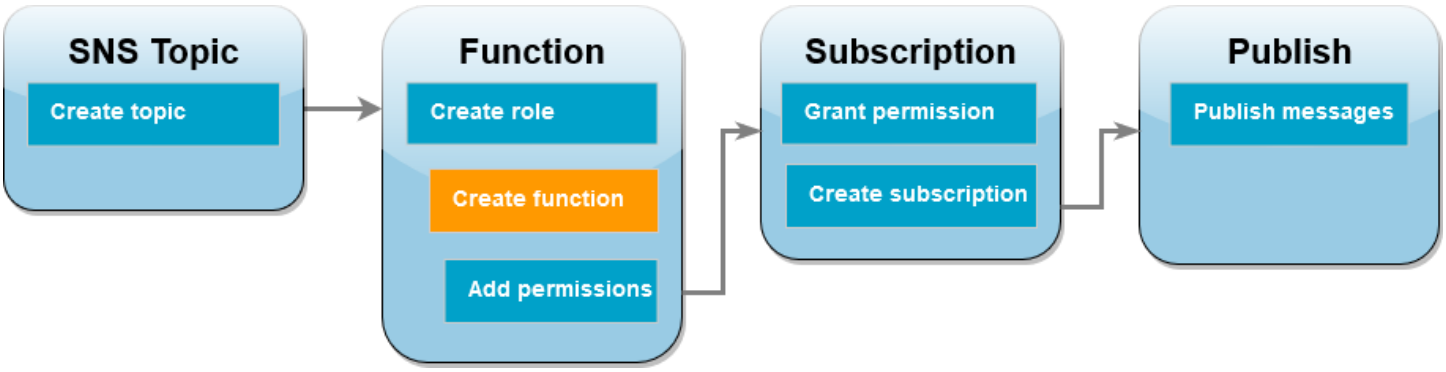
An execution role is an IAM role that grants a Lambda function permission to access AWS services and resources. Before you create your function in **account B**, you create a role that gives the

function basic permissions to write logs to CloudWatch Logs. We'll add the permissions to read from your Amazon SNS topic in a later step.

To create an execution role

1. In **account B** open the roles page in the IAM console.
2. Choose **Create role**.
3. For **Trusted entity type**, choose **AWS service**.
4. For **Use case**, choose **Lambda**.
5. Choose **Next**.
6. Add a basic permissions policy to the role by doing the following:
   a. In the **Permissions policies** search box, enter **`AWSLambdaBasicExecutionRole`**.
   b. Choose **Next**.
7. Finalize the role creation by doing the following:
   a. Under **Role details**, enter **`lambda-sns-role`** for **Role name**.
   b. Choose **Create role**.

Create a Lambda function (account B)



Create a Lambda function that processes your Amazon SNS messages. The function code logs the message contents of each record to Amazon CloudWatch Logs.

This tutorial uses the Node.js 24 runtime, but we've also provided example code in other runtime languages. You can select the tab in the following box to see code for the runtime you're interested in. The JavaScript code you'll use in this step is in the first example shown in the **JavaScript** tab.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");
```

```
            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
```

```
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;
```

```
    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SNS event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
```

```
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here
```

```
    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

To create the function

1. Create a directory for the project, and then switch to that directory.

```
mkdir sns-tutorial
cd sns-tutorial
```

2. Copy the sample JavaScript code into a new file named `index.js`.
3. Create a deployment package using the following `zip` command.

```
zip function.zip index.js
```

4. Run the following AWS CLI command to create your Lambda function in **account B**.

```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

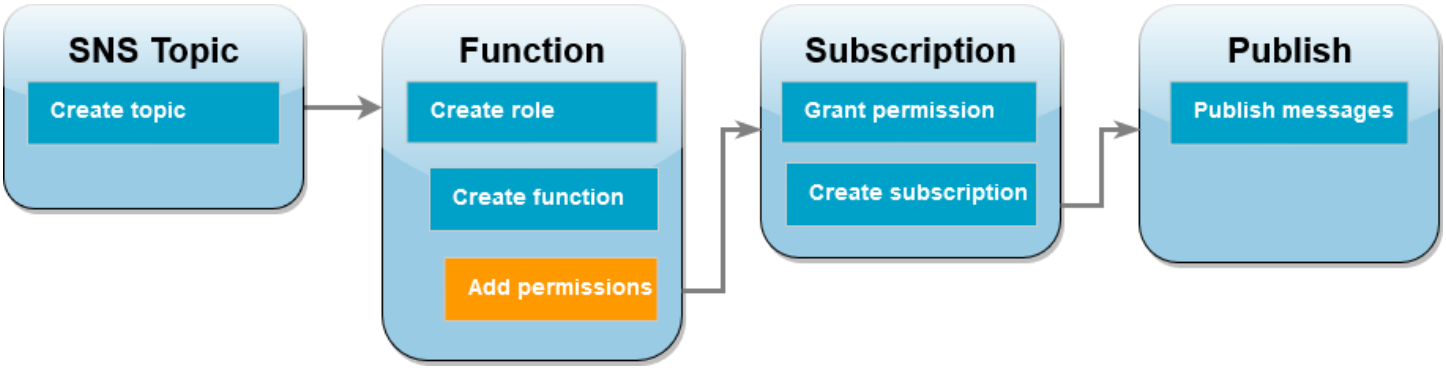
You should see output similar to the following.

```
{
    "FunctionName": "Function-With-SNS",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:Function-With-
SNS",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda_basic_role",
    "Handler": "index.handler",
    ...
```

```
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-
west-2::runtime:7d5f06b69c951da8a48b926ce280a9daf2e8bb1a74fc4a2672580c787d608206"
    }
}
```

5. Record the Amazon Resource Name (ARN) of your function. You'll need it later in the tutorial when you add permissions to allow Amazon SNS to invoke your function.

Add permissions to function (account B)



For Amazon SNS to invoke your function, you need to grant it permission in a statement on a resource-based policy. You add this statement using the AWS CLI `add-permission` command.

To grant Amazon SNS permission to invoke your function

- In **account B**, run the following AWS CLI command using the ARN for your Amazon SNS topic you recorded earlier.

```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

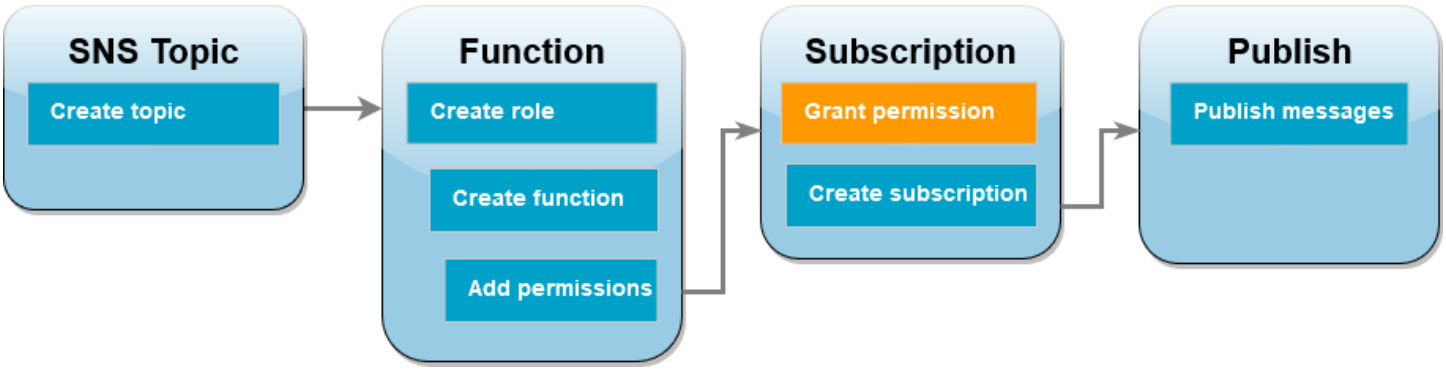
  You should see output similar to the following.

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
      \"arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda\"}},
      \"Action\":[\"lambda:InvokeFunction\"],
      \"Resource\":\"arn:aws:lambda:us-east-1:<AccountB_ID>:function:Function-With-
SNS\",
```

```
      \"Effect\":\"Allow\",\"Principal\":{\"Service\":\"sns.amazonaws.com\"},
      \"Sid\":\"function-with-sns\"}"
}
```

> **Note**
>
> If the account with the Amazon SNS topic is hosted in an opt-in AWS Region, you need to specify the region in the principal. For example, if you're working with an Amazon SNS topic in the Asia Pacific (Hong Kong) region, you need to specify `sns.ap-east-1.amazonaws.com` instead of `sns.amazonaws.com` for the principal.

Grant cross-account permission for Amazon SNS subscription (account A)



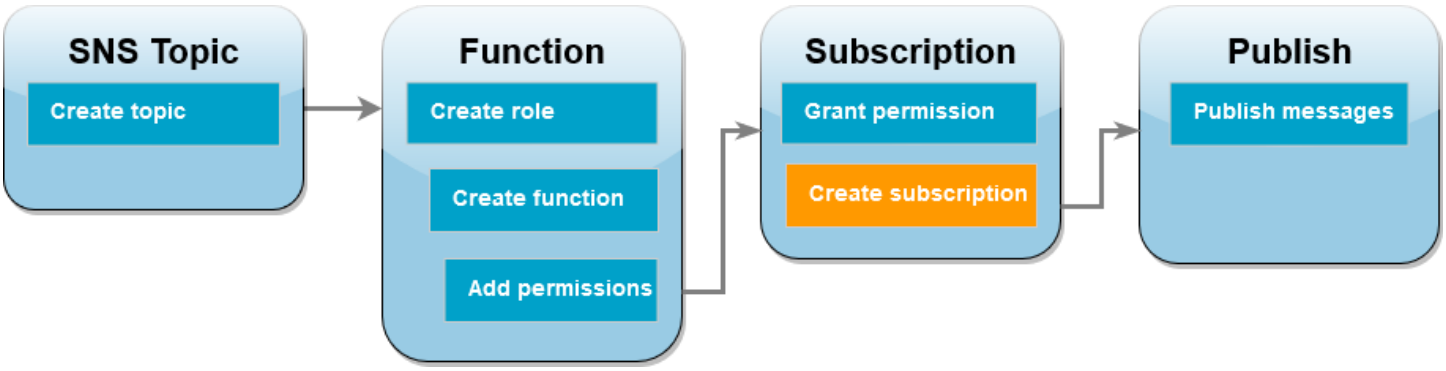
For your Lambda function in **account B** to subscribe to the Amazon SNS topic you created in **account A**, you need to grant permission for **account B** to subscribe to your topic. You grant this permission using the AWS CLI `add-permission` command.

To grant permission for account B to subscribe to the topic

- In **account A**, run the following AWS CLI command. Use the ARN for the Amazon SNS topic you recorded earlier.

```
aws sns add-permission --label lambda-access --aws-account-id <AccountB_ID> \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --action-name Subscribe ListSubscriptionsByTopic --profile accountA
```

Create a subscription (account B)



In **account B**, you now subscribe your Lambda function to the Amazon SNS topic you created at the beginning of the tutorial in **account A**. When a message is sent to this topic (`sns-topic-for-lambda`), Amazon SNS invokes your Lambda function `Function-With-SNS` in **account B**.

To create a subscription

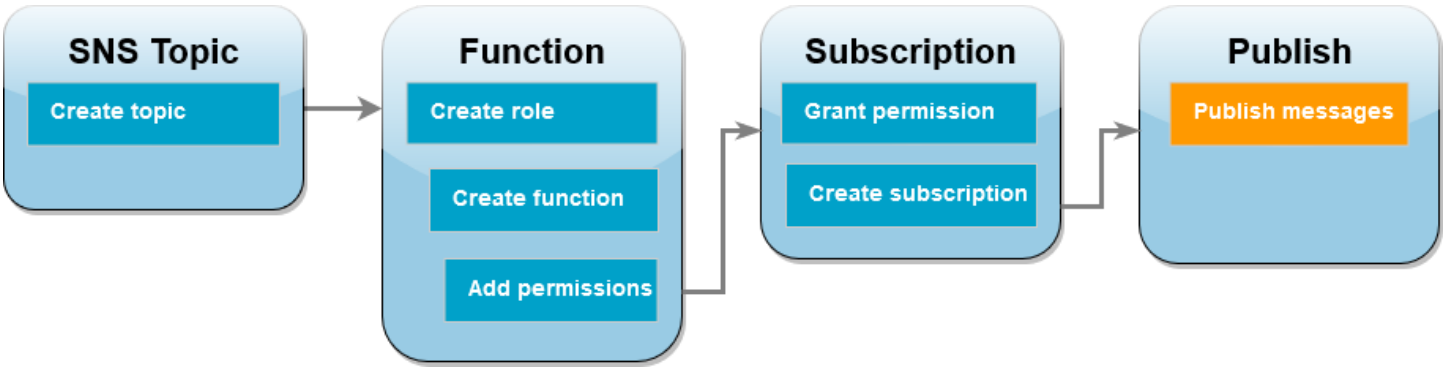
- In **account B**, run the following AWS CLI command. Use your default region you created your topic in and the ARNs for your topic and Lambda function.

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-
east-1:<AccountB_ID>:function:Function-With-SNS \
    --profile accountB
```

You should see output similar to the following.

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

Publish messages to topic (account A and account B)



Now that your Lambda function in **account B** is subscribed to your Amazon SNS topic in **account A**, it's time to test your setup by publishing messages to your topic. To confirm that Amazon SNS has invoked your Lambda function, you use CloudWatch Logs to view your function's output.

To publish a message to your topic and view your function's output

1. Enter `Hello World` into a text file and save it as `message.txt`.
2. From the same directory you saved your text file in, run the following AWS CLI command in **account A**. Use the ARN for your own topic.

```
aws sns publish --message file://message.txt --subject Test \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --profile accountA
```

   This will return a message ID with a unique identifier, indicating that Amazon SNS has accepted the message. Amazon SNS then attempts to deliver the message to the topic's subscribers. To confirm that Amazon SNS has invoked your Lambda function, use CloudWatch Logs to view your function's output:
3. In **account B**, open the Log groups page of the Amazon CloudWatch console.
4. Choose the log group for your function (`/aws/lambda/Function-With-SNS`).
5. Choose the most recent log stream.
6. If your function was correctly invoked, you'll see output similar to the following showing the contents of the message you published to your topic.

```
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO Processed
 message Hello World
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO done
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

In **Account A**, clean up your Amazon SNS topic.

To delete the Amazon SNS topic

1. Open the Topics page of the Amazon SNS console.
2. Select the topic you created.
3. Choose **Delete**.
4. Enter **`delete me`** in the text input field.
5. Choose **Delete**.

In **Account B**, clean up your execution role, Lambda function, and Amazon SNS subscription.

To delete the execution role

1. Open the Roles page of the IAM console.
2. Select the execution role that you created.
3. Choose **Delete**.
4. Enter the name of the role in the text input field and choose **Delete**.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the Amazon SNS subscription

1. Open the Subscriptions page of the Amazon SNS console.
2. Select the subscription you created.

3. Choose **Delete**, **Delete**.

Managing permissions in AWS Lambda

You can use AWS Identity and Access Management (IAM) to manage permissions in AWS Lambda. There are two main categories of permissions that you need to consider when working with Lambda functions:

- Permissions that your Lambda functions need to perform API actions and access other AWS resources
- Permissions that other AWS users and entities need to access your Lambda functions

Lambda functions often need to access other AWS resources, and perform various API operations on those resources. For example, you might have a Lambda function that responds to an event by updating entries in an Amazon DynamoDB database. In this case, your function needs permissions to access the database, as well as permissions to put or update items in that database.

You define the permissions that your Lambda function needs in a special IAM role called an execution role. In this role, you can attach a policy that defines every permission your function needs to access other AWS resources, and read from event sources. Every Lambda function must have an execution role. At a minimum, your execution role must have access to Amazon CloudWatch because Lambda functions log to CloudWatch Logs by default. You can attach the `AWSLambdaBasicExecutionRole` managed policy to your execution role to satisfy this requirement.

To give other AWS accounts, organizations, and services permissions to access your Lambda resources, you have a few options:

- You can use identity-based policies to grant other users access to your Lambda resources. Identity-based policies can apply to users directly, or to groups and roles that are associated with a user.
- You can use resource-based policies to give other accounts and AWS services permissions to access your Lambda resources. When a user tries to access a Lambda resource, Lambda considers both the user's identity-based policies and the resource's resource-based policy. When an AWS service such as Amazon Simple Storage Service (Amazon S3) calls your Lambda function, Lambda considers only the resource-based policy.
- You can use an attribute-based access control (ABAC) model to control access to your Lambda functions. With ABAC, you can attach tags to a Lambda function, pass them in certain API

requests, or attach them to the IAM principal making the request. Specify the same tags in the condition element of an IAM policy to control function access.

In AWS, it's a best practice to grant only the permissions required to perform a task (least-privilege permissions). To implement this in Lambda, we recommend starting with an AWS managed policy. You can use these managed policies as-is, or as a starting point for writing your own more restrictive policies.

To help you fine-tune your permissions for least-privilege access, Lambda provides some additional conditions you can include in your policies. For more information, see the section called “Resources and Conditions”.

For more information about IAM, see the *IAM User Guide*.

Defining Lambda function permissions with an execution role

A Lambda function's execution role is an AWS Identity and Access Management (IAM) role that grants the function permission to access AWS services and resources. For example, you might create an execution role that has permission to send logs to Amazon CloudWatch and upload trace data to AWS X-Ray. This page provides information on how to create, view, and manage a Lambda function's execution role.

Lambda automatically assumes your execution role when you invoke your function. You should avoid manually calling `sts:AssumeRole` to assume the execution role in your function code. If your use case requires that the role assumes itself, you must include the role itself as a trusted principal in your role's trust policy. For more information on how to modify a role trust policy, see Modifying a role trust policy (console) in the IAM User Guide.

In order for Lambda to properly assume your execution role, the role's trust policy must specify the Lambda service principal (`lambda.amazonaws.com`) as a trusted service.

Topics

- Creating an execution role in the IAM console
- Creating and managing roles with the AWS CLI
- Grant least privilege access to your Lambda execution role
- Viewing and updating permissions in the execution role
- Working with AWS managed policies in the execution role
- Using source function ARN to control function access behavior

Creating an execution role in the IAM console

By default, Lambda creates an execution role with minimal permissions when you create a function in the Lambda console. Specifically, this execution role includes the `AWSLambdaBasicExecutionRole` managed policy, which gives your function basic permissions to log events to Amazon CloudWatch Logs. You can select **Create default role** in the **Permissions** section.

You can choose an existing role by selecting **Use another role** in the **Permissions** section. If your Lambda function needs additional permissions to perform tasks such as updating entries in an Amazon DynamoDB database in response to events, you can create a custom execution role with

the necessary permissions. To do this, select **Use another role** in the **Permissions** section, which opens a drawer where you can customize your permissions.

To configure an execution role from Console

1. Enter a **role name** in the Role details section.
2. In the **Policy** section, select **Use existing policy**.
3. Select the AWS managed policies that you want to attach to your role. For example, if your function needs to access DynamoDB, select the **AWSLambdaDynamoDBExecutionRole** managed policy.
4. Choose **Create role**.

Alternatively, when you create a function in the Lambda console, you can attach any execution role that you previously created to the function. If you want to attach a new execution role to an existing function, follow the steps in Updating a function's execution role.

Creating and managing roles with the AWS CLI

To create an execution role with the AWS Command Line Interface (AWS CLI), use the **create-role** command. When using this command, you can specify the trust policy inline. A role's trust policy gives the specified principals permission to assume the role. In the following example, you grant the Lambda service principal permission to assume your role. Note that requirements for escaping quotes in the JSON string may vary depending on your shell.

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document '{"Version": "2012-10-17",       "Statement":
 [{ "Effect": "Allow", "Principal": {"Service": "lambda.amazonaws.com"}, "Action":
 "sts:AssumeRole"}]}'
```

You can also define the trust policy for the role using a separate JSON file. In the following example, `trust-policy.json` is a file in the current directory.

Example trust-policy.json

JSON

```
{
```

```
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document file://trust-policy.json
```

To add permissions to the role, use the **attach-policy-to-role** command. The following command adds the `AWSLambdaBasicExecutionRole` managed policy to the `lambda-ex` execution role.

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

After you create your execution role, attach it to your function. When you create a function in the Lambda console, you can attach any execution role that you previously created to the function. If you want to attach a new execution role to an existing function, follow the steps in the section called “Updating a function's execution role”.

Grant least privilege access to your Lambda execution role

When you first create an IAM role for your Lambda function during the development phase, you might sometimes grant permissions beyond what is required. Before publishing your function in the production environment, as a best practice, adjust the policy to include only the required permissions. For more information, see Apply least-privilege permissions in the *IAM User Guide*.

Use IAM Access Analyzer to help identify the required permissions for the IAM execution role policy. IAM Access Analyzer reviews your AWS CloudTrail logs over the date range that you specify and generates a policy template with only the permissions that the function used during that time. You can use the template to create a managed policy with fine-grained permissions, and then attach it

to the IAM role. That way, you grant only the permissions that the role needs to interact with AWS resources for your specific use case.

For more information, see Generate policies based on access activity in the *IAM User Guide*.

Viewing and updating permissions in the execution role

This topic covers how you can view and update your function's execution role.

Topics

- Viewing a function's execution role
- Updating a function's execution role

Viewing a function's execution role

To view a function's execution role, use the Lambda console.

To view a function's execution role (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose **Configuration**, and then choose **Permissions**.
4. Under **Execution role**, you can view the role that's currently being used as the function's execution role. For convenience, you can view all the resources and actions that the function can access under the **Resource summary** section. You can also choose a service from the dropdown list to see all permissions related to that service.

Updating a function's execution role

You can add or remove permissions from a function's execution role at any time, or configure your function to use a different role. If your function needs access to any other services or resources, you must add the necessary permissions to the execution role.

When you add permissions to your function, perform a trivial update to its code or configuration as well. This forces running instances of your function, which have outdated credentials, to stop and be replaced.

To update a function's execution role, you can use the Lambda console.

To update a function's execution role (console)

1. Open the Functions page of the Lambda console.
2. Choose the name of a function.
3. Choose **Configuration**, and then choose **Permissions**.
4. Under **Execution role**, choose **Edit**.
5. If you want to update your function to use a different role as the execution role, choose the new role in the dropdown menu under **Existing role**.

   > **Note**
   >
   > If you want to update the permissions within an existing execution role, you can only do so in the AWS Identity and Access Management (IAM) console.

   If you want to create a new role to use as the execution role, choose **Create a new role from AWS policy templates** under **Execution role**. Then, enter a name for your new role under **Role name**, and specify any policies you want to attach to the new role under **Policy templates**.
6. Choose **Save**.

Working with AWS managed policies in the execution role

The following AWS managed policies provide permissions that are required to use Lambda features.

Change	Description	Date
AWSLambdaMSKExecutionRole – Lambda added the kafka:DescribeClusterV2 permission to this policy.	`AWSLambdaMSKExecutionRole` grants permissions to read and access records from an Amazon Managed Streaming for Apache Kafka (Amazon MSK) cluster, manage elastic network interfaces (ENIs), and write to CloudWatch Logs.	June 17, 2022

Change	Description	Date
AWSLambdaBasicExecutionRole – Lambda started tracking changes to this policy.	`AWSLambdaBasicExecutionRole` grants permissions to upload logs to CloudWatch.	February 14, 2022
AWSLambdaDynamoDBExecutionRole – Lambda started tracking changes to this policy.	`AWSLambdaDynamoDBExecutionRole` grants permissions to read records from an Amazon DynamoDB stream and write to CloudWatch Logs.	February 14, 2022
AWSLambdaKinesisExecutionRole – Lambda started tracking changes to this policy.	`AWSLambdaKinesisExecutionRole` grants permissions to read events from an Amazon Kinesis data stream and write to CloudWatch Logs.	February 14, 2022
AWSLambdaMSKExecutionRole – Lambda started tracking changes to this policy.	`AWSLambdaMSKExecutionRole` grants permissions to read and access records from an Amazon Managed Streaming for Apache Kafka (Amazon MSK) cluster, manage elastic network interfaces (ENIs), and write to CloudWatch Logs.	February 14, 2022

Change	Description	Date
AWSLambdaSQSQueueExecutionRole – Lambda started tracking changes to this policy.	`AWSLambdaSQSQueueExecutionRole` grants permissions to read a message from an Amazon Simple Queue Service (Amazon SQS) queue and write to CloudWatch Logs.	February 14, 2022
AWSLambdaVPCAccessExecutionRole – Lambda started tracking changes to this policy.	`AWSLambdaVPCAccessExecutionRole` grants permissions to manage ENIs within an Amazon VPC and write to CloudWatch Logs.	February 14, 2022
AWSXRayDaemonWriteAccess – Lambda started tracking changes to this policy.	`AWSXRayDaemonWriteAccess` grants permissions to upload trace data to X-Ray.	February 14, 2022
CloudWatchLambdaInsightsExecutionRolePolicy – Lambda started tracking changes to this policy.	`CloudWatchLambdaInsightsExecutionRolePolicy` grants permissions to write runtime metrics to CloudWatch Lambda Insights.	February 14, 2022
AmazonS3ObjectLambdaExecutionRolePolicy – Lambda started tracking changes to this policy.	`AmazonS3ObjectLambdaExecutionRolePolicy` grants permissions to interact with Amazon Simple Storage Service (Amazon S3) object Lambda and to write to CloudWatch Logs.	February 14, 2022

For some features, the Lambda console attempts to add missing permissions to your execution role in a customer managed policy. These policies can become numerous. To avoid creating extra policies, add the relevant AWS managed policies to your execution role before enabling features.

When you use an event source mapping to invoke your function, Lambda uses the execution role to read event data. For example, an event source mapping for Kinesis reads events from a data stream and sends them to your function in batches.

When a service assumes a role in your account, you can include the `aws:SourceAccount` and `aws:SourceArn` global condition context keys in your role trust policy to limit access to the role to only requests that are generated by expected resources. For more information, see Cross-service confused deputy prevention for AWS Security Token Service.

In addition to the AWS managed policies, the Lambda console provides templates for creating a custom policy with permissions for additional use cases. When you create a function in the Lambda console, you can choose to create a new execution role with permissions from one or more templates. These templates are also applied automatically when you create a function from a blueprint, or when you configure options that require access to other services. Example templates are available in this guide's GitHub repository.

Using source function ARN to control function access behavior

It's common for your Lambda function code to make API requests to other AWS services. To make these requests, Lambda generates an ephemeral set of credentials by assuming your function's execution role. These credentials are available as environment variables during your function's invocation. When working with AWS SDKs, you don't need to provide credentials for the SDK directly in code. By default, the credential provider chain sequentially checks each place where you can set credentials and selects the first one available—usually the environment variables (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and `AWS_SESSION_TOKEN`).

Lambda injects the source function ARN into the credentials context if the request is an AWS API request that comes from within your execution environment. Lambda also injects the source function ARN for the following AWS API requests that Lambda makes on your behalf outside of your execution environment:

Service	Action	Reason
CloudWatch Logs	`CreateLogGroup` , `CreateLogStream` , `PutLogEvents`	To store logs into a CloudWatch Logs log group
X-Ray	`PutTraceSegments`	To send trace data to X-Ray
Amazon EFS	`ClientMount`	To connect your function to an Amazon Elastic File System (Amazon EFS) file system

Other AWS API calls that Lambda makes outside of your execution environment on your behalf using the same execution role don't contain the source function ARN. Examples of such API calls outside the execution environment include:

- Calls to AWS Key Management Service (AWS KMS) to automatically encrypt and decrypt your environment variables.
- Calls to Amazon Elastic Compute Cloud (Amazon EC2) to create elastic network interfaces (ENIs) for a VPC-enabled function.
- Calls to AWS services, such as Amazon Simple Queue Service (Amazon SQS), to read from an event source that's set up as an event source mapping.

With the source function ARN in the credentials context, you can verify whether a call to your resource came from a specific Lambda function's code. To verify this, use the `lambda:SourceFunctionArn` condition key in an IAM identity-based policy or service control policy (SCP).

> **Note**
>
> You cannot use the `lambda:SourceFunctionArn` condition key in resource-based policies.

With this condition key in your identity-based policies or SCPs, you can implement security controls for the API actions that your function code makes to other AWS services. This has a few key security applications, such as helping you identify the source of a credential leak.

> **Note**
>
> The `lambda:SourceFunctionArn` condition key is different from the `lambda:FunctionArn` and `aws:SourceArn` condition keys. The `lambda:FunctionArn` condition key applies only to event source mappings and helps define which functions your event source can invoke. The `aws:SourceArn` condition key applies only to policies where your Lambda function is the target resource, and helps define which other AWS services and resources can invoke that function. The `lambda:SourceFunctionArn` condition key can apply to any identity-based policy or SCP to define the specific Lambda functions that have permissions to make specific AWS API calls to other resources.

To use `lambda:SourceFunctionArn` in your policy, include it as a condition with any of the ARN condition operators. The value of the key must be a valid ARN.

For example, suppose your Lambda function code makes an `s3:PutObject` call that targets a specific Amazon S3 bucket. You might want to allow only one specific Lambda function to have `s3:PutObject` access that bucket. In this case, your function's execution role should have a policy attached that looks like this:

Example policy granting a specific Lambda function access to an Amazon S3 resource

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleSourceFunctionArn",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
```

```
                }
            }
        }
    ]
}
```

This policy allows only `s3:PutObject` access if the source is the Lambda function with ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda`. This policy doesn't allow `s3:PutObject` access to any other calling identity. This is true even if a different function or entity makes an `s3:PutObject` call with the same execution role.

> **Note**
>
> The `lambda:SourceFunctionARN` condition key doesn't support Lambda function versions or function aliases. If you use the ARN for a particular function version or alias, your function won't have permission to take the action you specify. Be sure to use the unqualified ARN for your function without a version or alias suffix.

You can also use `lambda:SourceFunctionArn` in SCPs. For example, suppose you want to restrict access to your bucket to either a single Lambda function's code or to calls from a specific Amazon Virtual Private Cloud (VPC). The following SCP illustrates this.

Example policy denying access to Amazon S3 under specific conditions

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
```

```
                        "vpc-12345678"
                    ]
                }
            }
        },
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

This policy denies all S3 actions unless they come from a specific Lambda function with ARN `arn:aws:lambda:*:123456789012:function:source_lambda`, or unless they come from the specified VPC. The `StringNotEqualsIfExists` operator tells IAM to process this condition only if the `aws:SourceVpc` key is present in the request. Similarly, IAM considers the `ArnNotEqualsIfExists` operator only if the `lambda:SourceFunctionArn` exists.

Granting other AWS entities access to your Lambda functions

To give other AWS accounts, organizations, and services permissions to access your Lambda resources, you have a few options:

- You can use identity-based policies to grant other users access to your Lambda resources. Identity-based policies can apply to users directly, or to groups and roles that are associated with a user.
- You can use resource-based policies to give other accounts and AWS services permissions to access your Lambda resources. When a user tries to access a Lambda resource, Lambda considers both the user's identity-based policies and the resource's resource-based policy. When an AWS service such as Amazon Simple Storage Service (Amazon S3) calls your Lambda function, Lambda considers only the resource-based policy.
- You can use an attribute-based access control (ABAC) model to control access to your Lambda functions. With ABAC, you can attach tags to a Lambda function, pass them in certain API requests, or attach them to the IAM principal making the request. Specify the same tags in the condition element of an IAM policy to control function access.

To help you fine-tune your permissions for least-privilege access, Lambda provides some additional conditions you can include in your policies. For more information, see the section called “Resources and Conditions”.

Identity-based IAM policies for Lambda

You can use identity-based policies in AWS Identity and Access Management (IAM) to grant users in your account access to Lambda. Identity-based policies can apply to users, user groups, or roles. You can also grant users in another account permission to assume a role in your account and access your Lambda resources.

Lambda provides AWS managed policies that grant access to Lambda API actions and, in some cases, access to other AWS services used to develop and manage Lambda resources. Lambda updates these managed policies as needed to ensure that your users have access to new features when they're released.

- AWSLambda_FullAccess – Grants full access to Lambda actions and other AWS services used to develop and maintain Lambda resources.
- AWSLambda_ReadOnlyAccess – Grants read-only access to Lambda resources.

- AWSLambdaRole – Grants permissions to invoke Lambda functions.

AWS managed policies grant permission to API actions without restricting the Lambda functions or layers that a user can modify. For finer-grained control, you can create your own policies that limit the scope of a user's permissions.

Topics

- Granting users access to a Lambda function
- Granting users access to a Lambda layer

Granting users access to a Lambda function

Use identity-based policies to allow users, user groups, or roles to perform operations on Lambda functions.

> **Note**
>
> For a function defined as a container image, the user permission to access the image must be configured in Amazon Elastic Container Registry (Amazon ECR). For an example, see Amazon ECR repository policies.

The following shows an example of a permissions policy with limited scope. It allows a user to create and manage Lambda functions named with a designated prefix (`intern-`), and configured with a designated execution role.

Example Function development policy

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetAccountSettings",
```

```
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DevelopFunctions",
            "Effect": "Allow",
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
        },
        {
            "Sid": "DevelopEventSourceMappings",
            "Effect": "Allow",
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "ArnLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
        },
        {
            "Sid": "PassExecutionRole",
            "Effect": "Allow",
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
```

```
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
        },
        {
            "Sid": "ViewLogs",
            "Effect": "Allow",
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
        }
    ]
}
```

The permissions in the policy are organized into statements based on the resources and conditions that they support.

- `ReadOnlyPermissions` – The Lambda console uses these permissions when you browse and view functions. They don't support resource patterns or conditions.

```
        "Action": [
            "lambda:GetAccountSettings",
            "lambda:GetEventSourceMapping",
            "lambda:GetFunction",
            "lambda:GetFunctionConfiguration",
            "lambda:GetFunctionCodeSigningConfig",
            "lambda:GetFunctionConcurrency",
            "lambda:ListEventSourceMappings",
            "lambda:ListFunctions",
            "lambda:ListTags",
            "iam:ListRoles"
        ],
        "Resource": "*"
```

- `DevelopFunctions` – Use any Lambda action that operates on functions prefixed with `intern-`, except `AddPermission` and `PutFunctionConcurrency`. `AddPermission` modifies the resource-based policy on the function and can have security implications.

`PutFunctionConcurrency` reserves scaling capacity for a function and can take capacity away from other functions.

```
        "NotAction": [
            "lambda:AddPermission",
            "lambda:PutFunctionConcurrency"
        ],
        "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- `DevelopEventSourceMappings` – Manage event source mappings on functions that are prefixed with `intern-`. These actions operate on event source mappings, but you can restrict them by function with a *condition*.

```
        "Action": [
            "lambda:DeleteEventSourceMapping",
            "lambda:UpdateEventSourceMapping",
            "lambda:CreateEventSourceMapping"
        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
            }
        }
```

- `PassExecutionRole` – View and pass only a role named `intern-lambda-execution-role`, which must be created and managed by a user with IAM permissions. `PassRole` is used when you assign an execution role to a function.

```
        "Action": [
            "iam:ListRolePolicies",
            "iam:ListAttachedRolePolicies",
            "iam:GetRole",
            "iam:GetRolePolicy",
            "iam:PassRole",
            "iam:SimulatePrincipalPolicy"
        ],
        "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- `ViewLogs` – Use CloudWatch Logs to view logs for functions that are prefixed with `intern-`.

```
        "Action": [
            "logs:*"
        ],
        "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

This policy allows a user to get started with Lambda, without putting other users' resources at risk. It doesn't allow a user to configure a function to be triggered by or call other AWS services, which requires broader IAM permissions. It also doesn't include permission to services that don't support limited-scope policies, like CloudWatch and X-Ray. Use the read-only policies for these services to give the user access to metrics and trace data.

When you configure triggers for your function, you need access to use the AWS service that invokes your function. For example, to configure an Amazon S3 trigger, you need permission to use the Amazon S3 actions that manage bucket notifications. Many of these permissions are included in the AWSLambda_FullAccess managed policy.

Granting users access to a Lambda layer

Use identity-based policies to allow users, user groups, or roles to perform operations on Lambda layers. The following policy grants a user permission to create layers and use them with functions. The resource patterns allow the user to work in any AWS Region and with any layer version, as long as the name of the layer starts with `test-`.

Example layer development policy

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "PublishLayers",
            "Effect": "Allow",
            "Action": [
                "lambda:PublishLayerVersion"
            ],
```

```
            "Resource": "arn:aws:lambda:*:*:layer:test-*"
        },
        {
            "Sid": "ManageLayerVersions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetLayerVersion",
                "lambda:DeleteLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*:*"
        }
    ]
}
```

You can also enforce layer use during function creation and configuration with the `lambda:Layer` condition. For example, you can prevent users from using layers published by other accounts. The following policy adds a condition to the `CreateFunction` and `UpdateFunctionConfiguration` actions to require that any layers specified come from account `123456789012`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ConfigureFunctions",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringLike": {
                    "lambda:Layer": [
                        "arn:aws:lambda:*:123456789012:layer:*:*"
                    ]
                }
            }
        }
```

```
    ]
}
```

To ensure that the condition applies, verify that no other statements grant the user permission to these actions.

Viewing resource-based IAM policies in Lambda

Lambda supports resource-based permissions policies for Lambda functions and layers. You can use resource-based policies to grant access to other AWS accounts, organizations, or services. Resource-based policies apply to a single function, version, alias, or layer version.

Console

To view a function's resource-based policy

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Permissions**.
4. Scroll down to **Resource-based policy** and then choose **View policy document**. The resource-based policy shows the permissions that are applied when another account or AWS service attempts to access the function. The following example shows a statement that allows Amazon S3 to invoke a function named `my-function` for a bucket named `amzn-s3-demo-bucket` in account `123456789012`.

   Example resource-based policy

```
{
    "Version":"2012-10-17",
    "Id": "default",
    "Statement": [
        {
            "Sid": "lambda-allow-s3-my-function",
            "Effect": "Allow",
            "Principal": {
              "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
```

```
            "Resource":  "arn:aws:lambda:us-east-2:123456789012:function:my-
function",
            "Condition": {
              "StringEquals": {
                "AWS:SourceAccount": "123456789012"
              },
              "ArnLike": {
                "AWS:SourceArn": "arn:aws:s3:::amzn-s3-demo-bucket"
              }
            }
        }
    ]
}
```

AWS CLI

To view a function's resource-based policy, use the `get-policy` command.

```
aws lambda get-policy \
  --function-name my-function \
  --output text
```

You should see the following output:

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"s3.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambd
east-2:123456789012:function:my-function","Condition":{"ArnLike":
{"AWS:SourceArn":"arn:aws:sns:us-east-2:123456789012:lambda*"}}}]}
```

For versions and aliases, append the version number or alias to the function name.

```
aws lambda get-policy --function-name my-function:PROD
```

To remove permissions from your function, use `remove-permission`.

```
aws lambda remove-permission \
  --function-name example \
```

```
  --statement-id sns
```

Use the `get-layer-version-policy` command to view the permissions on a layer.

```
aws lambda get-layer-version-policy \
  --layer-name my-layer \
  --version-number 3 \
  --output text
```

You should see the following output:

```
b0cd9796-d4eb-4564-939f-de7fe0b42236    {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:
west-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

Use `remove-layer-version-permission` to remove statements from the policy.

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3
 --statement-id engineering-org
```

Supported API actions

The following Lambda API actions support resource-based policies:

- CreateAlias
- DeleteAlias
- DeleteFunction
- DeleteFunctionConcurrency
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction
- GetFunctionConcurrency
- GetFunctionConfiguration

- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig
- Invoke
- InvokeFunctionUrl (permission only)
- ListAliases
- ListFunctionEventInvokeConfigs
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- TagResource
- UntagResource
- UpdateAlias
- UpdateFunctionCode
- UpdateFunctionEventInvokeConfig

Granting Lambda function access to AWS services

When you use an AWS service to invoke your function, you grant permission in a statement on a resource-based policy. You can apply the statement to the entire function, or limit the statement to a single version or alias.

> **Note**
>
> When you add a trigger to your function with the Lambda console, the console updates the function's resource-based policy to allow the service to invoke it. To grant permissions to other accounts or services that aren't available in the Lambda console, you can use the AWS CLI.

Add a statement with the add-permission command. The simplest resource-based policy statement allows a service to invoke a function. The following command grants Amazon Simple Notification Service permission to invoke a function named `my-function`.

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns \
  --principal sns.amazonaws.com \
  --output text
```

You should see the following output:

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

This lets Amazon SNS call the Invoke API action on the function, but it doesn't restrict the Amazon SNS topic that triggers the invocation. To ensure that your function is only invoked by a specific resource, specify the Amazon Resource Name (ARN) of the resource with the `source-arn` option. The following command only allows Amazon SNS to invoke the function for subscriptions to a topic named `my-topic`.

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns-my-topic \
  --principal sns.amazonaws.com \
  --source-arn arn:aws:sns:us-east-2:123456789012:my-topic
```

Some services can invoke functions in other accounts. If you specify a source ARN that has your account ID in it, that isn't an issue. For Amazon S3, however, the source is a bucket whose ARN doesn't have an account ID in it. It's possible that you could delete the bucket and another account could create a bucket with the same name. Use the `source-account` option with your account ID to ensure that only resources in your account can invoke the function.

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id s3-account \
```

```
  --principal s3.amazonaws.com \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

Granting function access to an organization

To grant permissions to an organization in AWS Organizations, specify the organization ID as the `principal-org-id`. The following add-permission command grants invocation access to all users in organization `o-a1b2c3d4e5f`.

```
aws lambda add-permission \
  --function-name example \
  --statement-id PrincipalOrgIDExample \
  --action lambda:InvokeFunction \
  --principal * \
  --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> In this command, `Principal` is `*`. This means that all users in the organization `o-a1b2c3d4e5f` get function invocation permissions. If you specify an AWS account or role as the `Principal`, then only that principal gets function invocation permissions, but only if they are also part of the `o-a1b2c3d4e5f` organization.

This command creates a resource-based policy that looks like the following:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "PrincipalOrgIDExample",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:example",
            "Condition": {
                "StringEquals": {
```

```
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        }
    ]
}
```

For more information, see aws:PrincipalOrgID in the *IAM user guide*.

Granting Lambda function access to other accounts

To share a function with another AWS account, add a cross-account permissions statement to the function's resource-based policy. Run the add-permission command and specify the account ID as the `principal`. The following example grants account `111122223333` permission to invoke `my-function` with the `prod` alias.

```
aws lambda add-permission \
  --function-name my-function:prod \
  --statement-id xaccount \
  --action lambda:InvokeFunction \
  --principal 111122223333 \
  --output text
```

You should see the following output:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-1:123456789012:function:my-function"}
```

The resource-based policy grants permission for the other account to access the function, but doesn't allow users in that account to exceed their permissions. Users in the other account must have the corresponding user permissions to use the Lambda API.

To limit access to a user or role in another account, specify the full ARN of the identity as the principal. For example, `arn:aws:iam::123456789012:user/developer`.

The alias limits which version the other account can invoke. It requires the other account to include the alias in the function ARN.

```
aws lambda invoke \
```

```
  --function-name arn:aws:lambda:us-east-2:123456789012:function:my-function:prod out
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

The function owner can then update the alias to point to a new version without the caller needing to change the way they invoke your function. This ensures that the other account doesn't need to change its code to use the new version, and it only has permission to invoke the version of the function associated with the alias.

You can grant cross-account access for most API actions that operate on an existing function. For example, you could grant access to `lambda:ListAliases` to let an account get a list of aliases, or `lambda:GetFunction` to let them download your function code. Add each permission separately, or use `lambda:*` to grant access to all actions for the specified function.

To grant other accounts permission for multiple functions, or for actions that don't operate on a function, we recommend that you use IAM roles.

Granting Lambda layer access to other accounts

To share a layer with another AWS account, add a cross-account permissions statement to the layer's resource-based policy. Run the add-layer-version-permission command and specify the account ID as the `principal`. In each statement, you can grant permission to a single account, all accounts, or an organization in AWS Organizations.

The following example grants account 111122223333 access to version 2 of the `bash-runtime` layer.

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

You should see output similar to the following:

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

Permissions apply only to a single layer version. Repeat the process each time that you create a new layer version.

To grant permission to all accounts in an AWS Organizations organization, use the `organization-id` option. The following example grants all accounts in organization `o-t194hfs8cz` permission to use version 3 of `my-layer`.

```
aws lambda add-layer-version-permission \
  --layer-name my-layer \
  --version-number 3 \
  --statement-id engineering-org \
  --principal '*' \
  --action lambda:GetLayerVersion \
  --organization-id o-t194hfs8cz \
  --output text
```

You should see the following output:

```
{"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

To grant permission to multiple accounts or organizations, you must add multiple statements.

Using attribute-based access control in Lambda

With attribute-based access control (ABAC), you can use tags to control access to your Lambda resources. You can attach tags to certain Lambda resources, attach them to certain API requests, or attach them to the AWS Identity and Access Management (IAM) principal making the request. For more information about how AWS grants attribute-based access, see Controlling access to AWS resources using tags in the *IAM User Guide*.

You can use ABAC to grant least privilege without specifying an Amazon Resource Name (ARN) or ARN pattern in the IAM policy. Instead, you can specify a tag in the condition element of an IAM

policy to control access. Scaling is easier with ABAC because you don't have to update your IAM policies when you create new resources. Instead, add tags to the new resources to control access.

In Lambda, tags work on the following resources:

- Functions–For more information on tagging functions see, the section called “Tags”.
- Code signing configurations–For more information on tagging code signing configurations, see the section called “Code signing configuration tags”.
- Event source mappings–For more information on tagging event source mappings, see the section called “Event source mapping tags”.

Tags aren't supported for layers.

You can use the following condition keys to write IAM policy rules based on tags:

- aws:ResourceTag/tag-key: Control access based on the tags that are attached to a Lambda resource.
- aws:RequestTag/tag-key: Require tags to be present in a request, such as when creating a new function.
- aws:PrincipalTag/tag-key: Control what the IAM principal (the person making the request) is allowed to do based on the tags that are attached to their IAM user or role.
- aws:TagKeys: Control whether specific tag keys can be used in a request.

You can only specify conditions for actions that support them. For a list of conditions supported by each Lambda action, see Actions, resources, and condition keys for AWS Lambda in the Service Authorization Reference. For **aws:ResourceTag/tag-key** support, refer to "Resource types defined by AWS Lambda." For **aws:RequestTag/tag-key** and **aws:TagKeys** support, refer to "Actions defined by AWS Lambda."

Topics

- Secure your functions by tag

Secure your functions by tag

The following steps demonstrate one way to set up permissions for functions using ABAC. In this example scenario, you'll create four IAM permissions policies. Then, you'll attach these policies to

a new IAM role. Finally, you'll create an IAM user and give that user permission to assume the new role.

Topics

- Prerequisites
- Step 1: Require tags on new functions
- Step 2: Allow actions based on tags attached to a Lambda function and IAM principal
- Step 3: Grant list permissions
- Step 4: Grant IAM permissions
- Step 5: Create the IAM role
- Step 6: Create the IAM user
- Step 7: Test the permissions
- Step 8: Clean up your resources

Prerequisites

Make sure that you have a Lambda execution role. You'll use this role when you grant IAM permissions and when you create a Lambda function.

Step 1: Require tags on new functions

When using ABAC with Lambda, it's a best practice to require that all functions have tags. This helps ensure that your ABAC permissions policies work as expected.

Create an IAM policy similar to the following example. This policy uses the aws:RequestTag/tag-key, aws:ResourceTag/tag-key, and aws:TagKeys condition keys to require that new functions and the IAM principal creating the functions both have the `project` tag. The `ForAllValues` modifier ensures that `project` is the only allowed tag. If you don't include the `ForAllValues` modifier, users can add other tags to the function as long as they also pass `project`.

Example– Require tags on new functions

JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:TagResource"
      ],
      "Resource": "arn:aws:lambda:*:*:function:*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/project": "${aws:PrincipalTag/project}",
          "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
        },
        "ForAllValues:StringEquals": {
          "aws:TagKeys": "project"
        }
      }
    }
  }
```

Step 2: Allow actions based on tags attached to a Lambda function and IAM principal

Create a second IAM policy using the aws:ResourceTag/tag-key condition key to require the principal's tag to match the tag that's attached to the function. The following example policy allows principals with the `project` tag to invoke functions with the `project` tag. If a function has any other tags, the action is denied.

Example– Require matching tags on function and IAM principal

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "lambda:InvokeFunction",
          "lambda:GetFunction"
        ],
        "Resource": "arn:aws:lambda:*:*:function:*",
        "Condition": {
```

```
          "StringEquals": {
            "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
          }
        }
      }
    ]
  }
```

Step 3: Grant list permissions

Create a policy that allows the principal to list Lambda functions and IAM roles. This allows the principal to see all Lambda functions and IAM roles on the console and when calling the API actions.

Example– Grant Lambda and IAM list permissions

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "AllResourcesLambdaNoTags",
        "Effect": "Allow",
        "Action": [
          "lambda:GetAccountSettings",
          "lambda:ListFunctions",
          "iam:ListRoles"
        ],
        "Resource": "*"
      }
    ]
  }
```

Step 4: Grant IAM permissions

Create a policy that allows **iam:PassRole**. This permission is required when you assign an execution role to a function. In the following example policy, replace the example ARN with the ARN of your Lambda execution role.

> **Note**
>
> Do not use the `ResourceTag` condition key in a policy with the `iam:PassRole` action. You cannot use the tag on an IAM role to control access to who can pass that role. For more information about permissions required to pass a role to a service, see Granting a user permissions to pass a role to an AWS service.

Example– Grant permission to pass the execution role

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "VisualEditor0",
        "Effect": "Allow",
        "Action": [
          "iam:PassRole"
        ],
        "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
      }
    ]
  }
```

Step 5: Create the IAM role

It's a best practice to use roles to delegate permissions. Create an IAM role called `abac-project-role`:

- On **Step 1: Select trusted entity**: Choose **AWS account** and then choose **This account**.
- On **Step 2: Add permissions**: Attach the four IAM policies that you created in the previous steps.
- On **Step 3: Name, review, and create**: Choose **Add tag**. For **Key**, enter `project`. Don't enter a **Value**.

Step 6: Create the IAM user

Create an IAM user called `abac-test-user`. In the **Set permissions** section, choose **Attach existing policies directly** and then choose **Create policy**. Enter the following policy definition. Replace *111122223333* with your AWS account ID. This policy allows `abac-test-user` to assume `abac-project-role`.

Example– Allow IAM user to assume ABAC role

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
    }
  }
```

Step 7: Test the permissions

1. Sign in to the AWS console as `abac-test-user`. For more information, see Sign in as an IAM user.
2. Switch to the `abac-project-role` role. For more information, see Switching to a role (console).
3. Create a Lambda function:
   - Under **Permissions**, choose **Change default execution role**, and then for **Execution role**, choose **Use an existing role**. Choose the same execution role that you used in Step 4: Grant IAM permissions.
   - Under **Advanced settings**, choose **Enable tags** and then choose **Add new tag**. For **Key**, enter `project`. Don't enter a **Value**.
4. Test the function.
5. Create a second Lambda function and add a different tag, such as `environment`. This operation should fail because the ABAC policy that you created in Step 1: Require tags on new functions only allows the principal to create functions with the `project` tag.

6. Create a third function without tags. This operation should fail because the ABAC policy that you created in Step 1: Require tags on new functions doesn't allow the principal to create functions without tags.

This authorization strategy allows you to control access without creating new policies for each new user. To grant access to new users, simply give them permission to assume the role that corresponds to their assigned project.

Step 8: Clean up your resources

To delete the IAM role

1. Open the Roles page of the IAM console.
2. Select the role that you created in step 5.
3. Choose **Delete**.
4. To confirm deletion, enter the role name in the text input field.
5. Choose **Delete**.

To delete the IAM user

1. Open the Users page of the IAM console.
2. Select the IAM user that you created in step 6.
3. Choose **Delete**.
4. To confirm deletion, enter the user name in the text input field.
5. Choose **Delete user**.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

Fine-tuning the Resources and Conditions sections of policies

You can restrict the scope of a user's permissions by specifying resources and conditions in an AWS Identity and Access Management (IAM) policy. Each action in a policy supports a combination of resource and condition types that varies depending on the behavior of the action.

Every IAM policy statement grants permission to an action that's performed on a resource. When the action doesn't act on a named resource, or when you grant permission to perform the action on all resources, the value of the resource in the policy is a wildcard (*). For many actions, you can restrict the resources that a user can modify by specifying the Amazon Resource Name (ARN) of a resource, or an ARN pattern that matches multiple resources.

By resource type, the general design of how to restrict the scope of an action is the following:

- Functions–Actions that operate on a function can be restricted to a specific function by function, version, or alias ARN.
- Event source mappings–Actions can be restricted to specific event source mapping resources by ARN. Event source mappings are always associated with a function. You can also use the `lambda:FunctionArn` condition to restrict actions by associated function.
- Layers–Actions related to layer usage and permissions act on a version of a layer.
- Code signing configuration–Actions can be restricted to specific code signing configuration resources by ARN.
- Tags–Use standard tag conditions. For more information, see the section called “Attribute-based access control”.

To restrict permissions by resource, specify the resource by ARN.

Lambda resource ARN format

- Function – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*
- Function version – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`1`*
- Function alias – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`TEST`*
- Event source mapping – `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:event-source-mapping:`*`fa123456-14a1-4fd2-9fec-83de64ad683de6d47`*

- Layer – `arn:aws:lambda:us-west-2:123456789012:layer:my-layer`
- Layer version – `arn:aws:lambda:us-west-2:123456789012:layer:my-layer:1`
- Code signing configuration – `arn:aws:lambda:us-west-2:123456789012:code-signing-config:my-csc`

For example, the following policy allows a user in AWS account 123456789012 to invoke a function named `my-function` in the US West (Oregon) AWS Region.

Example invoke function policy

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-function"
        }
    ]
}
```

This is a special case where the action identifier (`lambda:InvokeFunction`) differs from the API operation (Invoke). For other actions, the action identifier is the operation name prefixed by `lambda:`.

Sections

- Understanding the Condition section in policies
- Referencing functions in the Resource section of policies
- Supported IAM actions and function behaviors

Understanding the Condition section in policies

Conditions are an optional policy element that applies additional logic to determine if an action is allowed. In addition to common conditions that all actions support, Lambda defines condition types that you can use to restrict the values of additional parameters on some actions.

For example, the `lambda:Principal` condition lets you restrict the service or account that a user can grant invocation access to on a function's resource-based policy. The following policy lets a user grant permission to Amazon Simple Notification Service (Amazon SNS) topics to invoke a function named `test`.

Example manage function policy permissions

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ManageFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:test:*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "sns.amazonaws.com"
                }
            }
        }
    ]
}
```

The condition requires that the principal is Amazon SNS and not another service or account. The resource pattern requires that the function name is `test` and includes a version number or alias. For example, `test:v1`.

For more information on resources and conditions for Lambda and other AWS services, see Actions, resources, and condition keys for AWS services in the *Service Authorization Reference*.

Referencing functions in the Resource section of policies

You reference a Lambda function in a policy statement using an Amazon Resource Name (ARN). The format of a function ARN depends on whether you are referencing the whole function (unqualified) or a function version or alias (qualified).

When making Lambda API calls, users can specify a version or alias by passing a version ARN or alias ARN in the GetFunction `FunctionName` parameter, or by setting a value in the GetFunction `Qualifier` parameter. Lambda makes authorization decisions by comparing the resource element in the IAM policy with both the `FunctionName` and `Qualifier` passed in API calls. If there is a mismatch, Lambda denies the request.

Whether you are allowing or denying an action on your function, you must use the correct function ARN types in your policy statement to achieve the results that you expect. For example, if your policy references the unqualified ARN, Lambda accepts requests that reference the unqualified ARN but denies requests that reference a qualified ARN.

> **Note**
>
> You can't use a wildcard character (*) to match the account ID. For more information on accepted syntax, see IAM JSON policy reference in the *IAM User Guide*.

Example allowing invocation of an unqualified ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction"
        }
    ]
```

```
}
```

If your policy references a specific qualified ARN, Lambda accepts requests that reference that ARN but denies requests that reference the unqualified ARN or a different qualified ARN, for example, `myFunction:2`.

Example allowing invocation of a specific qualified ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:1"
        }
    ]
}
```

If your policy references any qualified ARN using `:*`, Lambda accepts any qualified ARN but denies requests that reference the unqualified ARN.

Example allowing invocation of any qualified ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
        }
    ]
```

```
}
```

If your policy references any ARN using *, Lambda accepts any qualified or unqualified ARN.

Example allowing invocation of any qualified or unqualified ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction*"
        }
    ]
}
```

Supported IAM actions and function behaviors

Actions define what can be permitted through IAM policies. For a list of actions supported in Lambda, see Actions, resources, and condition keys for AWS Lambda in the Service Authorization Reference. In most cases, when an IAM action permits an Lambda API action, the name of the IAM action is the same as the name of the Lambda API action, with the following exceptions:

API action	IAM action
Invoke	`lambda:InvokeFunction`
GetLayerVersion GetLayerVersionByArn	`lambda:GetLayerVersion`

In addition to the resources and conditions defined in the Service Authorization Reference, Lambda supports the following resources and conditions for certain actions. Many of these are related to

referencing functions in the resource section of policies. Actions that operate on a function can be restricted to a specific function by function, version, or alias ARN, as described in the following table.

Action	Resource	Condition
AddPermission RemovePermission Invoke (**Permission:** `lambda:InvokeFunction`)	Function version Function alias	N/A
UpdateFunctionConfiguration	N/A	`lambda:CodeSigningConfigArn`
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	Function alias	N/A

Security in AWS Lambda

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS compliance programs. To learn about the compliance programs that apply to AWS Lambda, see AWS services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Lambda. The following topics show you how to configure Lambda to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Lambda resources.

For more information about applying security principles to Lambda applications, see Security in Serverless Land.

Topics

- Data protection in AWS Lambda
- Using service-linked roles for Lambda
- Identity and Access Management for AWS Lambda
- Create a governance strategy for Lambda functions and layers
- Compliance validation for AWS Lambda
- Resilience in AWS Lambda
- Infrastructure security in AWS Lambda
- Securing workloads with public endpoints

- Using code signing to verify code integrity with Lambda

Data protection in AWS Lambda

The AWS shared responsibility model applies to data protection in AWS Lambda. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Lambda or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Sections

- Encryption in transit
- Data encryption at rest for AWS Lambda

Encryption in transit

Lambda API endpoints only support secure connections over HTTPS. When you manage Lambda resources with the AWS Management Console,AWS SDK, or the Lambda API, all communication is encrypted with Transport Layer Security (TLS). For a full list of API endpoints, see AWS Regions and endpoints in the AWS General Reference.

When you connect your function to a file system, Lambda uses encryption in transit for all connections. For more information, see Data encryption in Amazon EFS in the *Amazon Elastic File System User Guide*.

When you use environment variables, you can enable console encryption helpers to use client-side encryption to protect the environment variables in transit. For more information, see Securing Lambda environment variables.

Data encryption at rest for AWS Lambda

Lambda always provides at-rest encryption for the following resources using an AWS owned key or an AWS managed key:

- Environment variables
- Files that you upload to Lambda, including deployment packages and layer archives
- Event source mapping filter criteria objects

You can optionally configure Lambda to use a customer managed key to encrypt your environment variables, .zip deployment packages, and filter criteria objects.

Amazon CloudWatch Logs and AWS X-Ray also encrypt data by default, and can be configured to use a customer managed key. For details, see Encrypt log data in CloudWatch Logs and Data protection in AWS X-Ray.

Monitoring your encryption keys for Lambda

When you use an AWS KMS customer managed key with Lambda, you can use AWS CloudTrail. The following examples are CloudTrail events for `Decrypt`, `DescribeKey`, and `GenerateDataKey` calls made by Lambda to access data encrypted by your customer managed key.

Decrypt

If you used a AWS KMS customer managed key to encrypt your filter criteria object, Lambda sends a `Decrypt` request on your behalf when you try to access it in plaintext (for example, from a `ListEventSourceMappings` call). The following example event records the `Decrypt` operation:

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:05:46Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
```

```
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```

DescribeKey

If you used a AWS KMS customer managed key to encrypt your filter criteria object, Lambda sends a `DescribeKey` request on your behalf when you try to access it (for example, from a `GetEventSourceMapping` call). The following example event records the `DescribeKey` operation:

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
```

```
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        }
    },
    "eventTime": "2024-05-30T01:09:40Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "DescribeKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "54.240.197.238",
    "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36",
    "requestParameters": {
        "keyId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "tlsDetails": {
```

```
        "tlsVersion": "TLSv1.3",
        "cipherSuite": "TLS_AES_256_GCM_SHA384",
        "clientProvidedHostHeader": "kms.eu-west-1.amazonaws.com"
    },
    "sessionCredentialFromConsole": "true"
}
```

GenerateDataKey

When you use a AWS KMS customer managed key to encrypt your filter criteria object in a `CreateEventSourceMapping` or `UpdateEventSourceMapping` call, Lambda sends a `GenerateDataKey` request on your behalf to generate a data key to encrypt the filter criteria (envelope encryption). The following example event records the `GenerateDataKey` operation:

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:06:07Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:04:18Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
```

```
    "requestParameters": {
        "numberOfBytes": 32,
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management"
}
```


Using service-linked roles for Lambda

Lambda uses AWS Identity and Access Management (IAM) service-linked roles. A service-linked role is a unique type of IAM role that is linked directly to Lambda. Service-linked roles are predefined by Lambda and include permissions that the service requires to call other AWS services on your behalf.

Lambda defines the permissions of its service-linked roles, and only Lambda can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete a service-linked role only after first deleting their related resources. This protects your Lambda resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see AWS services that work with IAM and look for the services that have **Yes** in the **Service-linked roles** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for Lambda

Lambda uses the service-linked role named **AWSServiceRoleForLambda**. The service-linked role trusts the following services to assume the role:

- `lambda.amazonaws.com`

The role permissions policy named AWSLambdaServiceRolePolicy allows Lambda to complete the following actions on the specified resources:

- Action: `ec2:TerminateInstances` on `arn:aws:ec2:*:*:instance/*` with the condition that `ec2:ManagedResourceOperator` equals `scaler.lambda.amazonaws.com`
- Action: `ec2:DescribeInstanceStatus` and `ec2:DescribeInstances` on `*`

You must configure permissions to allow your users, groups, or roles to create, edit, or delete a service-linked role. For more information, see Service-linked role permissions in the *IAM User Guide*.

For managed policy updates, see Lambda managed policies.

Creating a service-linked role for Lambda

You don't need to manually create a service-linked role. When you create a Lambda capacity provider in the AWS Management Console, the AWS CLI, or the AWS API, Lambda creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you create a Lambda capacity provider, Lambda creates the service-linked role for you again.

You can also use the IAM console to create a service-linked role with the **AWSServiceRoleForLambda** use case. In the AWS CLI or the AWS API, create a service-linked role with the `lambda.amazonaws.com` service name. For more information, see Creating a service-

linked role in the *IAM User Guide*. If you delete this service-linked role, you can use this same process to create the role again.

Editing a service-linked role for Lambda

Lambda does not allow you to edit the AWSServiceRoleForLambda service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see Editing a service-linked role in the *IAM User Guide*.

Deleting a service-linked role for Lambda

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored or maintained. However, you must clean up the resources for your service-linked role before you can manually delete it.

> **Note**
>
> If the Lambda service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To delete Lambda resources used by the AWSServiceRoleForLambda

1. Remove all Lambda capacity providers from your account. You can do this using the Lambda console, CLI, or API.
2. Verify that no Lambda capacity providers remain in your account before attempting to delete the service-linked role.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the AWSServiceRoleForLambda service-linked role. For more information, see Deleting a service-linked role in the *IAM User Guide*.

Supported Regions for Lambda service-linked roles

Lambda does not support using service-linked roles in every Region where the service is available. AWSServiceRoleForLambda is supported in the following Regions.

Region name	Region identity	Support in Lambda
US East (N. Virginia)	us-east-1	Yes
US East (Ohio)	us-east-2	Yes
US West (N. California)	us-west-1	Yes
US West (Oregon)	us-west-2	Yes
Africa (Cape Town)	af-south-1	No
Asia Pacific (Hong Kong)	ap-east-1	Yes
Asia Pacific (Jakarta)	ap-southeast-3	Yes
Asia Pacific (Bangkok)	ap-southeast-7	Yes
Asia Pacific (Mumbai)	ap-south-1	Yes
Asia Pacific (Osaka)	ap-northeast-3	No
Asia Pacific (Seoul)	ap-northeast-2	No
Asia Pacific (Singapore)	ap-southeast-1	Yes
Asia Pacific (Sydney)	ap-southeast-2	Yes
Asia Pacific (Tokyo)	ap-northeast-1	Yes
Canada (Central)	ca-central-1	No
Europe (Frankfurt)	eu-central-1	Yes
Europe (Ireland)	eu-west-1	Yes
Europe (London)	eu-west-2	Yes
Europe (Milan)	eu-south-1	No
Europe (Paris)	eu-west-3	No

Region name	Region identity	Support in Lambda
Europe (Stockholm)	eu-north-1	No
Middle East (Bahrain)	me-south-1	No
Middle East (UAE)	me-central-1	No
South America (São Paulo)	sa-east-1	No
AWS GovCloud (US-East)	us-gov-east-1	No
AWS GovCloud (US-West)	us-gov-west-1	No

Identity and Access Management for AWS Lambda

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Lambda resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How AWS Lambda works with IAM
- Identity-based policy examples for AWS Lambda
- AWS managed policies for AWS Lambda
- Troubleshooting AWS Lambda identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting AWS Lambda identity and access)
- **Service administrator** - determine user access and submit permission requests (see How AWS Lambda works with IAM)
- **IAM administrator** - write policies to manage access (see Identity-based policy examples for AWS Lambda)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users to use federation with an identity provider to access AWS services using temporary credentials.

A *federated identity* is a user from your enterprise directory, web identity provider, or Directory Service that accesses AWS services using credentials from an identity source. Federated identities assume roles that provide temporary credentials.

For centralized access management, we recommend AWS IAM Identity Center. For more information, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How AWS Lambda works with IAM

Before you use IAM to manage access to Lambda, learn what IAM features are available to use with Lambda.

IAM feature	Lambda support
Identity-based policies	Yes
Resource-based policies	Yes
Policy actions	Yes
Policy resources	Yes
Policy condition keys (service-specific)	Yes
ACLs	No
ABAC (tags in policies)	Partial
Temporary credentials	Yes
Forward access sessions (FAS)	No
Service roles	Yes
Service-linked roles	Partial

To get a high-level view of how Lambda and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Lambda

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Lambda

To view examples of Lambda identity-based policies, see Identity-based policy examples for AWS Lambda.

Resource-based policies within Lambda

Supports resource-based policies: Yes

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

You can attach a resource-based policy to a Lambda function or layer. This policy defines which principals can perform actions on the function or layer.

To learn how to attach a resource-based policy to a function or layer, see Viewing resource-based IAM policies in Lambda.

Policy actions for Lambda

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Lambda actions, see Actions defined by AWS Lambda in the *Service Authorization Reference*.

Policy actions in Lambda use the following prefix before the action:

```
lambda
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "lambda:action1",
      "lambda:action2"
         ]
```

To view examples of Lambda identity-based policies, see Identity-based policy examples for AWS Lambda.

Policy resources for Lambda

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Lambda resource types and their ARNs, see Resource types defined by AWS Lambda in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by AWS Lambda.

To view examples of Lambda identity-based policies, see Identity-based policy examples for AWS Lambda.

Policy condition keys for Lambda

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Lambda condition keys, see Condition keys for AWS Lambda in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by AWS Lambda.

To view examples of Lambda identity-based policies, see Identity-based policy examples for AWS Lambda.

ACLs in Lambda

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Lambda

Supports ABAC (tags in policies): Partial

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes called tags. You can attach tags to IAM entities and AWS resources, then design ABAC policies to allow operations when the principal's tag matches the tag on the resource.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see Define permissions with ABAC authorization in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

For more information about tagging Lambda resources, see Using attribute-based access control in Lambda.

Using temporary credentials with Lambda

Supports temporary credentials: Yes

Temporary credentials provide short-term access to AWS resources and are automatically created when you use federation or switch roles. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM and AWS services that work with IAM in the *IAM User Guide*.

Forward access sessions for Lambda

Supports forward access sessions (FAS): No

Forward access sessions (FAS) use the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. For policy details when making FAS requests, see Forward access sessions.

Service roles for Lambda

Supports service roles: Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

In Lambda, a service role is known as an execution role.

> **Warning**
>
> Changing the permissions for an execution role might break Lambda functionality.

Service-linked roles for Lambda

Supports service-linked roles: Partial

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

Lambda doesn't have service-linked roles, but Lambda@Edge does. For more information, see Service-Linked Roles for Lambda@Edge in the *Amazon CloudFront Developer Guide*.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for AWS Lambda

By default, users and roles don't have permission to create or modify Lambda resources. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Create IAM policies (console) in the *IAM User Guide*.

For details about actions and resource types defined by Lambda, including the format of the ARNs for each of the resource types, see Actions, resources, and condition keys for AWS Lambda in the *Service Authorization Reference*.

Topics

- Policy best practices
- Using the Lambda console
- Allow users to view their own permissions

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Lambda resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Lambda console

To access the AWS Lambda console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Lambda resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

For an example policy that grants minimal access for function development, see Granting users access to a Lambda function. In addition to Lambda APIs, the Lambda console uses other services to display trigger configuration and let you add new triggers. If your users use Lambda with other services, they need access to those services as well. For details on configuring other services with Lambda, see Invoking Lambda with events from other AWS services.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
```

```
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policies for AWS Lambda

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

Topics

- AWS managed policy: AWSLambda_FullAccess
- AWS managed policy: AWSLambda_ReadOnlyAccess
- AWS managed policy: AWSLambdaBasicExecutionRole
- AWS managed policy: AWSLambdaBasicDurableExecutionRolePolicy
- AWS managed policy: AWSLambdaDynamoDBExecutionRole
- AWS managed policy: AWSLambdaENIManagementAccess
- AWS managed policy: AWSLambdaInvocation-DynamoDB

- AWS managed policy: AWSLambdaKinesisExecutionRole
- AWS managed policy: AWSLambdaMSKExecutionRole
- AWS managed policy: AWSLambdaRole
- AWS managed policy: AWSLambdaSQSQueueExecutionRole
- AWS managed policy: AWSLambdaVPCAccessExecutionRole
- AWS managed policy: AWSLambdaManagedEC2ResourceOperator
- AWS managed policy: AWSLambdaServiceRolePolicy
- Lambda updates to AWS managed policies

AWS managed policy: AWSLambda_FullAccess

This policy grants full access to Lambda actions. It also grants permissions to other AWS services that are used to develop and maintain Lambda resources.

You can attach the `AWSLambda_FullAccess` policy to your users, groups, and roles.

Permissions details

This policy includes the following permissions:

- `lambda` – Allows principals full access to Lambda.
- `cloudformation` – Allows principals to describe AWS CloudFormation stacks and list the resources in those stacks.
- `cloudwatch` – Allows principals to list Amazon CloudWatch metrics and get metric data.
- `ec2` – Allows principals to describe security groups, subnets, and VPCs.
- `iam` – Allows principals to get policies, policy versions, roles, role policies, attached role policies, and the list of roles. This policy also allows principals to pass roles to Lambda. The `PassRole` permission is used when you assign an execution role to a function. The `CreateServiceLinkedRole` permission is used when creating a service-linked role.
- `kms` – Allows principals to list aliases and describe key for volume encryption.
- `logs` – Allows principals to describe log streams, get log events, filter log events, and to start and stop Live Tail sessions.
- `states` – Allows principals to describe and list AWS Step Functions state machines.

- `tag` – Allows principals to get resources based on their tags.
- `xray` – Allows principals to get AWS X-Ray trace summaries and retrieve a list of traces specified by ID.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambda_FullAccess in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambda_ReadOnlyAccess

This policy grants read-only access to Lambda resources and to other AWS services that are used to develop and maintain Lambda resources.

You can attach the `AWSLambda_ReadOnlyAccess` policy to your users, groups, and roles.

Permissions details

This policy includes the following permissions:

- `lambda` – Allows principals to get and list all resources.
- `cloudformation` – Allows principals to describe and list AWS CloudFormation stacks and list the resources in those stacks.
- `cloudwatch` – Allows principals to list Amazon CloudWatch metrics and get metric data.
- `ec2` – Allows principals to describe security groups, subnets, and VPCs.
- `iam` – Allows principals to get policies, policy versions, roles, role policies, attached role policies, and the list of roles.
- `kms` – Allows principals to list aliases.
- `logs` – Allows principals to describe log streams, get log events, filter log events, and to start and stop Live Tail sessions.
- `states` – Allows principals to describe and list AWS Step Functions state machines.
- `tag` – Allows principals to get resources based on their tags.
- `xray` – Allows principals to get AWS X-Ray trace summaries and retrieve a list of traces specified by ID.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambda_ReadOnlyAccess in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaBasicExecutionRole

This policy grants permissions to upload logs to CloudWatch Logs.

You can attach the `AWSLambdaBasicExecutionRole` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaBasicExecutionRole in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaBasicDurableExecutionRolePolicy

This policy provides write permissions to CloudWatch Logs and read/write permissions to durable execution APIs used by Lambda durable functions. This policy provides the essential permissions required for Lambda durable functions, which use durable execution APIs to persist progress and maintain state across function invocations.

You can attach the `AWSLambdaBasicDurableExecutionRolePolicy` policy to your users, groups, and roles.

Permissions details

This policy includes the following permissions:

- `logs` – Allows principals to create log groups and log streams, and write log events to CloudWatch Logs.
- `lambda` – Allows principals to checkpoint durable execution state and retrieve durable execution state for Lambda durable functions.

To view more details about the policy, including the latest version of the JSON policy document, see AWSLambdaBasicDurableExecutionRolePolicy in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaDynamoDBExecutionRole

This policy grants permissions to read records from an Amazon DynamoDB stream and write to CloudWatch Logs.

You can attach the `AWSLambdaDynamoDBExecutionRole` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaDynamoDBExecutionRole in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaENIManagementAccess

This policy grants permissions to create, describe, and delete elastic network interfaces used by a VPC-enabled Lambda function.

You can attach the `AWSLambdaENIManagementAccess` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaENIManagementAccess in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaInvocation-DynamoDB

This policy grants read access to Amazon DynamoDB Streams.

You can attach the `AWSLambdaInvocation-DynamoDB` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaInvocation-DynamoDB in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaKinesisExecutionRole

This policy grants permissions to read events from an Amazon Kinesis data stream and write to CloudWatch Logs.

You can attach the `AWSLambdaKinesisExecutionRole` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaKinesisExecutionRole in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaMSKExecutionRole

This policy grants permissions to read and access records from an Amazon Managed Streaming for Apache Kafka cluster, manage elastic network interfaces, and write to CloudWatch Logs.

You can attach the `AWSLambdaMSKExecutionRole` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaMSKExecutionRole in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaRole

This policy grants permissions to invoke Lambda functions.

You can attach the `AWSLambdaRole` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaRole in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaSQSQueueExecutionRole

This policy grants permissions to read and delete messages from an Amazon Simple Queue Service queue, and grants write permissions to CloudWatch Logs.

You can attach the `AWSLambdaSQSQueueExecutionRole` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaSQSQueueExecutionRole in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaVPCAccessExecutionRole

This policy grants permissions to manage elastic network interfaces within an Amazon Virtual Private Cloud and write to CloudWatch Logs.

You can attach the `AWSLambdaVPCAccessExecutionRole` policy to your users, groups, and roles.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaVPCAccessExecutionRole in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaManagedEC2ResourceOperator

This policy enables automated Amazon Elastic Compute Cloud instance management for Lambda capacity providers. It grants permissions to the Lambda scaler service to perform instance lifecycle operations on your behalf.

You can attach the `AWSLambdaManagedEC2ResourceOperator` policy to your users, groups, and roles.

Permissions details

This policy includes the following permissions:

- `ec2:RunInstances` – Allows Lambda to launch new Amazon EC2 instances with the condition that ec2:ManagedResourceOperator equals scaler.lambda.amazonaws.com and restricts AMI usage to Amazon-owned images only.
- `ec2:DescribeInstances` and `ec2:DescribeInstanceStatus` – Allows Lambda to monitor instance status and retrieve instance information.

- `ec2:CreateTags` – Allows Lambda to tag Amazon EC2 resources for management and identification purposes.
- `ec2:DescribeAvailabilityZones` – Allows Lambda to view available zones for instance placement decisions.
- `ec2:DescribeCapacityReservations` – Allows Lambda to check capacity reservations for optimal instance placement.
- `ec2:DescribeInstanceTypes` and `ec2:DescribeInstanceTypeOfferings` – Allows Lambda to review available instance types and their offerings.
- `ec2:DescribeSubnets` – Allows Lambda to examine subnet configurations for network planning.
- `ec2:DescribeSecurityGroups` – Allows Lambda to retrieve security group information for network interface configuration.
- `ec2:CreateNetworkInterface` – Allows Lambda to create network interfaces and manage subnet and security group associations.
- `ec2:AttachNetworkInterface` – Allows Lambda to attach network interfaces to Amazon EC2 instances with the condition that `ec2:ManagedResourceOperator` equals scaler.lambda.amazonaws.com.

For more information about this policy, including the JSON policy document and policy versions, see AWSLambdaManagedEC2ResourceOperator in the *AWS Managed Policy Reference Guide*.

AWS managed policy: AWSLambdaServiceRolePolicy

This policy is attached to the service-linked role named AWSServiceRoleForLambda to allow Lambda to terminate instances managed as part of Lambda capacity providers.

Permissions details

This policy includes the following permissions:

- `ec2:TerminateInstances` – Allows Lambda to terminate EC2 instances with the condition that ec2:ManagedResourceOperator equals scaler.lambda.amazonaws.com.
- `ec2:DescribeInstanceStatus` and `ec2:DescribeInstances` – Allows Lambda to describe EC2 instances.

For more information about this policy, see Using service-linked roles for Lambda.

Lambda updates to AWS managed policies

Change	Description	Date
AWSLambdaManagedEC2ResourceOperator – New policy	Lambda added a new managed policy to enable automated Amazon EC2 instance management for Lambda capacity providers , allowing the scaler service to perform instance lifecycle operations.	November 30, 2025
AWSLambdaServiceRolePolicy – New policy	Lambda added a new managed policy for the service-linked role to allow Lambda to terminate instances managed as part of Lambda capacity providers.	November 30, 2025
AWSLambda_FullAccess – Change	Lambda updated the `AWSLambda_FullAccess` policy to allow the `kms:DescribeKey` and `iam:CreateServiceLinkedRole` actions.	November 30, 2025
AWSLambdaBasicDurableExecutionRolePolicy – New managed policy	Lambda released a new managed policy `AWSLambdaBasicDurableExecutionRolePolicy` that provides write permissions to CloudWatch Logs and read/write permissions to durable execution APIs used by Lambda durable functions.	December 1, 2025

Change	Description	Date
AWSLambda_ReadOnlyAccess and AWSLambda_FullAccess – Change	Lambda updated the `AWSLambda_ReadOnlyAccess` and `AWSLambda_FullAccess` policies to allow the `logs:StartLiveTail` and `logs:StopLiveTail` actions.	March 17, 2025
AWSLambdaVPCAccessExecutionRole – Change	Lambda updated the `AWSLambdaVPCAccessExecutionRole` policy to allow the action `ec2:DescribeSubnets` .	January 5, 2024
AWSLambda_ReadOnlyAccess – Change	Lambda updated the `AWSLambda_ReadOnlyAccess` policy to allow principals to list CloudFormation stacks.	July 27, 2023
AWS Lambda started tracking changes	AWS Lambda started tracking changes for its AWS managed policies.	July 27, 2023

Troubleshooting AWS Lambda identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Lambda and IAM.

Topics

- I am not authorized to perform an action in Lambda
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my Lambda resources

I am not authorized to perform an action in Lambda

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `lambda:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `lambda:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Lambda.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Lambda. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Lambda resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Lambda supports these features, see How AWS Lambda works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Create a governance strategy for Lambda functions and layers

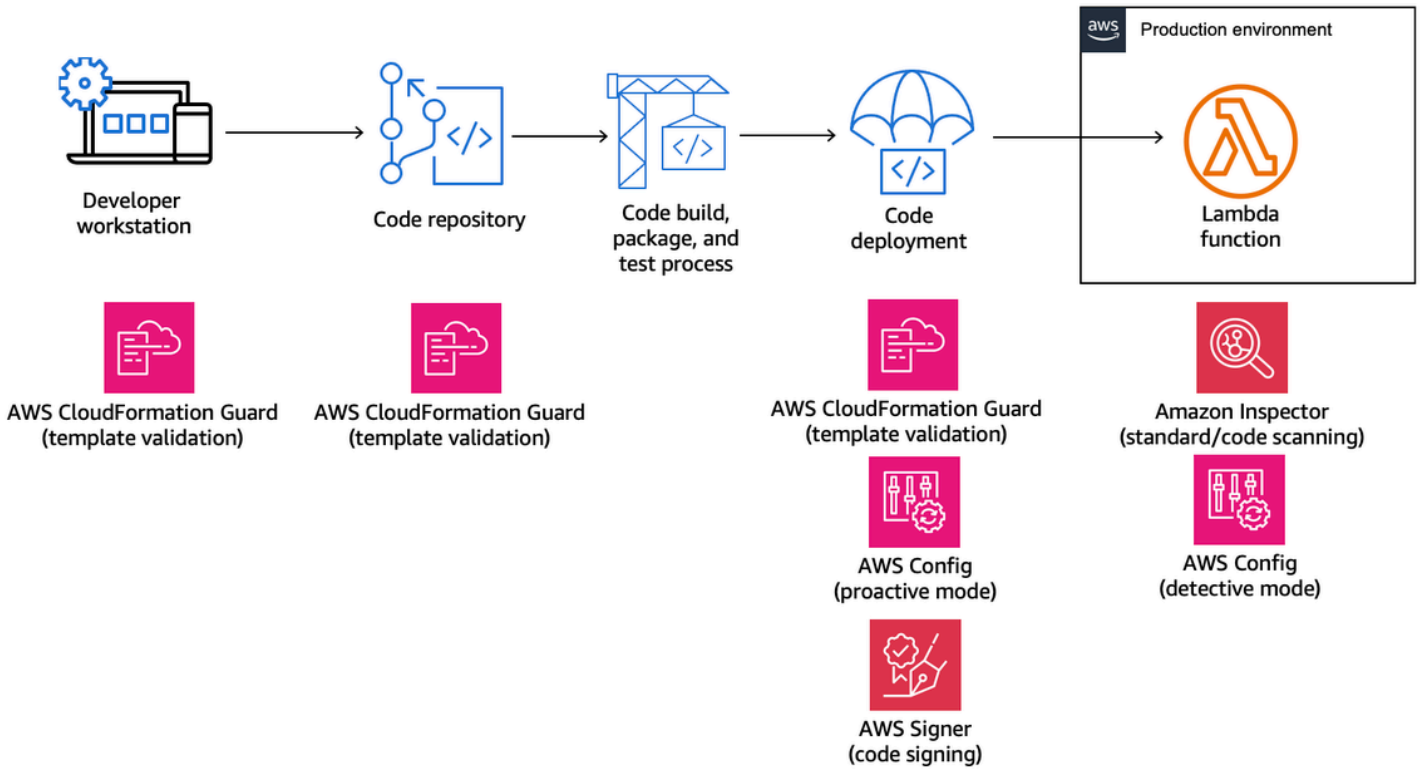
To build and deploy serverless, cloud-native applications, you must allow for agility and speed to market with appropriate governance and guardrails. You set business-level priorities, maybe emphasizing agility as the top priority, or alternatively emphasizing risk aversion via governance, guardrails, and controls. Realistically, you won't have an "either/or" strategy but an "and" strategy that balances both agility and guardrails in your software development lifecycle. No matter where these requirements fall in your company's lifecycle, governance capabilities are likely to become an implementation requirement in your processes and toolchains.

Here are a few examples of governance controls that an organization might implement for Lambda:

- Lambda functions must not be publicly accessible.
- Lambda functions must be attached to a VPC.
- Lambda functions should not use deprecated runtimes.
- Lambda functions must be tagged with a set of required tags.

- Lambda layers must not be accessible outside of the organization.
- Lambda functions with an attached security group must have matching tags between the function and security group.
- Lambda functions with an attached layer must use an approved version
- Lambda environment variables must be encrypted at rest with a customer managed key.

The following diagram is an example of an in-depth governance strategy that implements controls and policy throughout the software development and deployment process:



The following topics explain how to implement controls for developing and deploying Lambda functions in your organization, both for the startup and the enterprise. Your organization might already have tools in place. The following topics take a modular approach to these controls, so that you can pick and choose the components you actually need.

Topics

- Proactive controls for Lambda with AWS CloudFormation Guard
- Implement preventative controls for Lambda with AWS Config
- Detect non-compliant Lambda deployments and configurations with AWS Config

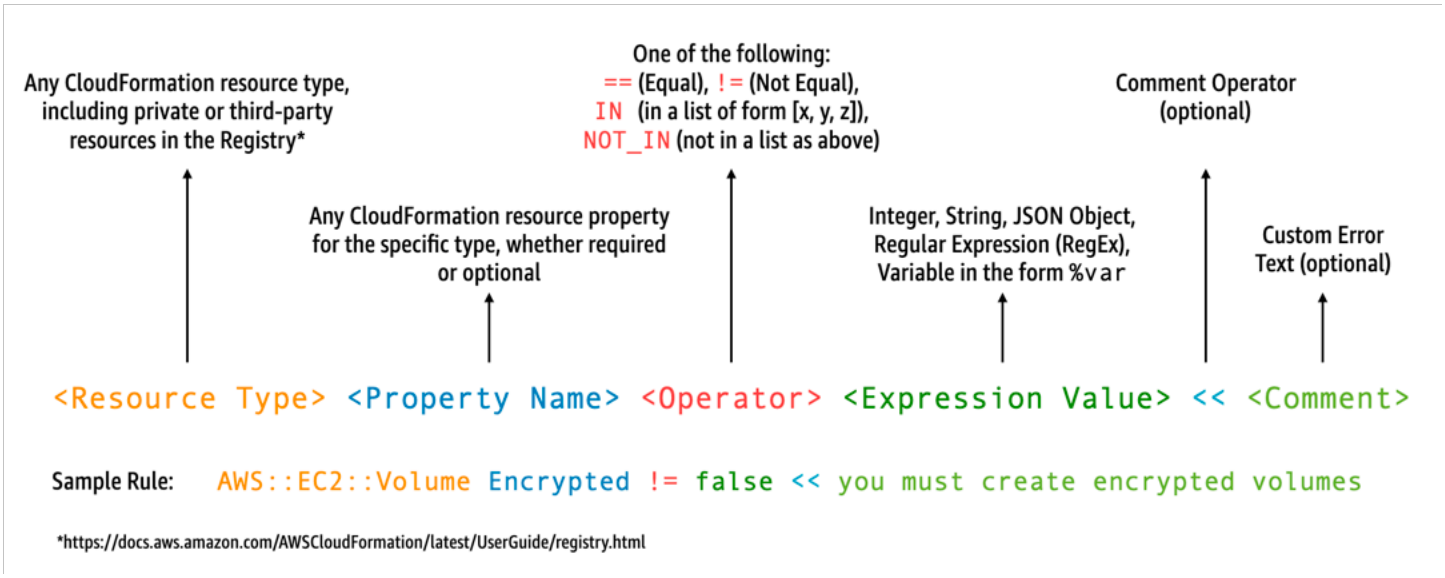
- Lambda code signing with AWS Signer
- Automate security assessments for Lambda with Amazon Inspector
- Implement observability for Lambda security and compliance

Proactive controls for Lambda with AWS CloudFormation Guard

AWS CloudFormation Guard is an open-source, general-purpose, policy-as-code evaluation tool. This can be used for preventative governance and compliance by validating Infrastructure as Code (IaC) templates and service compositions against policy rules. These rules can be customized based on your team or organizational requirements. For Lambda functions, the Guard rules can be used to control resource creation and configuration updates by defining the required property settings needed while creating or updating a Lambda function.

Compliance administrators define the list of controls and governance policies that are required for deploying and updating Lambda functions. Platform administrators implement the controls in CI/CD pipelines, as pre-commit validation webhooks with code repositories, and provide developers with command line tools for validating templates and code on local workstations. Developers author code, validate templates with command line tools, and then commit code to repositories, which are then automatically validated via the CI/CD pipelines prior to deployment into an AWS environment.

Guard allows you to write your rules and implement your controls with a domain-specific language as follows.



For example, suppose you want to ensure that developers choose only the latest runtimes. You could specify two different policies, one to identify runtimes that are already deprecated and another to identify runtimes that are to be deprecated soon. To do this, you might write the following `etc/rules.guard` file:

```
let lambda_functions = Resources.*[
    Type == "AWS::Lambda::Function"
]

rule lambda_already_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["dotnetcore3.1", "nodejs12.x", "python3.6", "python2.7",
 "dotnet5.0", "dotnetcore2.1", "ruby2.5", "nodejs10.x", "nodejs8.10", "nodejs4.3",
 "nodejs6.10", "dotnetcore1.0", "dotnetcore2.0", "nodejs4.3-edge", "nodejs"] <<Lambda
 function is using a deprecated runtime.>>
            }
        }
    }
}

rule lambda_soon_to_be_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["nodejs16.x", "nodejs14.x", "python3.7", "java8",
 "dotnet7", "go1.x", "ruby2.7", "provided"] <<Lambda function is using a runtime that
 is targeted for deprecation.>>
            }
        }
    }
}
```

Now suppose you write the following `iac/lambda.yaml` CloudFormation template that defines a Lambda function:

```
  Fn:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      CodeUri: src
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      Layers:
        - arn:aws:lambda:us-east-1:111122223333:layer:LambdaInsightsExtension:35
```

After installing the Guard utility, validate your template:

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

The output looks like this:

```
lambda.yaml Status = FAIL
FAILED rules
rules.guard/lambda_soon_to_be_deprecated_runtime
---
Evaluating data lambda.yaml against rules rules.guard
Number of non-compliant resources 1
Resource = Fn {
  Type      = AWS::Lambda::Function
  Rule = lambda_soon_to_be_deprecated_runtime {
    ALL {
      Check =  Runtime not IN
 ["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]
 {
        ComparisonError {
          Message          = Lambda function is using a runtime that is targeted for
 deprecation.
          Error            = Check was not compliant as property [/Resources/
Fn/Properties/Runtime[L:88,C:15]] was not present in [(resolved, Path=[L:0,C:0]
 Value=["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"])]
        }
          PropertyPath    = /Resources/Fn/Properties/Runtime[L:88,C:15]
          Operator        = NOT IN
          Value           = "python3.7"
          ComparedWith    =
 [["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]]
          Code:
                86.  Fn:
                87.    Type: AWS::Lambda::Function
                88.    Properties:
                89.      Runtime: python3.7
                90.      CodeUri: src
                91.      Handler: fn.handler

      }
    }
  }
}
```

Guard allows your developers to see from their local developer workstations that they need to update the template to use a runtime that is allowed by the organization. This happens prior to committing to a code repository and subsequently failing checks within a CI/CD pipeline. As a result, your developers get this feedback on how to develop compliant templates and shift their time to writing code that delivers business value. This control can be applied on the local developer workstation, in a pre-commit validation webhook, and/or in the CI/CD pipeline prior to deployment.

Caveats

If you're using AWS Serverless Application Model (AWS SAM) templates to define Lambda functions, be aware that you need to update the Guard rule to search for the `AWS::Serverless::Function` resource type as follows.

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

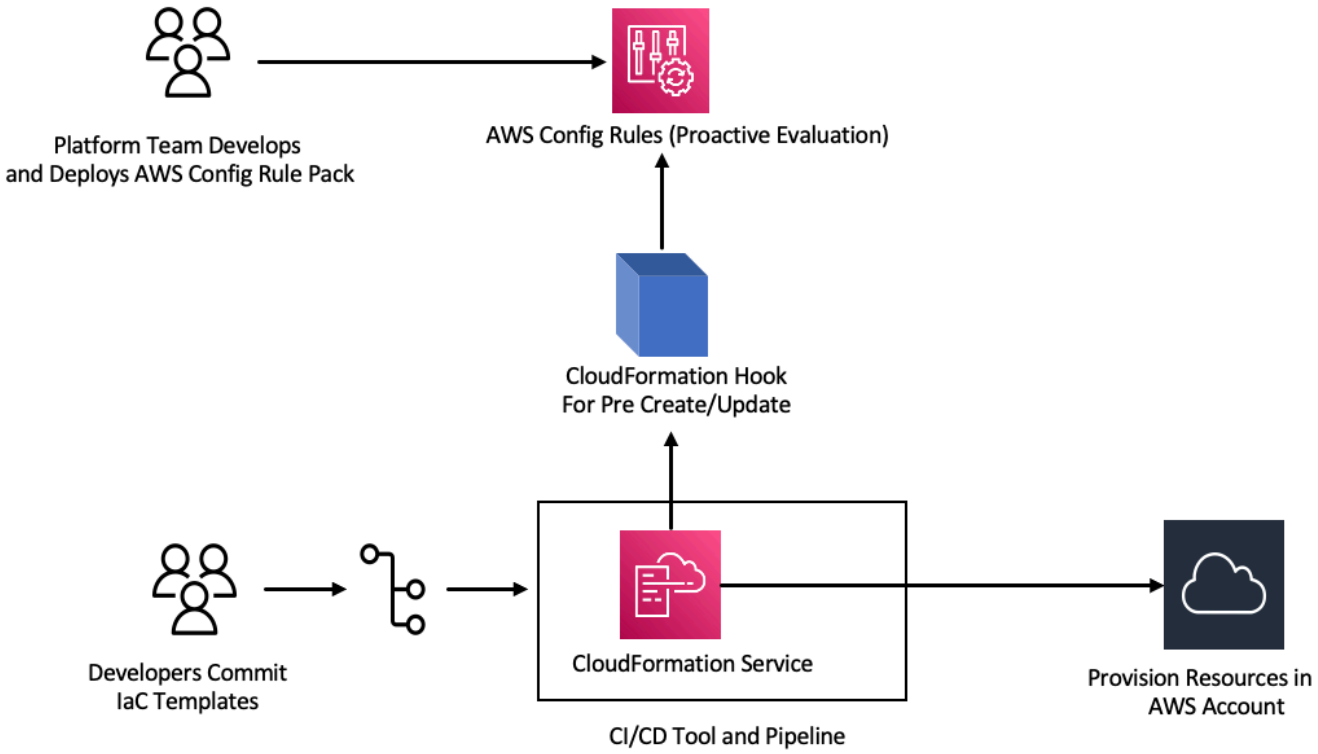
Guard also expects the properties to be included within the resource definition. Meanwhile, AWS SAM templates allow for properties to be specified in a separate Globals section. Properties that are defined in the Globals section are not validated with your Guard rules.

As outlined in the Guard troubleshooting documentation, be aware that Guard doesn't support short-form intrinsics like `!GetAtt` or `!Sub` and instead requires using the expanded forms: `Fn::GetAtt` and `Fn::Sub`. (The earlier example doesn't evaluate the Role property, so the short-form intrinsic was used for simplicity.)

Implement preventative controls for Lambda with AWS Config

It is essential to ensure compliance in your serverless applications as early in the development process as possible. In this topic, we cover how to implement preventative controls using AWS Config. This allows you to implement compliance checks earlier in the development process and enables you to implement the same controls in your CI/CD pipelines. This also standardizes your controls in a centrally managed repository of rules so that you can apply your controls consistently across your AWS accounts.

For example, suppose your compliance administrators defined a requirement to ensure that all Lambda functions include AWS X-Ray tracing. With AWS Config's proactive mode, you can run compliance checks on your Lambda function resources before deployment, reducing the risk of deploying improperly configured Lambda functions and saving developers time by giving them faster feedback on infrastructure as code templates. The following is a visualization of the flow for preventative controls with AWS Config:



Consider a requirement that all Lambda functions must have tracing enabled. In response, the platform team identifies the need for a specific AWS Config rule to run proactively across all

accounts. This rule flags any Lambda function that lacks a configured X-Ray tracing configuration as a non-compliant resource. The team develops a rule, packages it in a conformance pack, and deploys the conformance pack across all AWS accounts to ensure that all accounts in the organization uniformly apply these controls. You can write the rule in AWS CloudFormation Guard 2.x.x syntax, which takes the following form:

```
rule name when condition { assertion }
```

The following is a sample Guard rule that checks to ensure Lambda functions has tracing enabled:

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

The platform team takes further action by mandating that every AWS CloudFormation deployment invokes a pre-create/update hook. They assume full responsibility for developing this hook and configuring the pipeline, strengthening the centralized control of compliance rules and sustaining their consistent application across all deployments. To develop, package, and register a hook, see Developing AWS CloudFormation Hooks in the CloudFormation Command Line Interface (CFN-CLI) documentation. You can use the CloudFormation CLI to create the hook project:

```
cfn init
```

This command asks you for some basic information about your hook project and creates a project with following files in it:

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
template.yml
hook-role.yaml
```

As a hook developer, you need to add the desired target resource type in the `<hook-name>.json` configuration file. In the configuration below, a hook is configured to execute before any Lambda function is created using CloudFormation. You can add similar handlers for `preUpdate` and `preDelete` actions as well.

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
            "permissions": []
        }
    }
```

You also need to ensure that the CloudFormation hook has appropriate permissions to call the AWS Config APIs. You can do that by updating the role definition file named `hook-role.yaml`. The role definition file has the following trust policy by default, which allows CloudFormation to assume the role.

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

To allow this hook to call config APIs, you must add following permissions to the Policy statement. Then you submit the hook project using the `cfn submit` command, where CloudFormation creates a role for you with the required permissions.

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - "config:Describe*"
                  - "config:Get*"
                  - "config:List*"
                  - "config:SelectResourceConfig"
                Resource: "*
```

Next, you need to write a Lambda function in a `src/handler.py` file. Within this file, you find methods named `preCreate`, `preUpdate`, and `preDelete` already created when you initiated the project. You aim to write a common, reusable function that calls the AWS Config `StartResourceEvaluation` API in proactive mode using the AWS SDK for Python (Boto3). This API call takes resource properties as input and evaluates the resource against the rule definition.

```
def validate_lambda_tracing_config(resource_type, function_properties:
 MutableMapping[str, Any]) -> ProgressEvent:
  LOG.info("Fetching proactive data")
  config_client = boto3.client('config')
  resource_specs = {
      'ResourceId': 'MyFunction',
      'ResourceType': resource_type,
      'ResourceConfiguration': json.dumps(function_properties),
      'ResourceConfigurationSchemaType': 'CFN_RESOURCE_SCHEMA'
  }
  LOG.info("Resource Specifications:", resource_specs)
  eval_response = config_client.start_resource_evaluation(EvaluationMode='PROACTIVE',
 ResourceDetails=resource_specs, EvaluationTimeout=60)
  ResourceEvaluationId = eval_response.ResourceEvaluationId
  compliance_response =
 config_client.get_compliance_details_by_resource(ResourceEvaluationId=ResourceEvaluationId)
  LOG.info("Compliance Verification:",
 compliance_response.EvaluationResults[0].ComplianceType)
  if "NON_COMPLIANT" == compliance_response.EvaluationResults[0].ComplianceType:
      return ProgressEvent(status=OperationStatus.FAILED, message="Lambda function
 found with no tracing enabled : FAILED", errorCode=HandlerErrorCode.NonCompliant)
  else:
      return ProgressEvent(status=OperationStatus.SUCCESS, message="Lambda function
 found with tracing enabled : PASS.")
```

Now you can call the common function from the handler for the pre-create hook. Here's an example of the handler:

```
@hook.handler(HookInvocationPoint.CREATE_PRE_PROVISION)
def pre_create_handler(
        session: Optional[SessionProxy],
        request: HookHandlerRequest,
        callback_context: MutableMapping[str, Any],
        type_configuration: TypeConfigurationModel
) -> ProgressEvent:
    LOG.info("Starting execution of the hook")
```

```
    target_name = request.hookContext.targetName
    LOG.info("Target Name:", target_name)
    if "AWS::Lambda::Function" == target_name:
        return validate_lambda_tracing_config(target_name,
            request.hookContext.targetModel.get("resourceProperties")
        )
    else:
        raise exceptions.InvalidRequest(f"Unknown target type: {target_name}")
```

After this step you can register the hook and configure it to listen to all AWS Lambda function creation events.

A developer prepares the infrastructure as code (IaC) template for a serverless microservice using Lambda. This preparation includes adherence to internal standards, followed by locally testing and committing the template to the repository. Here's an example IaC template:

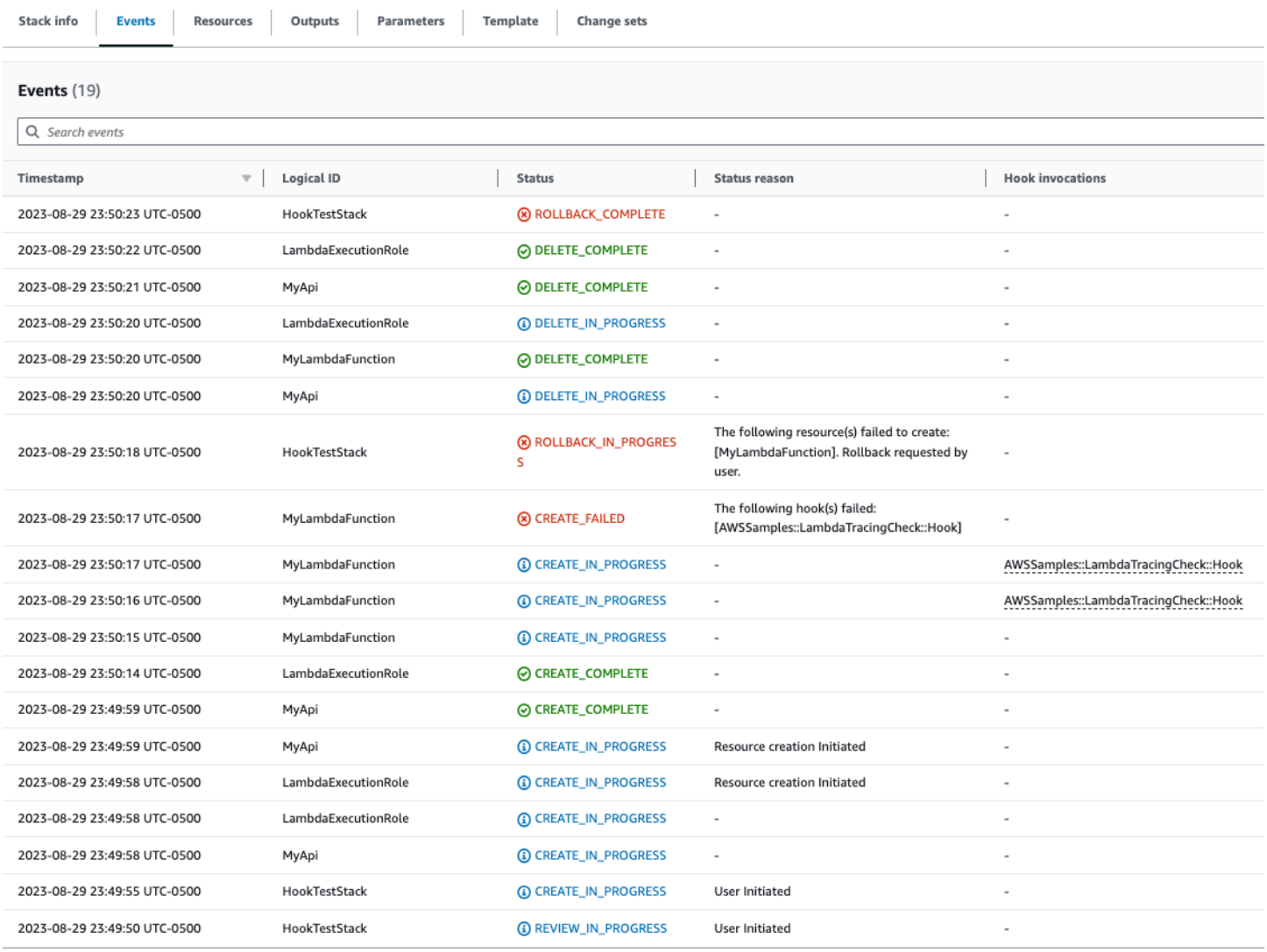
```
MyLambdaFunction:
Type: 'AWS::Lambda::Function'
Properties:
  Handler: index.handler
  Role: !GetAtt LambdaExecutionRole.Arn
  FunctionName: MyLambdaFunction
  Code:
    ZipFile: |
      import json

      def handler(event, context):
          return {
              'statusCode': 200,
              'body': json.dumps('Hello World!')
          }
  Runtime: python3.14
  TracingConfig:
      Mode: PassThrough
  MemorySize: 256
  Timeout: 10
```

As part of the CI/CD process, when the CloudFormation template is deployed, the CloudFormation service invokes the pre-create/update hook right before provisioning `AWS::Lambda::Function` resource type. The hook utilizes AWS Config rules running in proactive mode to verify that the Lambda function configuration includes the mandated tracing configuration. The response from the hook determines the next step. If compliant, the hook signals success, and CloudFormation

proceeds to provision the resources. If not, the CloudFormation stack deployment fails, the pipeline comes to an immediate halt, and the system records the details for subsequent review. Compliance notifications are sent to the relevant stakeholders.

You can find the hook success/fail information in the CloudFormation console:



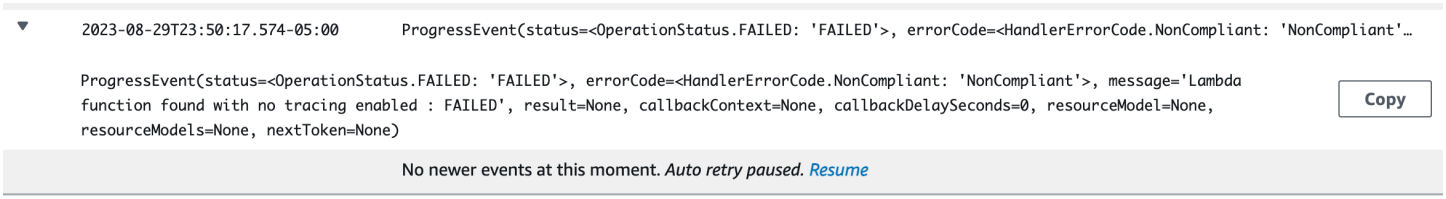
Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Events (19)

Search events

Timestamp	Logical ID	Status	Status reason	Hook invocations
2023-08-29 23:50:23 UTC-0500	HookTestStack	ROLLBACK_COMPLETE	-	-
2023-08-29 23:50:22 UTC-0500	LambdaExecutionRole	DELETE_COMPLETE	-	-
2023-08-29 23:50:21 UTC-0500	MyApi	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	LambdaExecutionRole	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:20 UTC-0500	MyLambdaFunction	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	MyApi	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:18 UTC-0500	HookTestStack	ROLLBACK_IN_PROGRESS	The following resource(s) failed to create: [MyLambdaFunction]. Rollback requested by user.	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_FAILED	The following hook(s) failed: [AWSSamples::LambdaTracingCheck::Hook]	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:16 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:15 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	-
2023-08-29 23:50:14 UTC-0500	LambdaExecutionRole	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:58 UTC-0500	MyApi	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:55 UTC-0500	HookTestStack	CREATE_IN_PROGRESS	User Initiated	-
2023-08-29 23:49:50 UTC-0500	HookTestStack	REVIEW_IN_PROGRESS	User Initiated	-

If you have logs enabled for your CloudFormation hook, you can capture the hook evaluation result. Here is a sample log for a hook with a failed status, indicating that the Lambda function does not have X-Ray enabled:



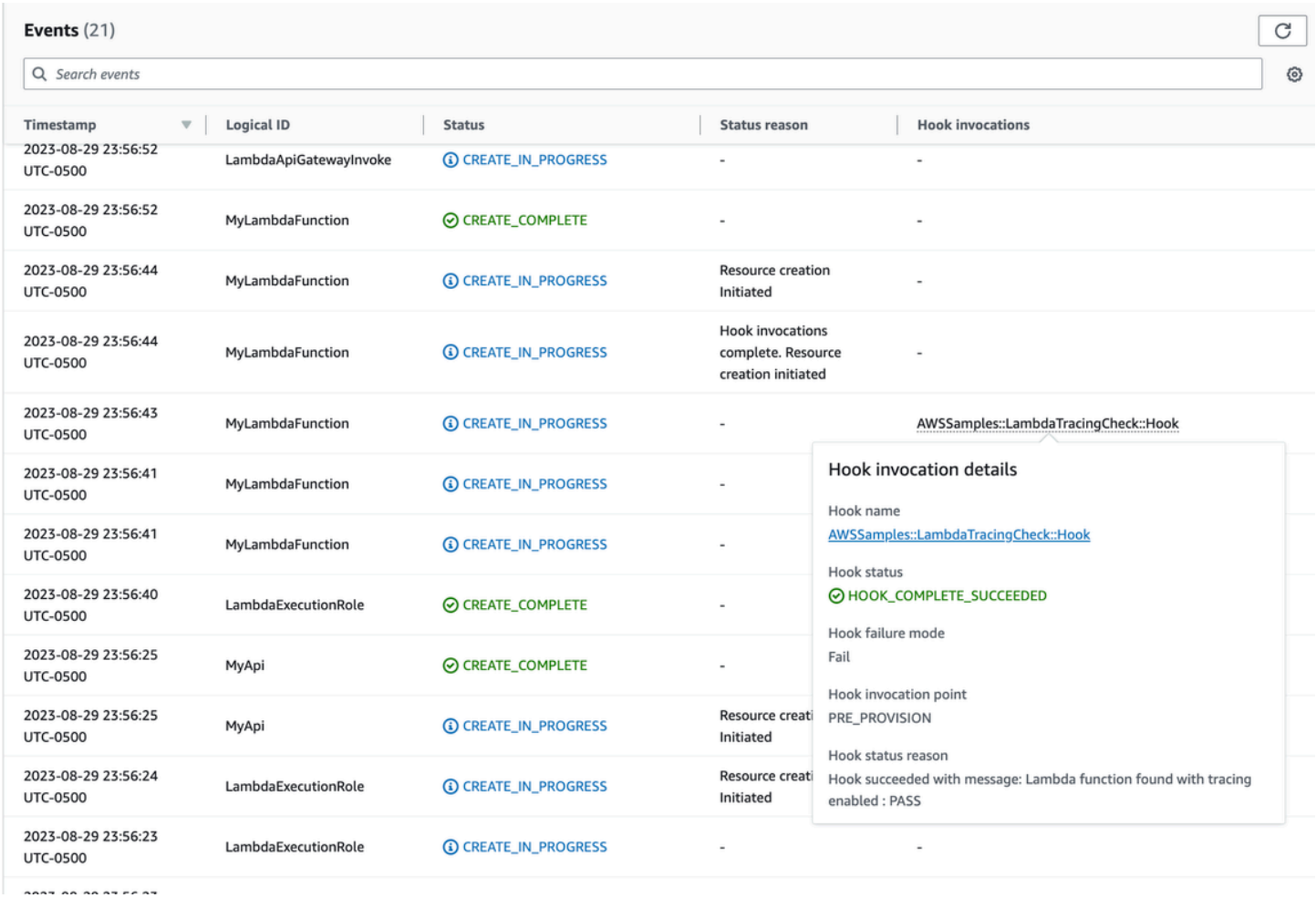
2023-08-29T23:50:17.574-05:00 ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'…

```
ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'>, message='Lambda
function found with no tracing enabled : FAILED', result=None, callbackContext=None, callbackDelaySeconds=0, resourceModel=None,
resourceModels=None, nextToken=None)
```

Copy

No newer events at this moment. *Auto retry paused.* Resume

If the developer chooses to change the IaC to update `TracingConfig Mode` value to `Active` and redeploy, the hook executes successfully and the stack proceeds with creating the Lambda resource.



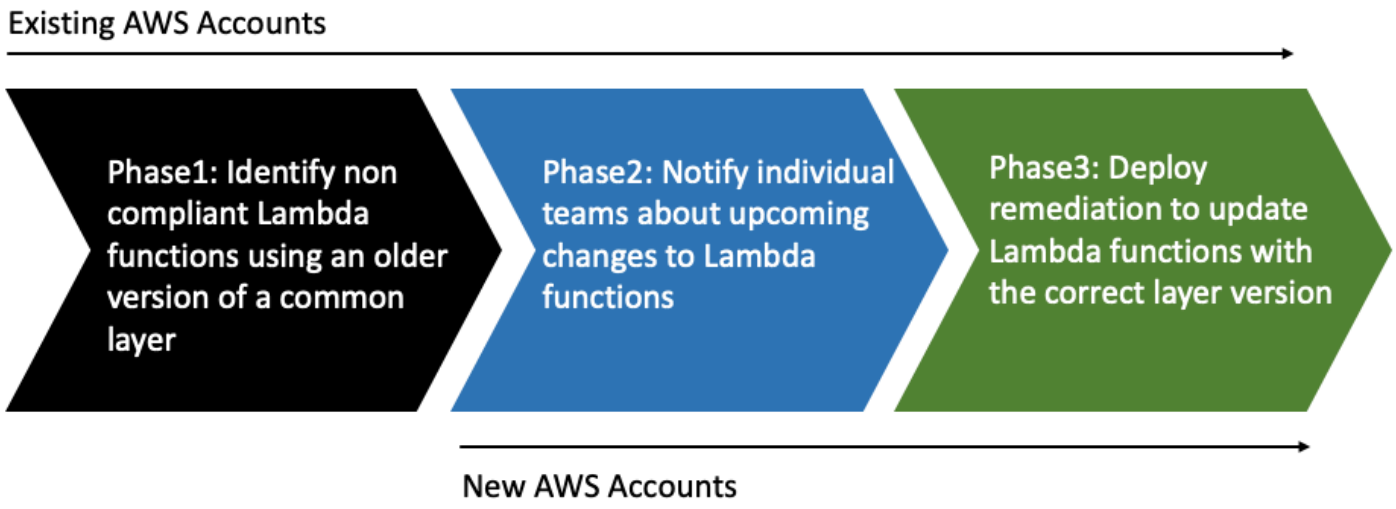
In this way, you can implement preventative controls with AWS Config in proactive mode when developing and deploying serverless resources in your AWS accounts. By integrating AWS Config rules into the CI/CD pipeline, you can identify and optionally block non-compliant resource deployments, such as Lambda functions that lack an active tracing configuration. This ensures that only resources that comply with the latest governance policies are deployed into your AWS environments.

Detect non-compliant Lambda deployments and configurations with AWS Config

In addition to proactive evaluation, AWS Config can also reactively detect resource deployments and configurations that do not comply with your governance policies. This is important because governance policies evolve as your organization learns and implements new best practices.

Consider a scenario where you set a brand new policy when deploying or updating Lambda functions: All Lambda functions must always use a specific, approved Lambda layer version. You can configure AWS Config to monitor new or updated functions for layer configurations. If AWS Config detects a function that is not using an approved layer version, it flags the function as a non-compliant resource. You can optionally configure AWS Config to automatically remediate the resource by specifying a remediation action using an AWS Systems Manager automation document. For example, you could write an automation document in Python using the AWS SDK for Python (Boto3), which updates the non-compliant function to point to the approved layer version. Thus, AWS Config serves as both a detective and corrective control, automating compliance management.

Let's break down this process into three important implementation phases:



Phase 1: Identify access resources

Start by activating AWS Config across your accounts and configuring it to record AWS Lambda functions. This allows AWS Config to observe when Lambda functions are created or updated. You can then configure custom policy rules to check for specific policy violations, which use AWS CloudFormation Guard syntax. Guard rules take the following general form:

```
rule name when condition { assertion }
```

Below is a sample rule that checks to ensure that a layer is not set to an old layer version:

```
rule desiredlayer when configuration.layers !empty {
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

Let's understand the rule syntax and structure:

- **Rule name:** The name of the rule in the provided example is `desiredlayer`.
- **Condition:** This clause specifies the condition under which the rule should be checked. In the provided example, the condition is `configuration.layers !empty`. This means the resource should be evaluated only when the `layers` property in the configuration isn't empty.
- **Assertion:** After the `when` clause, an assertion determines what the rule checks. The assertion `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` checks if any of the Lambda layer ARNs do not match the `OldLayerArn` value. If they do not match, the assertion is true and the rule passes; otherwise, it fails.

`CONFIG_RULE_PARAMETERS` is a special set of parameters that is configured with the AWS Config rule. In this case, `OldLayerArn` is a parameter inside `CONFIG_RULE_PARAMETERS`. This allows users to provide a specific ARN value that they consider old or deprecated, and then the rule checks if any Lambda functions are using this old ARN.

Phase 2: Visualize and design

AWS Config gathers configuration data and stores that data in Amazon Simple Storage Service (Amazon S3) buckets. You can use Amazon Athena to query this data directly from your S3 buckets. With Athena, you can aggregate this data at the organizational level, generating a holistic view of your resource configurations across all your accounts. To set up aggregation of resource configuration data, see Visualizing AWS Config data using Athena and Amazon Quick on the AWS Cloud Operations and Management blog.

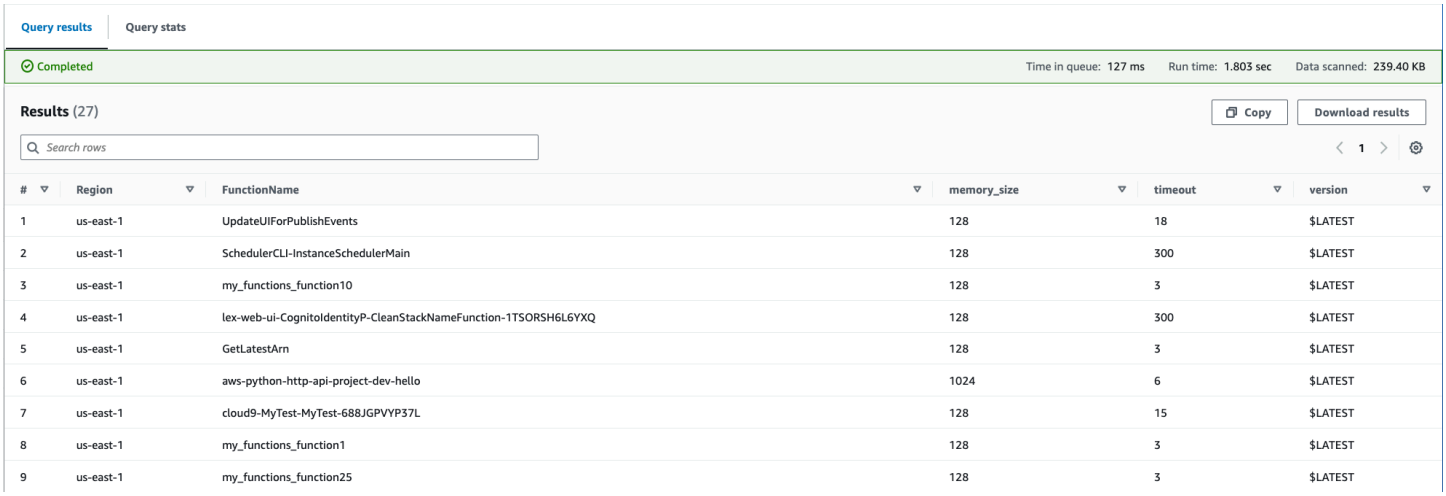
The following is a sample Athena query to identify all Lambda functions using a particular layer ARN:

```
WITH unnested AS (
```

```
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.version') AS version
  FROM
    unnested
  WHERE
    lambda_configuration LIKE '%arn:aws:lambda:us-
east-1:111122223333:layer:AnyGovernanceLayer:24%'
```

Here are results from the query:



Query results | Query stats

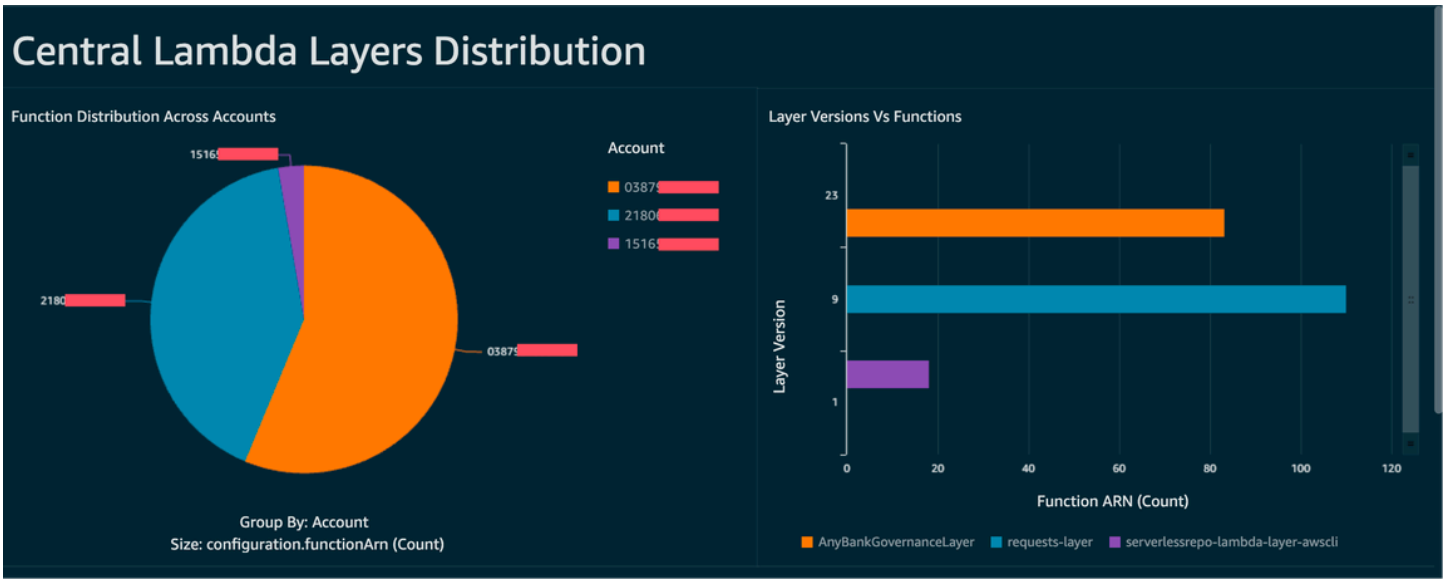
Completed — Time in queue: 127 ms — Run time: 1.803 sec — Data scanned: 239.40 KB

Results (27) — Copy — Download results

Search rows — 1

#	Region	FunctionName	memory_size	timeout	version
1	us-east-1	UpdateUIForPublishEvents	128	18	$LATEST
2	us-east-1	SchedulerCLI-InstanceSchedulerMain	128	300	$LATEST
3	us-east-1	my_functions_function10	128	3	$LATEST
4	us-east-1	lex-web-ui-CognitoIdentityP-CleanStackNameFunction-1TSORSH6L6YXQ	128	300	$LATEST
5	us-east-1	GetLatestArn	128	3	$LATEST
6	us-east-1	aws-python-http-api-project-dev-hello	1024	6	$LATEST
7	us-east-1	cloud9-MyTest-MyTest-688JGPVYP37L	128	15	$LATEST
8	us-east-1	my_functions_function1	128	3	$LATEST
9	us-east-1	my_functions_function25	128	3	$LATEST

With the AWS Config data aggregated across the organization, you can then create a dashboard using Amazon Quick. By importing your Athena results into Quick, you can visualize how well your Lambda functions adhere to the layer version rule. This dashboard can highlight compliant and non-compliant resources, which helps you to determine your enforcement policy, as outlined in the next section. The following image is an example dashboard that reports on the distribution of layer versions applied to functions within the organization.



Phase 3: Implement and enforce

You can now optionally pair your layer version rule that you created in phase 1 with a remediation action via a Systems Manager automation document, which you author as a Python script written with AWS SDK for Python (Boto3). The script calls the UpdateFunctionConfiguration API action for each Lambda function, updating the function configuration with the new layer ARN. Alternatively, you could have the script submit a pull request to the code repository to update the layer ARN. This way future code deployments are also updated with the correct layer ARN.

Lambda code signing with AWS Signer

AWS Signer is a fully managed code-signing service that allows you to validate your code against a digital signature to confirm that code is unaltered and from a trusted publisher. AWS Signer can be used in conjunction with AWS Lambda to verify that functions and layers are unaltered prior to deployment into your AWS environments. This protects your organization from malicious actors who might have gained credentials to create new or update existing functions.

To set up code signing for your Lambda functions, start by creating an S3 bucket with versioning enabled. After that, create a signing profile with AWS Signer, specify Lambda as the platform and then specify a period of days in which the signing profile is valid. Example:

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

Then use the signing profile and create a signing configuration with Lambda. You have to specify what to do when the signing configuration sees an artifact that does not match a digital signature that it expected: warn (but allow the deployment) or enforce (and block the deployment). The example below is configured to enforce and block deployments.

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

You now have AWS Signer configured with Lambda to block untrusted deployments. Let's assume you've finished coding a feature request and are now ready to deploy the function. The first step is to zip the code up with the appropriate dependencies and then sign the artifact using the signing profile that you created. You can do this by uploading the zip artifact to the S3 bucket and then starting a signing job.

```
aws signer start-signing-job \
--source 's3={bucketName=your-versioned-bucket,key=your-prefix/your-zip-
artifact.zip,version=QyaJ3c4qa50LXV.9VaZgXHlsGbvCXxpT}' \
--destination 's3={bucketName=your-versioned-bucket,prefix=your-prefix/}' \
--profile-name your-signer-id
```

You get an output as follows, where the `jobId` is the object that is created in the destination bucket and prefix and `jobOwner` is the 12-digit AWS account ID where the job was run.

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

And now you can deploy your function using the signed S3 object and the code signing configuration that you created.

```
  Fn:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://your-versioned-bucket/your-prefix/87a3522b-5c0b-4d7d-
b4e0-4255a8e05388.zip
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      CodeSigningConfigArn: !Ref pSigningConfigArn
```

You can alternatively test a function deployment with the original unsigned source zip artifact. The deployment should fail with the following message:

```
Lambda cannot deploy the function. The function or layer might be signed using a
 signature that the client is not configured to accept. Check the provided signature
 for unsigned.
```

If you are building and deploying your functions using the AWS Serverless Application Model (AWS SAM), the package command handles uploading the zip artifact to S3 and also starts the signing job and gets the signed artifact. You can do this with the following command and parameters:

```
sam package -t your-template.yaml \
--output-template-file your-output.yaml \
--s3-bucket your-versioned-bucket \
```

```
--s3-prefix your-prefix \
--signing-profiles your-signer-id
```

AWS Signer helps you verify that zip artifacts that are deployed into your accounts are trusted for deployment. You can include the process above in your CI/CD pipelines and require that all functions have a code signing configuration attached using the techniques outlined in previous topics. By using code signing with your Lambda function deployments, you prevent malicious actors who might have gotten credentials to create or update functions from injecting malicious code in your functions.

Automate security assessments for Lambda with Amazon Inspector

Amazon Inspector is a vulnerability management service that continually scans workloads for known software vulnerabilities and unintended network exposure. Amazon Inspector creates a finding that describes the vulnerability, identifies the affected resource, rates the severity of the vulnerability, and provides remediation guidance.

Amazon Inspector support provides continuous, automated security vulnerability assessments for Lambda functions and layers. Amazon Inspector provides two scan types for Lambda:

- **Lambda standard scanning (default):** Scans application dependencies within a Lambda function and its layers for package vulnerabilities.
- **Lambda code scanning**: Scans the custom application code in your functions and layers for code vulnerabilities. You can either activate Lambda standard scanning or activate Lambda standard scanning together with Lambda code scanning.

To enable Amazon Inspector, navigate to the Amazon Inspector console, expand the **Settings** section, and choose **Account Management**. On the **Accounts** tab, choose **Activate**, and then select one of the scan options.

You can enable Amazon Inspector for multiple accounts and delegate permissions to manage Amazon Inspector for the organization to specific accounts while setting up Amazon Inspector. While enabling, you need to grant Amazon Inspector permissions by creating the role: `AWSServiceRoleForAmazonInspector2`. The Amazon Inspector console allows you to create this role using a one-click option.

For Lambda standard scanning, Amazon Inspector initiates vulnerability scans of Lambda functions in the following situations:

- As soon as Amazon Inspector discovers an existing Lambda function.
- When you deploy a new Lambda function.
- When you deploy an update to the application code or dependencies of an existing Lambda function or its layers.
- Whenever Amazon Inspector adds a new common vulnerabilities and exposures (CVE) item to its database, and that CVE is relevant to your function.

For Lambda code scanning, Amazon Inspector evaluates your Lambda function application code using automated reasoning and machine learning that analyzes your application code for overall security compliance. If Amazon Inspector detects a vulnerability in your Lambda function application code, Amazon Inspector produces a detailed **Code Vulnerability** finding. For a list of possible detections, see the Amazon CodeGuru Detector Library.

To view the findings, go to the Amazon Inspector console. On the **Findings** menu, choose **By Lambda function** to display the security scan results that were performed on Lambda functions.

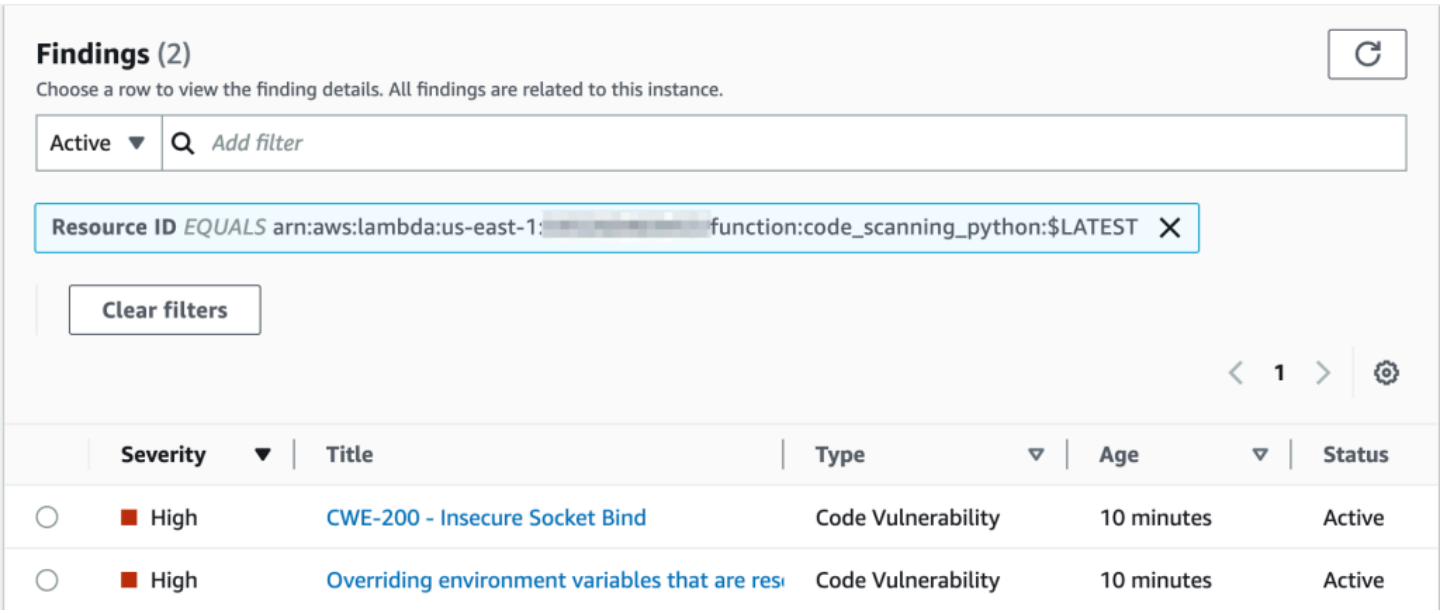
To exclude a Lambda function from standard scanning, tag the function with the following key-value pair:

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

To exclude a Lambda function from code scans, tag the function with the following key-value pair:

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

For example, as shown in following image, Amazon Inspector automatically detects vulnerabilities and categorizes the findings of type **Code Vulnerability**, which indicates that the vulnerability is in the code of the function, and not in one of the code-dependent libraries. You can check these details for a specific function or multiple functions at once.



You can dive further into each of these findings and learn how to remediate the issue.

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior.

Finding ID: arn:aws:inspector2:us-east-1:[redacted]:finding/[redacted]

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior or failure of the Lambda function.

Finding overview	
AWS account ID	[redacted]
Severity	High
Type	Code Vulnerability
Detector name	Override of reserved variable names in a Lambda function
Relevant CWE	--
Rule ID	Rule-434311
Detector tags	#availability, #aws-python-sdk, #aws-lambda, #data-integrity, #maintainability, #security, #security-context, #python
Fix available	Yes
Created at	March 29, 2023 10:08 AM (UTC-04:00)

Vulnerability details	
File path	lambda_function.py

Vulnerability location

```
3  import socket
4
5  def lambda_handler(event, context):
6
7      # print("Scenario 1");
8      os.environ['_HANDLER'] = 'hello'
9      # print("Scenario 1 ends")
10
11     # print("Scenario 2");
12     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     s.bind(('',0))
```

Suggested remediation

Your code attempts to override an environment variable that is reserved by the Lambda runtime environment. This can lead to unexpected behavior and might break the execution of your Lambda function.

While working with your Lambda functions, ensure that you comply with the naming conventions for your Lambda functions. For more information, see Working with Lambda environment variables.

You are responsible for the remediation suggestions that you accept. Always review remediation suggestions before accepting them. You might need to make edits to remediation suggestions to ensure that your code does what you intended.

Implement observability for Lambda security and compliance

AWS Config is a useful tool to find and fix non-compliant AWS Serverless resources. Every change you make to your serverless resources is recorded in AWS Config. Additionally, AWS Config allows you to store configuration snapshot data on S3. You can use Amazon Athena and Amazon Quick to make dashboards and see AWS Config data. In Detect non-compliant Lambda deployments and configurations with AWS Config, we discussed how we can visualize a certain configuration like Lambda layers. This topic expands on these concepts.

Visibility into Lambda configurations

You can use queries to pull important configurations like AWS account ID, Region, AWS X-Ray tracing configuration, VPC configuration, memory size, runtime, and tags. Here is a sample query you can use to pull this information from Athena:

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    account_id,
    tags,
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.runtime') AS version
    json_extract_scalar(lambda_json, '$.vpcConfig.SubnetIds') AS vpcConfig
    json_extract_scalar(lambda_json, '$.tracingConfig.mode') AS tracingConfig
```

```
  FROM
    unnested
```

You can use the query to build an Quick dashboard and visualize the data. To aggregate AWS resource configuration data, create tables in Athena, and build Quick dashboards on the data from Athena, see Visualizing AWS Config data using Athena and Amazon Quick on the AWS Cloud Operations and Management blog. Notably, this query also retrieves tag information for the functions. This allows for deeper insights into your workloads and environments, especially if you employ custom tags.



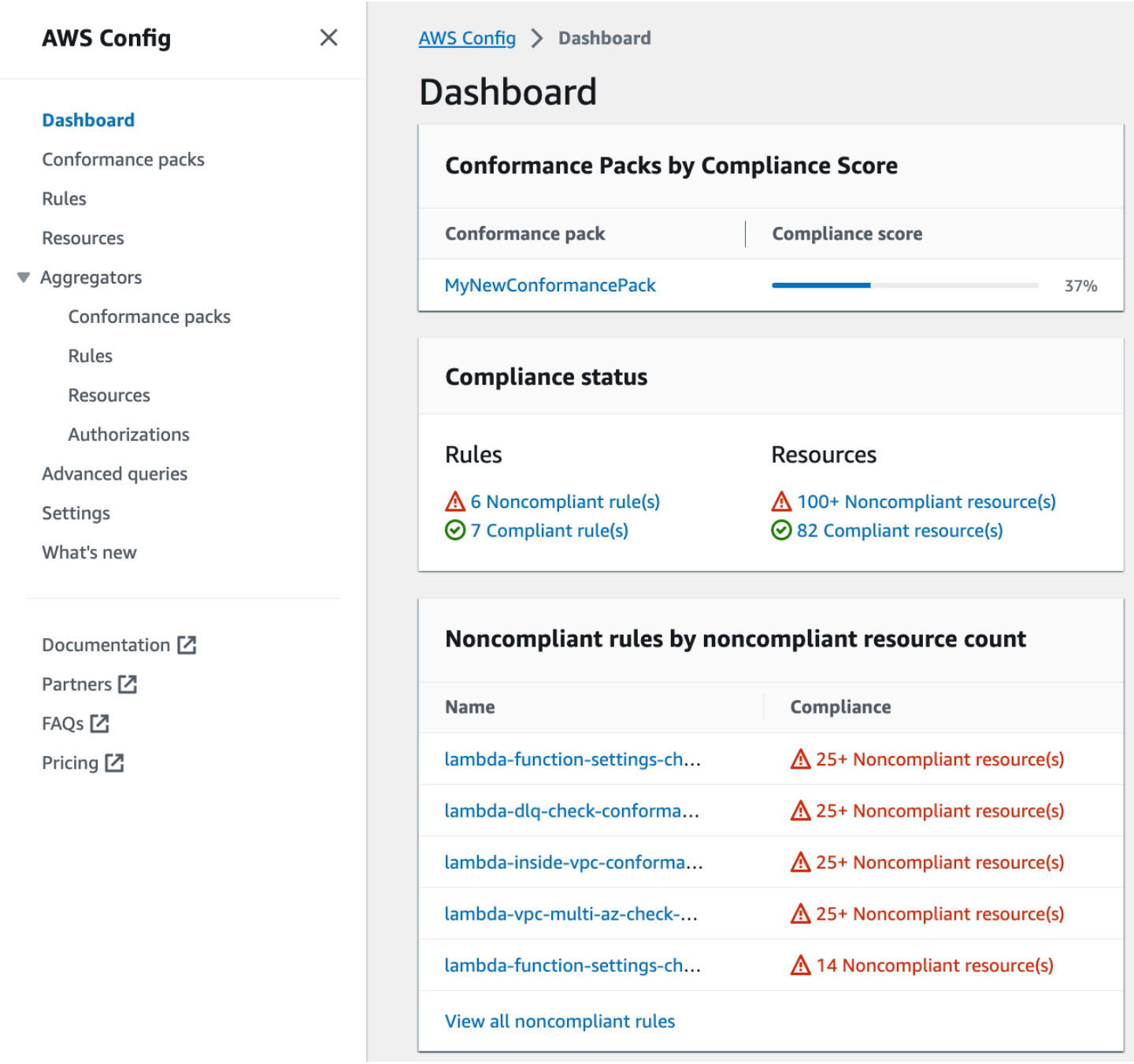
For more information on actions that you can take, see the Addressing the observability findings section later in this topic.

Visibility into Lambda compliance

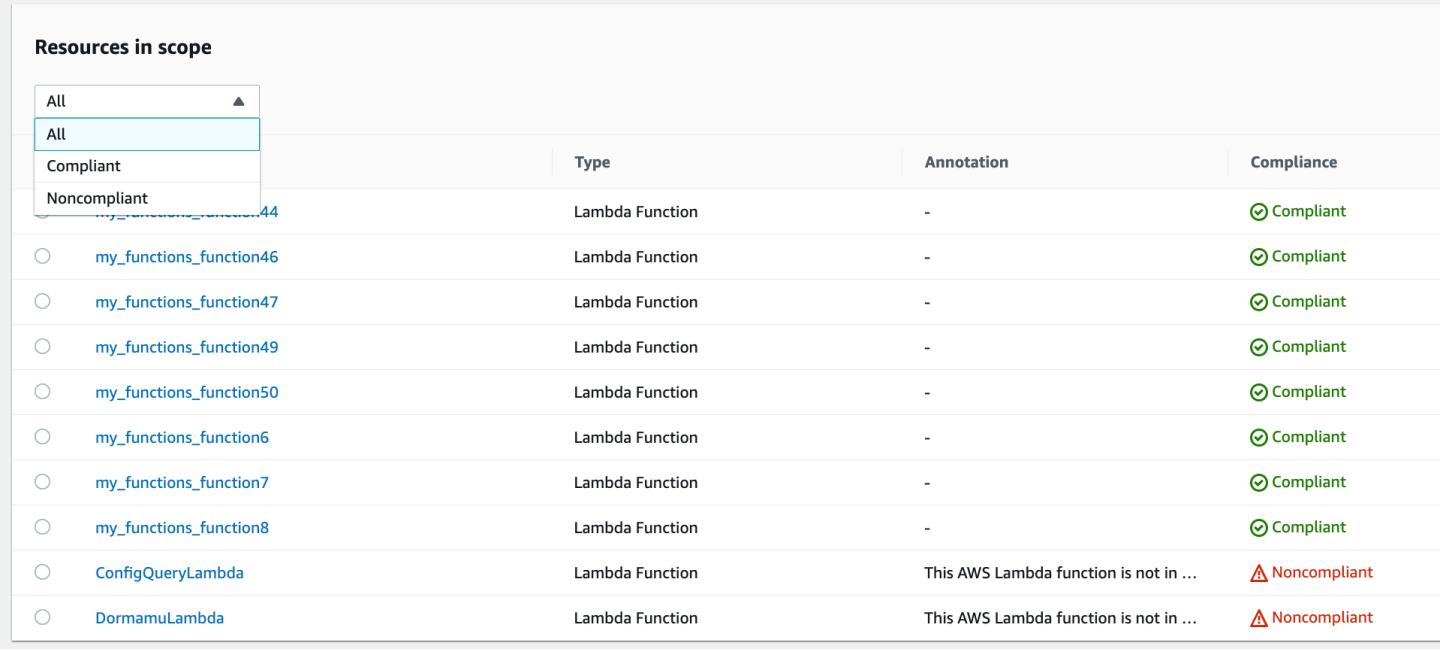
With the data generated by AWS Config, you can create organization-level dashboards to monitor compliance. This allows for consistent tracking and monitoring of:

- Compliance packs by compliance score
- Rules by non-compliant resources

- Compliance status

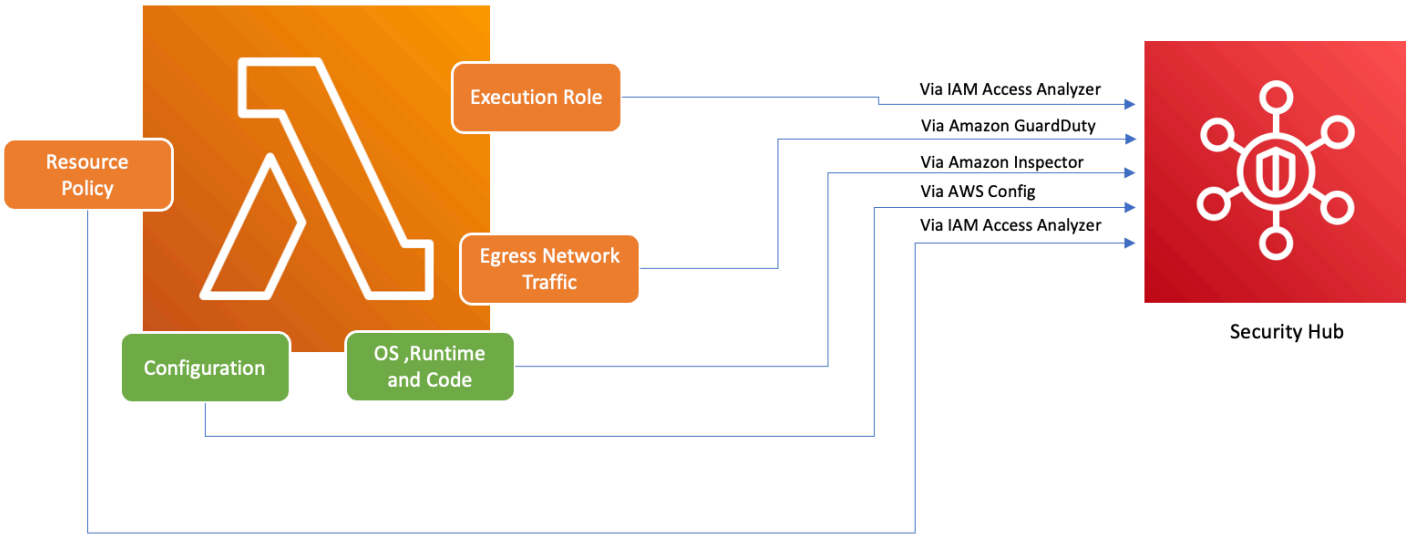


Check each rule to identify non-compliant resources for that rule. For example, if your organization mandates that all Lambda functions must be associated with a VPC and if you have deployed an AWS Config rule to identify compliance, you can select the `lambda-inside-vpc` rule in the list above.



For more information on actions that you can take, see the Addressing the observability findings section below.

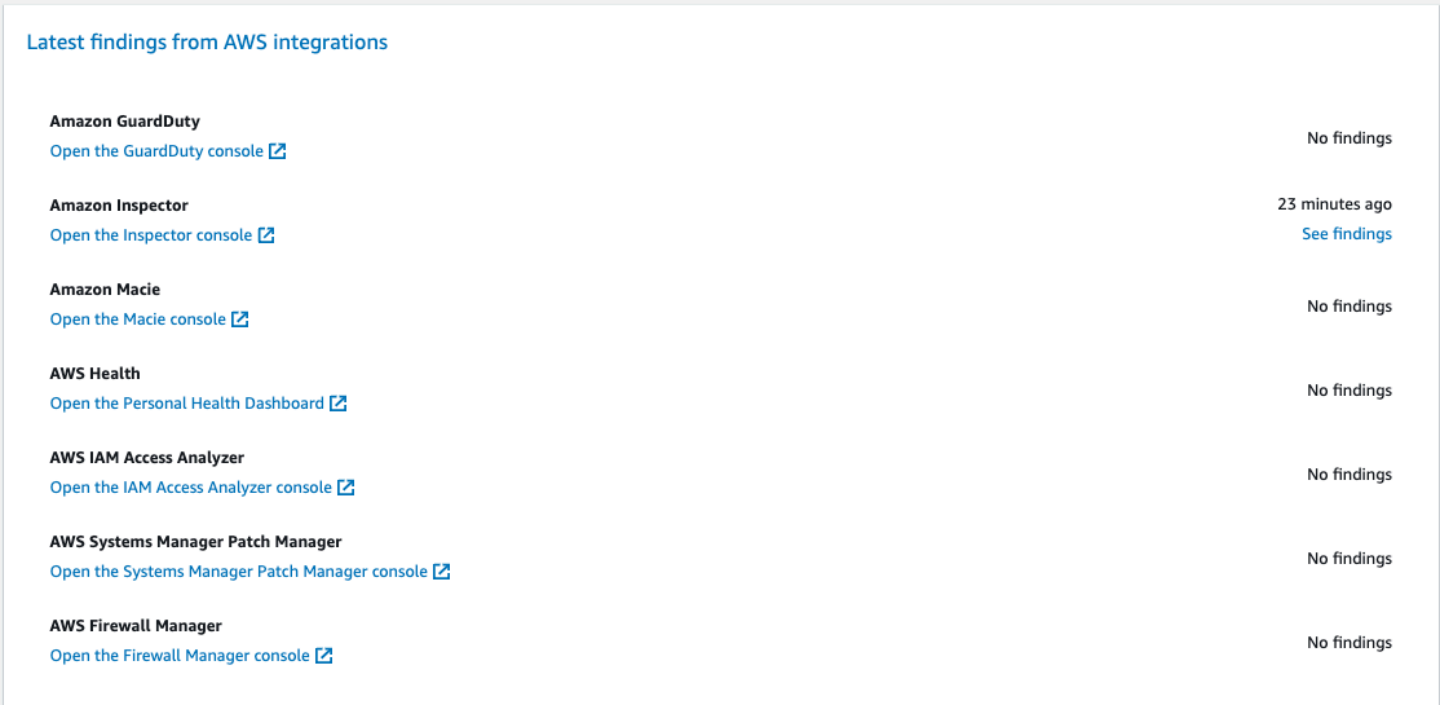
Visibility into Lambda function boundaries using Security Hub CSPM



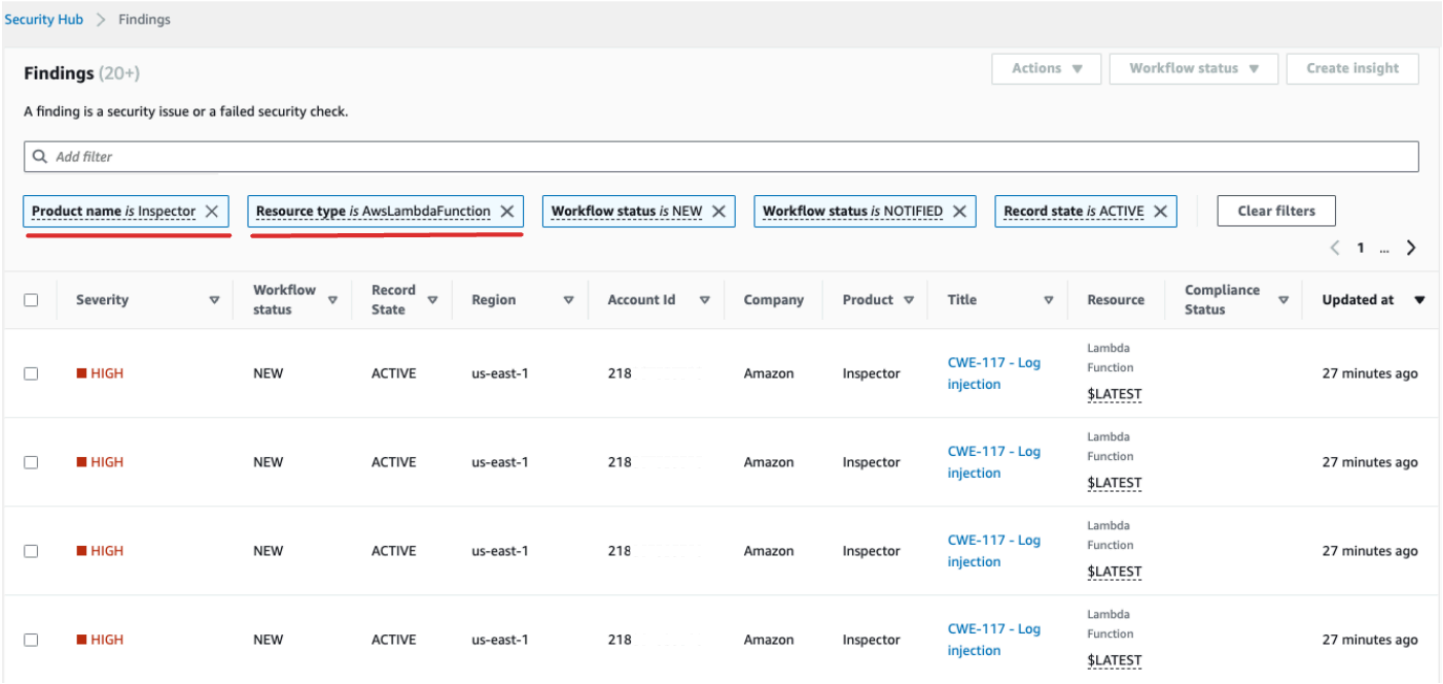
To ensure that AWS services including Lambda are used securely, AWS introduced the Foundational Security Best Practices v1.0.0. This set of best practices provides clear guidelines for securing resources and data in the AWS environment, emphasizing the importance of maintaining a strong security posture. The AWS Security Hub CSPM complements this by offering a unified security and compliance center. It aggregates, organizes, and prioritizes security findings from multiple

AWS services like Amazon Inspector, AWS Identity and Access Management Access Analyzer, and Amazon GuardDuty.

If you have Security Hub CSPM, Amazon Inspector, IAM Access Analyzer, and GuardDuty enabled within your AWS organization, Security Hub CSPM automatically aggregates findings from these services. For instance, let's consider Amazon Inspector. Using Security Hub CSPM, you can efficiently identify code and package vulnerabilities in Lambda functions. In the Security Hub CSPM console, navigate to the bottom section labeled **Latest findings from AWS integrations**. Here, you can view and analyze findings sourced from various integrated AWS services.



To see details, choose the **See findings** link in the second column. This displays a list of findings filtered by product, such as Amazon Inspector. To limit your search to Lambda functions, set `ResourceType` to `AwsLambdaFunction`. This displays findings from Amazon Inspector related to Lambda functions.



For GuardDuty, you can identify suspicious network traffic patterns. Such anomalies might suggest the existence of potentially malicious code within your Lambda function.

With IAM Access Analyzer, you can check policies, especially those with condition statements that grant function access to external entities. Moreover, IAM Access Analyzer evaluates permissions set when using the AddPermission operation in the Lambda API alongside an `EventSourceToken`.

Addressing the observability findings

Given the wide-ranging configurations possible for Lambda functions and their distinct requirements, a standardized automation solution for remediation might not suit every situation. Additionally, changes are implemented differently across various environments. If you encounter any configuration that seems non-compliant, consider the following guidelines:

1. **Tagging strategy**

   We recommend implementing a comprehensive tagging strategy. Each Lambda function should be tagged with key information such as:

   - **Owner:** The person or team responsible for the function.
   - **Environment:** Production, staging, development, or sandbox.
   - **Application:** The broader context to which this function belongs, if applicable.

2. **Owner outreach**

Instead of automating the breaking changes (like VPC configuration adjustment), proactively contact the owners of non-compliant functions (identified by the owner tag) providing them sufficient time to either:

- Adjust non-compliant configurations on Lambda functions.
- Provide an explanation and request an exception, or refine the compliance standards.

3. **Maintain a configuration management database (CMDB)**

   While tags can provide immediate context, maintaining a centralized CMDB can provide deeper insights. It can hold more granular information about each Lambda function, its dependencies, and other critical metadata. A CMDB is an invaluable resource for auditing, compliance checks, and identifying function owners.

As the landscape of serverless infrastructure continually evolves, it's essential to adopt a proactive stance towards monitoring. With tools like AWS Config, Security Hub CSPM, and Amazon Inspector, potential anomalies or non-compliant configurations can be swiftly identified. However, tools alone cannot ensure total compliance or optimal configurations. It's crucial to pair these tools with well-documented processes and best practices.

- **Feedback loop:** Once remediation steps are undertaken, ensure there's a feedback loop. This means periodically revisiting non-compliant resources to confirm if they've been updated or are still running with the same issues.
- **Documentation:** Always document the observations, actions taken, and any exceptions granted. Proper documentation not only helps during audits but also aids in enhancing the process for better compliance and security in the future.
- **Training and awareness:** Ensure that all stakeholders, especially Lambda function owners, are regularly trained and made aware of best practices, organizational policies, and compliance mandates. Regular workshops, webinars, or training sessions can go a long way in ensuring everyone is on the same page when it comes to security and compliance.

In conclusion, while tools and technologies provide robust capabilities to detect and flag potential issues, the human element—understanding, communication, training, and documentation—remains pivotal. Together, they form a potent combination to ensure that your Lambda functions and broader infrastructure remain compliant, secure, and optimized for your business needs.

Compliance validation for AWS Lambda

Third-party auditors assess the security and compliance of AWS Lambda as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see AWS services in scope by compliance program. For general information, see AWS compliance programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading reports in AWS artifact.

Your compliance responsibility when using Lambda is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. You can implement governance controls to ensure that your company's Lambda functions meet your compliance requirements. For more information, see Create a governance strategy for Lambda functions and layers.

Resilience in AWS Lambda

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS global infrastructure.

In addition to the AWS global infrastructure, Lambda offers several features to help support your data resiliency and backup needs.

- **Versioning** – You can use versioning in Lambda to save your function's code and configuration as you develop it. Together with aliases, you can use versioning to perform blue/green and rolling deployments. For details, see Manage Lambda function versions.
- **Scaling** – When your function receives a request while it's processing a previous request, Lambda launches another instance of your function to handle the increased load. Lambda automatically scales to handle 1,000 concurrent executions per Region, a quota that can be increased if needed. For details, see Understanding Lambda function scaling.

- **High availability** – Lambda runs your function in multiple Availability Zones to ensure that it is available to process events in case of a service interruption in a single zone. If you configure your function to connect to a virtual private cloud (VPC) in your account, specify subnets in multiple Availability Zones to ensure high availability. For details, see Giving Lambda functions access to resources in an Amazon VPC.
- **Reserved concurrency** – To make sure that your function can always scale to handle additional requests, you can reserve concurrency for it. Setting reserved concurrency for a function ensures that it can scale to, but not exceed, a specified number of concurrent invocations. This ensures that you don't lose requests due to other functions consuming all of the available concurrency. For details, see Configuring reserved concurrency for a function.
- **Retries** – For asynchronous invocations and a subset of invocations triggered by other services, Lambda automatically retries on error with delays between retries. Other clients and AWS services that invoke functions synchronously are responsible for performing retries. For details, see Understanding retry behavior in Lambda.
- **Dead-letter queue** – For asynchronous invocations, you can configure Lambda to send requests to a dead-letter queue if all retries fail. A dead-letter queue is an Amazon SNS topic or Amazon SQS queue that receives events for troubleshooting or reprocessing. For details, see Adding a dead-letter queue.

Infrastructure security in AWS Lambda

As a managed service, AWS Lambda is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access Lambda through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Securing workloads with public endpoints

For workloads that are accessible publicly, AWS provides a number of features and services that can help mitigate certain risks. This section covers authentication and authorization of application users and protecting API endpoints.

Authentication and authorization

Authentication relates to identity and authorization refers to actions. Use authentication to control who can invoke a Lambda function, and then use authorization to control what they can do. For many applications, IAM is sufficient for managing both control mechanisms.

For applications with external users, such as web or mobile applications, it is common to use JSON Web Tokens (JWTs) to manage authentication and authorization. Unlike traditional, server-based password management, JWTs are passed from the client on every request. They are a cryptographically secure way to verify identity and claims using data passed from the client. For Lambda-based applications, this allows you to secure every call to each API endpoint without relying on a central server for authentication.

You can implement JWTs with Amazon Cognito, a user directory service that can handle registration, authentication, account recovery, and other common account management operations. Amplify Framework provides libraries to simplify integrating this service into your frontend application. You can also consider third-party partner services like Auth0.

Given the critical security role of an identity provider service, it's important to use professional tooling to safeguard your application. It's not recommended that you write your own services to handle authentication or authorization. Any vulnerabilities in custom libraries may have significant implications for the security of your workload and its data.

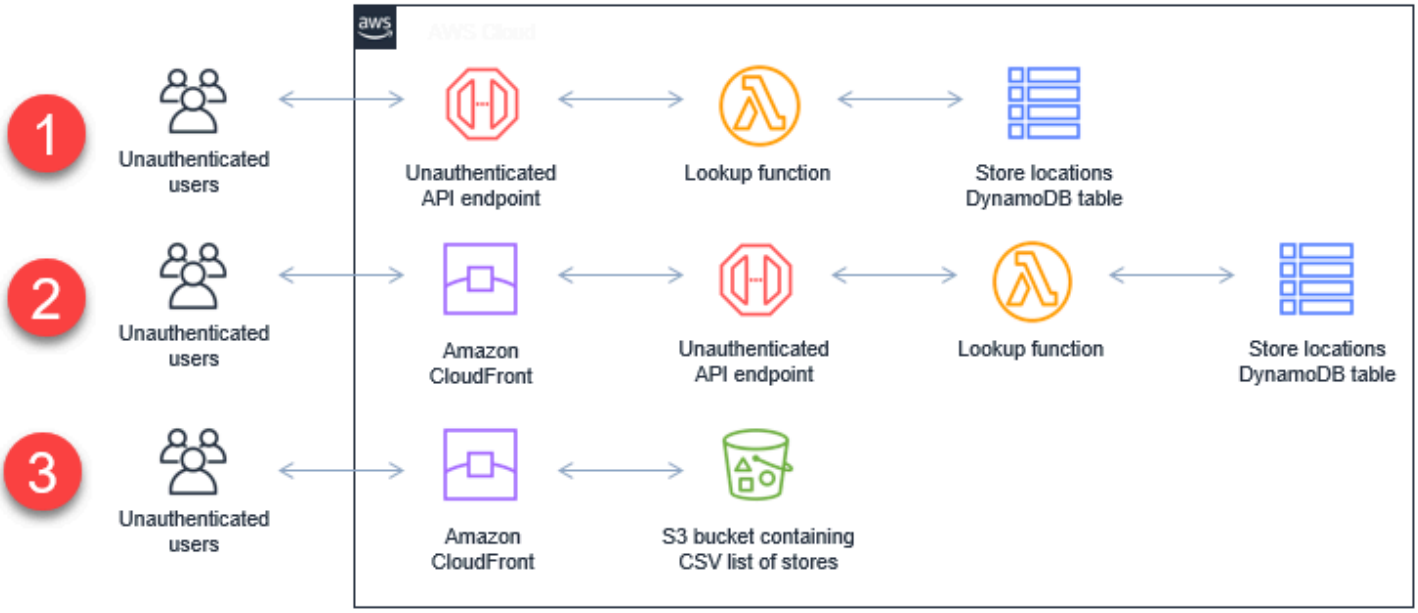
Protecting API endpoints

For serverless applications, the preferred way to serve a backend application publicly is to use Amazon API Gateway. This can help you protect an API from malicious users or spikes in traffic.

API Gateway offers two endpoint types for serverless developers: REST APIs and HTTP APIs. Both support authorization using AWS Lambda, IAM, or Amazon Cognito. When using IAM or Amazon Cognito, incoming requests are evaluated and if they are missing a required token or contain invalid authentication, the request is rejected. You are not charged for these requests and they do not count towards any throttling quotas.

Unauthenticated API routes may be accessed by anyone on the public internet so it's recommended that you limit the use of unauthenticated APIs. If you must use unauthenticated APIs, it's important to protect these against common risks, such as denial-of-service (DoS) attacks. Applying AWS WAF to these APIs can help protect your application from SQL injection and cross-site scripting (XSS) attacks. API Gateway also implements throttling at the AWS account-level and per-client level when API keys are used.

In many cases, the functionality provided by unauthenticated API can be achieved with an alternative approach. For example, a web application may provide a list of customer retail stores from a DynamoDB table to users who are not logged in. This request may originate from a frontend web application or from any other source that calls the URL endpoint. This diagram compares three solutions:



1. This unauthenticated API can be called by anyone on the internet. In a denial of service attack, it's possible to exhaust API throttling limits, Lambda concurrency, or DynamoDB provisioned read capacity on an underlying table.
2. A CloudFront distribution in front of the API endpoint with an appropriate time-to-live (TTL) configuration would absorb most of the traffic in a DoS attack, without changing the underlying solution for fetching the data.
3. Alternatively, for static data that rarely changes, the CloudFront distribution could serve the data' from an Amazon S3 bucket.

Using code signing to verify code integrity with Lambda

Code signing helps ensure that only trusted code is deployed to your Lambda functions. Using AWS Signer, you can create digitally signed code packages for your functions. When you add a code signing configuration to a function, Lambda verifies that all new code deployments are signed by a trusted source. Because code signing validation checks run at deployment time, there is no impact on function execution.

> **Important**
>
> Code signing configurations only prevent new deployments of unsigned code. If you add a code signing configuration to an existing function that has unsigned code, that code keeps running until you deploy a new code package.

When you enable code signing for a function, any layers that you add to the function must also be signed by an allowed signing profile.

There is no additional charge for using AWS Signer or code signing for AWS Lambda.

Signature validation

Lambda performs the following validation checks when you deploy a signed code package to your function:

1. **Integrity**: Validates that the code package has not been modified since it was signed. Lambda compares the hash of the package with the hash from the signature.
2. **Expiry**: Validates that the signature of the code package has not expired.
3. **Mismatch**: Validates that the code package is signed with an allowed signing profile
4. **Revocation**: Validates that the signature of the code package has not been revoked.

When you create a code signing configuration, you can use the UntrustedArtifactOnDeployment parameter to specify how Lambda should respond if the expiry, mismatch, or revocation checks fail. You can choose one of these actions:

- `Warn`: This is the default setting. Lambda allows the deployment of the code package, but issues a warning. Lambda issues a new Amazon CloudWatch metric (`SignatureValidationErrors`) and also stores the warning in the CloudTrail log.

- `Enforce` Lambda issues a warning (the same as for the `Warn` action) and blocks the deployment of the code package.

Topics

- Creating code signing configurations for Lambda
- Configuring IAM policies for Lambda code signing configurations
- Using tags on code signing configurations

Creating code signing configurations for Lambda

To enable code signing for a function, you create a *code signing configuration* and attach it to the function. A code signing configuration defines a list of allowed signing profiles and the policy action to take if any of the validation checks fail.

> **Note**
>
> Functions defined as container images do not support code signing.

Sections

- Configuration prerequisites
- Creating code signing configurations
- Enabling code signing for a function

Configuration prerequisites

Before you can configure code signing for a Lambda function, use AWS Signer to do the following:

- Create one or more signing profiles.
- Use a signing profile to create a signed code package for your function.

Creating code signing configurations

A code signing configuration defines a list of allowed signing profiles and the signature validation policy.

To create a code signing configuration (console)

1. Open the Code signing configurations page of the Lambda console.
2. Choose **Create configuration**.
3. For **Description**, enter a descriptive name for the configuration.
4. Under **Signing profiles**, add up to 20 signing profiles to the configuration.
   a. For **Signing profile version ARN**, choose a profile version's Amazon Resource Name (ARN), or enter the ARN.
   b. To add an additional signing profile, choose **Add signing profiles**.
5. Under **Signature validation policy**, choose **Warn** or **Enforce**.
6. Choose **Create configuration**.

Enabling code signing for a function

To enable code signing for a function, add a code signing configuration to the function.

> **Important**
>
> Code signing configurations only prevent new deployments of unsigned code. If you add a code signing configuration to an existing function that has unsigned code, that code keeps running until you deploy a new code package.

To associate a code signing configuration with a function (console)

1. Open the Functions page of the Lambda console.
2. Choose the function for which you want to enable code signing.
3. Open the **Configuration** tab.
4. Scroll down and choose **Code signing**.
5. Choose **Edit**.
6. In **Edit code signing**, choose a code signing configuration for this function.
7. Choose **Save**.

Configuring IAM policies for Lambda code signing configurations

To grant permission for a user to access Lambda code signing API operations, attach one or more policy statements to the user policy. For more information about user policies, see Identity-based IAM policies for Lambda.

The following example policy statement grants permission to create, update, and retrieve code signing configurations.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
          "lambda:CreateCodeSigningConfig",
          "lambda:UpdateCodeSigningConfig",
          "lambda:GetCodeSigningConfig"
        ],
      "Resource": "*"
    }
  ]
}
```

Administrators can use the `CodeSigningConfigArn` condition key to specify the code signing configurations that developers must use to create or update your functions.

The following example policy statement grants permission to create a function. The policy statement includes a `lambda:CodeSigningConfigArn` condition to specify the allowed code signing configuration. Lambda blocks `CreateFunction` API requests if the CodeSigningConfigArn parameter is missing or does not match the value in the condition.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
```

```
    {
      "Sid": "AllowReferencingCodeSigningConfig",
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "lambda:CodeSigningConfigArn": "arn:aws:lambda:us-
east-1:111122223333:code-signing-config:csc-0d4518bd353a0a7c6"
        }
      }
    }
  ]
}
```

Using tags on code signing configurations

You can tag code signing configurations to organize and manage your resources. Tags are free-form key-value pairs associated with your resources that are supported across AWS services. For more information about use cases for tags, see Common tagging strategies in the *Tagging AWS Resources and Tag Editor Guide*.

You can use the AWS Lambda API to view and update tags. You can also view and update tags while managing a specific code signing configuration in the Lambda console.

Sections

- Permissions required for working with tags
- Using tags with the Lambda console
- Using tags with the AWS CLI

Permissions required for working with tags

To allow an AWS Identity and Access Management (IAM) identity (user, group, or role) to read or set tags on a resource, grant it the corresponding permissions:

- **lambda:ListTags**–When a resource has tags, grant this permission to anyone who needs to call `ListTags` on it. For tagged functions, this permission is also necessary for `GetFunction`.

- **lambda:TagResource**–Grant this permission to anyone who needs to call `TagResource` or perform a tag on create.

Optionally, consider granting the **lambda:UntagResource** permission as well to allow `UntagResource` calls to the resource.

For more information, see Identity-based IAM policies for Lambda.

Using tags with the Lambda console

You can use the Lambda console to create code signing configurations that have tags, add tags to existing code signing configurations, and filter code signing configurations by tag.

To add a tag when you create a code signing configuration

1. Open Code signing configurations in the Lambda console.
2. From the header of the content pane, Choose **Create configuration**.
3. In the section at the top of the content pane, set up your code signing configuration. For more information about configuring code signing configurations, see the section called "Code signing".
4. In the **Tags** section, choose **Add new tag**.
5. In the **Key** field, enter your tag key. For information about tagging restrictions, see Tag naming limits and requirements in the *Tagging AWS Resources and Tag Editor Guide*.
6. Choose **Create configuration**.

To add a tag to an existing code signing configuration

1. Open Code signing configurations in the Lambda console.
2. Choose the name of your code signing configuration.
3. In the tabs below the **Detail** pane, choose **Tags**.
4. Choose **Manage tags**.
5. Choose **Add new tag**.
6. In the **Key** field, enter your tag key. For information about tagging restrictions, see Tag naming limits and requirements in the *Tagging AWS Resources and Tag Editor Guide*.
7. Choose **Save**.

To filter code signing configurations by tag

1. Open Code signing configurations in the Lambda console.
2. Choose the search box.
3. From the dropdown list, select your tag from below the **Tags** subheading.
4. Select **Use: "tag-name"** to see all code signing configurations tagged with this key, or choose an **Operator** to further filter by value.
5. Select your tag value to filter by a combination of tag key and value.

The search box also supports searching for tag keys. Enter the name of a key to find it in the list.

Using tags with the AWS CLI

You can add and remove tags on existing Lambda resources, including code signing configurations, with the Lambda API. You can also add tags when creating an code signing configuration, which allows you to keep a resource tagged through its entire lifecycle.

Updating tags with the Lambda tag APIs

You can add and remove tags for supported Lambda resources through the TagResource and UntagResource API operations.

You can call these operations using the AWS CLI. To add tags to an existing resource, use the `tag-resource` command. This example adds two tags, one with the key *`Department`* and one with the key *`CostCenter`*.

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

To remove tags, use the `untag-resource` command. This example removes the tag with the key *`Department`*.

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

Adding tags when creating a code signing configuration

To create a new Lambda code signing configuration with tags, use the CreateCodeSigningConfig API operation. Specify the `Tags` parameter. You can call this operation with the `create-code-signing-config` AWS CLI command and the `--tags` option. For more information about the CLI command, see create-code-signing-config in the *AWS CLI Command Reference*.

Before using the `Tags` parameter with `CreateCodeSigningConfig`, ensure that your role has permission to tag resources alongside the usual permissions needed for this operation. For more information about permissions for tagging, see the section called “Permissions required for working with tags”.

Viewing tags with the Lambda tag APIs

To view the tags that are applied to a specific Lambda resource, use the `ListTags` API operation. For more information, see ListTags.

You can call this operation with the `list-tags` AWS CLI command by providing an ARN (Amazon Resource Name).

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

Filtering resources by tag

You can use the AWS Resource Groups Tagging API GetResources API operation to filter your resources by tags. The `GetResources` operation receives up to 10 filters, with each filter containing a tag key and up to 10 tag values. You provide `GetResources` with a `ResourceType` to filter by specific resource types.

You can call this operation using the `get-resources` AWS CLI command. For examples of using `get-resources`, see get-resources in the *AWS CLI Command Reference*.

Monitoring, debugging, and troubleshooting Lambda functions

AWS Lambda integrates with other AWS services to help you monitor and troubleshoot your Lambda functions. Lambda automatically monitors Lambda functions on your behalf and reports metrics through Amazon CloudWatch. To help you monitor your code when it runs, Lambda automatically tracks the number of requests, the invocation duration per request, and the number of requests that result in an error.

You can use other AWS services to troubleshoot your Lambda functions. This section describes how to use these AWS services to monitor, trace, debug, and troubleshoot your Lambda functions and applications. For details about function logging and errors in each runtime, see individual runtime sections.

Sections

- Pricing
- Using CloudWatch metrics with Lambda
- Working with Lambda function logs
- Logging AWS Lambda API calls using AWS CloudTrail
- Visualize Lambda function invocations using AWS X-Ray
- Monitor function performance with Amazon CloudWatch Lambda Insights
- Monitoring Lambda applications
- Monitor application performance with Amazon CloudWatch Application Signals
- Remotely debug Lambda functions with Visual Studio Code

Pricing

CloudWatch has a perpetual free tier. Beyond the free tier threshold, CloudWatch charges for metrics, dashboards, alarms, logs, and insights. For more information, see Amazon CloudWatch pricing.

Using CloudWatch metrics with Lambda

When your AWS Lambda function finishes processing an event, Lambda automatically sends metrics about the invocation to Amazon CloudWatch. You don't need to grant any additional permissions to your execution role to receive function metrics, and there's no additional charge for these metrics.

There are many types of metrics associated with Lambda functions. These include invocation metrics, performance metrics, concurrency metrics, asynchronous invocation metrics, and event source mapping metrics. For more information, see the section called “Metric types”.

In the CloudWatch console, you can view these metrics and build graphs and dashboards with them. You can also set alarms to respond to changes in utilization, performance, or error rates. Lambda sends metric data to CloudWatch in 1-minute intervals. For more immediate insight into your Lambda function, you can create high-resolution custom metrics. Charges apply for custom metrics and CloudWatch alarms. For more information, see Amazon CloudWatch Pricing.

Viewing metrics for Lambda functions

Use the CloudWatch console to view metrics for your Lambda functions. In the console, you can filter and sort function metrics by function name, alias, version, or event source mapping UUID.

To view metrics on the CloudWatch console

1. Open the Metrics page (`AWS/Lambda` namespace) of the CloudWatch console.
2. On the **Browse** tab, under **Metrics**, choose any of the following dimensions:

   - **By Function Name** (`FunctionName`) – View aggregate metrics for all versions and aliases of a function.
   - **By Resource** (`Resource`) – View metrics for a version or alias of a function.
   - **By Executed Version** (`ExecutedVersion`) – View metrics for a combination of alias and version. Use the `ExecutedVersion` dimension to compare error rates for two versions of a function that are both targets of a weighted alias.
   - **By Event Source Mapping UUID** (`EventSourceMappingUUID`) – View metrics for an event source mapping.
   - **Across All Functions** (none) – View aggregate metrics for all functions in the current AWS Region.

3. Choose a metric. The metric should automatically appear in the visual graph, as well as under the **Graphed metrics** tab.

By default, graphs use the Sum statistic for all metrics. To choose a different statistic and customize the graph, use the options on the **Graphed metrics** tab.

> **Note**
>
> The timestamp on a metric reflects when the function was invoked. Depending on the duration of the invocation, this can be several minutes before the metric is emitted. For example, if your function has a 10-minute timeout, then look more than 10 minutes in the past for accurate metrics.

For more information about CloudWatch, see the Amazon CloudWatch User Guide.

Types of metrics for Lambda functions

This section describes the types of Lambda metrics available in the CloudWatch console.

Topics

- Invocation metrics
- Deployment metrics
- Performance metrics
- Concurrency metrics
- Asynchronous invocation metrics
- Event source mapping metrics

Invocation metrics

Invocation metrics are binary indicators of the outcome of a Lambda function invocation. View these metrics with the Sum statistic. For example, if the function returns an error, then Lambda sends the `Errors` metric with a value of 1. To get a count of the number of function errors that occurred each minute, view the Sum of the `Errors` metric with a period of 1 minute.

- `Invocations` – The number of times that your function code is invoked, including successful invocations and invocations that result in a function error. Invocations aren't recorded if

the invocation request is throttled or otherwise results in an invocation error. The value of `Invocations` equals the number of requests billed.

- `Errors` – The number of invocations that result in a function error. Function errors include exceptions that your code throws and exceptions that the Lambda runtime throws. The runtime returns errors for issues such as timeouts and configuration errors. To calculate the error rate, divide the value of `Errors` by the value of `Invocations`. Note that the timestamp on an error metric reflects when the function was invoked, not when the error occurred.
- `DeadLetterErrors` – For asynchronous invocation, the number of times that Lambda attempts to send an event to a dead-letter queue (DLQ) but fails. Dead-letter errors can occur due to incorrectly set resources or size limits.
- `DestinationDeliveryFailures` – For asynchronous invocation and supported event source mappings, the number of times that Lambda attempts to send an event to a destination but fails. For event source mappings, Lambda supports destinations for stream sources (DynamoDB and Kinesis). Delivery errors can occur due to permissions errors, incorrectly configured resources, or size limits. Errors can also occur if the destination you have configured is an unsupported type such as an Amazon SQS FIFO queue or an Amazon SNS FIFO topic.
- `Throttles` – The number of invocation requests that are throttled. When all function instances are processing requests and no concurrency is available to scale up, Lambda rejects additional requests with a `TooManyRequestsException` error. Throttled requests and other invocation errors don't count as either `Invocations` or `Errors`.

> **Note**
>
> With Lambda Managed Instances, Lambda provides granular throttle metrics that identify the specific constraint causing the throttle. When a throttle occurs on the execution environment, exactly one of the following sub-metrics is emitted with a value of 1, while the remaining three are emitted with a value of 0. The `Throttles` metric is always emitted alongside these sub-metrics.
>
> - `CPUThrottles` – Invocations throttled due to CPU exhaustion on the execution environment.
> - `MemoryThrottles` – Invocations throttled due to memory exhaustion on the execution environment.
> - `DiskThrottles` – Invocations throttled due to disk exhaustion on the execution environment.

    - `ConcurrencyThrottles` – Invocations throttled when the execution environment concurrency limit is reached.

- `OversizedRecordCount` – For Amazon DocumentDB event sources, the number of events your function receives from your change stream that are over 6 MB in size. Lambda drops the message and emits this metric.
- `ProvisionedConcurrencyInvocations` – The number of times that your function code is invoked using provisioned concurrency.
- `ProvisionedConcurrencySpilloverInvocations` – The number of times that your function code is invoked using standard concurrency when all provisioned concurrency is in use.
- `RecursiveInvocationsDropped` – The number of times that Lambda has stopped invocation of your function because it has detected that your function is part of an infinite recursive loop. Recursive loop detection monitors how many times a function is invoked as part of a chain of requests by tracking metadata added by supported AWS SDKs. By default, if your function is invoked as part of a chain of requests approximately 16 times, Lambda drops the next invocation. If you disable recursive loop detection, this metric is not emitted. For more information about this feature, see Use Lambda recursive loop detection to prevent infinite loops.

Deployment metrics

Deployment metrics provide information about Lambda function deployment events and related validation processes.

- `SignatureValidationErrors` – The number of times a code package deployment has occurred with signature validation failures when the code signing configuration policy is set to `Warn`. This metric is emitted when the expiry, mismatch, or revocation checks fail but the deployment is still allowed due to the `Warn` policy setting. For more information about code signing, see Using code signing to verify code integrity with Lambda.

Performance metrics

Performance metrics provide performance details about a single function invocation. For example, the `Duration` metric indicates the amount of time in milliseconds that your function spends processing an event. To get a sense of how fast your function processes events, view these metrics with the `Average` or `Max` statistic.

- `Duration` – The amount of time that your function code spends processing an event. The billed duration for an invocation is the value of `Duration` rounded up to the nearest millisecond. `Duration` does not include cold start time.
- `PostRuntimeExtensionsDuration` – The cumulative amount of time that the runtime spends running code for extensions after the function code has completed.
- `IteratorAge` – For DynamoDB, Kinesis, and Amazon DocumentDB event sources, the age of the last record in the event in milliseconds. This metric measures the time between when a stream receives the record and when the event source mapping sends the event to the function.
- `OffsetLag` – For self-managed Apache Kafka and Amazon Managed Streaming for Apache Kafka (Amazon MSK) event sources, the difference in offset between the last record written to a topic and the last record that your function's consumer group processed. Though a Kafka topic can have multiple partitions, this metric measures the offset lag at the topic level.

`Duration` also supports percentile (p) statistics. Use percentiles to exclude outlier values that skew `Average` and `Maximum` statistics. For example, the p95 statistic shows the maximum duration of 95 percent of invocations, excluding the slowest 5 percent. For more information, see Percentiles in the *Amazon CloudWatch User Guide*.

Concurrency metrics

Lambda reports concurrency metrics as an aggregate count of the number of instances processing events across a function, version, alias, or AWS Region. To see how close you are to hitting concurrency limits, view these metrics with the `Max` statistic.

- `ConcurrentExecutions` – The number of function instances that are processing events. If this number reaches your concurrent executions quota for the Region, or the reserved concurrency limit on the function, then Lambda throttles additional invocation requests.
- `ProvisionedConcurrentExecutions` – The number of function instances that are processing events using provisioned concurrency. For each invocation of an alias or version with provisioned concurrency, Lambda emits the current count. If your function is inactive or not receiving requests, Lambda doesn't emit this metric.
- `ProvisionedConcurrencyUtilization` – For a version or alias, the value of `ProvisionedConcurrentExecutions` divided by the total amount of provisioned concurrency configured. For example, if you configure a provisioned concurrency of 10 for your function, and your `ProvisionedConcurrentExecutions` is 7, then your `ProvisionedConcurrencyUtilization` is 0.7.

If your function is inactive or not receiving requests, Lambda doesn't emit this metric because it is based on `ProvisionedConcurrentExecutions`. Keep this in mind if you use `ProvisionedConcurrencyUtilization` as the basis for CloudWatch alarms.

- `UnreservedConcurrentExecutions` – For a Region, the number of events that functions without reserved concurrency are processing.
- `ClaimedAccountConcurrency` – For a Region, the amount of concurrency that is unavailable for on-demand invocations. `ClaimedAccountConcurrency` is equal to `UnreservedConcurrentExecutions` plus the amount of allocated concurrency (i.e. the total reserved concurrency plus total provisioned concurrency). For more information, see Working with the `ClaimedAccountConcurrency` metric.

Asynchronous invocation metrics

Asynchronous invocation metrics provide details about asynchronous invocations from event sources and direct invocations. You can set thresholds and alarms to notify you of certain changes. For example, when there's an undesired increase in the number of events queued for processing (`AsyncEventsReceived`). Or, when an event has been waiting a long time to be processed (`AsyncEventAge`).

- `AsyncEventsReceived` – The number of events that Lambda successfully queues for processing. This metric provides insight into the number of events that a Lambda function receives. Monitor this metric and set alarms for thresholds to check for issues. For example, to detect an undesirable number of events sent to Lambda, and to quickly diagnose issues resulting from incorrect trigger or function configurations. Mismatches between `AsyncEventsReceived` and `Invocations` can indicate a disparity in processing, events being dropped, or a potential queue backlog.
- `AsyncEventAge` – The time between when Lambda successfully queues the event and when the function is invoked. The value of this metric increases when events are being retried due to invocation failures or throttling. Monitor this metric and set alarms for thresholds on different statistics for when a queue buildup occurs. To troubleshoot an increase in this metric, look at the `Errors` metric to identify function errors and the `Throttles` metric to identify concurrency issues.
- `AsyncEventsDropped` – The number of events that are dropped without successfully executing the function. If you configure a dead-letter queue (DLQ) or `OnFailure` destination, then events are sent there before they're dropped. Events are dropped for various reasons. For example,

events can exceed the maximum event age or exhaust the maximum retry attempts, or reserved concurrency might be set to 0. To troubleshoot why events are dropped, look at the `Errors` metric to identify function errors and the `Throttles` metric to identify concurrency issues.

Event source mapping metrics

Event source mapping metrics provide insights into the processing behavior of your event source mapping.

Currently, event source mapping metrics are available for Amazon SQS, Kinesis, DynamoDB, Amazon MSK and self-managed Apache Kafka event sources.

For event source mapping with metrics config, you can also check all the ESM related metrics in the **Monitor** tab from the page Console **Lambda** > **Additional resources** > **event source mappings** now.

To enable metrics or an event source mapping (console)

1. Open the Functions page of the Lambda console.
2. Choose the function you want to enable metrics for.
3. Choose **Configuration**, then choose **Triggers**.
4. Choose the event source mapping that you want to enable metrics for, then choose **Edit**.
5. Under **Event source mapping configuration**, choose **Enable metrics** or select from the **Metrics** dropdown list.
6. Choose **Save**.

Alternatively, you can enable metrics for your event source mapping programmatically using the EventSourceMappingMetricsConfig object in your EventSourceMappingConfiguration. For example, the following UpdateEventSourceMapping CLI command enables metrics for an event source mapping:

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --metrics-config Metrics=EventCount
```

There are 3 metric goups: `EventCount`, `ErrorCount` and `KafkaMetrics`, and each group has multi metrics. Not every metric is available for each event source. The following table summarizes the supported metrics for each type of event source.

You must opt-in the metric group to receive metrics related metrics. for example set EventCount in metrics config to have: (`PolledEventCount`, `FilteredOutEventCount`, `InvokedEventCount`, `FailedInvokeEventCount`, `DroppedEventCount`, `OnFailureDestinationDeliveredEventCount`, and `DeletedEventCount`).

Event source mapping metric	Metric group	Amazon SQS	Kinesis and DynamoDB streams	Amazon MSK and self-managed Apache Kafka
`PolledEventCount`	`EventCount`	Yes	Yes	Yes
`FilteredOutEventCount`	`EventCount`	Yes	Yes	Yes
`InvokedEventCount`	`EventCount`	Yes	Yes	Yes
`FailedInvokeEventCount`	`EventCount`	Yes	Yes	Yes
`DroppedEventCount`	`EventCount`	No	Yes	Yes
`OnFailureDestinationDeliveredEventCount`	`EventCount`	No	Yes	Yes

Event source mapping metric	Metric group	Amazon SQS	Kinesis and DynamoDB streams	Amazon MSK and self-managed Apache Kafka
DeletedEventCount	EventCount	Yes	No	No
CommittedEventCount	EventCount	No	No	Yes
PollingErrorCount	ErrorCount	No	No	Yes
InvokeErrorCount	ErrorCount	No	No	Yes
OnFailureDestinationDeliveryErrorCount	ErrorCount	No	No	Yes
SchemaRegistryErrorCount	ErrorCount	No	No	Yes
CommitErrorCount	ErrorCount	No	No	Yes
MaxOffsetLag	KafkaMetrics	No	No	Yes
SumOffsetLag	KafkaMetrics	No	No	Yes

In addition, if your event source mapping is in provisioned mode, Lambda provides the following metric:

- `ProvisionedPollers` – For event source mappings in provisioned mode, the number of event pollers that are actively running. View this metric using the `MAX` math.
- (Amazon MSK and self-managed Apache Kafka event sources only) `EventPollerUnit` – For event source mappings in provisioned mode, the number of event poller units that are actively running. View this metric using the `SUM` math.
- (Amazon MSK and self-managed Apache Kafka event sources) `EventPollerThroughputInBytes` – For event source mappings in provisioned mode, the total record size of event pollers polled from the event source. It can tell you the current polling throughput. View this metric using the `SUM` math.

Here is more detail about each of the metric:

- `PolledEventCount` – The number of events that Lambda reads successfully from the event source. If Lambda polls for events but receives an empty poll (no new records), Lambda emits a 0 value for this metric. Use this metric to detect whether your event source mapping is correctly polling for new events.
- `FilteredOutEventCount` – For event source mapping with a filter criteria, the number of events filtered out by that filter criteria. Use this metric to detect whether your event source mapping is properly filtering out events. For events that match the filter criteria, Lambda emits a 0 metric.
- `InvokedEventCount` – The number of events that invoked your Lambda function. Use this metric to verify that events are properly invoking your function. If an event results in a function error or throttling, `InvokedEventCount` may count multiple times for the same polled event due to automatic retries.

  > **Warning**
  >
  > Lambda event source mappings process each event at least once, and duplicate processing of records can occur. Because of this, events may be counted multiple times in metrics that involve event counts.

- `FailedInvokeEventCount` – The number of events that Lambda tried to invoke your function with, but failed. Invocations can fail due to reasons such as network configuration issues, incorrect permissions, or a deleted Lambda function, version, or alias. If your event source mapping has partial batch responses enabled, `FailedInvokeEventCount` includes any event with a non-empty `BatchItemFailures` in the response.

> **Note**
>
> The timestamp for the `FailedInvokeEventCount` metric represents the end of the function invocation. This behavior differs from other Lambda invocation error metrics, which are timestamped at the start of the function invocation.

- `DroppedEventCount` – The number of events that Lambda dropped due to expiry or retry exhaustion. Specifically, this is the number of records that exceed your configured values for `MaximumRecordAgeInSeconds` or `MaximumRetryAttempts`. Importantly, this doesn't include the number of records that expire due to exceeding your event source's retention settings. Dropped events also excludes events that you send to an on-failure destination. Use this metric to detect an increasing backlog of events.
- `OnFailureDestinationDeliveredEventCount` – For event source mappings with an on-failure destination configured, the number of events sent to that destination. Use this metric to monitor for function errors related to invocations from this event source. If delivery to the destination fails, Lambda handles metrics as follows:
  - Lambda doesn't emit the `OnFailureDestinationDeliveredEventCount` metric.
  - For the `DestinationDeliveryFailures` metric, Lambda emits a 1.
  - For the `DroppedEventCount` metric, Lambda emits a number equal to the number of events that failed delivery.
- `DeletedEventCount` – The number of events that Lambda successfully deletes after processing. If Lambda tries to delete an event but fails, Lambda emits a 0 metric. Use this metric to ensure that successfully processed events are deleted from your event source.
- `CommittedEventCount` – The number of events that Lambda successfully committed after processing. It's a sum of the deltas of last and current committted offset from each partition in the Kafka event source mapping.
- `PollingErrorCount` – The number of errors that Lambda failed to poll requests from event source. Lambda only emits this metric data when error happened.
- `InvokeErrorCount` – The number of errors that Lambda failed to invoke your function. Notice the invocation is records in batch. The number is on batch level, not on record count level. Lambda only emits this metric data when error happened.
- `SchemaRegistryErrorCount` – The number of errors that Lambda failed to fetch the schema or deserialize with the scheme. Lambda only emits this metric data when error happened.

- `CommitErrorCount` – The number of errors that Lambda failed to commit to Kafka cluster. Lambda only emits this metric data when error happened.
- `MaxOffsetLag` – The max of offset lags (difference between latest and committed offsets) accross all partitions in the event source mapping.
- `SumOffsetLag` – The sum of the offset lags across all partitions in the event source mapping.

If your event source mapping is disabled, you won't receive event source mapping metrics. You may also see missing metrics if CloudWatch or Lambda is experiencing degraded availability.

Working with Lambda function logs

To help you troubleshoot failures, AWS Lambda automatically monitors Lambda functions on your behalf. You can view logs for Lambda functions using the Lambda console, the CloudWatch console, the AWS Command Line Interface (AWS CLI), the CloudWatch API. You can also configure Lambda to send logs to Amazon S3 and Firehose.

As long as your function's execution role has the necessary permissions, Lambda captures logs for all requests handled by your function and sends them to Amazon CloudWatch Logs, which is the default destination. You can also use the Lambda console to configure Amazon S3 or Firehose as logging destinations.

- **CloudWatch Logs** is the default logging destination for Lambda functions. CloudWatch Logs provides real-time log viewing and analysis capabilities, with support for creating metrics and alarms based on your log data.
- **Amazon S3** is economical for long-term storage, and services like Athena can be used to analyze logs. Latency is typically higher.
- **Firehose** offers managed streaming of logs to various destinations. If you need to send logs to other AWS services (for example, OpenSearch Service or Redshift Data API) or third-party platforms (like Datadog, New Relic, or Splunk), Firehose simplifies that process by providing pre-built integrations. You can also stream to custom HTTP endpoints without setting up additional infrastructure.

Choosing a service destination to send logs to

Consider the following key factors when choosing a service a destination for function logs:

- **Cost management varies by service.** Amazon S3 typically provides the most economical option for long-term storage, while CloudWatch Logs allows you to view logs, process logs, and set up alerts in real time. Firehose costs include both the streaming service and cost associated with what you configure it to stream to.
- **Analysis capabilities differ across services.** CloudWatch Logs excels at real-time monitoring and integrates natively with other CloudWatch features, such as Logs Insights and Live Tail. Amazon S3 works well with analysis tools like Athena and can integrate with various services, though it may require additional setup. Firehose simplifies direct streaming to specific AWS services (like OpenSearch Service and Redshift Data API) and supported third-party platforms (such as

Datadog and Splunk) by providing pre-built integrations, potentially reducing configuration work.

- **Setup and ease of use vary by service.** CloudWatch Logs is the default log destination - it works immediately with no additional configuration and provides straightforward log viewing and analysis through the CloudWatch console. If you need logs sent to Amazon S3, you'll need to do some initial setup in the Lambda console and configure bucket permissions. If you need logs sent directly to services like OpenSearch Service or third-party analytics platforms, Firehose can simplify that process.

Configuring log destinations

AWS Lambda supports multiple destinations for your function logs. This guide explains the available logging destinations and helps you choose the right option for your needs. Regardless of your chosen destination, Lambda provides options to control log format, filtering, and delivery.

Lambda supports both JSON and plain text formats for your function's logs. JSON structured logs provide enhanced searchability and enable automated analysis, while plain text logs offer simplicity and potentially reduced storage costs. You can control which logs Lambda sends to your chosen destination by configuring log levels for both system and application logs. Filtering helps you manage storage costs and makes it easier to find relevant log entries during debugging.

For detailed setup instructions for each destination, refer to the following sections:

- Sending Lambda function logs to CloudWatch Logs
- Sending Lambda function logs to Firehose
- Sending Lambda function logs to Amazon S3

Configuring advanced logging controls for Lambda functions

To give you more control over how your function logs are captured, processed, and consumed, Lambda offers the following logging configuration options:

- **Log format** - select between plain text and structured JSON format for your function’s logs.
- **Log level** - for JSON structured logs, choose the detail level of the logs Lambda sends to CloudWatch, such as `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`, and `TRACE`.
- **Log group** - choose the CloudWatch log group your function sends logs to.

To learn more about configuring advanced logging controls, refer to the following sections:

- Configuring JSON and plain text log formats
- Log-level filtering
- Configuring CloudWatch log groups

Configuring JSON and plain text log formats

Capturing your log outputs as JSON key value pairs makes it easier to search and filter when debugging your functions. With JSON formatted logs, you can also add tags and contextual information to your logs. This can help you to perform automated analysis of large volumes of log data. Unless your development workflow relies on existing tooling that consumes Lambda logs in plain text, we recommend that you select JSON for your log format.

> **Lambda Managed Instances**
>
> Lambda Managed Instances only support JSON log format. When you create a Managed Instances function, Lambda automatically configures the log format to JSON and you cannot change it to plain text. For more information about Managed Instances, see *Lambda Managed Instances*.

For all Lambda managed runtimes, you can choose whether your function's system logs are sent to CloudWatch Logs in unstructured plain text or JSON format. System logs are the logs that Lambda generates and are sometimes known as platform event logs.

For supported runtimes, when you use one of the supported built-in logging methods, Lambda can also output your function's application logs (the logs your function code generates) in structured JSON format. When you configure your function's log format for these runtimes, the configuration you choose applies to both system and application logs.

For supported runtimes, if your function uses a supported logging library or method, you don't need to make any changes to your existing code for Lambda to capture logs in structured JSON.

> **Note**
>
> Using JSON log formatting adds additional metadata and encodes log messages as JSON objects containing a series of key value pairs. Because of this, the size of your function's log messages can increase.

Supported runtimes and logging methods

Lambda currently supports the option to output JSON structured application logs for the following runtimes.

Language	Supported versions
Java	All Java runtimes except Java 8 on Amazon Linux 1
.NET	.NET 8 and later
Node.js	Node.js 16 and later
Python	Python 3.8 and later
Rust	n/a

For Lambda to send your function's application logs to CloudWatch in structured JSON format, your function must use the following built-in logging tools to output logs:

- **Java**: The `LambdaLogger` logger or Log4j2. For more information, see the section called "Logging".
- **.NET**: The `ILambdaLogger` instance on the context object. For more information, see Log and monitor C# Lambda functions.
- **Node.js** - The console methods `console.trace`, `console.debug`, `console.log`, `console.info`, `console.error`, and `console.warn`. For more information, see the section called "Logging".
- **Python**: The standard Python `logging` library. For more information, see the section called "Logging".

- **Rust**: The `tracing` crate. For more information, see the section called “Logging”.

For other managed Lambda runtimes, Lambda currently only natively supports capturing system logs in structured JSON format. However, you can still capture application logs in structured JSON format in any runtime by using logging tools such as Powertools for AWS Lambda that output JSON formatted log outputs.

Default log formats

Currently, the default log format for all Lambda runtimes is plain text. For Lambda Managed Instances, the log format is always JSON and cannot be changed.

If you’re already using logging libraries like Powertools for AWS Lambda to generate your function logs in JSON structured format, you don’t need to change your code if you select JSON log formatting. Lambda doesn’t double-encode any logs that are already JSON encoded, so your function’s application logs will continue to be captured as before.

JSON format for system logs

When you configure your function's log format as JSON, each system log item (platform event) is captured as a JSON object that contains key value pairs with the following keys:

- `"time"` - the time the log message was generated
- `"type"` - the type of event being logged
- `"record"` - the contents of the log output

The format of the `"record"` value varies according to the type of event being logged. For more information see the section called “Telemetry API `Event` object types”. For more information about the log levels assigned to system log events, see the section called “System log level event mapping”.

For comparison, the following two examples show the same log output in both plain text and structured JSON formats. Note that in most cases, system log events contain more information when output in JSON format than when output in plain text.

Example plain text:

```
2024-03-13 18:56:24.046000 fbe8c1   INIT_START  Runtime Version:
 python:3.12.v18  Runtime Version ARN: arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0
```

Example structured JSON:

```
{
  "time": "2024-03-13T18:56:24.046Z",
  "type": "platform.initStart",
  "record": {
    "initializationType": "on-demand",
    "phase": "init",
    "runtimeVersion": "python:3.12.v18",
    "runtimeVersionArn": "arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0"
  }
}
```

> **Note**
>
> The the section called “Telemetry API” always emits platform events such as START and REPORT in JSON format. Configuring the format of the system logs Lambda sends to CloudWatch doesn’t affect Lambda Telemetry API behavior.

JSON format for application logs

When you configure your function's log format as JSON, application log outputs written using supported logging libraries and methods are captured as a JSON object that contains key value pairs with the following keys.

- `"timestamp"` - the time the log message was generated
- `"level"` - the log level assigned to the message
- `"message"` - the contents of the log message
- `"requestId"` (Python, .NET, and Node.js) or `"AWSrequestId"` (Java) - the unique request ID for the function invocation

Depending on the runtime and logging method that your function uses, this JSON object may also contain additional key pairs. For example, in Node.js, if your function uses `console` methods to log error objects using multiple arguments, The JSON object will contain extra key value pairs with the keys `errorMessage`, `errorType`, and `stackTrace`. To learn more about JSON formatted logs in different Lambda runtimes, see the section called “Logging”, the section called “Logging”, and the section called “Logging”.

> **Note**
>
> The key Lambda uses for the timestamp value is different for system logs and application logs. For system logs, Lambda uses the key `"time"` to maintain consistency with Telemetry API. For application logs, Lambda follows the conventions of the supported runtimes and uses `"timestamp"`.

For comparison, the following two examples show the same log output in both plain text and structured JSON formats.

Example plain text:

```
2024-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example structured JSON:

```
{
    "timestamp":"2024-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

Setting your function's log format

To configure the log format for your function, you can use the Lambda console or the AWS Command Line Interface (AWS CLI). You can also configure a function’s log format using the CreateFunction and UpdateFunctionConfiguration Lambda API commands, the AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, and the CloudFormation AWS::Lambda::Function resource.

Changing your function's log format doesn't affect existing logs stored in CloudWatch Logs. Only new logs will use the updated format.

If you change your function's log format to JSON and do not set log level, then Lambda automatically sets your function's application log level and system log level to INFO. This means that Lambda sends only log outputs of level INFO and lower to CloudWatch Logs. To learn more about application and system log-level filtering see the section called "Log-level filtering"

> **Note**
>
> For Python runtimes, when your function's log format is set to plain text, the default log-level setting is WARN. This means that Lambda only sends log outputs of level WARN and lower to CloudWatch Logs. Changing your function's log format to JSON changes this default behavior. To learn more about logging in Python, see the section called "Logging".

For Node.js functions that emit embedded metric format (EMF) logs, changing your function's log format to JSON could result in CloudWatch being unable to recognize your metrics.

> **Important**
>
> If your function uses Powertools for AWS Lambda (TypeScript) or the open-sourced EMF client libraries to emit EMF logs, update your Powertools and EMF libraries to the latest versions to ensure that CloudWatch can continue to parse your logs correctly. If you switch to the JSON log format, we also recommend that you carry out testing to ensure compatibility with your function's embedded metrics. For further advice about node.js functions that emit EMF logs, see the section called "Using embedded metric format (EMF) client libraries with structured JSON logs".

To configure a function's log format (console)

1. Open the Functions page of the Lambda console.
2. Choose a function
3. On the function configuration page, choose **Monitoring and operations tools**.
4. In the **Logging configuration** pane, choose **Edit**.
5. Under **Log content**, for **Log format** select either **Text** or **JSON**.

6. Choose **Save**.

To change the log format of an existing function (AWS CLI)

- To change the log format of an existing function, use the update-function-configuration command. Set the `LogFormat` option in `LoggingConfig` to either `JSON` or `Text`.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON
```

To set log format when you create a function (AWS CLI)

- To configure log format when you create a new function, use the `--logging-config` option in the create-function command. Set `LogFormat` to either `JSON` or `Text`. The following example command creates a Node.js function that outputs logs in structured JSON.

  If you don't specify a log format when you create a function, Lambda will use the default log format for the runtime version you select. For information about default logging formats, see the section called "Default log formats".

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON
```

Log-level filtering

Lambda can filter your function's logs so that only logs of a certain detail level or lower are sent to CloudWatch Logs. You can configure log-level filtering separately for your function's system logs (the logs that Lambda generates) and application logs (the logs that your function code generates).

For the section called "Supported runtimes and logging methods", you don't need to make any changes to your function code for Lambda to filter your function's application logs.

For all other runtimes and logging methods , your function code must output log events to `stdout` or `stderr` as JSON formatted objects that contain a key value pair with the key `"level"`. For example, Lambda interprets the following output to `stdout` as a DEBUG level log.

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2024-11-02T16:51:31.587199Z"}')
```

If the `"level"` value field is invalid or missing, Lambda will assign the log output the level INFO. For Lambda to use the timestamp field, you must specify the time in valid RFC 3339 timestamp format. If you don't supply a valid timestamp, Lambda will assign the log the level INFO and add a timestamp for you.

When naming the timestamp key, follow the conventions of the runtime you are using. Lambda supports most common naming conventions used by the managed runtimes.

> **Note**
>
> To use log-level filtering, your function must be configured to use the JSON log format. The default log format for all Lambda managed runtimes is currently plain text. To learn how to configure your function's log format to JSON, see the section called “Setting your function's log format”.

For application logs (the logs generated by your function code), you can choose between the following log levels.

Log level	Standard usage
TRACE (most detail)	The most fine-grained information used to trace the path of your code's execution
DEBUG	Detailed information for system debugging
INFO	Messages that record the normal operation of your function
WARN	Messages about potential errors that may lead to unexpected behavior if unaddressed

Log level	Standard usage
ERROR	Messages about problems that prevent the code from performing as expected
FATAL (least detail)	Messages about serious errors that cause the application to stop functioning

When you select a log level, Lambda sends logs at that level and lower to CloudWatch Logs. For example, if you set a function's application log level to WARN, Lambda doesn't send log outputs at the INFO and DEBUG levels. The default application log level for log filtering is INFO.

When Lambda filters your function's application logs, log messages with no level will be assigned the log level INFO.

For system logs (the logs generated by the Lambda service), you can choose between the following log levels.

Log level	Usage
DEBUG (most detail)	Detailed information for system debugging
INFO	Messages that record the normal operation of your function
WARN (least detail)	Messages about potential errors that may lead to unexpected behavior if unaddressed

When you select a log level, Lambda sends logs at that level and lower. For example, if you set a function's system log level to INFO, Lambda doesn't send log outputs at the DEBUG level.

By default, Lambda sets the system log level to INFO. With this setting, Lambda automatically sends `"start"` and `"report"` log messages to CloudWatch. To receive more or less detailed system logs, change the log level to DEBUG or WARN. To see a list of the log levels that Lambda maps different system log events to, see the section called "System log level event mapping".

Configuring log-level filtering

To configure application and system log-level filtering for your function, you can use the Lambda console or the AWS Command Line Interface (AWS CLI). You can also configure a function's log level using the CreateFunction and UpdateFunctionConfiguration Lambda API commands, the AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource, and the CloudFormation AWS::Lambda::Function resource.

Note that if you set your function's log level in your code, this setting takes precedence over any other log level settings you configure. For example, if you use the Python `logging setLevel()` method to set your function's logging level to INFO, this setting takes precedence over a setting of WARN that you configure using the Lambda console.

To configure an existing function's application or system log level (console)

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. On the function configuration page, choose **Monitoring and operations tools**.
4. In the **Logging configuration** pane, choose **Edit**.
5. Under **Log content**, for **Log format** ensure **JSON** is selected.
6. Using the radio buttons, select your desired **Application log level** and **System log level** for your function.
7. Choose **Save**.

To configure an existing function's application or system log level (AWS CLI)

- To change the application or system log level of an existing function, use the update-function-configuration command. Use `--logging-config` to set `SystemLogLevel` to one of `DEBUG`, `INFO`, or `WARN`. Set `ApplicationLogLevel` to one of `DEBUG`, `INFO`, `WARN`, `ERROR`, or `FATAL`.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

To configure log-level filtering when you create a function

- To configure log-level filtering when you create a new function, use `--logging-config` to set the `SystemLogLevel` and `ApplicationLogLevel` keys in the create-function command. Set `SystemLogLevel` to one of `DEBUG`, `INFO`, or `WARN`. Set `ApplicationLogLevel` to one of `DEBUG`, `INFO`, `WARN`, `ERROR`, or `FATAL`.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

System log level event mapping

For system level log events generated by Lambda, the following table defines the log level assigned to each event. To learn more about the events listed in the table, see the section called "Event schema reference"

Event name	Condition	Assigned log level
initStart	runtimeVersion is set	INFO
initStart	runtimeVersion is not set	DEBUG
initRuntimeDone	status=success	DEBUG
initRuntimeDone	status!=success	WARN
initReport	initializationType!=on-demand	INFO
initReport	initializationType=on-demand	DEBUG
initReport	status!=success	WARN
restoreStart	runtimeVersion is set	INFO

Event name	Condition	Assigned log level
restoreStart	runtimeVersion is not set	DEBUG
restoreRuntimeDone	status=success	DEBUG
restoreRuntimeDone	status!=success	WARN
restoreReport	status=success	INFO
restoreReport	status!=success	WARN
start	-	INFO
runtimeDone	status=success	DEBUG
runtimeDone	status!=success	WARN
report	status=success	INFO
report	status!=success	WARN
extension	state=success	INFO
extension	state!=success	WARN
logSubscription	-	INFO
telemetrySubscription	-	INFO
logsDropped	-	WARN

Note

The the section called “Telemetry API” always emits the complete set of platform events. Configuring the level of the system logs Lambda sends to CloudWatch doesn’t affect Lambda Telemetry API behavior.

Application log-level filtering with custom runtimes

When you configure application log-level filtering for your function, behind the scenes Lambda sets the application log level in the runtime using the `AWS_LAMBDA_LOG_LEVEL` environment variable. Lambda also sets your function's log format using the `AWS_LAMBDA_LOG_FORMAT` environment variable. You can use these variables to integrate Lambda advanced logging controls into a custom runtime.

For the ability to configure logging settings for a function using a custom runtime with the Lambda console, AWS CLI, and Lambda APIs, configure your custom runtime to check the value of these environment variables. You can then configure your runtime's loggers in accordance with the log format and log levels you select.

Sending Lambda function logs to CloudWatch Logs

By default, Lambda automatically captures logs for all function invocations and sends them to CloudWatch Logs, provided your function's execution role has the necessary permissions. These logs are, by default, stored in a log group named /aws/lambda/*<function-name>*. To enhance debugging, you can insert custom logging statements into your code, which Lambda will seamlessly integrate with CloudWatch Logs. If needed, you can configure your function to send logs to a different group using the Lambda console, AWS CLI, or Lambda API. See the section called "Configure CloudWatch function logs" to learn more.

You can view logs for Lambda functions using the Lambda console, the CloudWatch console, the AWS Command Line Interface (AWS CLI), or the CloudWatch API. For more information, see to Viewing CloudWatch logs for Lambda functions.

> **Note**
>
> It may take 5 to 10 minutes for logs to show up after a function invocation.

Required IAM permissions

Your execution role needs the following permissions to upload logs to CloudWatch Logs:

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

To learn more, see Using identity-based policies (IAM policies) for CloudWatch Logs in the *Amazon CloudWatch User Guide*.

You can add these CloudWatch Logs permissions using the `AWSLambdaBasicExecutionRole` AWS managed policy provided by Lambda. To add this policy to your role, run the following command:

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

For more information, see the section called "AWS managed policies".

Pricing

There is no additional charge for using Lambda logs; however, standard CloudWatch Logs charges apply. For more information, see CloudWatch pricing.

Configuring CloudWatch log groups

By default, CloudWatch automatically creates a log group named `/aws/lambda/<function name>` for your function when it's first invoked. To configure your function to send logs to an existing log group, or to create a new log group for your function, you can use the Lambda console or the AWS CLI. You can also configure custom log groups using the CreateFunction and UpdateFunctionConfiguration Lambda API commands and the AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function resource.

You can configure multiple Lambda functions to send logs to the same CloudWatch log group. For example, you could use a single log group to store logs for all of the Lambda functions that make up a particular application. When you use a custom log group for a Lambda function, the log streams Lambda creates include the function name and function version. This ensures that the mapping between log messages and functions is preserved, even if you use the same log group for multiple functions.

The log stream naming format for custom log groups follows this convention:

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

Note that when configuring a custom log group, the name you select for your log group must follow the CloudWatch Logs naming rules. Additionally, custom log group names mustn't begin

with the string `aws/`. If you create a custom log group beginning with `aws/`, Lambda won't be able to create the log group. As a result of this, your function's logs won't be sent to CloudWatch.

To change a function's log group (console)

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. On the function configuration page, choose **Monitoring and operations tools**.
4. In the **Logging configuration** pane, choose **Edit**.
5. In the **Logging group** pane, for **CloudWatch log group**, choose **Custom**.
6. Under **Custom log group**, enter the name of the CloudWatch log group you want your function to send logs to. If you enter the name of an existing log group, then your function will use that group. If no log group exists with the name that you enter, then Lambda will create a new log group for your function with that name.

To change a function's log group (AWS CLI)

- To change the log group of an existing function, use the update-function-configuration command.

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogGroup=myLogGroup
```

To specify a custom log group when you create a function (AWS CLI)

- To specify a custom log group when you create a new Lambda function using the AWS CLI, use the `--logging-config` option. The following example command creates a Node.js Lambda function that sends logs to a log group named `myLogGroup`.

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogGroup=myLogGroup
```

Execution role permissions

For your function to send logs to CloudWatch Logs, it must have the logs:PutLogEvents permission. When you configure your function's log group using the Lambda console, Lambda will add this permission to the role under the following conditions:

- The service destination is set to CloudWatch Logs
- Your function's execution role doesn't have permissions to upload logs to CloudWatch Logs (the default destination)

> **Note**
>
> Lambda does not add any Put permission for Amazon S3 or Firehose log destinations.

When Lambda adds this permission, it gives the function permission to send logs to any CloudWatch Logs log group.

To prevent Lambda from automatically updating the function's execution role and edit it manually instead, expand **Permissions** and uncheck **Add required permissions**.

When you configure your function's log group using the AWS CLI, Lambda won't automatically add the `logs:PutLogEvents` permission. Add the permission to your function's execution role if it doesn't already have it. This permission is included in the AWSLambdaBasicExecutionRole managed policy.

CloudWatch logging for Lambda Managed Instances

When using Lambda Managed Instances, there are additional considerations for sending logs to CloudWatch Logs:

VPC networking requirements

Lambda Managed Instances run on customer-owned EC2 instances within your VPC. To send logs to CloudWatch Logs and traces to X-Ray, you must ensure that these AWS APIs are routable from your VPC. You have several options:

- **AWS PrivateLink (recommended)**: Use AWS PrivateLink to create VPC endpoints for CloudWatch Logs and X-Ray services. This allows your instances to access these services privately without

requiring an internet gateway or NAT gateway. For more information, see Using CloudWatch Logs with interface VPC endpoints.

- **NAT Gateway**: Configure a NAT gateway to allow outbound internet access from your private subnets.
- **Internet Gateway**: For public subnets, ensure your VPC has an internet gateway configured.

If CloudWatch Logs or X-Ray APIs are not routable from your VPC, your function logs and traces will not be delivered.

Concurrent invocations and log attribution

Lambda Managed Instances execution environments can process multiple invocations concurrently. When multiple invocations run simultaneously, their log entries are interleaved in the same log stream. To effectively filter and analyze logs from concurrent invocations, you should ensure each log entry includes the AWS request ID.

We recommend one of the following approaches:

- **Use default Lambda runtime loggers (recommended)**: The default logging libraries provided by Lambda managed runtimes automatically include the request ID in each log entry.
- **Implement structured JSON logging**: If you're building a custom runtime or need custom logging, implement JSON-formatted logs that include the request ID in each entry. Lambda Managed Instances only support the JSON log format. Include the `requestId` field in your JSON logs to enable filtering by invocation:

```
{
  "timestamp": "2025-01-15T10:30:00.000Z",
  "level": "INFO",
  "requestId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "message": "Processing request"
}
```

With request ID attribution, you can filter CloudWatch Logs log entries for a specific invocation using CloudWatch Logs Insights queries. For example:

```
fields @timestamp, @message
| filter requestId = "a1b2c3d4-e5f6-7890-abcd-ef1234567890"
| sort @timestamp asc
```

For more information about Lambda Managed Instances logging requirements, see Understanding the Lambda Managed Instances execution environment.

Viewing CloudWatch logs for Lambda functions

You can view Amazon CloudWatch logs for your Lambda function using the Lambda console, the CloudWatch console, or the AWS Command Line Interface (AWS CLI). Follow the instructions in the following sections to access your function's logs.

Stream function logs with CloudWatch Logs Live Tail

Amazon CloudWatch Logs Live Tail helps you quickly troubleshoot your functions by displaying a streaming list of new log events directly in the Lambda console. You can view and filter ingested logs from your Lambda functions in real time, helping you to detect and resolve issues quickly.

> **Note**
>
> Live Tail sessions incur costs by session usage time, per minute. For more information about pricing, see Amazon CloudWatch Pricing.

Comparing Live Tail and --log-type Tail

There are several differences between CloudWatch Logs Live Tail and the LogType: Tail option in the Lambda API (`--log-type Tail` in the AWS CLI):

- `--log-type Tail` returns only the first 4 KB of the invocation logs. Live Tail does not share this limit, and can receive up to 500 log events per second.
- `--log-type Tail` captures and sends the logs with the response, which can impact the function's response latency. Live Tail does not affect function response latency.
- `--log-type Tail` supports synchronous invocations only. Live Tail works for both synchronous and asynchronous invocations.

> **Note**
>
> Lambda Managed Instances does not support the `--log-type Tail` option. Use CloudWatch Logs Live Tail or query CloudWatch Logs directly to view logs for Managed Instances functions.

Permissions

The following permissions are required to start and stop CloudWatch Logs Live Tail sessions:

- `logs:DescribeLogGroups`
- `logs:StartLiveTail`
- `logs:StopLiveTail`

Start a Live Tail session in the Lambda console

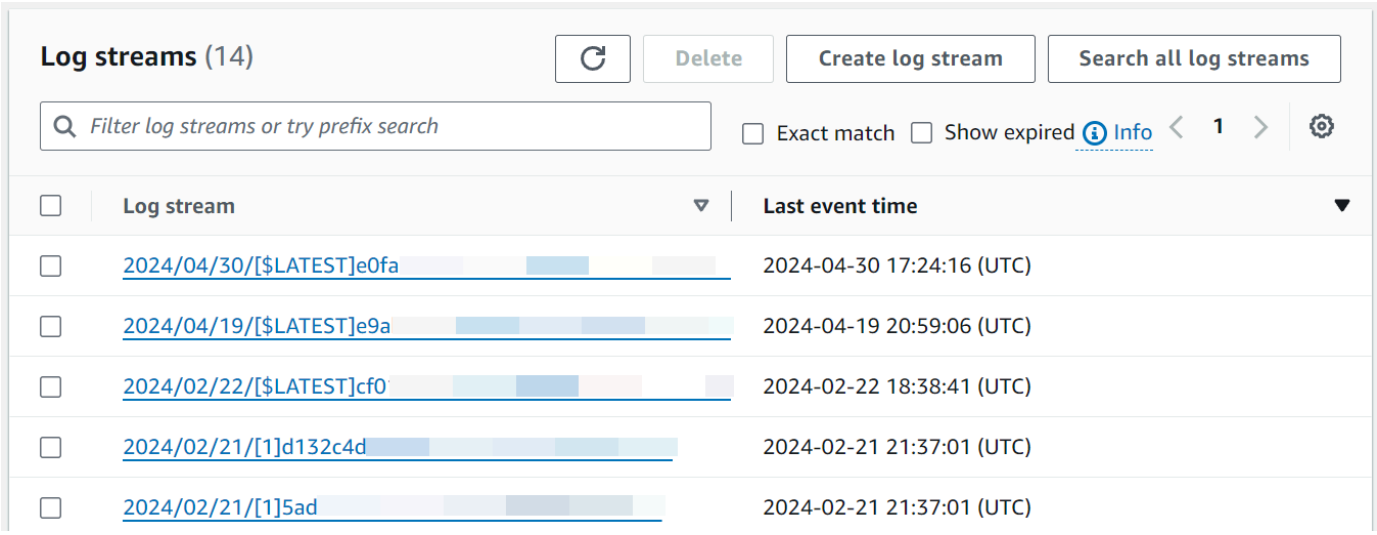
1. Open the Functions page of the Lambda console.
2. Choose the name of the function.
3. Choose the **Test** tab.
4. In the **Test event** pane, choose **CloudWatch Logs Live Tail**.
5. For **Select log groups**, the function's log group is selected by default. You can select up to five log groups at a time.
6. (Optional) To display only log events that contain certain words or other strings, enter the word or string in the **Add filter pattern** box. The filters field is case-sensitive. You can include multiple terms and pattern operators in this field, including regular expressions (regex). For more information about pattern syntax, see Filter pattern syntax. in the *Amazon CloudWatch Logs User Guide*.
7. Choose **Start**. Matching log events begin appearing in the window.
8. To stop the Live Tail session, choose **Stop**.

> **Note**
>
> The Live Tail session automatically stops after 15 minutes of inactivity or when the Lambda console session times out.

Access function logs using the console

1. Open the Functions page of the Lambda console.
2. Select a function.
3. Choose the **Monitor** tab.

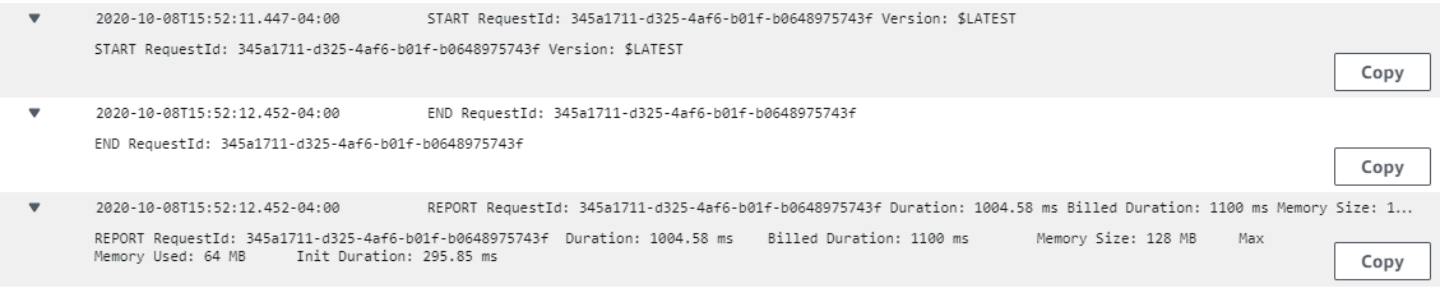
4. Choose **View CloudWatch logs** to open the CloudWatch console.
5. Scroll down and choose the **Log stream** for the function invocations you want to look at.



Each instance of a Lambda function has a dedicated log stream. If a function scales up, each concurrent instance has its own log stream. Each time a new execution environment is created in response to an invocation, this generates a new log stream. The naming convention for log streams is:

```
YYYY/MM/DD[Function version][Execution environment GUID]
```

A single execution environment writes to the same log stream during its lifetime. The log stream contains messages from that execution environment and also any output from your Lambda function's code. Every message is timestamped, including your custom logs. Even if your function does not log any output from your code, there are three minimal log statements generated per invocation (START, END and REPORT):



These logs show:

- **RequestId** – this is a unique ID generated per request. If the Lambda function retries a request, this ID does not change and appears in the logs for each subsequent retry.
- **Start/End** – these bookmark a single invocation, so every log line between these belongs to the same invocation.
- **Duration** – the total invocation time for the handler function, excluding `INIT` code.
- **Billed Duration** – applies rounding logic for billing purposes.
- **Memory Size** – the amount of memory allocated to the function.
- **Max Memory Used** – the maximum amount of memory used during the invocation.
- **Init Duration** – the time taken to run the `INIT` section of code, outside of the main handler.

Access logs with the AWS CLI

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the AWS CLI version 2.

You can use the AWS CLI to retrieve logs for an invocation using the `--log-type` command option. The response contains a `LogResult` field that contains up to 4 KB of base64-encoded logs from the invocation.

Example retrieve a log ID

The following example shows how to retrieve a *log ID* from the `LogResult` field for a function named `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail
```

You should see the following output:

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example decode the logs

In the same command prompt, use the `base64` utility to decode the logs. The following example shows how to retrieve base64-encoded logs for `my-function`.

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

You should see the following output:

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

The `base64` utility is available on Linux, macOS, and Ubuntu on Windows. macOS users may need to use `base64 -D`.

Example get-logs.sh script

In the same command prompt, use the following script to download the last five log events. The script uses `sed` to remove quotes from the output file, and sleeps for 15 seconds to allow time for the logs to become available. The output includes the response from Lambda and the output from the `get-log-events` command.

Copy the contents of the following code sample and save in your Lambda project directory as `get-logs.sh`.

The **cli-binary-format** option is required if you're using AWS CLI version 2. To make this the default setting, run `aws configure set cli-binary-format raw-in-base64-out`. For more

information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide for Version 2*.

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS and Linux (only)

In the same command prompt, macOS and Linux users may need to run the following command to ensure the script is executable.

```
chmod -R 755 get-logs.sh
```

Example retrieve the last five log events

In the same command prompt, run the following script to get the last five log events.

```
./get-logs.sh
```

You should see the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


Parsing logs and structured logging

With CloudWatch Logs Insights, you can search and analyze log data using a specialized query syntax. It performs queries over multiple log groups and provides powerful filtering using glob and regular expressions pattern matching.

You can take advantage of these capabilities by implementing structured logging in your Lambda functions. Structured logging organizes your logs into a pre-defined format, making it easier to query for. Using log levels is an important first step in generating filter-friendly logs that separate informational messages from warnings or errors. For example, consider the following Node.js code:

```
exports.handler = async (event) => {
    console.log("console.log - Application is fine")
    console.info("console.info - This is the same as console.log")
    console.warn("console.warn - Application provides a warning")
```

```
    console.error("console.error - An error occurred")
}
```

The resulting CloudWatch log file contains a separate field specifying the log level:

```
START RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109 Version: $LATEST
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    INFO    console.log - Application is fine
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    INFO    console.info - This is the same as console.log
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    WARN    console.warn - Application provides a warning
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    ERROR   console.error - An error occurred
END RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109
REPORT RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109  Duration: 3.13 ms       Billed Duration: 100 ms Memory Size: 128 MB
Max Memory Used: 64 MB  Init Duration: 142.18 ms
```

A CloudWatch Logs Insights query can then filter on log level. For example, to query for errors only, you can use the following query:

```
fields @timestamp, @message
| filter @message like /ERROR/
| sort @timestamp desc
```

JSON structured logging

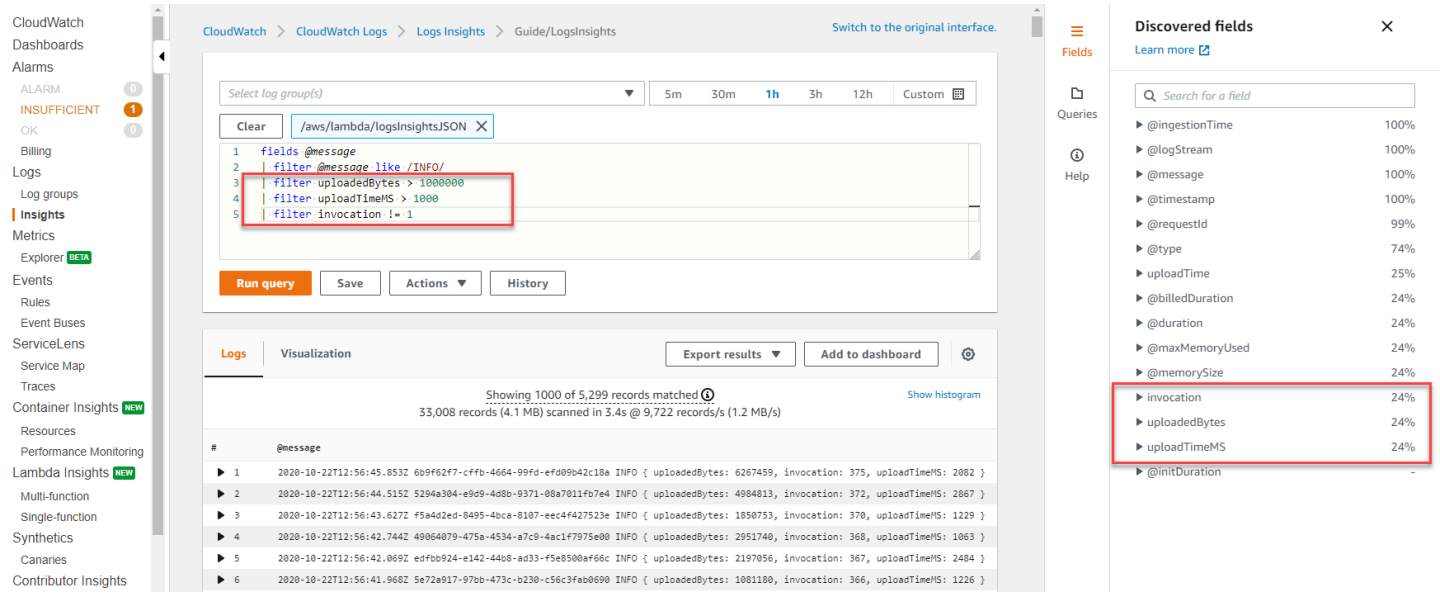
JSON is commonly used to provide structure for application logs. In the following example, the logs have been converted to JSON to output three distinct values:

```
START RequestId: 22fda338-7b46-4c49-9531-efd6e8568480 Version: $LATEST
2020-10-22T11:35:59.216Z        22fda338-7b46-4c49-9531-efd6e8568480    INFO    { uploadedBytes: 5453396,
invocation: 5, uploadTimeMS: 4519 }
END RequestId: 22fda338-7b46-4c49-9531-efd6e8568480
REPORT RequestId: 22fda338-7b46-4c49-9531-efd6e8568480  Duration: 1.25 ms       Billed Duration: 100 ms Memory
Size: 128 MB    Max Memory Used: 65 MB
```

The CloudWatch Logs Insights feature automatically discovers values in JSON output and parses the messages as fields, without the need for custom glob or regular expression. By using the JSON-structured logs, the following query finds invocations where the uploaded file was larger than 1 MB, the upload time was more than 1 second, and the invocation was not a cold start:

```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
| filter invocation != 1
```

This query might produce the following result:



The discovered fields in JSON are automatically populated in the *Discovered fields* menu on the right side. Standard fields emitted by the Lambda service are prefixed with '@', and you can query on these fields in the same way. Lambda logs always include the fields @timestamp, @logStream, @message, @requestId, @duration, @billedDuration, @type, @maxMemoryUsed, @memorySize. If X-Ray is enabled for a function, logs also include @xrayTraceId and @xraySegmentId.

When an AWS event source such as Amazon S3, Amazon SQS, or Amazon EventBridge invokes your function, the entire event is provided to the function as a JSON object input. By logging this event in the first line of the function, you can then query on any of the nested fields using CloudWatch Logs Insights.

Useful Insights queries

The following table shows example Insights queries that can be useful for monitoring Lambda functions.

Description	Example query syntax

The last 100 errors	`fields Timestamp, LogLevel, Message`
	`| filter LogLevel == "ERR"`
	`| sort @timestamp desc`
	`| limit 100`

The top 100 highest billed invocations	`filter @type = "REPORT"`

Description	Example query syntax

	`| fields @requestId, @billedDuration`
	`| sort by @billedDuration desc`
	`| limit 100`

Percentage of cold starts in total invocations	`filter @type = "REPORT"`
	`| stats sum(strcontains(@message,`
	` "Init Duration"))/count(*) * 100 as`
	`  coldStartPct, avg(@duration)`
	`  by bin(5m)`

Percentile report of Lambda duration	`filter @type = "REPORT"`
	`| stats`
	`  avg(@billedDuration) as Average,`
	`  percentile(@billedDuration, 99)`
	` as NinetyNinth,`
	`  percentile(@billedDuration, 95)`
	` as NinetyFifth,`
	`  percentile(@billedDuration, 90)`
	` as Ninetieth`
	`  by bin(30m)`

Percentile report of Lambda memory usage	`filter @type="REPORT"`
	`| stats avg(@maxMemoryUsed/1024/102`
	`4) as mean_MemoryUsed,`
	`  min(@maxMemoryUsed/1024/1024) as`
	` min_MemoryUsed,`
	`  max(@maxMemoryUsed/1024/1024) as`
	` max_MemoryUsed,`
	`  percentile(@maxMemoryUsed/1`
	`024/1024, 95) as Percentile95`

Invocations using 100% of assigned memory	`filter @type = "REPORT" and @maxMemor`
	`yUsed=@memorySize`
	`| stats`
	`  count_distinct(@requestId)`
	`  by bin(30m)`

Description	Example query syntax

Average memory used across invocations	`avgMemoryUsedPERC,`
	`    avg(@billedDuration) as avgDurati`
	`onMS`
	`    by bin(5m)`

Visualization of memory statistics	`filter @type = "REPORT"`
	`| stats`
	`    max(@maxMemoryUsed / 1024 / 1024)`
	` as maxMemMB,`
	`    avg(@maxMemoryUsed / 1024 / 1024)`
	` as avgMemMB,`
	`    min(@maxMemoryUsed / 1024 / 1024)`
	` as minMemMB,`
	`    (avg(@maxMemoryUsed / 1024 /`
	` 1024) / max(@memorySize / 1024 /`
	` 1024)) * 100 as avgMemUsedPct,`
	`    avg(@billedDuration) as avgDurati`
	`onMS`
	`    by bin(30m)`

Invocations where Lambda exited	`filter @message like /Process exited/` `| stats count() by bin(30m)`
Invocations that timed out	`filter @message like /Task timed out/` `| stats count() by bin(30m)`

Latency report	`filter @type = "REPORT"`
	`| stats avg(@duration), max(@dura`
	`tion), min(@duration)`
	`  by bin(5m)`

Description	Example query syntax

Over-provisioned memory	`filter @type = "REPORT"`
	`| stats max(@memorySize / 1024 /`
	`1024) as provisonedMemMB,`
	`min(@maxMemoryUsed / 1024 /`
	`1024) as smallestMemReqMB,`
	`avg(@maxMemoryUsed / 1024 /`
	`1024) as avgMemUsedMB,`
	`max(@maxMemoryUsed / 1024 /`
	`1024) as maxMemUsedMB,`
	`provisonedMemMB - maxMemUse`
	`dMB as overProvisionedMB`

Log visualization and dashboards

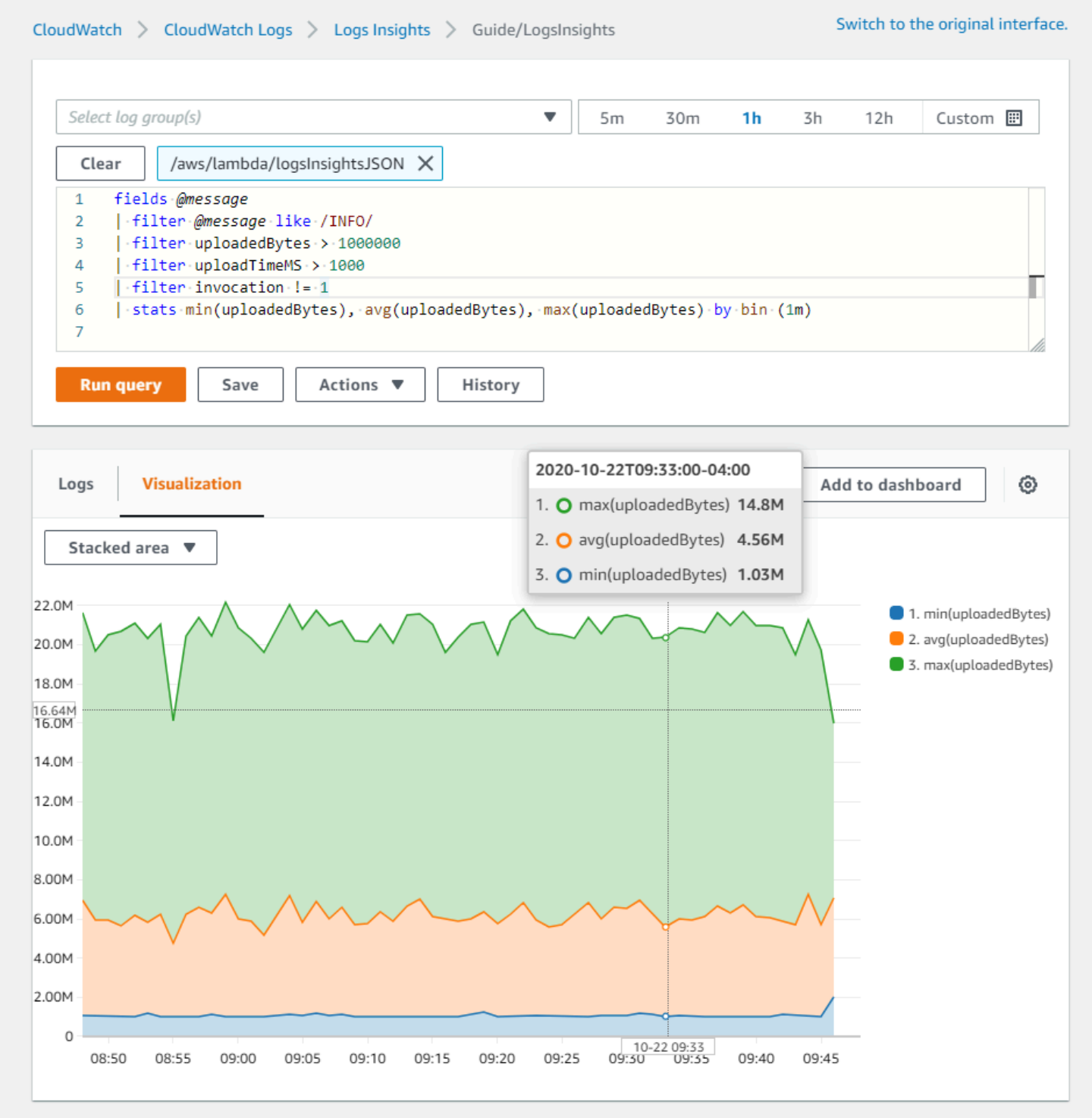
For any CloudWatch Logs Insights query, you can export the results to markdown or CSV format. In some cases, it might be more useful to create visualizations from queries, providing there is at least one aggregation function. The `stats` function allows you to define aggregations and grouping.

The previous *logInsightsJSON* example filtered on upload size and upload time and excluded first invocations. This resulted in a table of data. For monitoring a production system, it may be more useful to visualize minimum, maximum, and average file sizes to find outliers. To do this, apply the stats function with the required aggregates, and group on a time value such as every minute:

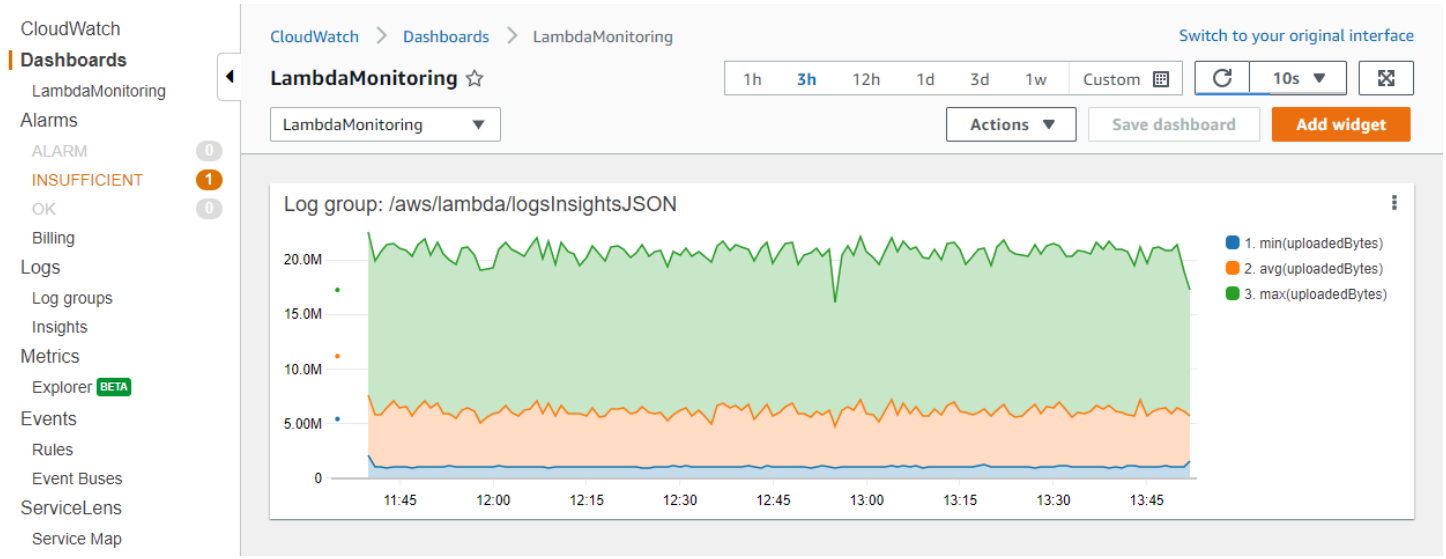
For example, consider the following query. This is the same example query from the the section called “JSON structured logging” section, but with additional aggregation functions:

```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
| filter invocation != 1
| stats min(uploadedBytes), avg(uploadedBytes), max(uploadedBytes) by bin (1m)
```

We included these aggregates because it may be more useful to visualize minimum, maximum, and average file sizes to find outliers. You can view the results in the **Visualization** tab:



After you have finished building the visualization, you can optionally add the graph to a CloudWatch dashboard. To do this, choose **Add to dashboard** above the visualization. This adds the query as a widget and enables you to select automatic refresh intervals, making it easier to continuously monitor the results:



Sending Lambda function logs to Firehose

The Lambda console now offers the option to send function logs to Firehose. This enables real-time streaming of your logs to various destinations supported by Firehose, including third-party analytics tools and custom endpoints.

> **Note**
>
> You can configure Lambda function logs to be sent to Firehose using the Lambda console, AWS CLI, AWS CloudFormation, and all AWS SDKs.

Pricing

For details on pricing, see Amazon CloudWatch pricing.

Required permissions for Firehose log destination

When using the Lambda console to configure Firehose as your function's log destination, you need:

1. The required IAM permissions to use CloudWatch Logs with Lambda.
2. To set up subscription filters with Firehose. This filter defines which log events are delivered to your Firehose stream.

Sending Lambda function logs to Firehose

In the Lambda console, you can send function logs directly to Firehose after creating a new function. To do this, complete these steps:

1. Sign in to the AWS Management Console and open the Lambda console.
2. Choose your function's name.
3. Choose the **Configuration** tab.
4. Choose the **Monitoring and operations tools** tab.
5. In the "Logging configuration" section, choose **Edit**.
6. In the "Log content" section, select a log format.
7. In the "Log destination" section, complete the following steps:
   a. Select a destination service.
   b. Choose to **Create a new log group** or use an **Existing log group**.

      > **Note**
      >
      > If choosing an existing log group for a Firehose destination, ensure the log group you choose is a `Delivery` log group type.

   c. Choose a Firehose stream.
   d. The CloudWatch `Delivery` log group will appear.
8. Choose **Save**.

> **Note**
>
> If the IAM role provided in the console doesn't have the required permission, then the destination setup will fail. To fix this, refer to Required permissions for Firehose log destination to provide the required permissions.

Cross-Account Logging

You can configure Lambda to send logs to Firehose delivery stream in a different AWS account. This requires setting up a destination and configuring appropriate permissions in both accounts.

For detailed instructions on setting up cross-account logging, including required IAM roles and policies, see Setting up a new cross-account subscription in the CloudWatch Logs documentation.

Sending Lambda function logs to Amazon S3

You can configure your Lambda function to send logs directly to Amazon S3 using the Lambda console. This feature provides a cost-effective solution for long-term log storage and enables powerful analysis options using services like Athena.

> **Note**
>
> You can configure Lambda function logs to be sent to Amazon S3 using the Lambda console, AWS CLI, AWS CloudFormation, and all AWS SDKs.

Pricing

For details on pricing, see Amazon CloudWatch pricing.

Required permissions for Amazon S3 log destination

When using the Lambda console to configure Amazon S3 as your function's log destination, you need:

1. The required IAM permissions to use CloudWatch Logs with Lambda.
2. To Set up a CloudWatch Logs subscriptions filter to send Lambda function logs to Amazon S3. This filter defines which log events are delivered to your Amazon S3 bucket.

Set up a CloudWatch Logs subscriptions filter to send Lambda function logs to Amazon S3

To send logs from CloudWatch Logs to Amazon S3, you need to create a subscription filter. This filter defines which log events are delivered to your Amazon S3 bucket. Your Amazon S3 bucket must be in the same Region as your log group.

To create a subscription filter for Amazon S3

1. Create an Amazon Simple Storage Service (Amazon S3) bucket. We recommend that you use a bucket that was created specifically for CloudWatch Logs. However, if you want to use an existing bucket, skip to step 2.

   Run the following command, replacing the placeholder Region with the Region you want to use:

   ```
   aws s3api create-bucket --bucket amzn-s3-demo-bucket2 --create-bucket-configuration
    LocationConstraint=region
   ```

   > **Note**
   >
   > `amzn-s3-demo-bucket2` is an example Amazon S3 bucket name. It is *reserved*. For this procedure to work, you must to replace it with your unique Amazon S3 bucket name.

   The following is example output:

   ```
   {
       "Location": "/amzn-s3-demo-bucket2"
   }
   ```

2. Create the IAM role that grants CloudWatch Logs permission to put data into your Amazon S3 bucket. This policy includes a aws:SourceArn global condition context key to help prevent the confused deputy security issue. For more information, see Confused deputy prevention.

   a. Use a text editor to create a trust policy in a file `~/TrustPolicyForCWL.json` as follows:

   ```
   {
       "Statement": {
           "Effect": "Allow",
           "Principal": { "Service": "logs.amazonaws.com" },
           "Condition": {
               "StringLike": {
                   "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
               }
            },
           "Action": "sts:AssumeRole"
       }
   ```

```
}
```

b. Use the create-role command to create the IAM role, specifying the trust policy file. Note of the returned Role.Arn value, as you will need it in a later step:

```
aws iam create-role \
 --role-name CWLtoS3Role \
 --assume-role-policy-document file://~/TrustPolicyForCWL.json
{
    "Role": {
        "AssumeRolePolicyDocument": {
            "Statement": {
                "Action": "sts:AssumeRole",
                "Effect": "Allow",
                "Principal": {
                    "Service": "logs.amazonaws.com"
                },
                "Condition": {
                    "StringLike": {
                        "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
                    }
                }
            }
        },
        "RoleId": "AAOIIAH450GAB4HC5F431",
        "CreateDate": "2015-05-29T13:46:29.431Z",
        "RoleName": "CWLtoS3Role",
        "Path": "/",
        "Arn": "arn:aws:iam::123456789012:role/CWLtoS3Role"
    }
}
```

3. Create a permissions policy to define what actions CloudWatch Logs can do on your account. First, use a text editor to create a permissions policy in a file ~/PermissionsForCWL.json:

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:PutObject"],
      "Resource": ["arn:aws:s3:::amzn-s3-demo-bucket2/*"]
    }
  ]
```

```
}
```

Associate the permissions policy with the role using the following `put-role-policy` command:

```
aws iam put-role-policy --role-name CWLtoS3Role --policy-name Permissions-Policy-For-S3 --policy-document file://~/PermissionsForCWL.json
```

4. Create a `Delivery` log group or use a existing `Delivery` log group.

```
aws logs create-log-group --log-group-name my-logs --log-group-class DELIVERY --region REGION_NAME
```

5. `PutSubscriptionFilter` to set up destination

```
aws logs put-subscription-filter
--log-group-name my-logs
--filter-name my-lambda-delivery
--filter-pattern ""
--destination-arn arn:aws:s3:::amzn-s3-demo-bucket2
--role-arn arn:aws:iam::123456789012:role/CWLtoS3Role
--region REGION_NAME
```

Sending Lambda function logs to Amazon S3

In the Lambda console, you can send function logs directly to Amazon S3 after creating a new function. To do this, complete these steps:

1. Sign in to the AWS Management Console and open the Lambda console.
2. Choose your function's name.
3. Choose the **Configuration** tab.
4. Choose the **Monitoring and operations tools** tab.
5. In the "Logging configuration" section, choose **Edit**.
6. In the "Log content" section, select a log format.
7. In the "Log destination" section, complete the following steps:
   a. Select a destination service.
   b. Choose to **Create a new log group** or use an **Existing log group**.

> **Note**
>
> If choosing an existing log group for an Amazon S3 destination, ensure the log group you choose is a `Delivery` log group type.

c. Choose an Amazon S3 bucket to be the destination for your function logs.

d. The CloudWatch `Delivery` log group will appear.

8. Choose **Save**.

> **Note**
>
> If the IAM role provided in the console doesn't have the required permissions, then the destination setup will fail. To fix this, refer to Required permissions for Amazon S3 log destination.

Cross-Account Logging

You can configure Lambda to send logs to an Amazon S3 bucket in a different AWS account. This requires setting up a destination and configuring appropriate permissions in both accounts.

For detailed instructions on setting up cross-account logging, including required IAM roles and policies, see Setting up a new cross-account subscription in the CloudWatch Logs documentation.

Logging AWS Lambda API calls using AWS CloudTrail

AWS Lambda is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service. CloudTrail captures API calls for Lambda as events. The calls captured include calls from the Lambda console and code calls to the Lambda API operations. Using the information collected by CloudTrail, you can determine the request that was made to Lambda, the IP address from which the request was made, when it was made, and additional details.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root user or user credentials.
- Whether the request was made on behalf of an IAM Identity Center user.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

CloudTrail is active in your AWS account when you create the account and you automatically have access to the CloudTrail **Event history**. The CloudTrail **Event history** provides a viewable, searchable, downloadable, and immutable record of the past 90 days of recorded management events in an AWS Region. For more information, see Working with CloudTrail Event history in the *AWS CloudTrail User Guide*. There are no CloudTrail charges for viewing the **Event history**.

For an ongoing record of events in your AWS account past 90 days, create a trail or a CloudTrail Lake event data store.

CloudTrail trails

A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. All trails created using the AWS Management Console are multi-Region. You can create a single-Region or a multi-Region trail by using the AWS CLI. Creating a multi-Region trail is recommended because you capture activity in all AWS Regions in your account. If you create a single-Region trail, you can view only the events logged in the trail's AWS Region. For more information about trails, see Creating a trail for your AWS account and Creating a trail for an organization in the *AWS CloudTrail User Guide*.

You can deliver one copy of your ongoing management events to your Amazon S3 bucket at no charge from CloudTrail by creating a trail, however, there are Amazon S3 storage charges. For

more information about CloudTrail pricing, see AWS CloudTrail Pricing. For information about Amazon S3 pricing, see Amazon S3 Pricing.

CloudTrail Lake event data stores

CloudTrail Lake lets you run SQL-based queries on your events. CloudTrail Lake converts existing events in row-based JSON format to Apache ORC format. ORC is a columnar storage format that is optimized for fast retrieval of data. Events are aggregated into *event data stores*, which are immutable collections of events based on criteria that you select by applying advanced event selectors. The selectors that you apply to an event data store control which events persist and are available for you to query. For more information about CloudTrail Lake, see Working with AWS CloudTrail Lake in the *AWS CloudTrail User Guide*.

CloudTrail Lake event data stores and queries incur costs. When you create an event data store, you choose the pricing option you want to use for the event data store. The pricing option determines the cost for ingesting and storing events, and the default and maximum retention period for the event data store. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

Lambda data events in CloudTrail

Data events provide information about the resource operations performed on or in a resource (for example, reading or writing to an Amazon S3 object). These are also known as data plane operations. Data events are often high-volume activities. By default, CloudTrail doesn't log most data events, and the CloudTrail **Event history** doesn't record them.

One CloudTrail data event that is logged by default for supported services is `LambdaESMDisabled`. To learn more about using this event to help troubleshoot issues with Lambda event source mappings, see the section called "Using CloudTrail to troubleshoot disabled Lambda event sources".

Additional charges apply for data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

You can log data events for the `AWS::Lambda::Function` resource type by using the CloudTrail console, AWS CLI, or CloudTrail API operations. For more information about how to log data events, see Logging data events with the AWS Management Console and Logging data events with the AWS Command Line Interface in the *AWS CloudTrail User Guide*.

The following table lists the Lambda resource type for which you can log data events. The **Data event type (console)** column shows the value to choose from the **Data event type** list on the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

Data event type (console)	resources.type value	Data APIs logged to CloudTrail
Lambda	`AWS::Lambda::Function`	Invoke

You can configure advanced event selectors to filter on the `eventName`, `readOnly`, and `resources.ARN` fields to log only those events that are important to you. The following example is the JSON view of a data event configuration that logs events for a specific function only. For more information about these fields, see AdvancedFieldSelector in the *AWS CloudTrail API Reference*.

```
[
  {
    "name": "function-invokes",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::Lambda::Function"
        ]
      },
      {
        "field": "resources.ARN",
        "equals": [
          "arn:aws:lambda:us-east-1:111122223333:function:hello-world"
```

```
        ]
      }
    ]
  }
]
```

Lambda management events in CloudTrail

Management events provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail logs management events.

Lambda supports logging the following actions as management events in CloudTrail log files.

> **Note**
>
> In the CloudTrail log file, the `eventName` might include date and version information, but it is still referring to the same public API action. For example the, `GetFunction` action appears as `GetFunction20150331v2`. The following list specifies when the event name differs from the API action name.

- AddLayerVersionPermission
- AddPermission (event name: `AddPermission20150331v2`)
- CreateAlias (event name: `CreateAlias20150331`)
- CreateEventSourceMapping (event name: `CreateEventSourceMapping20150331`)
- CreateFunction (event name: `CreateFunction20150331`)

  (The `Environment` and `ZipFile` parameters are omitted from the CloudTrail logs for `CreateFunction`.)
- CreateFunctionUrlConfig
- DeleteAlias (event name: `DeleteAlias20150331`)
- DeleteCodeSigningConfig
- DeleteEventSourceMapping (event name: `DeleteEventSourceMapping20150331`)
- DeleteFunction (event name: `DeleteFunction20150331`)
- DeleteFunctionConcurrency (event name: `DeleteFunctionConcurrency20171031`)

- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias (event name: `GetAlias20150331`)
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig
- GetFunctionConfiguration
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayerVersion (event name: `PublishLayerVersion20181031`)

  (The `ZipFile` parameter is omitted from the CloudTrail logs for `PublishLayerVersion`.)
- PublishVersion (event name: `PublishVersion20150331`)
- PutFunctionConcurrency (event name: `PutFunctionConcurrency20171031`)
- PutFunctionCodeSigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- PutRuntimeManagementConfig
- RemovePermission (event name: `RemovePermission20150331v2`)
- TagResource (event name: `TagResource20170331v2`)
- UntagResource (event name: `UntagResource20170331v2`)
- UpdateAlias (event name: `UpdateAlias20150331`)
- UpdateCodeSigningConfig
- UpdateEventSourceMapping (event name: `UpdateEventSourceMapping20150331`)
- UpdateFunctionCode (event name: `UpdateFunctionCode20150331v2`)

  (The `ZipFile` parameter is omitted from the CloudTrail logs for `UpdateFunctionCode`.)
- UpdateFunctionConfiguration (event name: `UpdateFunctionConfiguration20150331v2`)

(The `Environment` parameter is omitted from the CloudTrail logs for `UpdateFunctionConfiguration`.)

- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

Using CloudTrail to troubleshoot disabled Lambda event sources

When you change the state of an event source mapping using the UpdateEventSourceMapping API action, the API call is logged as a management event in CloudTrail. Event source mappings can also transition directly to the `Disabled` state due to errors.

For the following services, Lambda publishes the `LambdaESMDisabled` data event to CloudTrail when your event source transitions to the Disabled state:

- Amazon Simple Queue Service (Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda doesn't support this event for any other event source mapping types.

To receive alerts when event source mappings for supported services transition to the `Disabled` state, set up an alarm in Amazon CloudWatch using the `LambdaESMDisabled` CloudTrail event. To learn more about setting up a CloudWatch alarm, see Creating CloudWatch alarms for CloudTrail events: examples.

The `serviceEventDetails` entity in the `LambdaESMDisabled` event message contains one of the following error codes.

`RESOURCE_NOT_FOUND`

The resource specified in the request does not exist.

`FUNCTION_NOT_FOUND`

The function attached to the event source does not exist.

`REGION_NAME_NOT_VALID`

A Region name provided to the event source or function is invalid.

AUTHORIZATION_ERROR

Permissions have not been set, or are misconfigured.

FUNCTION_IN_FAILED_STATE

The function code does not compile, has encountered an unrecoverable exception, or a bad deployment has occurred.

Lambda event examples

An event represents a single request from any source and includes information about the requested API operation, the date and time of the operation, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so events don't appear in any specific order.

The following example shows CloudTrail log entries for the `GetFunction` and `DeleteFunction` actions.

> **Note**
>
> The `eventName` might include date and version information, such as `"GetFunction20150331"`, but it is still referring to the same public API.

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
```

```
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "errorMessage": "User: arn:aws:iam::111122223333:user/myUserName is not
 authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-
west-2:111122223333:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    },
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction20150331",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "requestParameters": {
        "functionName": "basic-node-task"
      },
      "responseElements": null,
      "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    }
  ]
}
```

For information about CloudTrail record contents, see CloudTrail record contents in the *AWS CloudTrail User Guide*.

Visualize Lambda function invocations using AWS X-Ray

You can use AWS X-Ray to visualize the components of your application, identify performance bottlenecks, and troubleshoot requests that resulted in an error. Your Lambda functions send trace data to X-Ray, and X-Ray processes the data to generate a service map and searchable trace summaries.

Lambda supports two tracing modes for X-Ray: `Active` and `PassThrough`. With `Active` tracing, Lambda automatically creates trace segments for function invocations and sends them to X-Ray. `PassThrough` mode, on the other hand, simply propagates the tracing context to downstream services. If you've enabled `Active` tracing for your function, Lambda automatically sends traces to X-Ray for sampled requests. Typically, an upstream service, such as Amazon API Gateway or an application hosted on Amazon EC2 that is instrumented with the X-Ray SDK, decides whether incoming requests should be traced, then adds that sampling decision as a tracing header. Lambda uses that header to decide to send traces or not. Traces from upstream message producers, such as Amazon SQS, are automatically linked to traces from downstream Lambda functions, creating an end-to-end view of the entire application. For more information, see Tracing event-driven applications in the *AWS X-Ray Developer Guide*.

> **Note**
>
> X-Ray tracing is currently not supported for Lambda functions with Amazon Managed Streaming for Apache Kafka (Amazon MSK), self-managed Apache Kafka, Amazon MQ with ActiveMQ and RabbitMQ, or Amazon DocumentDB event source mappings.

To toggle active tracing on your Lambda function with the console, follow these steps:

To turn on active tracing

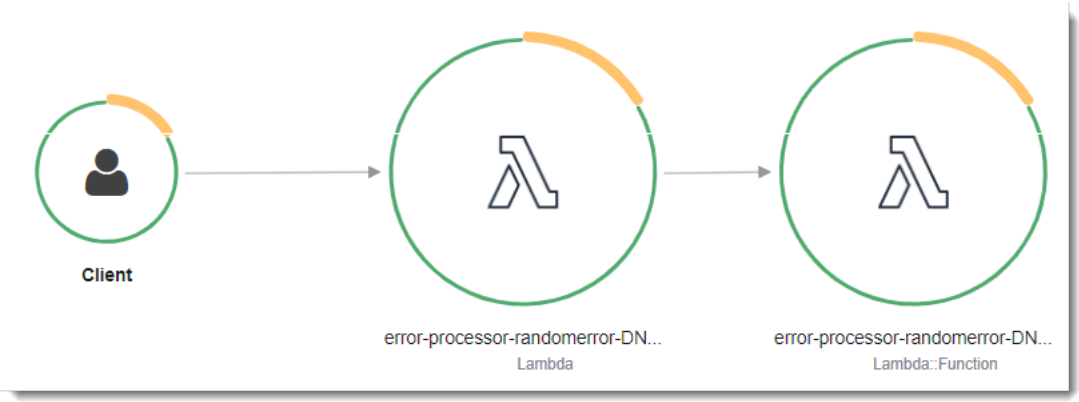
1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose **Configuration** and then choose **Monitoring and operations tools**.
4. Under **Additional monitoring tools**, choose **Edit**.
5. Under **CloudWatch Application Signals and AWS X-Ray**, choose **Enable** for **Lambda service traces**.
6. Choose **Save**.

Your function needs permission to upload trace data to X-Ray. When you activate tracing in the Lambda console, Lambda adds the required permissions to your function's execution role. Otherwise, add the AWSXRayDaemonWriteAccess policy to the execution role.

X-Ray doesn't trace all requests to your application. X-Ray applies a sampling algorithm to ensure that tracing is efficient, while still providing a representative sample of all requests. The sampling rate is 1 request per second and 5 percent of additional requests. You can't configure the X-Ray sampling rate for your functions.

Understanding X-Ray traces

In X-Ray, a *trace* records information about a request that is processed by one or more *services*. Lambda records 2 segments per trace, which creates two nodes on the service graph. The following image highlights these two nodes:



The first node on the left represents the Lambda service, which receives the invocation request. The second node represents your specific Lambda function.

The segment recorded for the Lambda service, `AWS::Lambda`, covers all the steps required to prepare the Lambda execution environment. This includes scheduling the MicroVM, creating or unfreezing an execution environment with the resources you have configured, as well as downloading your function code and all layers.

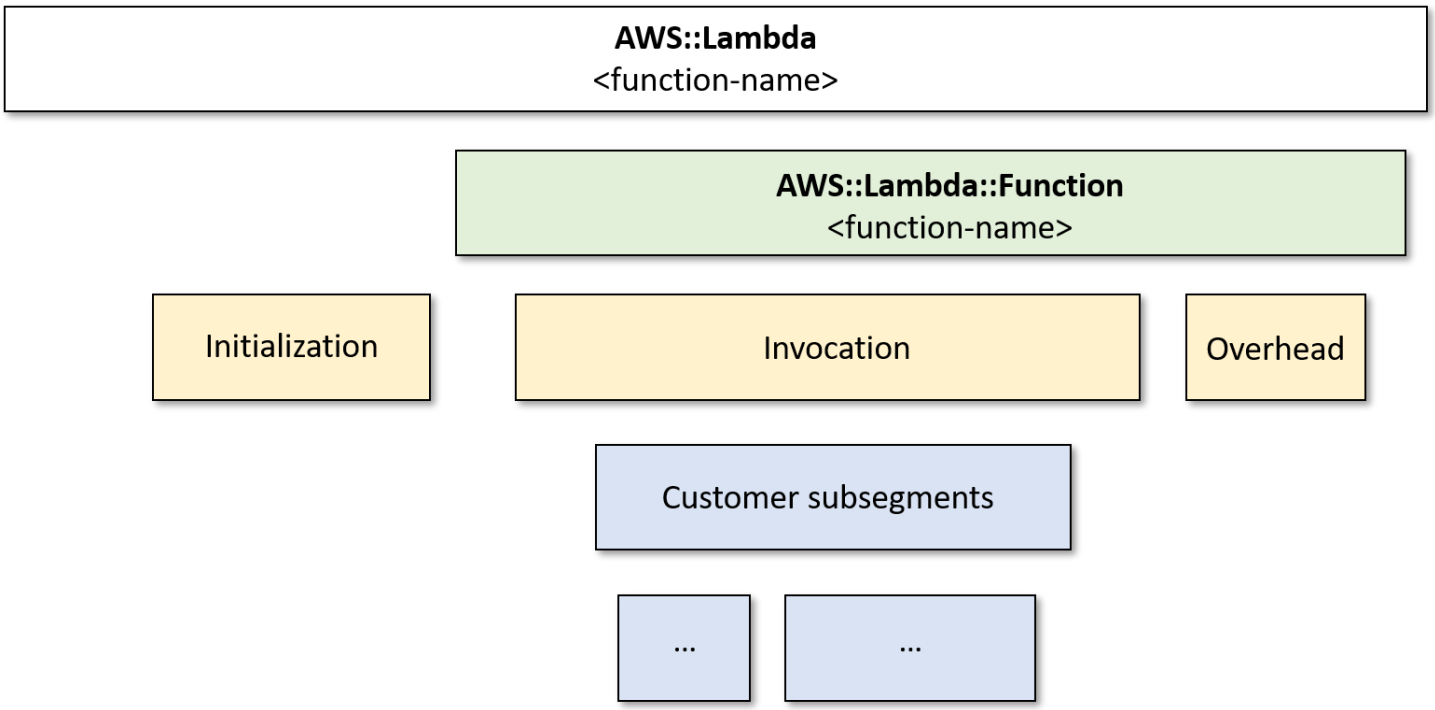
The `AWS::Lambda::Function` segment is for the work done by the function.

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.

This change affects the subsegments of the function segment. The following paragraphs describe both the old and new formats for these subsegments.
These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

Old-style AWS X-Ray Lambda segment structure

The old-style X-Ray structure for the `AWS::Lambda` segment looks like the following:



In this format, the function segment has subsegments for `Initialization`, `Invocation`, and `Overhead`. For Lambda SnapStart only, there is also a `Restore` subsegment (not shown on this diagram).

The `Initialization` subsegment represents the init phase of the Lambda execution environment lifecycle. During this phase, Lambda initializes extensions, initializes the runtime, and runs the function's initialization code.

The `Invocation` subsegment represents the invoke phase where Lambda invokes the function handler. This begins with runtime and extension registration and it ends when the runtime is ready to send the response.

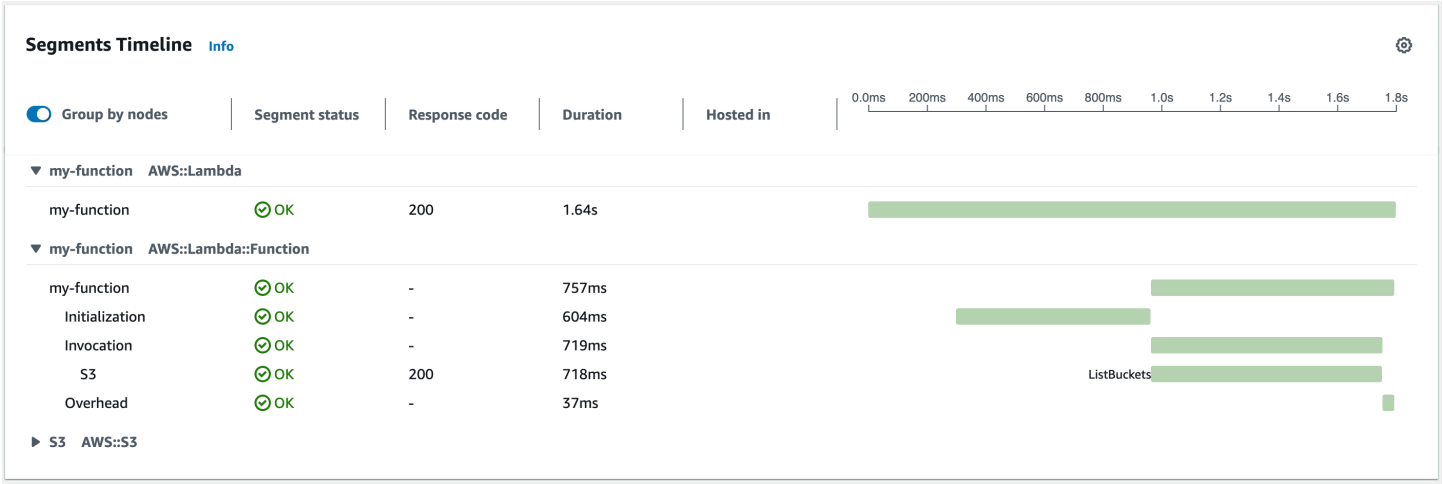
(Lambda SnapStart only) The `Restore` subsegment shows the time it takes for Lambda to restore a snapshot, load the runtime, and run any after-restore runtime hooks. The process of restoring snapshots can include time spent on activities outside the MicroVM. This time is reported in the `Restore` subsegment. You aren't charged for the time spent outside the microVM to restore a snapshot.

The `Overhead` subsegment represents the phase that occurs between the time when the runtime sends the response and the signal for the next invoke. During this time, the runtime finishes all tasks related to an invoke and prepares to freeze the sandbox.

> **Important**
>
> You can use the X-Ray SDK to extend the `Invocation` subsegment with additional subsegments for downstream calls, annotations, and metadata. You can't access the function segment directly or record work done outside of the handler invocation scope.

For more information about Lambda execution environment phases, see the section called "Execution environment".

An example trace using the old-style X-Ray structure is shown in the following diagram.

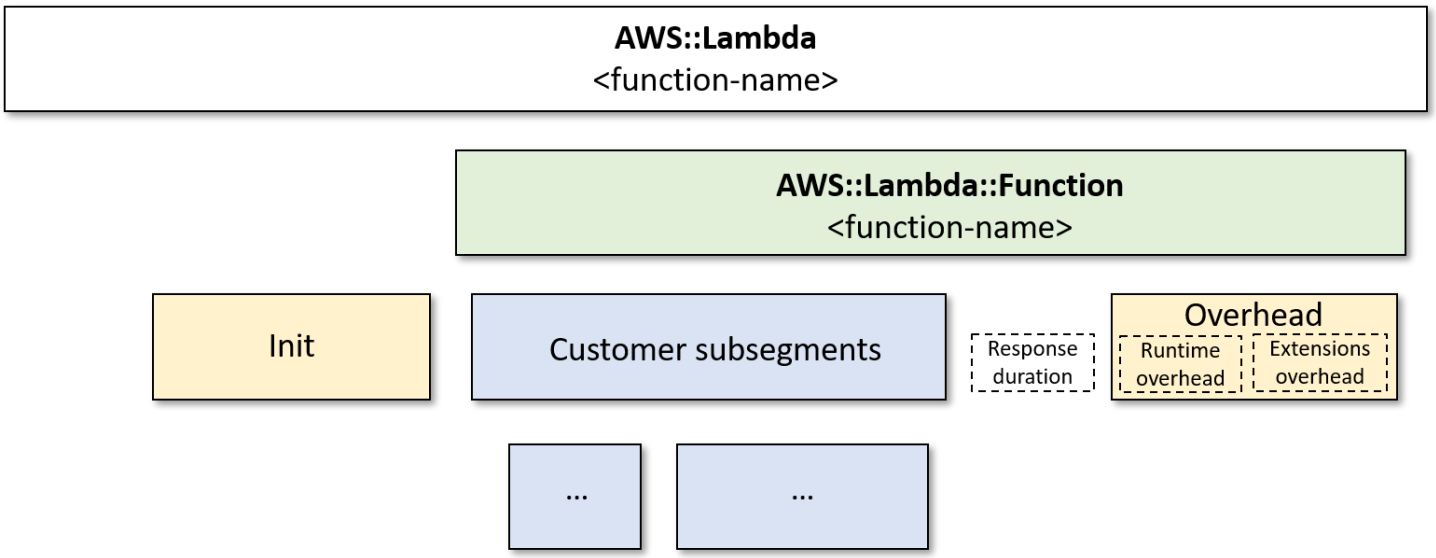


Note the two segments in the example. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.

> **Note**
>
> Occasionally, you may notice a large gap between the function initialization and invocation phases in your X-Ray traces. For functions using provisioned concurrency, this is because Lambda initializes your function instances well in advance of invocation. For functions using unreserved (on-demand) concurrency, Lambda may proactively initialize a function instance, even if there's no invocation. Visually, both of these cases show up as a time gap between the initialization and invocation phases.

New-style AWS X-Ray Lambda segment structure

The new-style X-Ray structure for the `AWS::Lambda` segment looks like the following:

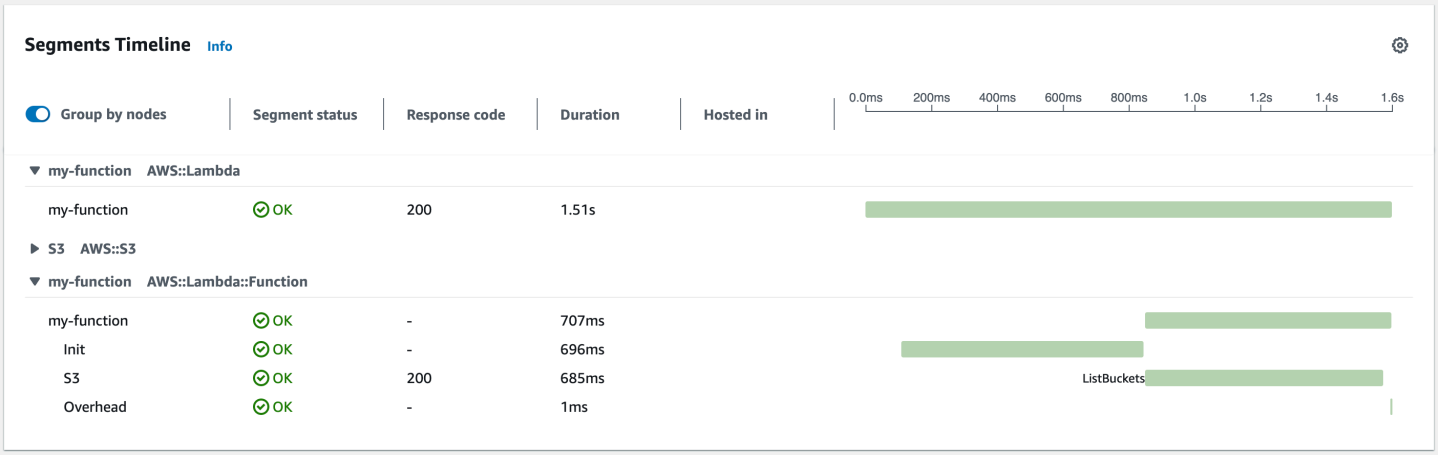


In this new format, The `Init` subsegment represents the init phase of the Lambda execution environment lifecycle as before.

There is no invocation segment in the new format. Instead, customer subsegments are attached directly to the `AWS::Lambda::Function` segment. This segment contains the following metrics as annotations:

- `aws.responseLatency` - the time taken for the function to run
- `aws.responseDuration` - the time taken to transfer the response to the customer
- `aws.runtimeOverhead` - the amount of additional time the runtime needed to finish
- `aws.extensionOverhead` - the amount of additional time the extensions needed to finish

An example trace using the new-style X-Ray structure is shown in the following diagram.



Note the two segments in the example. Both are named **my-function**, but one has an origin of `AWS::Lambda` and the other has an origin of `AWS::Lambda::Function`. If the `AWS::Lambda` segment shows an error, the Lambda service had an issue. If the `AWS::Lambda::Function` segment shows an error, your function had an issue.

See the following topics for a language-specific introduction to tracing in Lambda:

- Instrumenting Node.js code in AWS Lambda
- Instrumenting Python code in AWS Lambda
- Instrumenting Ruby code in AWS Lambda
- Instrumenting Java code in AWS Lambda
- Instrumenting Go code in AWS Lambda
- Instrumenting C# code in AWS Lambda

For a full list of services that support active instrumentation, see Supported AWS services in the AWS X-Ray Developer Guide.

Default tracing behavior in Lambda

If you do not have `Active` tracing turned on, Lambda defaults to `PassThrough` tracing mode.

In `PassThrough` mode, Lambda forwards the X-Ray tracing header to downstream services, but does not send traces automatically. This is true even if the tracing header contains a decision to sample the request. If the upstream service does not provide an X-Ray tracing header, Lambda

generates a header and makes the decision not to sample. However, you can send your own traces by calling tracing libraries from your function code.

> **Note**
>
> Previously, Lambda would send traces automatically when upstream services, such as Amazon API Gateway, added a tracing header. By not sending traces automatically, Lambda gives you the control to trace the functions that are important to you. If your solution depends on this passive tracing behavior, switch to `Active` tracing.

Execution role permissions

Lambda needs the following permissions to send trace data to X-Ray. Add them to your function's execution role.

- xray:PutTraceSegments
- xray:PutTelemetryRecords

These permissions are included in the AWSXRayDaemonWriteAccess managed policy.

Enabling `Active` tracing with the Lambda API

To manage tracing configuration with the AWS CLI or AWS SDK, use the following API operations:

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

The following example AWS CLI command enables active tracing on a function named **my-function**.

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

Tracing mode is part of the version-specific configuration when you publish a version of your function. You can't change the tracing mode on a published version.

Enabling `Active` tracing with CloudFormation

To activate tracing on an `AWS::Lambda::Function` resource in an CloudFormation template, use the `TracingConfig` property.

Example function-inline.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

For an AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` resource, use the `Tracing` property.

Example template.yml – Tracing configuration

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

Monitor function performance with Amazon CloudWatch Lambda Insights

Amazon CloudWatch Lambda Insights collects and aggregates Lambda function runtime performance metrics and logs for your serverless applications. This page describes how to enable and use Lambda Insights to diagnose issues with your Lambda functions.

Sections

- How Lambda Insights monitors serverless applications
- Pricing
- Supported runtimes
- Enabling Lambda Insights in the Lambda console
- Enabling Lambda Insights programmatically
- Using the Lambda Insights dashboard
- Example workflow to detect function anomalies
- Example workflow using queries to troubleshoot a function
- What's next?

How Lambda Insights monitors serverless applications

CloudWatch Lambda Insights is a monitoring and troubleshooting solution for serverless applications running on AWS Lambda. The solution collects, aggregates, and summarizes system-level metrics including CPU time, memory, disk and network usage. It also collects, aggregates, and summarizes diagnostic information such as cold starts and Lambda worker shutdowns to help you isolate issues with your Lambda functions and resolve them quickly.

Lambda Insights uses a new CloudWatch Lambda Insights extension, which is provided as a Lambda layer. When you enable this extension on a Lambda function for a supported runtime, it collects system-level metrics and emits a single performance log event for every invocation of that Lambda function. CloudWatch uses embedded metric formatting to extract metrics from the log events. For more information, see Using AWS Lambda extensions.

The Lambda Insights layer extends the `CreateLogStream` and `PutLogEvents` for the `/aws/lambda-insights/` log group.

Pricing

When you enable Lambda Insights for your Lambda function, Lambda Insights reports 8 metrics per function and every function invocation sends about 1KB of log data to CloudWatch. You only pay for the metrics and logs reported for your function by Lambda Insights. There are no minimum fees or mandatory service usage policies. You do not pay for Lambda Insights if the function is not invoked. For a pricing example, see Amazon CloudWatch pricing.

Supported runtimes

You can use Lambda Insights with any of the runtimes that support Lambda extensions.

Enabling Lambda Insights in the Lambda console

You can enable Lambda Insights enhanced monitoring on new and existing Lambda functions. When you enable Lambda Insights on a function in the Lambda console for a supported runtime, Lambda adds the Lambda Insights extension as a layer to your function, and verifies or attempts to attach the `CloudWatchLambdaInsightsExecutionRolePolicy` policy to your function's execution role.

To enable Lambda Insights in the Lambda console

1. Open the Functions page of the Lambda console.
2. Choose your function.
3. Choose the **Configuration** tab.
4. On the left menu, choose **Monitoring and operations tools**.
5. On the **Additional monitoring tools** pane, choose **Edit**.
6. Under **CloudWatch Lambda Insights**, turn on **Enhanced monitoring**.
7. Choose **Save**.

Enabling Lambda Insights programmatically

You can also enable Lambda Insights using the AWS Command Line Interface (AWS CLI), AWS Serverless Application Model (SAM) CLI, CloudFormation, or the AWS Cloud Development Kit (AWS CDK). When you enable Lambda Insights programmatically on a function for a supported runtime, CloudWatch attaches the `CloudWatchLambdaInsightsExecutionRolePolicy` policy to your function's execution role.

For more information, see Getting started with Lambda Insights in the *Amazon CloudWatch User Guide*.

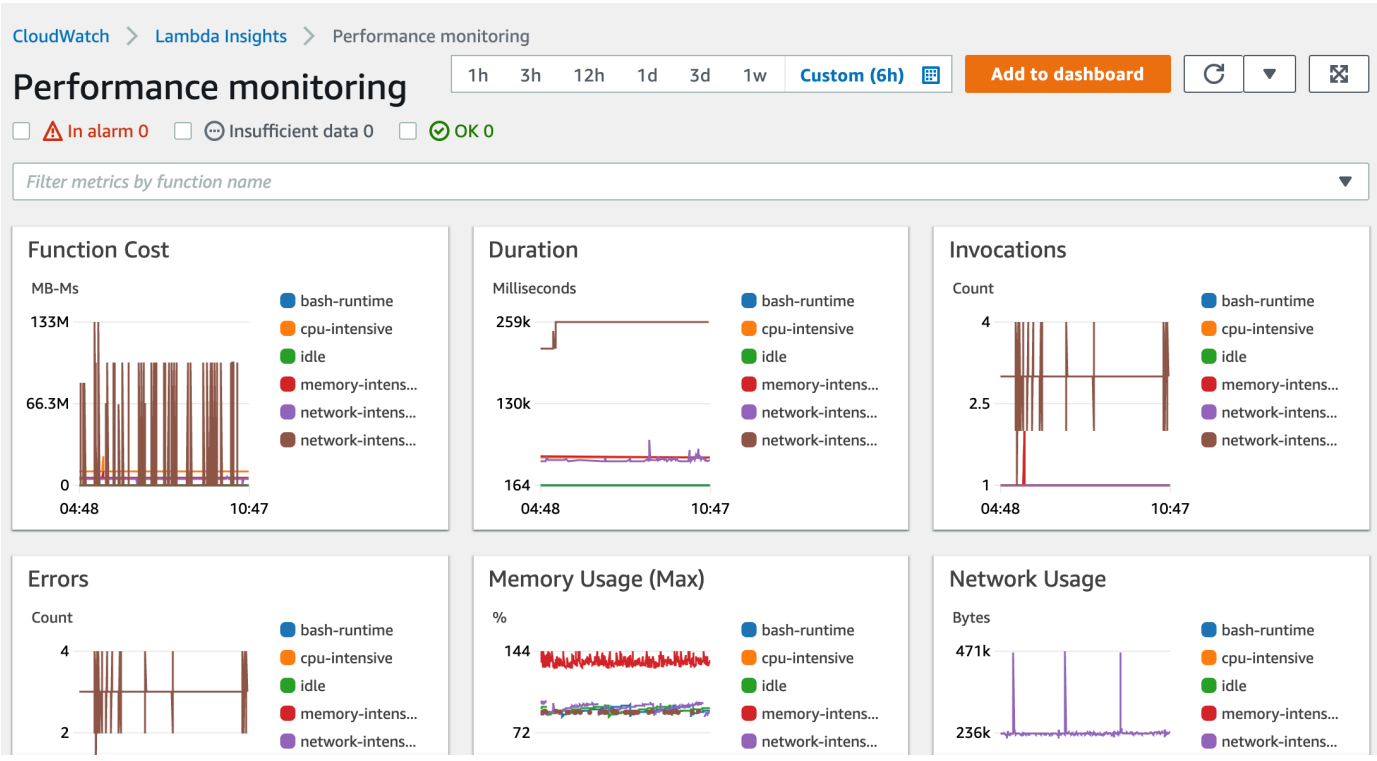
Using the Lambda Insights dashboard

The Lambda Insights dashboard has two views in the CloudWatch console: the multi-function overview and the single-function view. The multi-function overview aggregates the runtime metrics for the Lambda functions in the current AWS account and Region. The single-function view shows the available runtime metrics for a single Lambda function.

You can use the Lambda Insights dashboard multi-function overview in the CloudWatch console to identify over- and under-utilized Lambda functions. You can use the Lambda Insights dashboard single-function view in the CloudWatch console to troubleshoot individual requests.

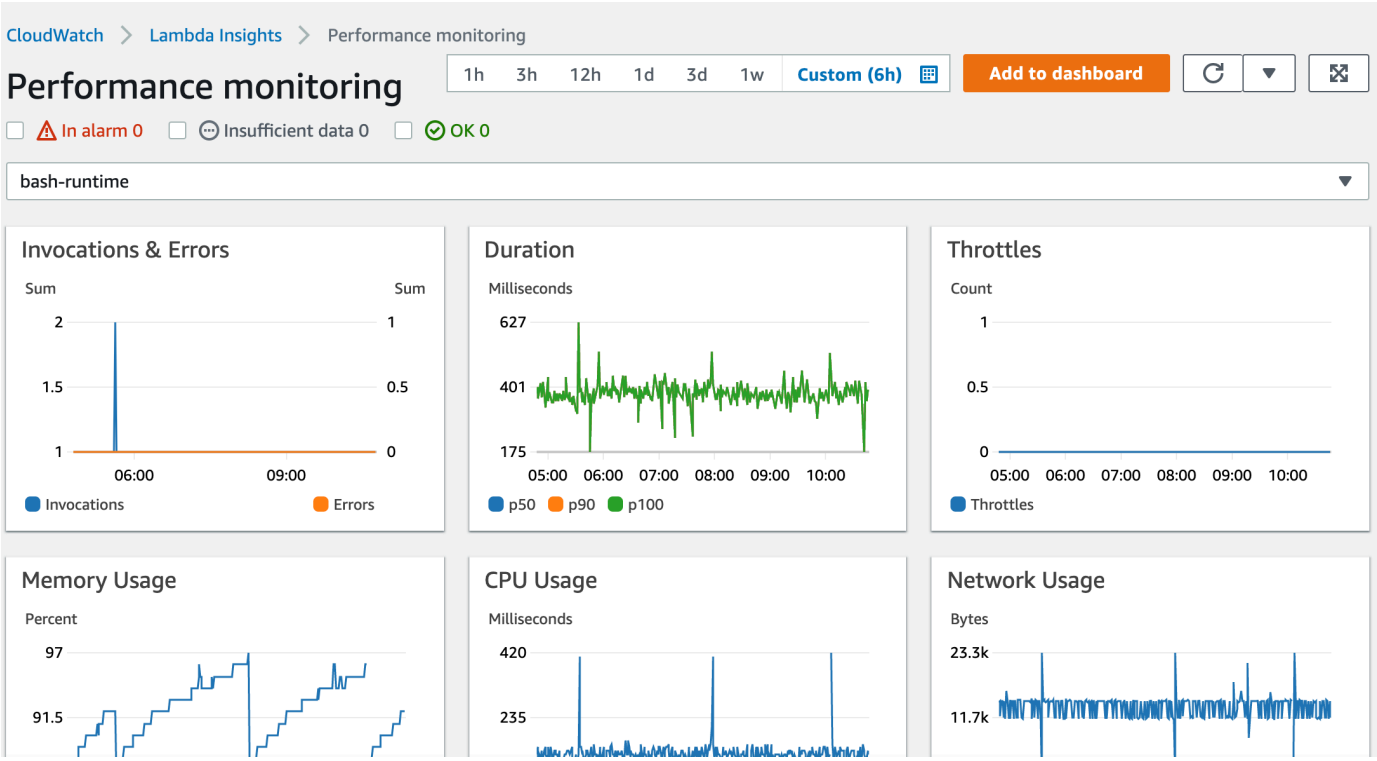
To view the runtime metrics for all functions

1. Open the Multi-function page in the CloudWatch console.
2. Choose from the predefined time ranges, or choose a custom time range.
3. (Optional) Choose **Add to dashboard** to add the widgets to your CloudWatch dashboard.



To view the runtime metrics of a single function

1. Open the Single-function page in the CloudWatch console.
2. Choose from the predefined time ranges, or choose a custom time range.
3. (Optional) Choose **Add to dashboard** to add the widgets to your CloudWatch dashboard.



For more information, see Creating and working with widgets on CloudWatch dashboards.

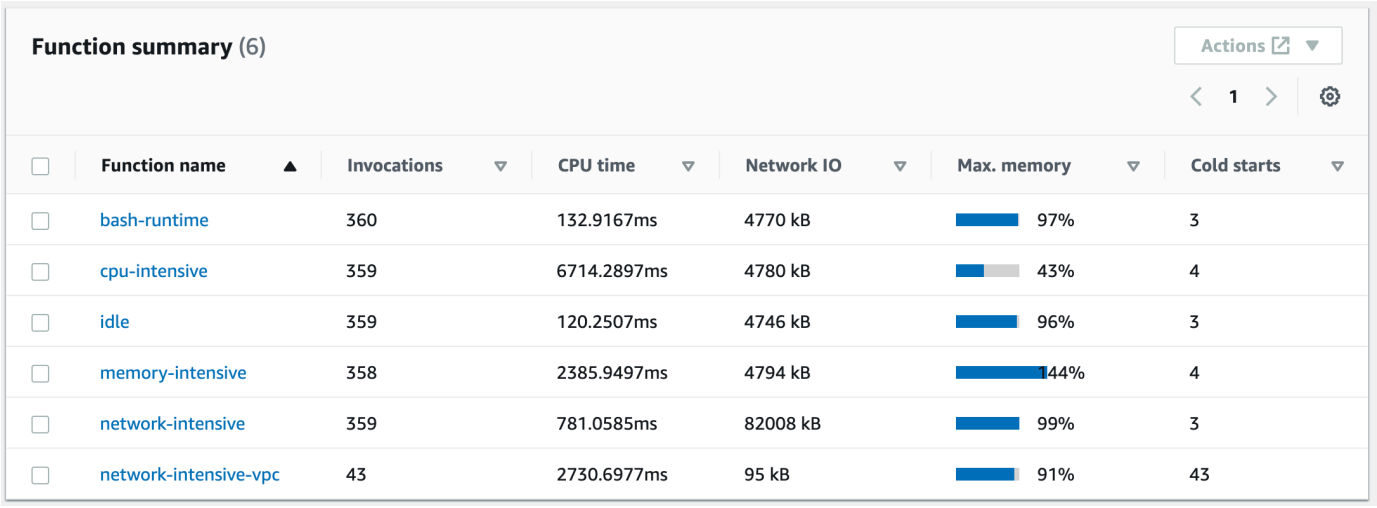
Example workflow to detect function anomalies

You can use the multi-function overview on the Lambda Insights dashboard to identify and detect compute memory anomalies with your function. For example, if the multi-function overview indicates that a function is using a large amount of memory, you can view detailed memory utilization metrics in the **Memory Usage** pane. You can then go to the Metrics dashboard to enable anomaly detection or create an alarm.

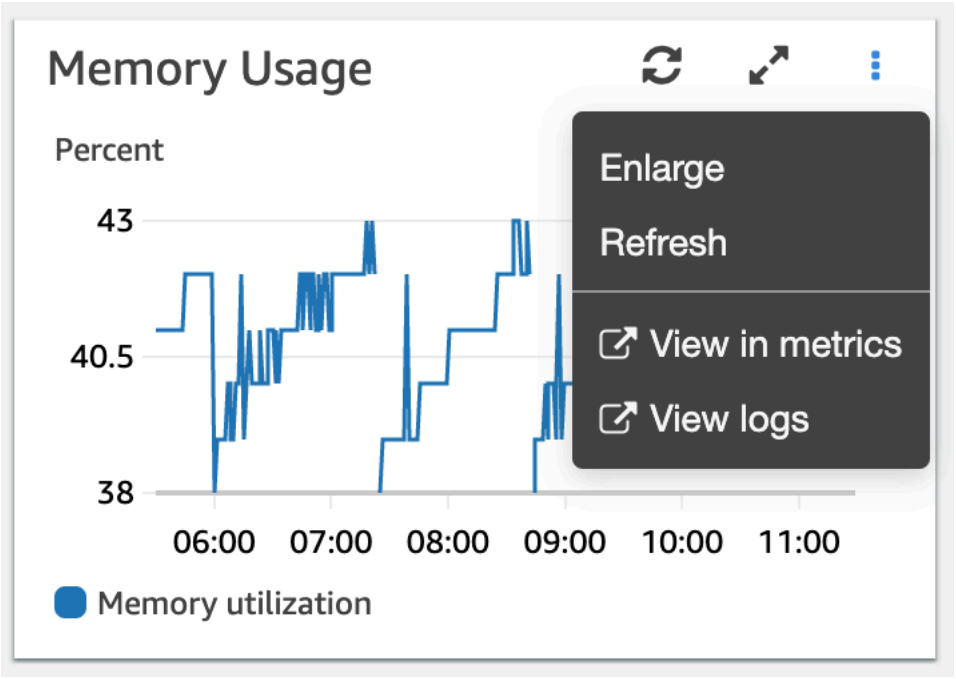
To enable anomaly detection for a function

1. Open the Multi-function page in the CloudWatch console.
2. Under **Function summary**, choose your function's name.

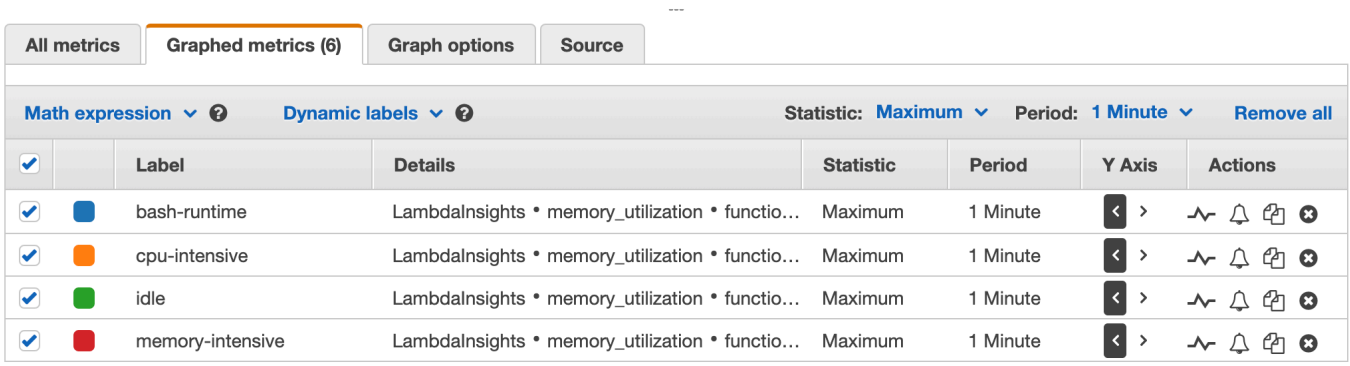
The single-function view opens with the function runtime metrics.



3. On the **Memory Usage** pane, choose the three vertical dots, and then choose **View in metrics** to open the **Metrics** dashboard.



4. On the **Graphed metrics** tab, in the **Actions** column, choose the first icon to enable anomaly detection for the function.



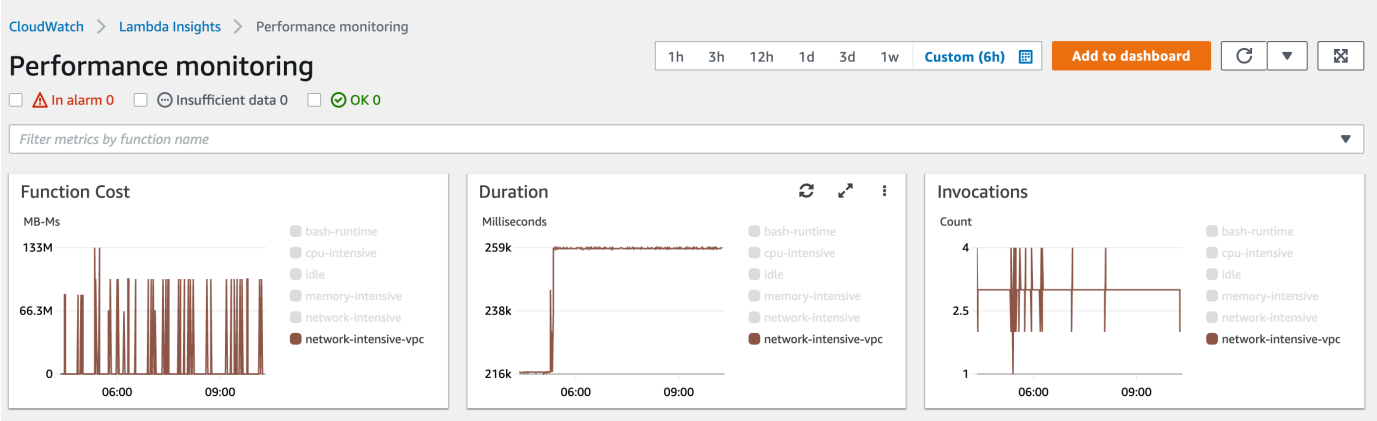
For more information, see Using CloudWatch Anomaly Detection.

Example workflow using queries to troubleshoot a function

You can use the single-function view on the Lambda Insights dashboard to identify the root cause of a spike in function duration. For example, if the multi-function overview indicates a large increase in function duration, you can pause on or choose each function in the **Duration** pane to determine which function is causing the increase. You can then go to the single-function view and review the **Application logs** to determine the root cause.

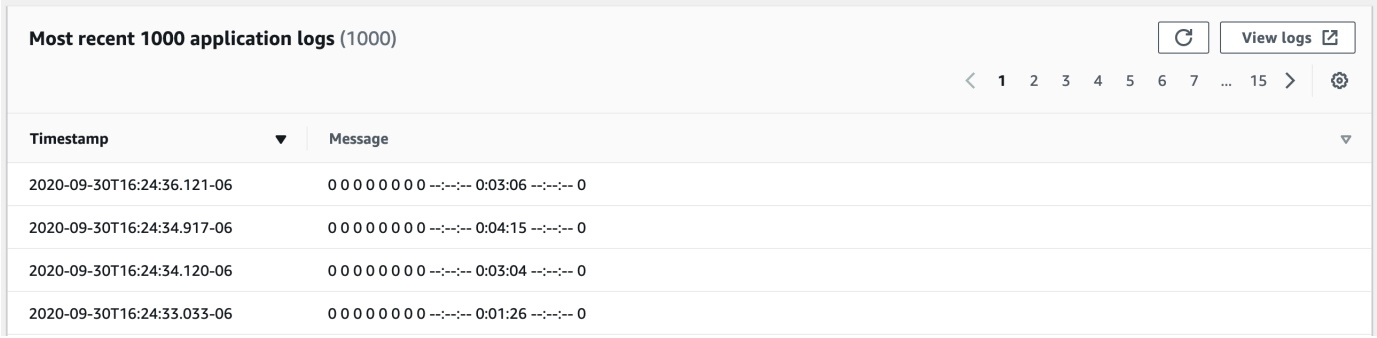
To run queries on a function

1. Open the Multi-function page in the CloudWatch console.
2. In the **Duration** pane, choose your function to filter the duration metrics.

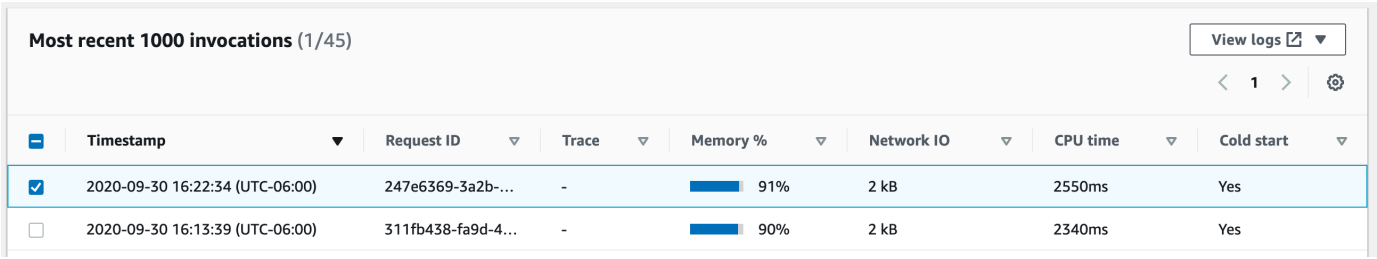


3. Open the Single-function page.
4. Choose the **Filter metrics by function name** dropdown list, and then choose your function.
5. To view the **Most recent 1000 application logs**, choose the **Application logs** tab.

6. Review the **Timestamp** and **Message** to identify the invocation request that you want to troubleshoot.



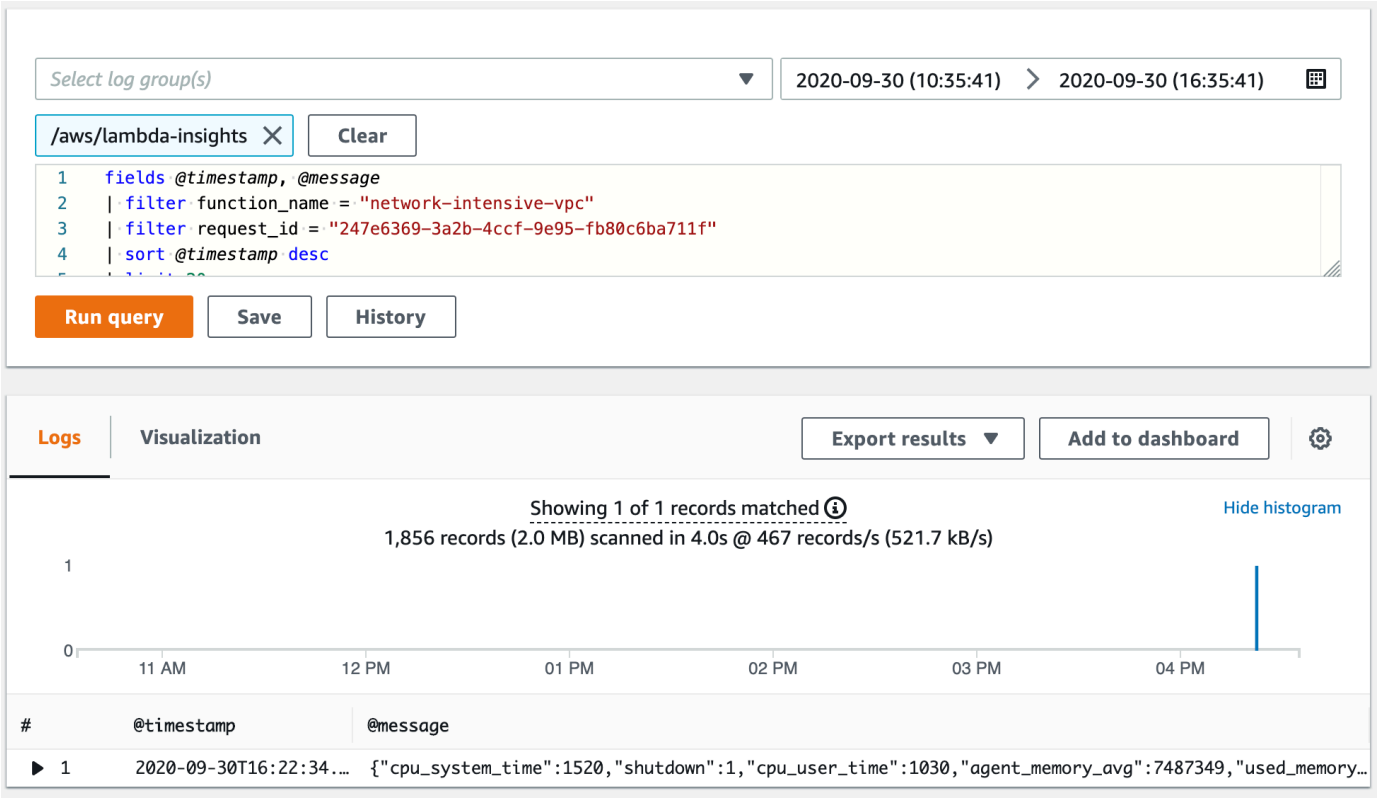
7. To show the **Most recent 1000 invocations**, choose the **Invocations** tab.
8. Select the **Timestamp** or **Message** for the invocation request that you want to troubleshoot.



9. Choose the **View logs** dropdown list, and then choose **View performance logs**.

   An autogenerated query for your function opens in the **Logs Insights** dashboard.

10. Choose **Run query** to generate a **Logs** message for the invocation request.



What's next?

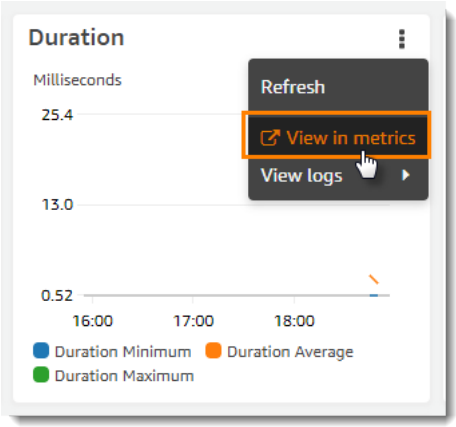
- Learn how to create a CloudWatch Logs dashboard in Create a Dashboard in the *Amazon CloudWatch User Guide*.
- Learn how to add queries to a CloudWatch Logs dashboard in Add Query to Dashboard or Export Query Results in the *Amazon CloudWatch User Guide*.

Monitoring Lambda applications

The **Applications** section of the Lambda console includes a **Monitoring** tab where you can review an Amazon CloudWatch dashboard with aggregate metrics for the resources in your application.

To monitor a Lambda application

1. Open the Lambda console Applications page.
2. Choose **Monitoring**.
3. To see more details about the metrics in any graph, choose **View in metrics** from the drop-down menu.



   The graph appears in a new tab, with the relevant metrics listed below the graph. You can customize your view of this graph, changing the metrics and resources shown, the statistic, the period, and other factors to get a better understanding of the current situation.

By default, the Lambda console shows a basic dashboard. You can customize this page by adding one or more Amazon CloudWatch dashboards to your application template with the AWS::CloudWatch::Dashboard resource type. When your template includes one or more dashboards, the page shows your dashboards instead of the default dashboard. You can switch between dashboards with the drop-down menu on the top right of the page. The following example creates a dashboard with a single widget that graphs the number of invocations of a function named `my-function`.

Example function dashboard template

```
Resources:
  MyDashboard:
```

```
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
      DashboardBody: |
        {
            "widgets": [
                {
                    "type": "metric",
                    "width": 12,
                    "height": 6,
                    "properties": {
                        "metrics": [
                            [
                                "AWS/Lambda",
                                "Invocations",
                                "FunctionName",
                                "my-function",
                                {
                                    "stat": "Sum",
                                    "label": "MyFunction"
                                }
                            ],
                            [
                                {
                                    "expression": "SUM(METRICS())",
                                    "label": "Total Invocations"
                                }
                            ]
                        ],
                        "region": "us-east-1",
                        "title": "Invocations",
                        "view": "timeSeries",
                        "stacked": false
                    }
                }
            ]
        }
```

For more information about authoring CloudWatch dashboards and widgets, see Dashboard body structure and syntax in the *Amazon CloudWatch API Reference*.

Monitor application performance with Amazon CloudWatch Application Signals

Amazon CloudWatch Application Signals is an application performance monitoring (APM) solution that enables developers and operators to monitor the health and performance of their serverless applications built using Lambda. You can enable Application Signals in one-click from the Lambda console, and you don't need to add any instrumentation code or external dependencies to your Lambda function. After you enable Application Signals, you can view all collected metrics and traces in the CloudWatch console. This page describes how to enable and view Application Signals telemetry data for your applications.

Topics

- How Application Signals integrates with Lambda
- Pricing
- Supported runtimes
- Enabling Application Signals in the Lambda console
- Using the Application Signals dashboard

How Application Signals integrates with Lambda

Application Signals automatically instruments your Lambda functions using enhanced AWS Distro for OpenTelemetry (ADOT) libraries, provided via a Lambda layer. Application Signals reads data collected by the layer and generates dashboards with key performance metrics for your applications.

You can attach this layer in one-click by enabling Application Signals in the Lambda console. When you enable Application Signals from the console, Lambda does the following on your behalf:

- Updates your function's execution role to include the `CloudWatchLambdaApplicationSignalsExecutionRolePolicy`. This policy provides write access to AWS X-Ray and CloudWatch log groups used for Application Signals.
- Adds a layer to your function which automatically instruments the function to capture telemetry data such as requests, availability, latency, errors, and faults. To ensure that Application Signals works properly, remove any existing X-Ray SDK instrumentation code from your function. Custom X-Ray SDK instrumentation code can interfere with the layer-provided instrumentation.

- Adds the `AWS_LAMBDA_EXEC_WRAPPER` environment variable to your function, and sets its value to `/opt/otel-instrument`. This environment variable modifies your function's startup behavior to utilize the Application Signals layer, and is required for proper instrumentation. If this environment variable already exists, ensure that it's set to the required value.

Pricing

Using Application Signals for your Lambda functions incurs costs. For pricing information, see Amazon CloudWatch pricing.

Supported runtimes

The Application Signals integration with Lambda works with the following runtimes:

- .NET 8
- Java 11
- Java 17
- Java 21
- Python 3.10
- Python 3.11
- Python 3.12
- Python 3.13
- Node.js 18.x
- Node.js 20.x
- Node.js 22.x

Enabling Application Signals in the Lambda console

You can enable Application Signals on any existing Lambda function using a supported runtime. The following steps describe how to enable Application Signals in one-click in the Lambda console.

To enable Application Signals in the Lambda console

1. Open the Functions page of the Lambda console.

2. Choose your function.
3. Choose the **Configuration** tab.
4. On the left menu, choose **Monitoring and operations tools**.
5. On the **Additional monitoring tools** pane, choose **Edit**.
6. Under **CloudWatch Application Signals and AWS X-Ray**, and under **Application Signals**, choose **Enable**.
7. Choose **Save**.

If this is your first time enabling Application Signals for your function, you must also do a one-time service discovery setup for Application Signals in the CloudWatch console. After you complete this one-time service discovery setup, Application Signals automatically discovers any additional Lambda functions that you enable Application Signals for, across all Regions.

> **Note**
>
> After you invoke your updated function, it can take up to 10 minutes for service data to start appearing in the Application Signals dashboard in the CloudWatch console.

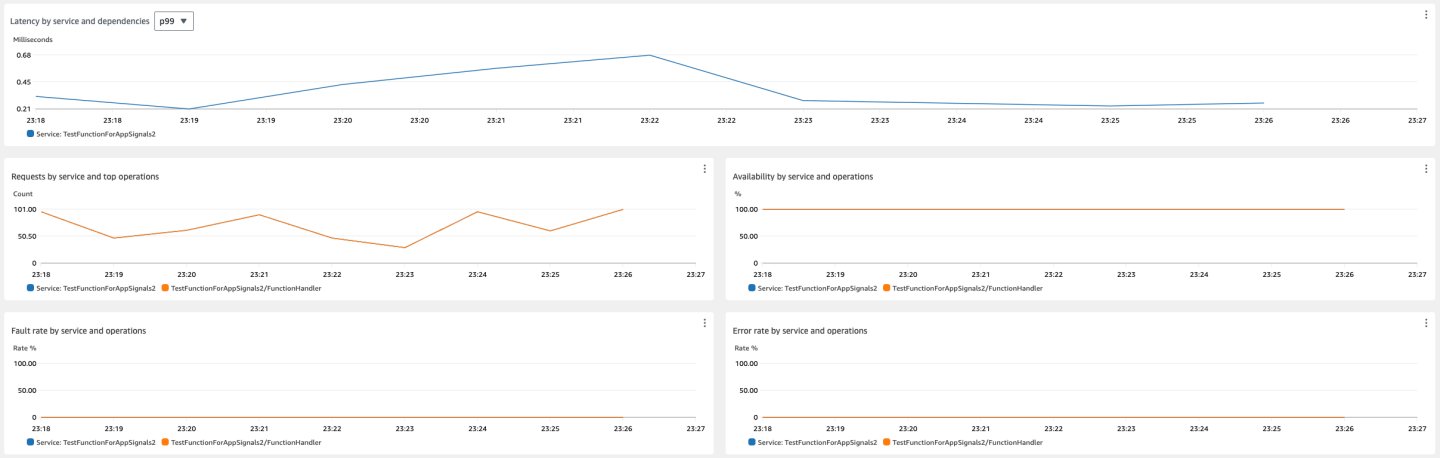
Using the Application Signals dashboard

After you enable Application Signals for your function, you can visualize your application metrics in the CloudWatch console. You can quickly view the associated Application Signals dashboard from the Lambda console with the following steps:

To view the Application Signals dashboard for your function

1. Open the Functions page of the Lambda console.
2. Choose your function.
3. Choose the **Monitor** tab.
4. Choose the **View Application Signals** button. This takes you directly to the Application Signals overview for your service in the CloudWatch console.

For example, the following screenshot shows metrics for latency, number of requests, availability, fault rate, and error rate for a function across a 10 minute time window.



To make the most out of your integration with Application Signals, you can create service-level objectives (SLOs) for your aplication. For example, you can create latency SLOs to ensure your application responds quickly to user requests, and availability SLOs to track uptime. SLOs can help you detect performance degradation or outages before they impact your users. For more information, see Service level objectives (SLOs) in the Amazon CloudWatch User Guide.

Remotely debug Lambda functions with Visual Studio Code

With the remote debugging feature in the AWS Toolkit for Visual Studio Code, you can debug your Lambda functions running directly in the AWS cloud. This is useful when investigating issues that are difficult to replicate locally or diagnose only with logs.

With remote debugging, you can:

- Set breakpoints in your Lambda function code.
- Step through code execution in real-time.
- Inspect variables and state during runtime.
- Debug Lambda functions deployed to AWS, including those in VPCs or with specific IAM permissions.

Supported runtimes

Remote debugging is supported for the following runtimes:

- Python (AL2023)
- Java
- JavaScript/Node.js (AL2023)

> **Note**
>
> Remote debugging is supported for both x86_64 and arm64 architectures.

Security and remote debugging

Remote debugging operates within existing Lambda security boundaries. Users can attach layers to a function using the `UpdateFunctionConfiguration` permission, which already has the ability to access function environment variables and configuration. Remote debugging doesn't extend beyond these existing permissions. Instead, it adds extra security controls through secure tunneling and automatic session management. Additionally, remote debugging is entirely a customer-controlled feature that requires explicit permissions and actions:

- **IoT Secure Tunnel Creation**: The AWS Toolkit must create an IoT secure tunnel, which only occurs with the user's explicit permission using `iot:OpenTunnel`.
- **Debug Layer Attachment and Token Management**: The debugging process maintains security through these controls:
  - The debugging layer must be attached to the Lambda function and this process requires the following permissions: `lambda:UpdateFunctionConfiguration` and `lambda:GetLayerVersion`.
  - A security token (generated via `iot:OpenTunnel`) must be updated in the function environment variable before each debug session, which also requires `lambda:UpdateFunctionConfiguration`.
  - For security, this token is automatically rotated and the debug layer is automatically removed at the end of each debug session and cannot be reused.

> **Note**
>
> Remote debugging is supported for both x86_64 and arm64 architectures.

Prerequisites

Before you begin remote debugging, ensure you have the following:

1. A Lambda function deployed to your AWS account.
2. AWS Toolkit for Visual Studio Code. See Setting up the AWS Toolkit for Visual Studio Code for installation instructions.
3. The version of the AWS Toolkit you have installed is **3.69.0** or later.
4. AWS credentials configured in AWS Toolkit for Visual Studio Code. For more information, see Authentication and access control.

Remotely debug Lambda functions

Follow these steps to start a remote debugging session:

1. Open the AWS Explorer in VS Code by selecting the AWS icon in the left sidebar.
2. Expand the Lambda section to see your functions.

3. Right-click on the function you want to debug.
4. From the context menu, select **Remotely invoke**.
5. In the invoke window that opens, check the box for **Enable debugging**.
6. Click **Invoke** to start the remote debugging session.

> **Note**
>
> Lambda functions have a 250MB combined limit for function code and all attached layers. The remote debugging layer adds approximately 40MB to your function's size.

A remote debugging session ends when you:

- Choose **Remove Debug Setup** from the Remote invoke configuration screen.
- Select the disconnect icon in the VS Code debugging controls.
- Select the handler file in the VS Code editor.

> **Note**
>
> The debug layer is automatically removed after 60 seconds of inactivity following your last invoke.

Disable remote debugging

There are three ways to disable this feature:

- **Deny Function Updates**: Set `lambda:UpdateFunctionConfiguration` to deny.
- **Restrict IoT Permissions**: Deny IoT-related permissions
- **Block Debug Layers**: Deny `lambda:GetLayerVersion` for the following ARNs:
  - `arn:aws:lambda:*:*:layer:LDKLayerX86:*`
  - `arn:aws:lambda:*:*:layer:LDKLayerArm64:*`

> **Note**
>
> Disabling this feature prevents the debugging layer from being added during function configuration updates.

Additional information

For more information on using Lambda in VS Code, refer to Developing Lambda functions locally with VS Code.

For detailed instructions on troubleshooting, advanced use cases, and region availability, see Remote debugging Lambda functions in the AWS Toolkit for Visual Studio Code User Guide.

Managing Lambda dependencies with layers

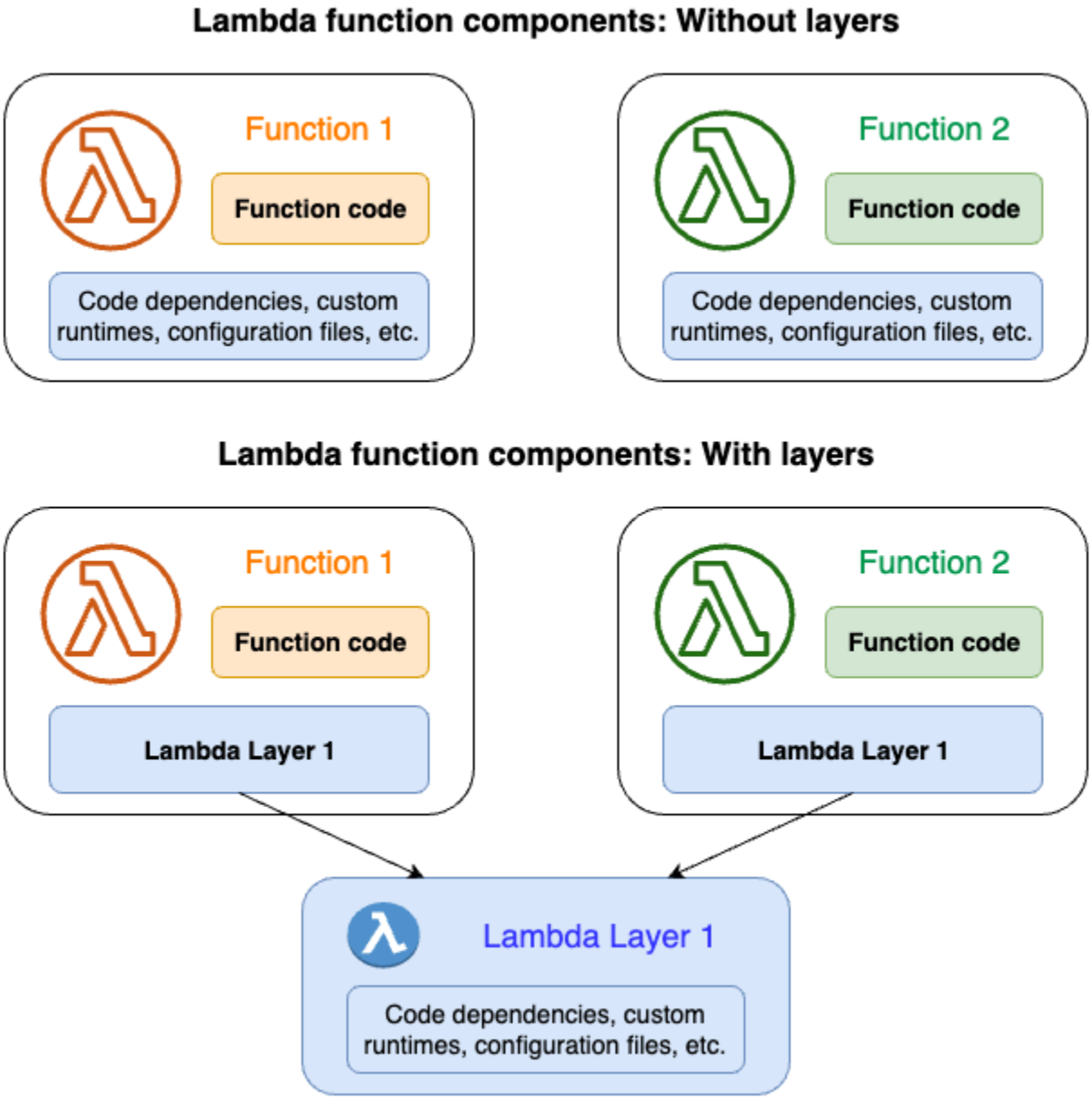
A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.

There are multiple reasons why you might consider using layers:

- **To reduce the size of your deployment packages.** Instead of including all of your function dependencies along with your function code in your deployment package, put them in a layer. This keeps deployment packages small and organized.
- **To separate core function logic from dependencies.** With layers, you can update your function dependencies independent of your function code, and vice versa. This promotes separation of concerns and helps you focus on your function logic.
- **To share dependencies across multiple functions.** After you create a layer, you can apply it to any number of functions in your account. Without layers, you need to include the same dependencies in each individual deployment package.
- **To use the Lambda console code editor.** The code editor is a useful tool for testing minor function code updates quickly. However, you can't use the editor if your deployment package size is too large. Using layers reduces your package size and can unlock usage of the code editor.
- **To lock an embedded SDK version.**The embedded SDKs may change without notice as AWS releases new services and features. You can lock a version of the SDK by creating a Lambda layer with the specific version needed. The function then always uses the version in the layer, even if the version embedded in the service changes.

If you're working with Lambda functions in Go or Rust, we recommend against using layers. For Go and Rust functions, you provide your function code as an executable, which includes your compiled function code along with all of its dependencies. Putting your dependencies in a layer forces your function to manually load additional assemblies during the initialization phase, which can increase cold start times. For optimal performance for Go and Rust functions, include your dependencies along with your deployment package.

The following diagram illustrates the high-level architectural differences between two functions that share dependencies. One uses Lambda layers, and the other does not.



When you add a layer to a function, Lambda extracts the layer contents into the `/opt` directory in your function's execution environment. All natively supported Lambda runtimes include paths to specific directories within the `/opt` directory. This gives your function access to your layer content. For more information about these specific paths and how to properly package your layers, see the section called "Packaging layers".

You can include up to five layers per function. Also, you can use layers only with Lambda functions deployed as a .zip file archive. For functions defined as a container image, package your preferred

runtime and all code dependencies when you create the container image. For more information, see Working with Lambda layers and extensions in container images on the AWS Compute Blog.

Topics

- How to use layers
- Layers and layer versions
- Packaging your layer content
- Creating and deleting layers in Lambda
- Adding layers to functions
- Using AWS CloudFormation with layers
- Using AWS SAM with layers

How to use layers

To create a layer, package your dependencies into a .zip file, similar to how you create a normal deployment package. More specifically, the general process of creating and using layers involves these three steps:

- **First, package your layer content.** This means creating a .zip file archive. For more information, see the section called "Packaging layers".
- **Next, create the layer in Lambda.** For more information, see the section called "Creating and deleting layers".
- **Add the layer to your function(s).** For more information, see the section called "Adding layers".

Layers and layer versions

A layer version is an immutable snapshot of a specific version of a layer. When you create a new layer, Lambda creates a new layer version with a version number of 1. Each time you publish an update to the layer, Lambda increments the version number and creates a new layer version.

Every layer version is identified by a unique Amazon Resource Name (ARN). When adding a layer to the function, you must specify the exact layer version you want to use (for example, `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`).

Packaging your layer content

A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.

This section explains how to properly package your layer content. For more conceptual information about layers and why you might consider using them, see *Lambda layers*.

The first step to creating a layer is to bundle all of your layer content into a .zip file archive. Because Lambda functions run on Amazon Linux, your layer content must be able to compile and build in a Linux environment.

To ensure that your layer content works properly in a Linux environment, we recommend creating your layer content using a tool like Docker.

Topics

- Layer paths for each Lambda runtime

Layer paths for each Lambda runtime

When you add a layer to a function, Lambda loads the layer content into the `/opt` directory of that execution environment. For each Lambda runtime, the `PATH` variable already includes specific folder paths within the `/opt` directory. To ensure that Lambda picks up your layer content, your layer .zip file should have its dependencies in one of the following folder paths:

Runtime	Path
Node.js	`nodejs/node_modules`
	`nodejs/node18/node_modules` (`NODE_PATH`)
	`nodejs/node20/node_modules` (`NODE_PATH`)
	`nodejs/node22/node_modules` (`NODE_PATH`)
Python	`python`
	`python/lib/` *`python3.x`* `/site-packages` (site directories)

Runtime	Path
Java	`java/lib` (`CLASSPATH`)
Ruby	`ruby/gems/3.4.0` (`GEM_PATH`)
	`ruby/lib` (`RUBYLIB`)
All runtimes	`bin` (`PATH`)
	`lib` (`LD_LIBRARY_PATH`)

The following examples show how you can structure the folders in your layer .zip archive.

Node.js

Example file structure for the AWS X-Ray SDK for Node.js

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example

```
python/              # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

Ruby

Example file structure for the JSON gem

```
json.zip
# ruby/gems/3.4.0/
               | build_info
               | cache
               | doc
```

```
                | extensions
                | gems
                | # json-2.1.0
                # specifications
                  # json-2.1.0.gemspec
```

Java

Example file structure for the Jackson JAR file

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example file structure for the jq library

```
jq.zip
# bin/jq
```

For language-specific instructions on packaging, creating, and adding a layer, refer to the following pages:

- **Node.js** – the section called “Layers”
- **Python** – the section called “Layers”
- **Ruby** – the section called “Layers”
- **Java** – the section called “Layers”

We recommend **against** using layers to manage dependencies for Lambda functions written in Go and Rust. This is because Lambda functions written in these languages compile into a single executable, which you provide to Lambda when you deploy your function. This executable contains your compiled function code, along with all of its dependencies. Using layers not only complicates this process, but also leads to increased cold start times because your functions need to manually load extra assemblies into memory during the init phase.

To use external dependencies with Go and Rust Lambda functions, include them directly in your deployment package.

Creating and deleting layers in Lambda

A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.

This section explains how to create and delete layers in Lambda. For more conceptual information about layers and why you might consider using them, see *Lambda layers*.

After you've packaged your layer content, the next step is to create the layer in Lambda. This section demonstrates how to create and delete layers using the Lambda console or the Lambda API only. To create a layer using AWS CloudFormation, see the section called "Layers with CloudFormation". To create a layer using the AWS Serverless Application Model (AWS SAM), see the section called "Layers with AWS SAM".

Topics

- Creating a layer
- Deleting a layer version

Creating a layer

To create a layer, you can either upload the .zip file archive from your local machine or from Amazon Simple Storage Service (Amazon S3). Lambda extracts the layer contents into the `/opt` directory when setting up the execution environment for the function.

Layers can have one or more layer versions. When you create a layer, Lambda sets the layer version to version 1. You can change the permissions on an existing layer version at any time. However, to update the code or make other configuration changes, you must create a new version of the layer.

To create a layer (console)

1. Open the Layers page of the Lambda console.
2. Choose **Create layer**.
3. Under **Layer configuration**, for **Name**, enter a name for your layer.
4. (Optional) For **Description**, enter a description for your layer.
5. To upload your layer code, do one of the following:
   - To upload a .zip file from your computer, choose **Upload a .zip file**. Then, choose **Upload** to select your local .zip file.

- To upload a file from Amazon S3, choose **Upload a file from Amazon S3**. Then, for **Amazon S3 link URL**, enter a link to the file.

6. (Optional) For **Compatible architectures**, choose one value or both values. For more information, see the section called "Instruction sets (ARM/x86)".
7. (Optional) For **Compatible runtimes**, choose the runtimes that your layer is compatible with.
8. (Optional) For **License**, enter any necessary license information.
9. Choose **Create**.

Alternatively, you can run the publish-layer-version AWS Command Line Interface (CLI) command. Example:

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip --
compatible-runtimes python3.14
```

Each time that you run `publish-layer-version`, Lambda creates a new version of the layer.

Deleting a layer version

To delete a layer version, use the DeleteLayerVersion API operation. For example, run the delete-layer-version AWS CLI command with the layer name and layer version specified.

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

When you delete a layer version, you can no longer configure a Lambda function to use it. However, any function that already uses the version continues to have access to it. Also, Lambda never reuses version numbers for a layer name.

When calculating quotas, deleting a layer version means it's no longer counted as part of the default 75 GB quota for storage of functions and layers. However, for functions that consume a deleted layer version, the layer content still counts towards the function's deployment package size quota (i.e. 250MB for .zip file archives).

Adding layers to functions

A Lambda layer is a .zip file archive that contains supplementary code or data. Layers usually contain library dependencies, a custom runtime, or configuration files.

This section explains how to add a layer to a Lambda function. For more conceptual information about layers and why you might consider using them, see *Lambda layers*.

Before you can configure a Lambda function to use a layer, you must:

- Package your layer content
- Create a layer in Lambda
- Make sure that you have permission to call the GetLayerVersion API on the layer version. For functions in your AWS account, you must have this permission in your user policy. To use a layer in another account, the owner of that account must grant your account permission in a resource-based policy. For examples, see the section called "Layer access for other accounts".

You can add up to five layers to a Lambda function. The total unzipped size of the function and all layers cannot exceed the unzipped deployment package size quota of 250 MB. For more information, see *Lambda quotas*.

Your functions can continue to use any layer version that you've already added, even after that layer version has been deleted, or after your permission to access the layer is revoked. However, you cannot create a new function that uses a deleted layer version.

To add a layer to a function

1. Open the Functions page of the Lambda console.
2. Choose the function.
3. Scroll down to the **Layers** section, and then choose **Add a layer**.
4. Under **Choose a layer**, choose a layer source:
    a. **AWS layers**: Choose from the list of AWS-managed extensions.
    b. **Custom layers**: Choose a layer created in your AWS account.
    c. **Specify an ARN**: To use a layer from a different AWS account, such as a third-party extension, enter the Amazon Resource Name (ARN).
5. Choose **Add**.

The order in which you add the layers is the order in which Lambda merges the layer content into the execution environment. You can change the layer merge order using the console.

To update layer merge order for your function (console)

1. Open the Functions page of the Lambda console.
2. Choose the function to configure.
3. Under **Layers**, choose **Edit**
4. Choose one of the layers.
5. Choose **Merge earlier** or **Merge later** to adjust the order of the layers.
6. Choose **Save**.

Layers are versioned. The content of each layer version is immutable. The owner of a layer can release new layer versions to provide updated content. You can use the console to update the layer version attached to your functions.

To update layer versions for your function (console)

1. Open the Layers page of the Lambda console.
2. Choose the layer you want to update the version for.
3. Choose the **Functions using this version** tab.
4. Choose the functions you want to modify, then choose **Edit**.
5. For **Layer version**, choose the layer version to change to.
6. Choose **Update functions**.

You cannot update function layer versions across AWS accounts.

Finding layer information

To find layers in your account that are compatible with your function's runtime, use the ListLayers API. For example, you can use the following list-layers AWS Command Line Interface (CLI) command:

```
aws lambda list-layers --compatible-runtime python3.14
```

You should see output similar to the following:

```
{
    "Layers": [
        {
            "LayerName": "my-layer",
            "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
            "LatestMatchingVersion": {
                "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
                "Version": 2,
                "Description": "My layer",
                "CreatedDate": "2025-04-15T00:37:46.592+0000",
                "CompatibleRuntimes": [
                    "python3.14"
                ]
            }
        }
    ]
}
```

To list all layers in your account, omit the `--compatible-runtime` option. The response details show the latest version of each layer.

You can also get the latest version of a layer using the ListLayerVersions API. For example, you can use the following `list-layer-versions` CLI command:

```
aws lambda list-layer-versions --layer-name my-layer
```

You should see output similar to the following:

```
{
    "LayerVersions": [
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
            "Version": 2,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:37:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        },
        {
```

```
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:1",
            "Version": 1,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:27:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        }
    ]
}
```

Using AWS CloudFormation with layers

You can use CloudFormation to create a layer and associate the layer with your Lambda function. The following example template creates a layer named `my-lambda-layer` and attaches the layer to the Lambda function using the **Layers** property.

In this example, the template specifies the Amazon Resource Name (ARN) of an existing IAM execution role. You can also create a new execution role in the template using the CloudFormation AWS::IAM::Role resource.

Your function doesn't need any special permissions to use layers.

```
---
Description: CloudFormation Template for Lambda Function with Lambda Layer
Resources:
  MyLambdaLayer:
    Type: AWS::Lambda::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      Content:
        S3Bucket: amzn-s3-demo-bucket
        S3Key: my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-lambda-function
      Runtime: python3.9
      Handler: index.handler
      Timeout: 10
      Role: arn:aws:iam::111122223333:role/my_lambda_role
      Layers:
        - !Ref MyLambdaLayer
```

Using AWS SAM with layers

You can use the AWS Serverless Application Model (AWS SAM) to automate the creation of layers in your application. The `AWS::Serverless::LayerVersion` resource type creates a layer version that you can reference from your Lambda function configuration.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS SAM Template for Lambda Function with Lambda Layer

Resources:
  MyLambdaLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      ContentUri: s3://amzn-s3-demo-bucket/my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: MyLambdaFunction
      Runtime: python3.9
      Handler: app.handler
      CodeUri: s3://amzn-s3-demo-bucket/my-function
      Layers:
        - !Ref MyLambdaLayer
```

Augment Lambda functions using Lambda extensions

You can use Lambda extensions to augment your Lambda functions. For example, use Lambda extensions to integrate functions with your preferred monitoring, observability, security, and governance tools. You can choose from a broad set of tools that AWS Lambda Partners provides, or you can create your own Lambda extensions.

Lambda supports external and internal extensions. An external extension runs as an independent process in the execution environment and continues to run after the function invocation is fully processed. Because extensions run as separate processes, you can write them in a different language than the function. All Lambda runtimes support extensions.

An internal extension runs as part of the runtime process. Your function accesses internal extensions by using wrapper scripts or in-process mechanisms such as `JAVA_TOOL_OPTIONS`. For more information, see Modifying the runtime environment.

You can add extensions to a function using the Lambda console, the AWS Command Line Interface (AWS CLI), or infrastructure as code (IaC) services and tools such as CloudFormation, AWS Serverless Application Model (AWS SAM), and Terraform.

You are charged for the execution time that the extension consumes (in 1 ms increments). There is no cost to install your own extensions. For more pricing information for extensions, see AWS Lambda Pricing. For pricing information for partner extensions, see those partners' websites. See the section called “Extensions partners” for a list of official partner extensions.

Topics

- Execution environment
- Impact on performance and resources
- Permissions
- Configuring Lambda extensions
- AWS Lambda extensions partners
- Using the Lambda Extensions API to create extensions
- Accessing real-time telemetry data for extensions using the Telemetry API

Execution environment

Lambda invokes your function in an execution environment, which provides a secure and isolated runtime environment. The execution environment manages the resources required to run your function and provides lifecycle support for the function's runtime and extensions.

The lifecycle of the execution environment includes the following phases:

- **`Init`**: In this phase, Lambda creates or unfreezes an execution environment with the configured resources, downloads the code for the function and all layers, initializes any extensions, initializes the runtime, and then runs the function's initialization code (the code outside the main handler). The `Init` phase happens either during the first invocation, or in advance of function invocations if you have enabled provisioned concurrency.

  The `Init` phase is split into three sub-phases: `Extension init`, `Runtime init`, and `Function init`. These sub-phases ensure that all extensions and the runtime complete their setup tasks before the function code runs.

  When Lambda SnapStart is activated, the `Init` phase happens when you publish a function version. Lambda saves a snapshot of the memory and disk state of the initialized execution environment, persists the encrypted snapshot, and caches it for low-latency access. If you have a before-checkpoint runtime hook, then the code runs at the end of `Init` phase.
- **`Restore`** (SnapStart only): When you first invoke a SnapStart function and as the function scales up, Lambda resumes new execution environments from the persisted snapshot instead of initializing the function from scratch. If you have an after-restore runtime hook, the code runs at the end of the `Restore` phase. You are charged for the duration of after-restore runtime hooks. The runtime must load and after-restore runtime hooks must complete within the timeout limit (10 seconds). Otherwise, you'll get a SnapStartTimeoutException. When the `Restore` phase completes, Lambda invokes the function handler (the Invoke phase).
- **`Invoke`**: In this phase, Lambda invokes the function handler. After the function runs to completion, Lambda prepares to handle another function invocation.
- **`Shutdown`**: This phase is triggered if the Lambda function does not receive any invocations for a period of time. In the `Shutdown` phase, Lambda shuts down the runtime, alerts the extensions to let them stop cleanly, and then removes the environment. Lambda sends a `Shutdown` event to each extension, which tells the extension that the environment is about to be shut down.

During the `Init` phase, Lambda extracts layers containing extensions into the `/opt` directory in the execution environment. Lambda looks for extensions in the `/opt/extensions/` directory, interprets each file as an executable bootstrap for launching the extension, and starts all extensions in parallel.

Impact on performance and resources

The size of your function's extensions counts towards the deployment package size limit. For a .zip file archive, the total unzipped size of the function and all extensions cannot exceed the unzipped deployment package size limit of 250 MB.

Extensions can impact the performance of your function because they share function resources such as CPU, memory, and storage. For example, if an extension performs compute-intensive operations, you may see your function's execution duration increase.

Each extension must complete its initialization before Lambda invokes the function. Therefore, an extension that consumes significant initialization time can increase the latency of the function invocation.

To measure the extra time that the extension takes after the function execution, you can use the `PostRuntimeExtensionsDuration` function metric. To measure the increase in memory used, you can use the `MaxMemoryUsed` metric. To understand the impact of a specific extension, you can run different versions of your functions side by side.

> **Note**
>
> `MaxMemoryUsed` metric is one of the Metrics collected by Lambda Insights and not a Lambda native metric.

Permissions

Extensions have access to the same resources as functions. Because extensions are executed within the same environment as the function, permissions are shared between the function and the extension.

For a .zip file archive, you can create an CloudFormation template to simplify the task of attaching the same extension configuration—including AWS Identity and Access Management (IAM) permissions—to multiple functions.

Configuring Lambda extensions

Configuring extensions (.zip file archive)

You can add an extension to your function as a Lambda layer. Using layers enables you to share extensions across your organization or to the entire community of Lambda developers. You can add one or more extensions to a layer. You can register up to 10 extensions for a function.

You add the extension to your function using the same method as you would for any layer. For more information, see *Lambda layers*.

Add an extension to your function (console)

1. Open the Functions page of the Lambda console.
2. Choose a function.
3. Choose the **Code** tab if it is not already selected.
4. Under **Layers**, choose **Edit**.
5. For **Choose a layer**, choose **Specify an ARN**.
6. For **Specify an ARN**, enter the Amazon Resource Name (ARN) of an extension layer.
7. Choose **Add**.

Using extensions in container images

You can add extensions to your container image. The ENTRYPOINT container image setting specifies the main process for the function. Configure the ENTRYPOINT setting in the Dockerfile, or as an override in the function configuration.

You can run multiple processes within a container. Lambda manages the lifecycle of the main process and any additional processes. Lambda uses the Extensions API to manage the extension lifecycle.

Example: Adding an external extension

An external extension runs in a separate process from the Lambda function. Lambda starts a process for each extension in the `/opt/extensions/` directory. Lambda uses the Extensions API to manage the extension lifecycle. After the function has run to completion, Lambda sends a `Shutdown` event to each external extension.

Example of adding an external extension to a Python base image

```
FROM public.ecr.aws/lambda/python:3.11

# Copy and install the app
COPY /app /app
WORKDIR /app
RUN pip install -r requirements.txt

# Add an extension from the local directory into /opt/extensions
ADD my-extension.zip /opt/extensions
CMD python ./my-function.py
```

Next steps

To learn more about extensions, we recommend the following resources:

- For a basic working example, see Building Extensions for AWS Lambda on the AWS Compute Blog.
- For information about extensions that AWS Lambda Partners provides, see Introducing AWS Lambda Extensions on the AWS Compute Blog.
- To view available example extensions and wrapper scripts, see AWS Lambda Extensions on the AWS Samples GitHub repository.

AWS Lambda extensions partners

AWS Lambda has partnered with several third party entities to provide extensions to integrate with your Lambda functions. The following list details third party extensions that are ready for you to use at any time.

- AppDynamics – Provides automatic instrumentation of Node.js or Python Lambda functions, providing visibility and alerting on function performance.
- Axiom – Provides dashboards for monitoring Lambda function performance and aggregate system-level metrics.
- Datadog – Provides comprehensive, real-time visibility to your serverless applications through the use of metrics, traces, and logs.
- Dynatrace – Provides visibility into traces and metrics, and leverages AI for automated error detection and root cause analysis across the entire application stack.
- Elastic – Provides Application Performance Monitoring (APM) to identify and resolve root cause issues using correlated traces, metrics, and logs.
- Epsagon – Listens to invocation events, stores traces, and sends them in parallel to Lambda function executions.
- Fastly– Protects your Lambda functions from suspicious activity, such as injection-style attacks, account takeover via credential stuffing, malicious bots, and API abuse.
- HashiCorp Vault – Manages secrets and makes them available for developers to use within function code, without making functions Vault aware.
- Honeycomb – Observability tool for debugging your app stack.
- Lumigo – Profiles Lambda function invocations and collects metrics for troubleshooting issues in serverless and microservice environments.
- New Relic – Runs alongside Lambda functions, automatically collecting, enhancing, and transporting telemetry to New Relic's unified observability platform.
- Sedai – An autonomous cloud management platform, powered by AI/ML, that delivers continuous optimization for cloud operations teams to maximize cloud cost savings, performance, and availability at scale.
- Sentry – Diagnose, fix, and optimize performance of Lambda functions.
- Site24x7 – Achieve real-time observability into your Lambda environments
- Splunk – Collects high-resolution, low-latency metrics for efficient and effective monitoring of Lambda functions.

- Sumo Logic – Provides visibility into the health and performance of serverless applications.
- Salt Security – Simplifies API posture governance and API security for Lambda functions through automated setup and support for diverse runtimes.

AWS managed extensions

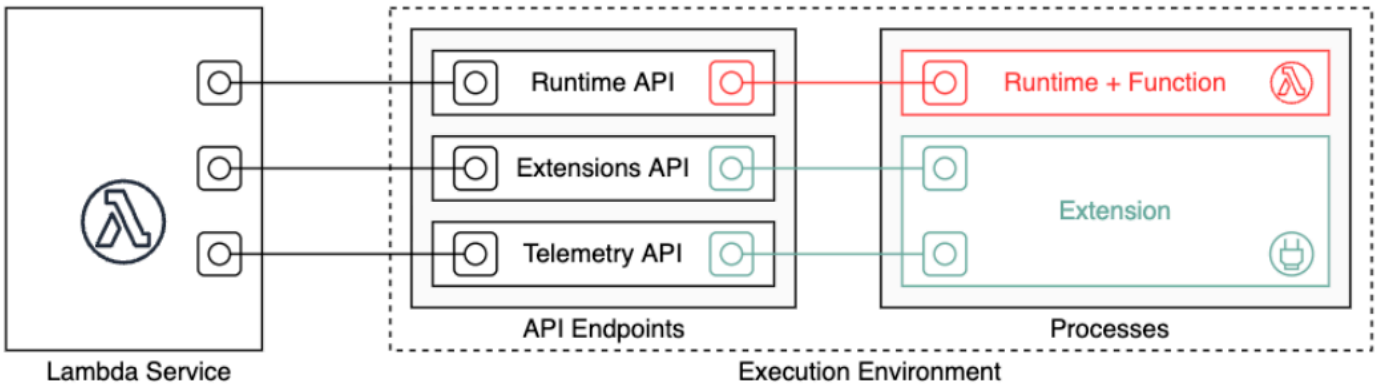
AWS provides its own managed extensions, including:

- AWS AppConfig – Use feature flags and dynamic data to update your Lambda functions. You can also use this extension to update other dynamic configuration, such as ops throttling and tuning.
- Amazon CodeGuru Profiler – Improves application performance and reduces cost by pinpointing an application's most expensive line of code and providing recommendations for improving code.
- CloudWatch Lambda Insights – Monitor, troubleshoot, and optimize the performance of your Lambda functions through automated dashboards.
- AWS Distro for OpenTelemetry (ADOT) – Enables functions to send trace data to AWS monitoring services such as AWS X-Ray, and to destinations that support OpenTelemetry such as Honeycomb and Lightstep.
- AWS Parameters and Secrets – Securely retrieve parameters from AWS Systems Manager Parameter Store and secrets from AWS Secrets Manager in Lambda functions.

For additional extensions samples and demo projects, see AWS Lambda Extensions.

Using the Lambda Extensions API to create extensions

Lambda function authors use extensions to integrate Lambda with their preferred tools for monitoring, observability, security, and governance. Function authors can use extensions from AWS, AWS Partners, and open-source projects. For more information on using extensions, see Introducing AWS Lambda Extensions on the AWS Compute Blog. This section describes how to use the Lambda Extensions API, the Lambda execution environment lifecycle, and the Lambda Extensions API reference.



As an extension author, you can use the Lambda Extensions API to integrate deeply into the Lambda execution environment. Your extension can register for function and execution environment lifecycle events. In response to these events, you can start new processes, run logic, and control and participate in all phases of the Lambda lifecycle: initialization, invocation, and shutdown. In addition, you can use the Runtime Logs API to receive a stream of logs.

An extension runs as an independent process in the execution environment and can continue to run after the function invocation is fully processed. Because extensions run as processes, you can write them in a different language than the function. We recommend that you implement extensions using a compiled language. In this case, the extension is a self-contained binary that is compatible with supported runtimes. All Lambda runtimes support extensions. If you use a non-compiled language, ensure that you include a compatible runtime in the extension.

Lambda also supports *internal extensions*. An internal extension runs as a separate thread in the runtime process. The runtime starts and stops the internal extension. An alternative way to integrate with the Lambda environment is to use language-specific environment variables and wrapper scripts. You can use these to configure the runtime environment and modify the startup behavior of the runtime process.

You can add extensions to a function in two ways. For a function deployed as a .zip file archive, you deploy your extension as a layer. For a function defined as a container image, you add the extensions to your container image.

> **Note**
>
> For example extensions and wrapper scripts, see AWS Lambda Extensions on the AWS Samples GitHub repository.

Topics

- Lambda execution environment lifecycle
- Extensions API reference

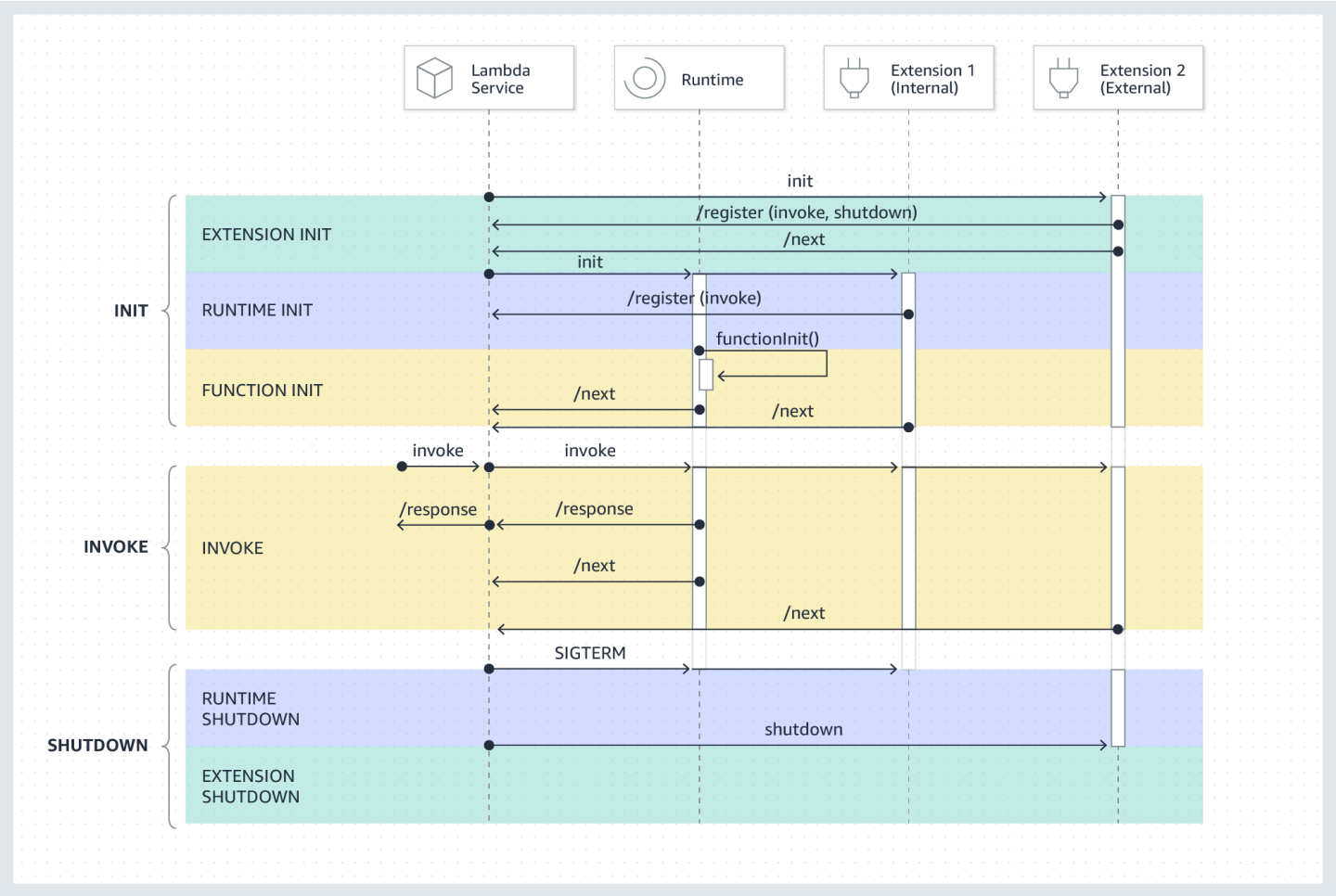
Lambda execution environment lifecycle

The lifecycle of the execution environment includes the following phases:

- **`Init`**: In this phase, Lambda creates or unfreezes an execution environment with the configured resources, downloads the code for the function and all layers, initializes any extensions, initializes the runtime, and then runs the function's initialization code (the code outside the main handler). The `Init` phase happens either during the first invocation, or in advance of function invocations if you have enabled provisioned concurrency.

  The `Init` phase is split into three sub-phases: `Extension init`, `Runtime init`, and `Function init`. These sub-phases ensure that all extensions and the runtime complete their setup tasks before the function code runs.
- **`Invoke`**: In this phase, Lambda invokes the function handler. After the function runs to completion, Lambda prepares to handle another function invocation.
- **`Shutdown`**: This phase is triggered if the Lambda function does not receive any invocations for a period of time. In the `Shutdown` phase, Lambda shuts down the runtime, alerts the extensions to let them stop cleanly, and then removes the environment. Lambda sends a `Shutdown` event to each extension, which tells the extension that the environment is about to be shut down.

Each phase starts with an event from Lambda to the runtime and to all registered extensions. The runtime and each extension signal completion by sending a `Next` API request. Lambda freezes the execution environment when each process has completed and there are no pending events.



Topics

- Init phase
- Invoke phase
- Shutdown phase
- Permissions and configuration
- Failure handling
- Troubleshooting extensions

Init phase

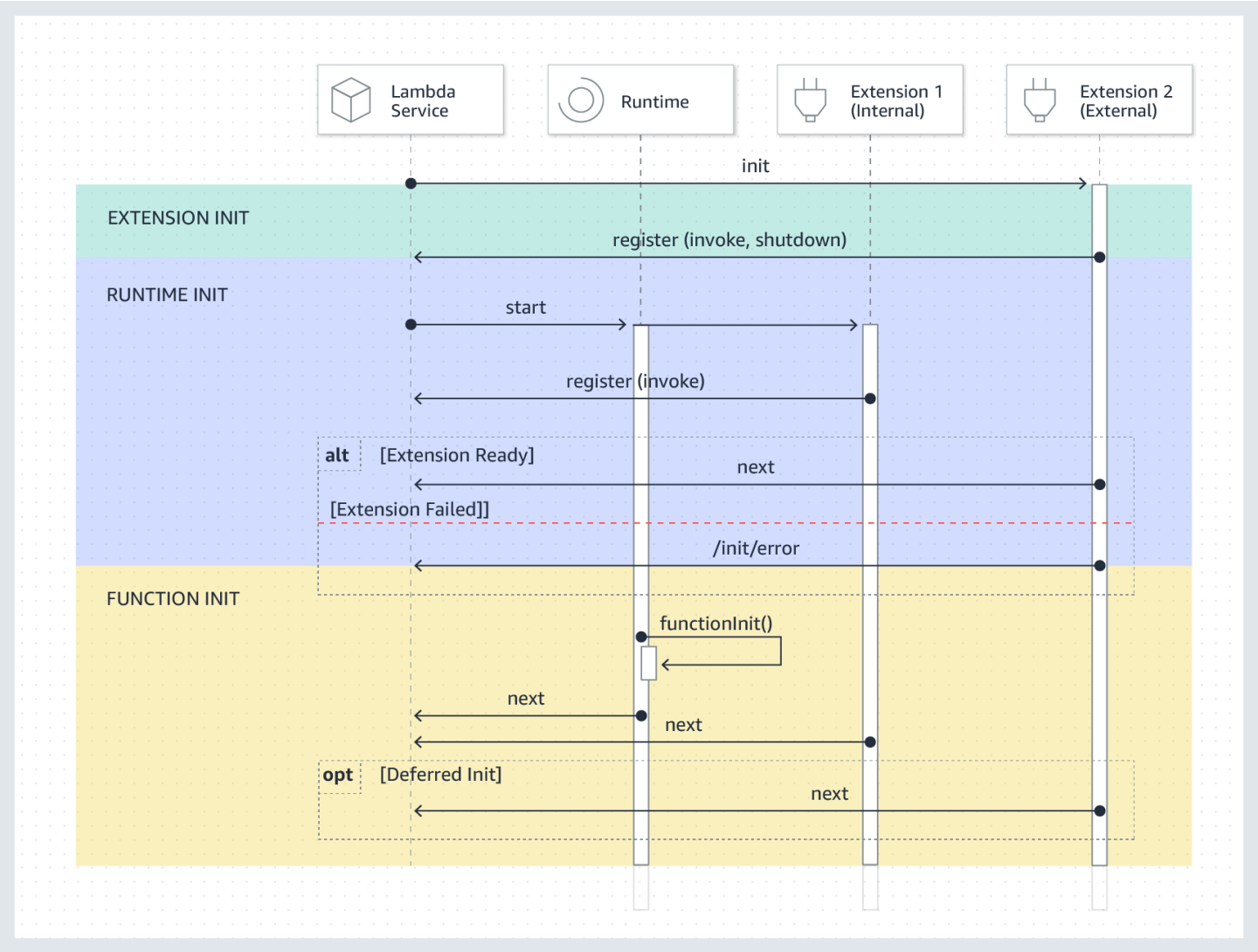
During the `Extension init` phase, each extension needs to register with Lambda to receive events. Lambda uses the full file name of the extension to validate that the extension has completed the bootstrap sequence. Therefore, each `Register` API call must include the `Lambda-Extension-Name` header with the full file name of the extension.

You can register up to 10 extensions for a function. This limit is enforced through the `Register` API call.

After each extension registers, Lambda starts the `Runtime init` phase. The runtime process calls `functionInit` to start the `Function init` phase.

The `Init` phase completes after the runtime and each registered extension indicate completion by sending a `Next` API request.

> **Note**
>
> Extensions can complete their initialization at any point in the `Init` phase.



Invoke phase

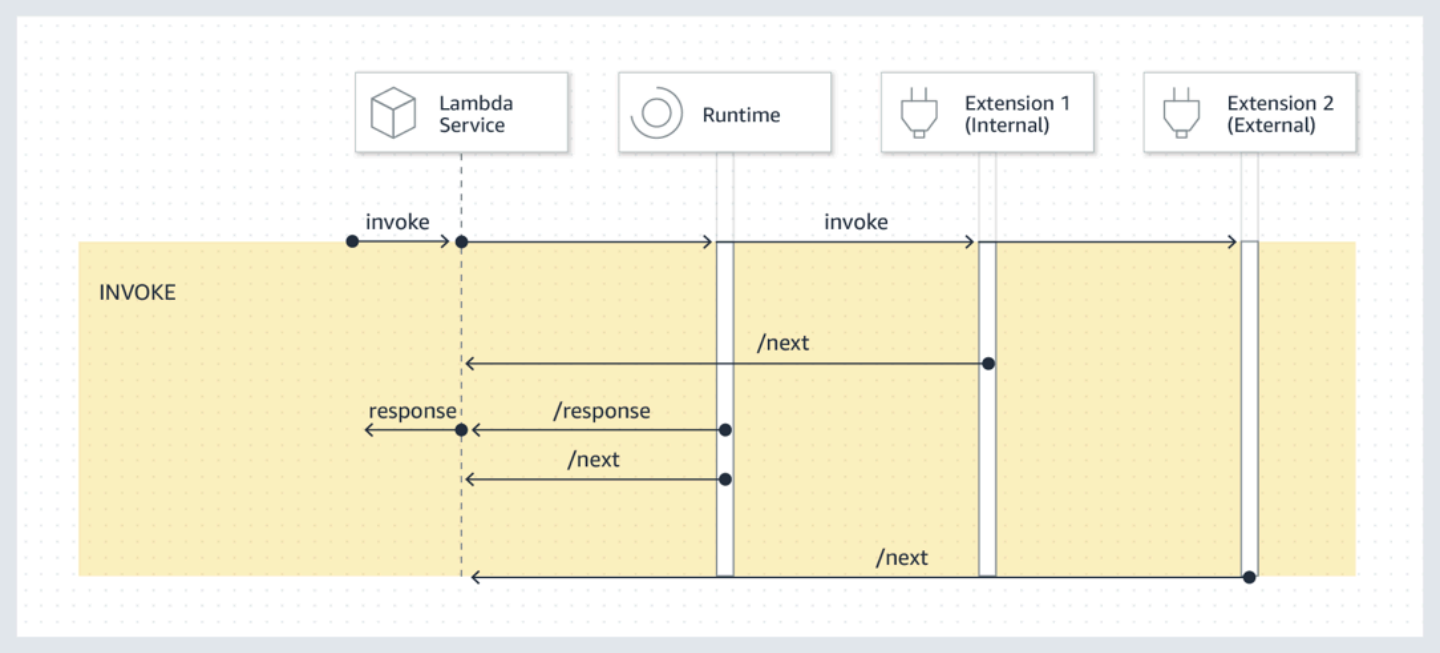
When a Lambda function is invoked in response to a `Next` API request, Lambda sends an `Invoke` event to the runtime and to each extension that is registered for the `Invoke` event.

> **Note**
>
> **Lambda Managed Instances:** Extensions for Lambda Managed Instances functions cannot register for the `Invoke` event. Because Lambda Managed Instances supports concurrent invocations within a single execution environment, the `Invoke` event is not supported. Extensions can only register for the `Shutdown` event. If you need to track when invocations start and finish, use the `platform.report` platform event through the Telemetry API.

During the invocation, external extensions run in parallel with the function. They also continue running after the function has completed. This enables you to capture diagnostic information or to send logs, metrics, and traces to a location of your choice.

After receiving the function response from the runtime, Lambda returns the response to the client, even if extensions are still running.

The `Invoke` phase ends after the runtime and all extensions signal that they are done by sending a `Next` API request.



Event payload: The event sent to the runtime (and the Lambda function) carries the entire request, headers (such as `RequestId`), and payload. The event sent to each extension contains metadata that describes the event content. This lifecycle event includes the type of the event, the time that the function times out (`deadlineMs`), the `requestId`, the invoked function's Amazon Resource Name (ARN), and tracing headers.

Extensions that want to access the function event body can use an in-runtime SDK that communicates with the extension. Function developers use the in-runtime SDK to send the payload to the extension when the function is invoked.

Here is an example payload:

```
{
    "eventType": "INVOKE",
```

```
    "deadlineMs": 676051,
    "requestId": "3da1f2dc-3222-475e-9205-e2e6c6318895",
    "invokedFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:ExtensionTest",
    "tracing": {
        "type": "X-Amzn-Trace-Id",
        "value":
 "Root=1-5f35ae12-0c0fec141ab77a00bc047aa2;Parent=2be948a625588e32;Sampled=1"
    }
 }
```

Duration limit: The function's timeout setting limits the duration of the entire `Invoke` phase. For example, if you set the function timeout as 360 seconds, the function and all extensions need to complete within 360 seconds. Note that there is no independent post-invoke phase. The duration is the total time it takes for your runtime and all your extensions' invocations to complete and is not calculated until the function and all extensions have finished running.

Performance impact and extension overhead: Extensions can impact function performance. As an extension author, you have control over the performance impact of your extension. For example, if your extension performs compute-intensive operations, the function's duration increases because the extension and the function code share the same CPU resources. In addition, if your extension performs extensive operations after the function invocation completes, the function duration increases because the `Invoke` phase continues until all extensions signal that they are completed.

> **Note**
>
> Lambda allocates CPU power in proportion to the function's memory setting. You might see increased execution and initialization duration at lower memory settings because the function and extension processes are competing for the same CPU resources. To reduce the execution and initialization duration, try increasing the memory setting.

To help identify the performance impact introduced by extensions on the `Invoke` phase, Lambda outputs the `PostRuntimeExtensionsDuration` metric. This metric measures the cumulative time spent between the runtime `Next` API request and the last extension `Next` API request. To measure the increase in memory used, use the `MaxMemoryUsed` metric. For more information about function metrics, see Using CloudWatch metrics with Lambda.

Function developers can run different versions of their functions side by side to understand the impact of a specific extension. We recommend that extension authors publish expected resource consumption to make it easier for function developers to choose a suitable extension.

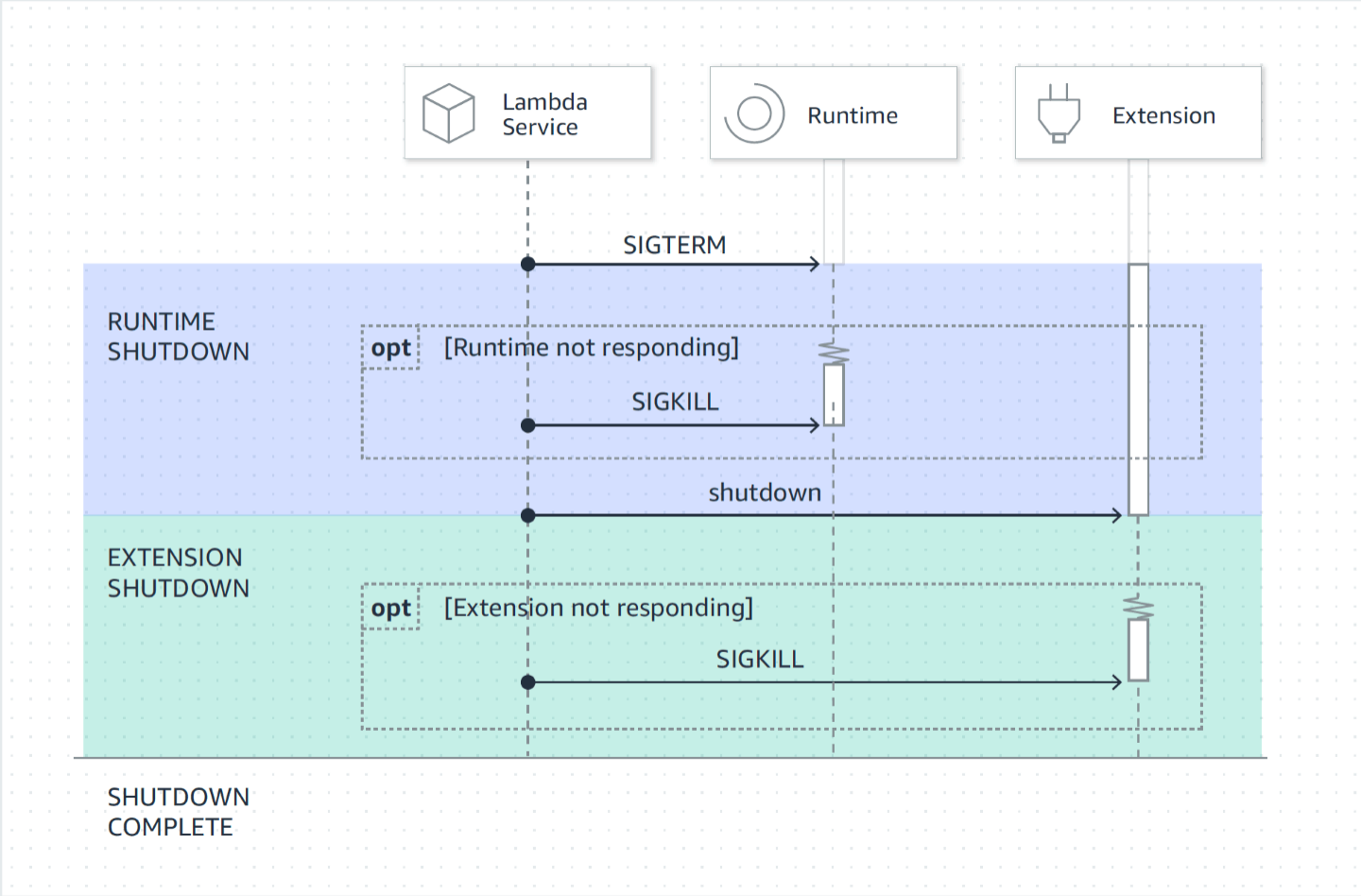
Shutdown phase

When Lambda is about to shut down the runtime, it sends a `Shutdown` to each registered external extension. Extensions can use this time for final cleanup tasks. The `Shutdown` event is sent in response to a `Next` API request.

Duration limit: The maximum duration of the `Shutdown` phase depends on the configuration of registered extensions:

- 0 ms – A function with no registered extensions
- 500 ms – A function with a registered internal extension
- 2,000 ms – A function with one or more registered external extensions

If the runtime or an extension does not respond to the `Shutdown` event within the limit, Lambda ends the process using a `SIGKILL` signal.



Event payload: The `Shutdown` event contains the reason for the shutdown and the time remaining in milliseconds.

The `shutdownReason` includes the following values:

- SPINDOWN – Normal shutdown
- TIMEOUT – Duration limit timed out
- FAILURE – Error condition, such as an `out-of-memory` event

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

Permissions and configuration

Extensions run in the same execution environment as the Lambda function. Extensions also share resources with the function, such as CPU, memory, and `/tmp` disk storage. In addition, extensions use the same AWS Identity and Access Management (IAM) role and security context as the function.

File system and network access permissions: Extensions run in the same file system and network name namespace as the function runtime. This means that extensions need to be compatible with the associated operating system. If an extension requires any additional outbound network traffic rules, you must apply these rules to the function configuration.

> **Note**
>
> Because the function code directory is read-only, extensions cannot modify the function code.

Environment variables: Extensions can access the function's environment variables, except for the following variables that are specific to the runtime process:

- `AWS_EXECUTION_ENV`
- `AWS_LAMBDA_LOG_GROUP_NAME`
- `AWS_LAMBDA_LOG_STREAM_NAME`
- `AWS_XRAY_CONTEXT_MISSING`
- `AWS_XRAY_DAEMON_ADDRESS`
- `LAMBDA_RUNTIME_DIR`
- `LAMBDA_TASK_ROOT`
- `_AWS_XRAY_DAEMON_ADDRESS`
- `_AWS_XRAY_DAEMON_PORT`
- `_HANDLER`

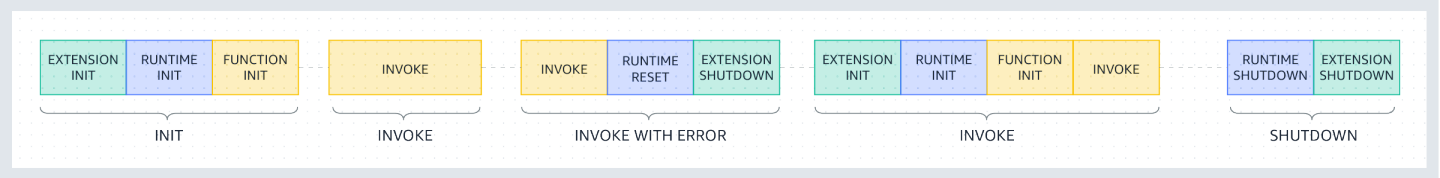
> **Note**
>
> **Detecting Lambda Managed Instances:** Extensions can check the `AWS_LAMBDA_INITIALIZATION_TYPE` environment variable to determine if they are running on Lambda Managed Instances versus Lambda (default) functions. This is the recommended method for extensions to adapt their behavior based on the execution environment type.

Failure handling

Initialization failures: If an extension fails, Lambda restarts the execution environment to enforce consistent behavior and to encourage fail fast for extensions. Also, for some customers, the extensions must meet mission-critical needs such as logging, security, governance, and telemetry collection.

Invoke failures (such as out of memory, function timeout): Because extensions share resources with the runtime, memory exhaustion affects them. When the runtime fails, all extensions and the runtime itself participate in the `Shutdown` phase. In addition, the runtime is restarted—either automatically as part of the current invocation, or via a deferred re-initialization mechanism.

If there is a failure (such as a function timeout or runtime error) during `Invoke`, the Lambda service performs a reset. The reset behaves like a `Shutdown` event. First, Lambda shuts down the runtime, then it sends a `Shutdown` event to each registered external extension. The event includes the reason for the shutdown. If this environment is used for a new invocation, the extension and runtime are re-initialized as part of the next invocation.



For a more detailed explanation of the previous diagram, see Failures during the invoke phase.

Extension logs: Lambda sends the log output of extensions to CloudWatch Logs. Lambda also generates an additional log event for each extension during `Init`. The log event records the name and registration preference (event, config) on success, or the failure reason on failure.

Troubleshooting extensions

- If a `Register` request fails, make sure that the `Lambda-Extension-Name` header in the `Register` API call contains the full file name of the extension.
- If the `Register` request fails for an internal extension, make sure that the request does not register for the Shutdown event.

Extensions API reference

The OpenAPI specification for the extensions API version **2020-01-01** is available here: extensions-api.zip

You can retrieve the value of the API endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable. To send a `Register` request, use the prefix `2020-01-01/` before each API path. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API methods

- Register
- Next
- Init error
- Exit error

Register

During `Extension init`, all extensions need to register with Lambda to receive events. Lambda uses the full file name of the extension to validate that the extension has completed the bootstrap sequence. Therefore, each `Register` API call must include the `Lambda-Extension-Name` header with the full file name of the extension.

Internal extensions are started and stopped by the runtime process, so they are not permitted to register for the Shutdown event.

Path – `/extension/register`

Method – **POST**

Request headers

- `Lambda-Extension-Name` – The full file name of the extension. Required: yes. Type: string.
- `Lambda-Extension-Accept-Feature` – Use this to specify optional Extensions features during registration. Required: no. Type: comma separated string. Features available to specify using this setting:
  - `accountId` – If specified, the Extension registration response will contain the account ID associated with the Lambda function that you're registering the Extension for.

Request body parameters

- `events` – Array of the events to register for. Required: no. Type: array of strings. Valid strings: `INVOKE`, `SHUTDOWN`.

> **Note**
>
> **Lambda Managed Instances:** Extensions for Lambda Managed Instances functions can only register for the `SHUTDOWN` event. Attempting to register for the `INVOKE` event will result in an error. This is because Lambda Managed Instances supports concurrent invocations within a single execution environment.

Response headers

- `Lambda-Extension-Identifier` – Generated unique agent identifier (UUID string) that is required for all subsequent requests.

Response codes

- 200 – Response body contains the function name, function version, and handler name.
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Extension should exit promptly.

Example request body

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example response body

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
}
```

Example response body with optional accountId feature

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

Next

Extensions send a `Next` API request to receive the next event, which can be an `Invoke` event or a `Shutdown` event. The response body contains the payload, which is a JSON document that contains event data.

The extension sends a `Next` API request to signal that it is ready to receive new events. This is a blocking call.

Do not set a timeout on the GET call, as the extension can be suspended for a period of time until there is an event to return.

Path – `/extension/event/next`

Method – **GET**

Request headers

- `Lambda-Extension-Identifier` – Unique identifier for extension (UUID string). Required: yes. Type: UUID string.

Response headers

- `Lambda-Extension-Event-Identifier` – Unique identifier for the event (UUID string).

Response codes

- 200 – Response contains information about the next event (`EventInvoke` or `EventShutdown`).
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Extension should exit promptly.

Init error

The extension uses this method to report an initialization error to Lambda. Call it when the extension fails to initialize after it has registered. After Lambda receives the error, no further API calls succeed. The extension should exit after it receives the response from Lambda.

Path – `/extension/init/error`

Method – **POST**

Request headers

- `Lambda-Extension-Identifier` – Unique identifier for extension. Required: yes. Type: UUID string.
- `Lambda-Extension-Function-Error-Type` – Error type that the extension encountered. Required: yes. This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

Request body parameters

- `ErrorRequest` – Information about the error. Required: no.

This field is a JSON object with the following structure:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Note that Lambda accepts any value for `errorType`.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

Example Function error

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Response codes

- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Extension should exit promptly.

Exit error

The extension uses this method to report an error to Lambda before exiting. Call it when you encounter an unexpected failure. After Lambda receives the error, no further API calls succeed. The extension should exit after it receives the response from Lambda.

Path – `/extension/exit/error`

Method – POST

Request headers

- `Lambda-Extension-Identifier` – Unique identifier for extension. Required: yes. Type: UUID string.
- `Lambda-Extension-Function-Error-Type` – Error type that the extension encountered. Required: yes. This header consists of a string value. Lambda accepts any string, but we recommend a format of <category.reason>. For example:
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

Request body parameters

- `ErrorRequest` – Information about the error. Required: no.

This field is a JSON object with the following structure:

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

Note that Lambda accepts any value for `errorType`.

The following example shows a Lambda function error message in which the function could not parse the event data provided in the invocation.

Example Function error

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

Response codes

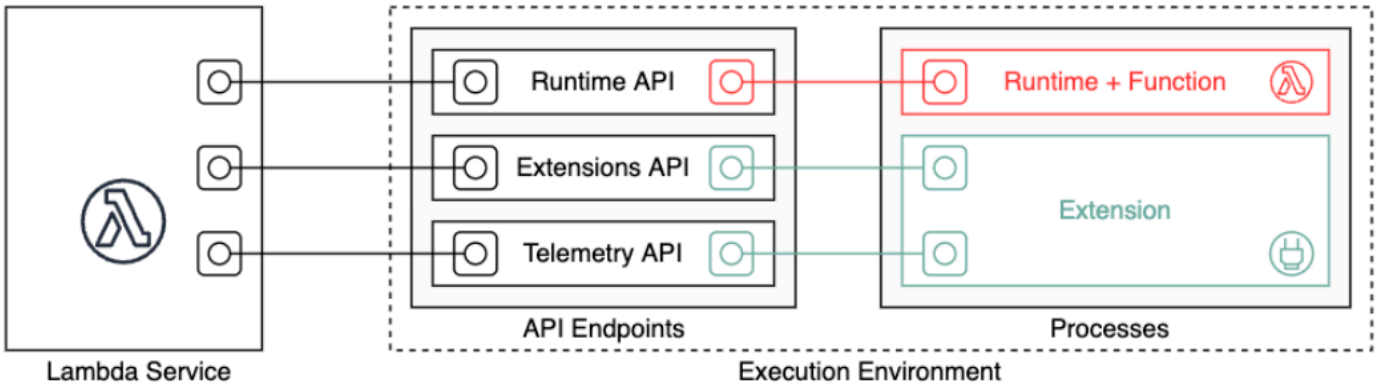
- 202 – Accepted
- 400 – Bad Request
- 403 – Forbidden
- 500 – Container error. Non-recoverable state. Extension should exit promptly.

Accessing real-time telemetry data for extensions using the Telemetry API

The Telemetry API enables your extensions to receive telemetry data directly from Lambda. During function initialization and invocation, Lambda automatically captures telemetry, including logs, platform metrics, and platform traces. The Telemetry API enables extensions to access this telemetry data directly from Lambda in near real time.

Within the Lambda execution environment, you can subscribe your Lambda extensions to telemetry streams. After subscribing, Lambda automatically sends all telemetry data to your extensions. You then have the flexibility to process, filter, and dispatch the data to your preferred destination, such as an Amazon Simple Storage Service (Amazon S3) bucket or a third-party observability tools provider.

The following diagram shows how the Extensions API and Telemetry API link extensions to Lambda from within the execution environment. Additionally, the Runtime API connects your runtime and function to Lambda.



Important

The Lambda Telemetry API supersedes the Lambda Logs API. **While the Logs API remains fully functional, we recommend using only the Telemetry API going forward.** You can subscribe your extension to a telemetry stream using either the Telemetry API or the Logs API. After subscribing using one of these APIs, any attempt to subscribe using the other API returns an error.

Lambda Managed Instances schema version requirement

Lambda Managed Instances support only the `2025-01-29` schema version of the Telemetry API. When subscribing to telemetry streams for Managed Instance functions, you **must** use `"schemaVersion": "2025-01-29"` in your subscription request. Using previous schema versions will result in events being rejected by Lambda.
The `2025-01-29` schema version is backward compatible and can be used with both Lambda Managed Instances and Lambda (default) functions. We recommend using this version for all new extensions to ensure compatibility across both deployment models.

Extensions can use the Telemetry API to subscribe to three different telemetry streams:

- **Platform telemetry** – Logs, metrics, and traces, which describe events and errors related to the execution environment runtime lifecycle, extension lifecycle, and function invocations.
- **Function logs** – Custom logs that the Lambda function code generates.
- **Extension logs** – Custom logs that the Lambda extension code generates.

Note

Lambda sends logs and metrics to CloudWatch, and traces to X-Ray (if you've activated tracing), even if an extension subscribes to telemetry streams.

Sections

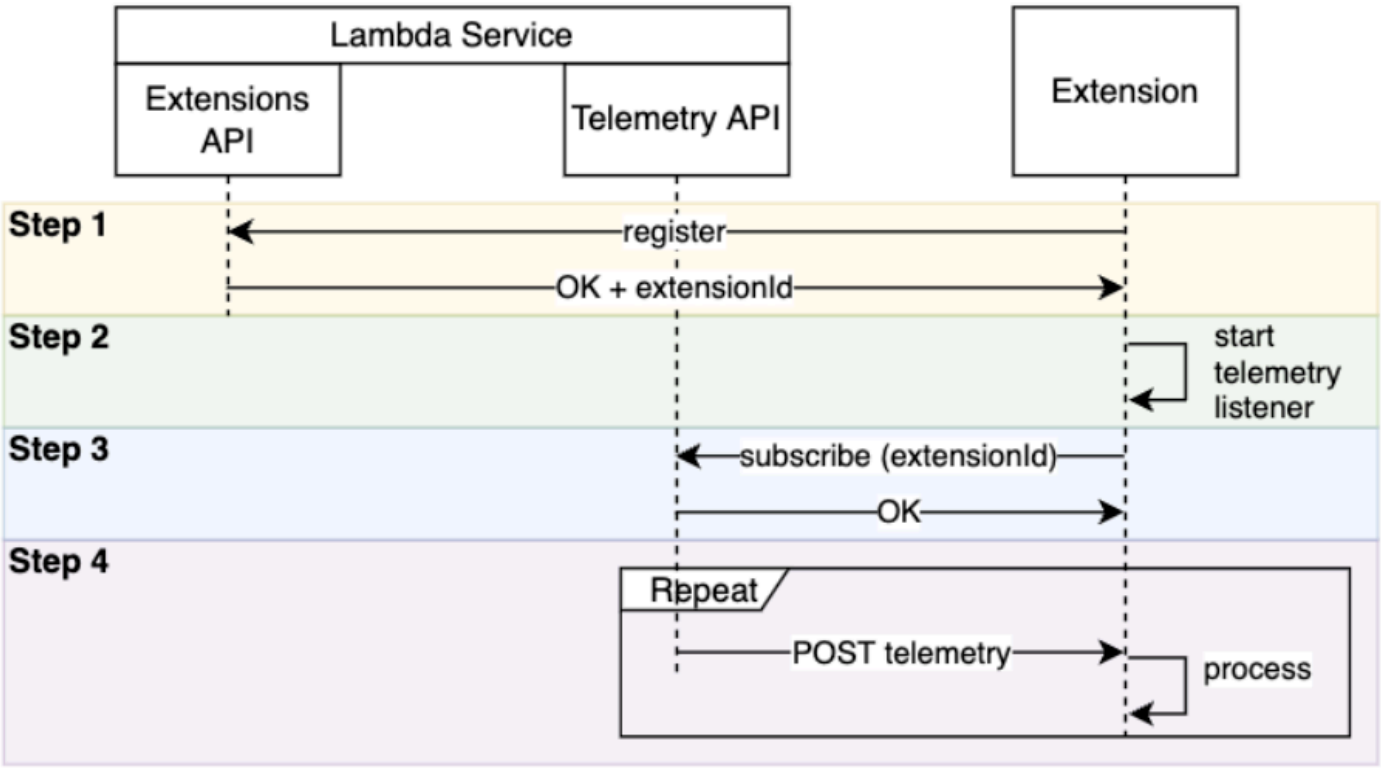
- Creating extensions using the Telemetry API
- Registering your extension
- Creating a telemetry listener
- Specifying a destination protocol
- Configuring memory usage and buffering
- Sending a subscription request to the Telemetry API
- Inbound Telemetry API messages
- Lambda Telemetry API reference
- Lambda Telemetry API Event schema reference

- Converting Lambda Telemetry API Event objects to OpenTelemetry Spans
- Using the Lambda Logs API

Creating extensions using the Telemetry API

Lambda extensions run as independent processes in the execution environment. Extensions can continue to run after function invocation completes. Because extensions are separate processes, you can write them in a language different from the function code. We recommend writing extensions using a compiled language such as Golang or Rust. This way, the extension is a self-contained binary that can be compatible with any supported runtime.

The following diagram illustrates a four-step process to create an extension that receives and processes telemetry data using the Telemetry API.



Here is each step in more detail:

1. Register your extension using the the section called “Extensions API”. This provides you with a `Lambda-Extension-Identifier`, which you'll need in the following steps. For more information about how to register your extension, see the section called “Registering your extension”.

2. Create a telemetry listener. This can be a basic HTTP or TCP server. Lambda uses the URI of the telemetry listener to send telemetry data to your extension. For more information, see the section called “Creating a telemetry listener”.
3. Using the Subscribe API in the Telemetry API, subscribe your extension to the desired telemetry streams. You'll need the URI of your telemetry listener for this step. For more information, see the section called “Sending a subscription request to the Telemetry API”.
4. Get telemetry data from Lambda via the telemetry listener. You can do any custom processing of this data, such as dispatching the data to Amazon S3 or to an external observability service.

> **Note**
>
> A Lambda function's execution environment can start and stop multiple times as part of its lifecycle. In general, your extension code runs during function invocations, and also up to 2 seconds during the shutdown phase. We recommend batching the telemetry as it arrives to your listener. Then, use the `Invoke` and `Shutdown` lifecycle events to send each batch to their desired destinations.

Registering your extension

Before you can subscribe to telemetry data, you must register your Lambda extension. Registration occurs during the extension initialization phase. The following example shows an HTTP request to register an extension.

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

If the request succeeds, the subscriber receives an HTTP 200 success response. The response header contains the `Lambda-Extension-Identifier`. The response body contains other properties of the function.

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
```

```
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

For more information, see the the section called “Extensions API reference”.

Creating a telemetry listener

Your Lambda extension must have a listener that handles incoming requests from the Telemetry API. The following code shows an example telemetry listener implementation in Golang:

```
// Starts the server in a goroutine where the log events will be sent
func (s *TelemetryApiListener) Start() (string, error) {
 address := listenOnAddress()
 l.Info("[listener:Start] Starting on address", address)
 s.httpServer = &http.Server{Addr: address}
 http.HandleFunc("/", s.http_handler)
 go func() {
  err := s.httpServer.ListenAndServe()
  if err != http.ErrServerClosed {
   l.Error("[listener:goroutine] Unexpected stop on Http Server:", err)
   s.Shutdown()
  } else {
   l.Info("[listener:goroutine] Http Server closed:", err)
  }
 }()
 return fmt.Sprintf("http://%s/", address), nil
}

// http_handler handles the requests coming from the Telemetry API.
// Everytime Telemetry API sends log events, this function will read them from the
 response body
// and put into a synchronous queue to be dispatched later.
// Logging or printing besides the error cases below is not recommended if you have
 subscribed to
// receive extension logs. Otherwise, logging here will cause Telemetry API to send new
 logs for
// the printed lines which may create an infinite loop.
func (s *TelemetryApiListener) http_handler(w http.ResponseWriter, r *http.Request) {
 body, err := ioutil.ReadAll(r.Body)
 if err != nil {
```

```
  l.Error("[listener:http_handler] Error reading body:", err)
  return
 }

 // Parse and put the log messages into the queue
 var slice []interface{}
 _ = json.Unmarshal(body, &slice)

 for _, el := range slice {
  s.LogEventsQueue.Put(el)
 }

 l.Info("[listener:http_handler] logEvents received:", len(slice), " LogEventsQueue
 length:", s.LogEventsQueue.Len())
 slice = nil
}
```

Specifying a destination protocol

When you subscribe to receive telemetry using the Telemetry API, you can specify a destination protocol in addition to the destination URI:

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda accepts two protocols for receiving telemetry:

- **HTTP (recommended)** – Lambda delivers telemetry to a local HTTP endpoint (`http://sandbox.localdomain:${PORT}/${PATH}`) as an array of records in JSON format. The `$PATH` parameter is optional. Lambda supports only HTTP, not HTTPS. Lambda delivers telemetry through POST requests.
- **TCP** – Lambda delivers telemetry to a TCP port in Newline delimited JSON (NDJSON) format.

> **Note**
>
> We strongly recommend using HTTP rather than TCP. With TCP, the Lambda platform cannot acknowledge when it delivers telemetry to the application layer. Therefore, if your extension crashes, you might lose telemetry. HTTP does not have this limitation.

Before subscribing to receive telemetry, establish the local HTTP listener or TCP port. During setup, note the following:

- Lambda sends telemetry only to destinations that are inside the execution environment.
- Lambda retries to send telemetry (with backoff) in the absence of a listener, or if the POST request encounters an error. If the telemetry listener crashes, it resumes receiving telemetry after Lambda restarts the execution environment.
- Lambda reserves port 9001. There are no other port number restrictions or recommendations.

Configuring memory usage and buffering

Memory usage in an execution environment grows linearly with the number of subscribers. Subscriptions consume memory resources because each one opens a new memory buffer to store telemetry data. Buffer memory usage contributes to the overall memory consumption in the execution environment.

When subscribing to receive telemetry through the Telemetry API, you have the option to buffer telemetry data and deliver it to subscribers in batches. To optimize memory usage, you can specify a buffering configuration:

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
    }
}
```

Parameter	Description	Defaults and limits
`maxBytes`	The maximum volume of telemetry (in bytes) to buffer in memory.	Default: 262,144 Minimum: 262,144 Maximum: 1,048,576
`maxItems`	The maximum number of events to buffer in memory.	Default: 10,000 Minimum: 1,000 Maximum: 10,000
`timeoutMs`	The maximum time (in milliseconds) to buffer a batch.	Default: 1,000 Minimum: 25 Maximum: 30,000

When setting up buffering, keep these points in mind:

- If any of the input streams are closed, Lambda flushes the logs. For example, this can occur if the runtime crashes.
- Each subscriber can customize their buffering configuration in their subscription request.
- When determining the buffer size for reading the data, anticipate receiving payloads as large as `2 * maxBytes + metadataBytes`, where `maxBytes` is a component of your buffering setup. To gauge the amount of `metadataBytes` to consider, review the following metadata. Lambda appends metadata similar to this to each record:

```
{
   "time": "2022-08-20T12:31:32.123Z",
   "type": "function",
   "record": "Hello World"
}
```

- If the subscriber cannot process incoming telemetry fast enough, or if your function code generates very high log volume, Lambda might drop records to keep memory utilization bounded. When this occurs, Lambda sends a `platform.logsDropped` event.

Sending a subscription request to the Telemetry API

Lambda extensions can subscribe to receive telemetry data by sending a subscription request to the Telemetry API. The subscription request should contain information about the types of events that you want the extension to subscribe to. In addition, the request can contain delivery destination information and a buffering configuration.

Before sending a subscription request, you must have an extension ID (`Lambda-Extension-Identifier`). When you register your extension with the Extensions API, you obtain an extension ID from the API response.

Subscription occurs during the extension initialization phase. The following example shows an HTTP request to subscribe to all three telemetry streams: platform telemetry, function logs, and extension logs.

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2025-01-29",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

If the request succeeds, then the subscriber receives an HTTP 200 success response.

```
HTTP/1.1 200 OK
"OK"
```

Inbound Telemetry API messages

After subscribing using the Telemetry API, an extension automatically starts to receive telemetry from Lambda via POST requests. Each POST request body contains an array of `Event` objects. Each `Event` has the following schema:

```
{
   time: String,
   type: String,
   record: Object
}
```

- The `time` property defines when the Lambda platform generated the event. This is different from when the event actually occurred. The string value of `time` is a timestamp in ISO 8601 format.
- The `type` property defines the event type. The following table describes all possible values.
- The `record` property defines a JSON object that contains the telemetry data. The schema of this JSON object depends on the `type`.

Event ordering with concurrent invocations

For Lambda Managed Instances, multiple function invocations can execute concurrently within the same execution environment. In this case, the order of `platform.start` and `platform.report` events is not guaranteed between different concurrent invocations. Extensions must handle events from multiple invocations running in parallel and should not assume sequential ordering.
To properly attribute events to specific invocations, extensions should use the `requestId` field present in these platform events. Each invocation has a unique request ID that remains consistent across all events for that invocation, allowing extensions to correlate events correctly even when they arrive out of order.

The following table summarizes all types of `Event` objects, and links to the Telemetry API `Event` schema reference for each event type.

Category	Event type	Description	Event record schema
Platform event	`platform.initStart`	Function initialization started.	the section called "`platform.initStart`" schema
Platform event	`platform.initRuntimeDone`	Function initialization completed.	the section called "`platform.initRuntimeDone`" schema
Platform event	`platform.initReport`	A report of function initialization.	the section called "`platform.initReport`" schema
Platform event	`platform.start`	Function invocation started.	the section called "`platform.start`" schema
Platform event	`platform.runtimeDone`	The runtime finished processing an event with either success or failure.	the section called "`platform.runtimeDone`" schema
Platform event	`platform.report`	A report of function invocation.	the section called "`platform.report`" schema
Platform event	`platform.restoreStart`	Runtime restore started.	the section called "`platform.restoreStart`" schema
Platform event	`platform.restoreRuntimeDone`	Runtime restore completed.	the section called "`platform.restoreRu`

Category	Event type	Description	Event record schema
			ntimeDone ” schema
Platform event	platform. restoreReport	Report of runtime restore.	the section called “platform. restoreReport ” schema
Platform event	platform. telemetry Subscription	The extension subscribed to the Telemetry API.	the section called “platform. telemetry Subscription ” schema
Platform event	platform. logsDropped	Lambda dropped log entries.	the section called “platform. logsDropped ” schema
Function logs	function	A log line from function code.	the section called “function” schema
Extension logs	extension	A log line from extension code.	the section called “extension ” schema

Lambda Telemetry API reference

Use the Lambda Telemetry API endpoint to subscribe extensions to telemetry streams. You can retrieve the Telemetry API endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable. To send an API request, append the API version (`2022-07-01/`) and `telemetry/`. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

For the OpenAPI Specification (OAS) definition of the subscription responses version `2025-01-29`, see the following:

- **HTTP** – telemetry-api-http-schema.zip
- **TCP** – telemetry-api-tcp-schema.zip

API operations

- Subscribe

Subscribe

To subscribe to a telemetry stream, a Lambda extension can send a Subscribe API request.

- **Path** – `/telemetry`
- **Method** – PUT
- **Headers**
  - `Content-Type: application/json`
- **Request body parameters**
  - **schemaVersion**
    - Required: Yes
    - Type: String
    - Valid values: `"2025-01-29"`, `"2022-12-13"`, or `"2022-07-01"`
    - **Note:** Lambda Managed Instances require `"2025-01-29"`. This version is backward compatible with Lambda (default) functions.
  - **destination** – The configuration settings that define the telemetry event destination and the protocol for event delivery.

- Required: Yes
- Type: Object

```
{
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080"
}
```

- **protocol** – The protocol that Lambda uses to send telemetry data.
  - Required: Yes
  - Type: String
  - Valid values: `"HTTP"|"TCP"`
- **URI** – The URI to send telemetry data to.
  - Required: Yes
  - Type: String
- For more information, see the section called “Specifying a destination protocol”.

- **types** – The types of telemetry that you want the extension to subscribe to.
  - Required: Yes
  - Type: Array of strings
  - Valid values: `"platform"|"function"|"extension"`
- **buffering** – The configuration settings for event buffering.
  - Required: No
  - Type: Object

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

  - **maxItems** – The maximum number of events to buffer in memory.
    - Required: No
    - Type: Integer

  - Default: 1,000
  - Minimum: 1,000
  - Maximum: 10,000
- **maxBytes** – The maximum volume of telemetry (in bytes) to buffer in memory.
  - Required: No
  - Type: Integer
  - Default: 262,144
  - Minimum: 262,144
  - Maximum: 1,048,576
- **timeoutMs** – The maximum time (in milliseconds) to buffer a batch.
  - Required: No
  - Type: Integer
  - Default: 1,000
  - Minimum: 25
  - Maximum: 30,000
- For more information, see the section called “Configuring memory usage and buffering”.

Example Subscribe API request

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2025-01-29",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
```

```
}
```

If the Subscribe request succeeds, the extension receives an HTTP 200 success response:

```
HTTP/1.1 200 OK
"OK"
```

If the Subscribe request fails, the extension receives an error response. For example:

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

Here are some additional response codes that the extension can receive:

- 200 – Request completed successfully
- 202 – Request accepted. Subscription request response in local testing environment
- 400 – Bad request
- 500 – Service error

Lambda Telemetry API Event schema reference

Use the Lambda Telemetry API endpoint to subscribe extensions to telemetry streams. You can retrieve the Telemetry API endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable. To send an API request, append the API version (`2022-07-01/`) and `telemetry/`. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

For the OpenAPI Specification (OAS) definition of the subscription responses version `2025-01-29`, see the following:

- **HTTP** – telemetry-api-http-schema.zip
- **TCP** – telemetry-api-tcp-schema.zip

The following table is a summary of all the types of `Event` objects that the Telemetry API supports.

Category	Event type	Description	Event record schema
Platform event	`platform.initStart`	Function initialization started.	the section called "`platform.initStart`" schema
Platform event	`platform.initRuntimeDone`	Function initialization completed.	the section called "`platform.initRuntimeDone`" schema
Platform event	`platform.initReport`	A report of function initialization.	the section called "`platform.initReport`" schema
Platform event	`platform.start`	Function invocation started.	the section called "`platform.start`" schema

Category	Event type	Description	Event record schema
Platform event	`platform.runtimeDone`	The runtime finished processing an event with either success or failure.	the section called "`platform.runtimeDone`" schema
Platform event	`platform.report`	A report of function invocation.	the section called "`platform.report`" schema
Platform event	`platform.restoreStart`	Runtime restore started.	the section called "`platform.restoreStart`" schema
Platform event	`platform.restoreRuntimeDone`	Runtime restore completed.	the section called "`platform.restoreRuntimeDone`" schema
Platform event	`platform.restoreReport`	Report of runtime restore.	the section called "`platform.restoreReport`" schema
Platform event	`platform.telemetrySubscription`	The extension subscribed to the Telemetry API.	the section called "`platform.telemetrySubscription`" schema
Platform event	`platform.logsDropped`	Lambda dropped log entries.	the section called "`platform.logsDropped`" schema

Category	Event type	Description	Event record schema
Function logs	`function`	A log line from function code.	the section called "`function`" schema
Extension logs	`extension`	A log line from extension code.	the section called "`extension`" schema

Contents

- Telemetry API Event object types
  - platform.initStart
  - platform.initRuntimeDone
  - platform.initReport
  - platform.start
  - platform.runtimeDone
  - platform.report
  - platform.restoreStart
  - platform.restoreRuntimeDone
  - platform.restoreReport
  - platform.extension
  - platform.telemetrySubscription
  - platform.logsDropped
  - function
  - extension
- Shared object types
  - InitPhase
  - InitReportMetrics
  - InitType
  - ReportMetrics
  - RestoreReportMetrics
  - RuntimeDoneMetrics

- Span
- Status
- TraceContext
- TracingType

Telemetry API Event object types

This section details the types of `Event` objects that the Lambda Telemetry API supports. In the event descriptions, a question mark (?) indicates that the attribute may not be present in the object.

`platform.initStart`

A `platform.initStart` event indicates that the function initialization phase has started. A `platform.initStart` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

The `PlatformInitStart` object has the following attributes:

- **functionName** – `String`
- **functionVersion** – `String`
- **initializationType** – `the section called “InitType”` object
- **instanceId?** – `String`
- **instanceMaxMemory?** – `Integer`
- **phase** – `the section called “InitPhase”` object
- **runtimeVersion?** – `String`
- **runtimeVersionArn?** – `String`

The following is an example `Event` of type `platform.initStart`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
```

```
    "type": "platform.initStart",
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

`platform.initRuntimeDone`

A `platform.initRuntimeDone` event indicates that the function initialization phase has completed. A `platform.initRuntimeDone` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

The `PlatformInitRuntimeDone` object has the following attributes:

- **initializationType** – the section called “InitType” object
- **phase** – the section called “InitPhase” object
- **status** – the section called “Status” object
- **spans?** – List of the section called “Span” objects

The following is an example `Event` of type `platform.initRuntimeDone`:

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initRuntimeDone",
    "record": {
        "initializationType": "on-demand"
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
```

```
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 70.5
            }
        ]
    }
}
```

`platform.initReport`

A `platform.initReport` event contains an overall report of the function initialization phase. A `platform.initReport` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

The `PlatformInitReport` object has the following attributes:

- **errorType?** – string
- **initializationType** – the section called "InitType" object
- **phase** – the section called "InitPhase" object
- **metrics** – the section called "InitReportMetrics" object
- **spans?** – List of the section called "Span" objects
- **status** – the section called "Status" object

The following is an example `Event` of type `platform.initReport`:

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initReport",
    "record": {
        "initializationType": "on-demand",
        "status": "success",
        "phase": "init",
        "metrics": {
            "durationMs": 125.33
        },
        "spans": [
            {
```

```
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 90.1
            }
        ]
    }
}
```

`platform.start`

A `platform.start` event indicates that the function invocation phase has started. A `platform.start` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

The `PlatformStart` object has the following attributes:

- **requestId** – `String`
- **version?** – `String`
- **tracing?** – the section called “TraceContext”

The following is an example `Event` of type `platform.start`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.start",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "version": "$LATEST",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        }
    }
}
```

platform.runtimeDone

A platform.runtimeDone event indicates that the function invocation phase has completed. A platform.runtimeDone Event object has the following shape:

> **Lambda Managed Instances**
>
> The platform.runtimeDone event is not supported for Lambda Managed Instances. Extensions running on Managed Instances will not receive this event because extensions cannot subscribe to the INVOKE event on Managed Instances. Due to the concurrent execution model where multiple invocations can be processed simultaneously, extensions cannot perform post-invoke processing for individual invocations as they traditionally do on Lambda (default) functions.
> For Managed Instances, the responseLatency and responseDuration spans that are normally included in platform.runtimeDone are instead available in the platform.report event. See the section called "platform.report" for details.

```
Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

The PlatformRuntimeDone object has the following attributes:

- **errorType?** – String
- **metrics?** – the section called "RuntimeDoneMetrics" object
- **requestId** – String
- **status** – the section called "Status" object
- **spans?** – List of the section called "Span" objects
- **tracing?** – the section called "TraceContext" object

The following is an example Event of type platform.runtimeDone:

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.runtimeDone",
```

```
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "status": "success",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ],
        "metrics": {
            "durationMs": 140.0,
            "producedBytes": 16
        }
    }
}
```

`platform.report`

A `platform.report` event contains an overall report of the function invoke phase. A `platform.report` `Event` object has the following shape:

> **Lambda Managed Instances**
>
> The `platform.report` event for Lambda Managed Instances has different metrics and spans compared to Lambda (default) functions. For Managed Instances:
>
> - **Spans**: Contains `responseLatency` and `responseDuration` instead of `extensionOverhead`. The `extensionOverhead` span is not available because extensions cannot subscribe to the `INVOKE` event on Managed Instances due to the concurrent execution model.
> - **Metrics**: Only includes `durationMs`. The following metrics are not included: `billedDurationMs`, `initDurationMs`, `maxMemoryUsedMB`, and `memorySizeMB`. These per-invoke metrics are not applicable in the concurrent execution environment.

For resource utilization metrics, use Monitoring Lambda Managed Instances or Lambda Insights.

```
Event: Object
- time: String
- type: String = platform.report
- record: PlatformReport
```

The `PlatformReport` object has the following attributes:

- **metrics** – the section called "ReportMetrics" object
- **requestId** – `String`
- **spans?** – List of the section called "Span" objects
- **status** – the section called "Status" object
- **tracing?** – the section called "TraceContext" object

The following is an example `Event` of type `platform.report`:

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.report",
    "record": {
        "metrics": {
            "billedDurationMs": 694,
            "durationMs": 693.92,
            "initDurationMs": 397.68,
            "maxMemoryUsedMB": 84,
            "memorySizeMB": 128
        },
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
    }
}
```

`platform.restoreStart`

A `platform.restoreStart` event indicates that a function environment restoration event started. In an environment restoration event, Lambda creates the environment from a cached

snapshot rather than initializing it from scratch. For more information, see Lambda SnapStart. A `platform.restoreStart` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

The `PlatformRestoreStart` object has the following attributes:

- **functionName** – `String`
- **functionVersion** – `String`
- **instanceId?** – `String`
- **instanceMaxMemory?** – `String`
- **runtimeVersion?** – `String`
- **runtimeVersionArn?** – `String`

The following is an example `Event` of type `platform.restoreStart`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreStart",
    "record": {
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

`platform.restoreRuntimeDone`

A `platform.restoreRuntimeDone` event indicates that a function environment restoration event completed. In an environment restoration event, Lambda creates the environment from a cached snapshot rather than initializing it from scratch. For more information, see Lambda SnapStart. A `platform.restoreRuntimeDone` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

The `PlatformRestoreRuntimeDone` object has the following attributes:

- **errorType?** – `String`
- **spans?** – List of the section called "Span" objects
- **status** – the section called "Status" object

The following is an example `Event` of type `platform.restoreRuntimeDone`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreRuntimeDone",
    "record": {
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ]
    }
}
```

`platform.restoreReport`

A `platform.restoreReport` event contains an overall report of a function restoration event. A `platform.restoreReport` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

The `PlatformRestoreReport` object has the following attributes:

- **errorType?** – string
- **metrics?** – the section called "RestoreReportMetrics" object
- **spans?** – List of the section called "Span" objects
- **status** – the section called "Status" object

The following is an example `Event` of type `platform.restoreReport`:

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreReport",
    "record": {
        "status": "success",
        "metrics": {
            "durationMs": 15.19
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 30.0
            }
        ]
    }
}
```

`platform.extension`

An `extension` event contains logs from the extension code. An `extension` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

The `PlatformExtension` object has the following attributes:

- **events** – List of `String`
- **name** – `String`
- **state** – `String`

The following is an example `Event` of type `platform.extension`:

```
{
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
    }
}
```

`platform.telemetrySubscription`

A `platform.telemetrySubscription` event contains information about an extension subscription. A `platform.telemetrySubscription` `Event` object has the following shape:

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

The `PlatformTelemetrySubscription` object has the following attributes:

- **name** – `String`
- **state** – `String`
- **types** – List of `String`

The following is an example `Event` of type `platform.telemetrySubscription`:

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

platform.logsDropped

A platform.logsDropped event contains information about dropped events. Lambda emits the platform.logsDropped event when a function outputs logs at too high a rate for Lambda to process them. When Lambda can't send logs to CloudWatch or to the extension subscribed to Telemetry API at the rate the function produces them, it drops logs to prevent the function's execution from slowing down. A platform.logsDropped Event object has the following shape:

```
Event: Object
- time: String
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

The PlatformLogsDropped object has the following attributes:

- **droppedBytes** – Integer
- **droppedRecords** – Integer
- **reason** – String

The following is an example Event of type platform.logsDropped:

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.logsDropped",
    "record": {
        "droppedBytes": 12345,
        "droppedRecords": 123,
        "reason": "Some logs were dropped because the downstream consumer is slower
 than the logs production rate"
    }
}
```

function

A function event contains logs from the function code. A function Event object has the following shape:

```
Event: Object
- time: String
- type: String = function
```

```
- record: {}
```

The format of the `record` field depends on whether your function's logs are formatted in plain text or JSON format. to learn more about log format configuration options, see the section called "Log formats"

The following is an example `Event` of type `function` where the log format is plain text:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
}
```

The following is an example `Event` of type `function` where the log format is JSON:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": {
        "timestamp": "2022-10-12T00:03:50.000Z",
        "level": "INFO",
        "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
        "message": "Hello world, I am a function!"
    }
}
```

Note

If the schema version you're using is older than the `2022-12-13` version, then the `"record"` is always rendered as a string even when your function's logging format is configured as JSON. For Lambda Managed Instances, you must use schema version `2025-01-29`.

`extension`

A `extension` event contains logs from the extension code. A `extension` `Event` object has the following shape:

```
Event: Object
```

```
- time: String
- type: String = extension
- record: {}
```

The format of the `record` field depends on whether your function's logs are formatted in plain text or JSON format. to learn more about log format configuration options, see the section called "Log formats"

The following is an example `Event` of type `extension` where the log format is plain text:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
}
```

The following is an example `Event` of type `extension` where the log format is JSON:

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": {
       "timestamp": "2022-10-12T00:03:50.000Z",
       "level": "INFO",
       "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
       "message": "Hello world, I am an extension!"
    }
}
```

Note

If the schema version you're using is older than the `2022-12-13` version, then the `"record"` is always rendered as a string even when your function's logging format is configured as JSON. For Lambda Managed Instances, you must use schema version `2025-01-29`.

Shared object types

This section details the types of shared objects that the Lambda Telemetry API supports.

`InitPhase`

A string enum that describes the phase when the initialization step occurs. In most cases, Lambda runs the function initialization code during the `init` phase. However, in some error cases, Lambda may re-run the function initialization code during the `invoke` phase. (This is called a *suppressed init*.)

- **Type** – `String`
- **Valid values** – `init`|`invoke`|`snap-start`

`InitReportMetrics`

An object that contains metrics about an initialization phase.

- **Type** – `Object`

An `InitReportMetrics` object has the following shape:

```
InitReportMetrics: Object
- durationMs: Double
```

The following is an example `InitReportMetrics` object:

```
{
    "durationMs": 247.88
}
```

`InitType`

A string enum that describes how Lambda initialized the environment.

- **Type** – `String`
- **Valid values** – `on-demand`|`provisioned-concurrency`

`ReportMetrics`

An object that contains metrics about a completed phase.

- **Type** – `Object`

A `ReportMetrics` object has the following shape:

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

The following is an example `ReportMetrics` object:

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

`RestoreReportMetrics`

An object that contains metrics about a completed restoration phase.

- **Type** – `Object`

A `RestoreReportMetrics` object has the following shape:

```
RestoreReportMetrics: Object
- durationMs: Double
```

The following is an example `RestoreReportMetrics` object:

```
{
    "durationMs": 15.19
}
```

`RuntimeDoneMetrics`

An object that contains metrics about an invocation phase.

- **Type** – `Object`

A `RuntimeDoneMetrics` object has the following shape:

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

The following is an example `RuntimeDoneMetrics` object:

```
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

Span

An object that contains details about a span. A span represents a unit of work or operation in a trace. For more information about spans, see Span on the **Tracing API** page of the OpenTelemetry Docs website.

Lambda supports the following spans for the `platform.RuntimeDone` event:

- The `responseLatency` span describes how long it took your Lambda function to start sending the response.
- The `responseDuration` span describes how long it took your Lambda function to finish sending the entire response.
- The `runtimeOverhead` span describes how long it took the Lambda runtime to signal that it is ready to process the next function invoke. This is how long the runtime took to call the next invocation API to get the next event after returning your function response.

The following is an example `responseLatency` span object:

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
        "durationMs": 23.02
      }
```

`Status`

An object that describes the status of an initialization or invocation phase. If the status is either `failure` or `error`, then the `Status` object also contains an `errorType` field describing the error.

- **Type** – `Object`
- **Valid status values** – `success|failure|error|timeout`

`TraceContext`

An object that describes the properties of a trace.

- **Type** – `Object`

A `TraceContext` object has the following shape:

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
- value: String
```

The following is an example `TraceContext` object:

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

`TracingType`

A string enum that describes the type of tracing in a the section called “TraceContext” object.

- **Type** – `String`
- **Valid values** – `X-Amzn-Trace-Id`

Converting Lambda Telemetry API Event objects to OpenTelemetry Spans

The AWS Lambda Telemetry API schema is semantically compatible with OpenTelemetry (OTel). This means that you can convert your AWS Lambda Telemetry API `Event` objects to OpenTelemetry (OTel) Spans. When converting, you shouldn't map a single `Event` object to a single OTel Span. Instead, you should present all three events related to a lifecycle phase in a single OTel Span. For example, the `start`, `runtimeDone`, and `runtimeReport` events represent a single function invocation. Present all three of these events as a single OTel Span.

You can convert your events using Span Events or Child (nested) Spans. The tables on this page describe the mappings between Telemetry API schema properties and OTel Span properties for both approaches. For more information about OTel Spans, see Span on the **Tracing API** page of the OpenTelemetry Docs website.

Sections

- Map to OTel Spans with Span Events
- Map to OTel Spans with Child Spans

Map to OTel Spans with Span Events

In the following tables, e represents the event coming from the telemetry source.

Mapping the *Start events

OpenTelemetry	Lambda Telemetry API schema
`Span.Name`	Your extension generates this value based on the `type` field.
`Span.StartTime`	Use `e.time`.
`Span.EndTime`	N/A, because the event hasn't completed yet.
`Span.Kind`	Set to `Server`.
`Span.Status`	Set to `Unset`.

OpenTelemetry	Lambda Telemetry API schema
`Span.TraceId`	Parse the AWS X-Ray header found in `e.tracing.value` , then use the `TraceId` value.
`Span.ParentId`	Parse the X-Ray header found in `e.tracing.value` , then use the `Parent` value.
`Span.SpanId`	Use `e.tracing.spanId` if available. Otherwise, generate a new `SpanId`.
`Span.SpanContext.TraceState`	N/A for an X-Ray trace context.
`Span.SpanContext.TraceFlags`	Parse the X-Ray header found in `e.tracing.value` , then use the `Sampled` value.
`Span.Attributes`	Your extension can add any custom values here.

Mapping the *RuntimeDone events

OpenTelemetry	Lambda Telemetry API schema
`Span.Name`	Your extension generates the value based on the `type` field.
`Span.StartTime`	Use `e.time` from the matching `*Start` event. Alternatively, use `e.time - e.metrics.durationMs` .
`Span.EndTime`	N/A, because the event hasn't completed yet.
`Span.Kind`	Set to `Server`.
`Span.Status`	If `e.status` doesn't equal `success`, then set to `Error`.

OpenTelemetry	Lambda Telemetry API schema
	Otherwise, set to 0k.
`Span.Events[]`	Use `e.spans[]` .
`Span.Events[i].Name`	Use `e.spans[i].name` .
`Span.Events[i].Time`	Use `e.spans[i].start` .
`Span.TraceId`	Parse the AWS X-Ray header found in `e.tracing.value` , then use the `TraceId` value.
`Span.ParentId`	Parse the X-Ray header found in `e.tracing.value` , then use the `Parent` value.
`Span.SpanId`	Use the same `SpanId` from the `*Start` event. If unavailable, then use `e.tracing.spanId` , or generate a new `SpanId`.
`Span.SpanContext.TraceState`	N/A for an X-Ray trace context.
`Span.SpanContext.TraceFlags`	Parse the X-Ray header found in `e.tracing.value` , then use the `Sampled` value.
`Span.Attributes`	Your extension can add any custom values here.

Mapping the *Report events

OpenTelemetry	Lambda Telemetry API schema
`Span.Name`	Your extension generates the value based on the `type` field.
`Span.StartTime`	Use `e.time` from the matching `*Start` event.

OpenTelemetry	Lambda Telemetry API schema
	Alternatively, use `e.time - e.metrics.durationMs` .
`Span.EndTime`	Use `e.time`.
`Span.Kind`	Set to `Server`.
`Span.Status`	Use the same value as the `*RuntimeDone` event.
`Span.TraceId`	Parse the AWS X-Ray header found in `e.tracing.value` , then use the `TraceId` value.
`Span.ParentId`	Parse the X-Ray header found in `e.tracing.value` , then use the `Parent` value.
`Span.SpanId`	Use the same `SpanId` from the `*Start` event. If unavailable, then use `e.tracing.spanId` , or generate a new `SpanId`.
`Span.SpanContext.TraceState`	N/A for an X-Ray trace context.
`Span.SpanContext.TraceFlags`	Parse the X-Ray header found in `e.tracing.value` , then use the `Sampled` value.
`Span.Attributes`	Your extension can add any custom values here.

Map to OTel Spans with Child Spans

The following table describes how to convert Lambda Telemetry API events into OTel Spans with Child (nested) Spans for `*RuntimeDone` Spans. For `*Start` and `*Report` mappings, refer to the tables in the section called “Map to OTel Spans with Span Events”, as they're the same for Child Spans. In this table, e represents the event coming from the telemetry source.

Mapping the *RuntimeDone events

OpenTelemetry	Lambda Telemetry API schema
`Span.Name`	Your extension generates the value based on the `type` field.
`Span.StartTime`	Use `e.time` from the matching `*Start` event. Alternatively, use `e.time - e.metrics.durationMs` .
`Span.EndTime`	N/A, because the event hasn't completed yet.
`Span.Kind`	Set to `Server`.
`Span.Status`	If `e.status` doesn't equal `success`, then set to `Error`. Otherwise, set to `Ok`.
`Span.TraceId`	Parse the AWS X-Ray header found in `e.tracing.value` , then use the `TraceId` value.
`Span.ParentId`	Parse the X-Ray header found in `e.tracing.value` , then use the `Parent` value.
`Span.SpanId`	Use the same `SpanId` from the `*Start` event. If unavailable, then use `e.tracing.spanId` , or generate a new `SpanId`.
`Span.SpanContext.TraceState`	N/A for an X-Ray trace context.
`Span.SpanContext.TraceFlags`	Parse the X-Ray header found in `e.tracing.value` , then use the `Sampled` value.
`Span.Attributes`	Your extension can add any custom values here.
`ChildSpan[i].Name`	Use `e.spans[i].name` .

OpenTelemetry	Lambda Telemetry API schema
`ChildSpan[i].StartTime`	Use `e.spans[i].start` .
`ChildSpan[i].EndTime`	Use `e.spans[i].start + e.spans[i].durations` .
`ChildSpan[i].Kind`	Same as parent `Span.Kind` .
`ChildSpan[i].Status`	Same as parent `Span.Status` .
`ChildSpan[i].TraceId`	Same as parent `Span.TraceId` .
`ChildSpan[i].ParentId`	Use parent `Span.SpanId` .
`ChildSpan[i].SpanId`	Generate a new `SpanId`.
`ChildSpan[i].SpanContext.TraceState`	N/A for an X-Ray trace context.
`ChildSpan[i].SpanContext.TraceFlags`	Same as parent `Span.SpanContext.TraceFlags` .

Using the Lambda Logs API

Important

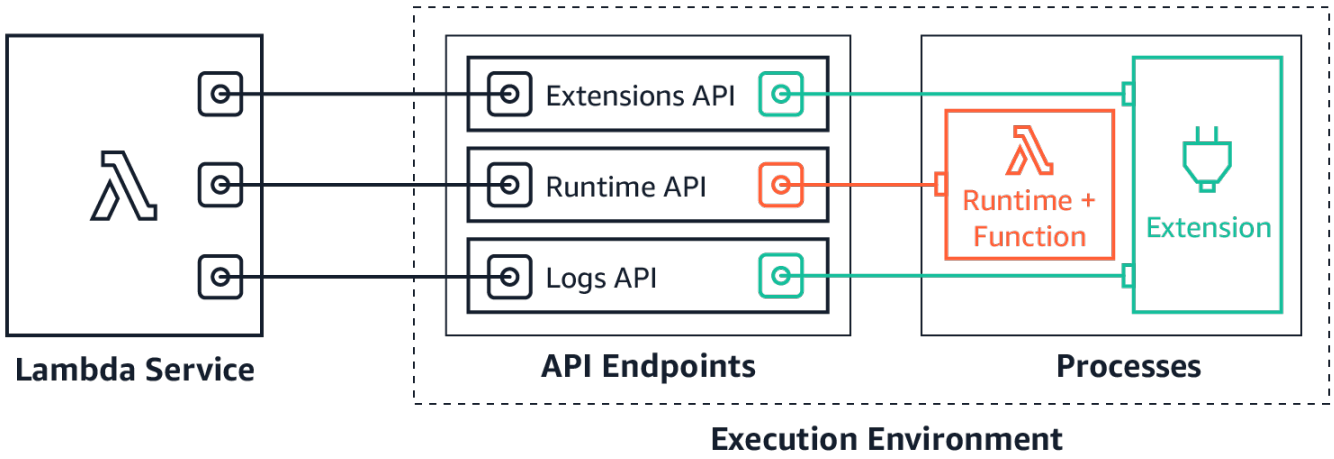
The Lambda Telemetry API supersedes the Lambda Logs API. **While the Logs API remains fully functional, we recommend using only the Telemetry API going forward.** You can subscribe your extension to a telemetry stream using either the Telemetry API or the Logs API. After subscribing using one of these APIs, any attempt to subscribe using the other API returns an error.

Lambda Managed Instances do not support Logs API

Lambda Managed Instances do not support the Logs API. If you are using Managed Instance functions, use the Telemetry API instead. The Telemetry API provides enhanced capabilities for collecting and processing telemetry data from your Lambda functions.

Lambda automatically captures runtime logs and streams them to Amazon CloudWatch. This log stream contains the logs that your function code and extensions generate, and also the logs that Lambda generates as part of the function invocation.

Lambda extensions can use the Lambda Runtime Logs API to subscribe to log streams directly from within the Lambda execution environment. Lambda streams the logs to the extension, and the extension can then process, filter, and send the logs to any preferred destination.



The Logs API allows extensions to subscribe to three different logs streams:

- Function logs that the Lambda function generates and writes to `stdout` or `stderr`.
- Extension logs that extension code generates.
- Lambda platform logs, which record events and errors related to invocations and extensions.

> **Note**
>
> Lambda sends all logs to CloudWatch, even when an extension subscribes to one or more of the log streams.

Topics

- Subscribing to receive logs
- Memory usage
- Destination protocols
- Buffering configuration
- Example subscription
- Sample code for Logs API
- Logs API reference
- Log messages

Subscribing to receive logs

A Lambda extension can subscribe to receive logs by sending a subscription request to the Logs API.

To subscribe to receive logs, you need the extension identifier (`Lambda-Extension-Identifier`). First register the extension to receive the extension identifier. Then subscribe to the Logs API during initialization. After the initialization phase completes, Lambda does not process subscription requests.

> **Note**
>
> Logs API subscription is idempotent. Duplicate subscribe requests do not result in duplicate subscriptions.

Memory usage

Memory usage increases linearly as the number of subscribers increases. Subscriptions consume memory resources because each subscription opens a new memory buffer to store the logs. To help optimize memory usage, you can adjust the buffering configuration. Buffer memory usage counts towards overall memory consumption in the execution environment.

Destination protocols

You can choose one of the following protocols to receive the logs:

1. **HTTP** (recommended) – Lambda delivers logs to a local HTTP endpoint (`http://sandbox.localdomain:${PORT}/${PATH}`) as an array of records in JSON format. The `$PATH` parameter is optional. Note that only HTTP is supported, not HTTPS. You can choose to receive logs through PUT or POST.
2. **TCP** – Lambda delivers logs to a TCP port in Newline delimited JSON (NDJSON) format.

We recommend using HTTP rather than TCP. With TCP, the Lambda platform cannot acknowledge when it delivers logs to the application layer. Therefore, you might lose logs if your extension crashes. HTTP does not share this limitation.

We also recommend setting up the local HTTP listener or the TCP port before subscribing to receive logs. During setup, note the following:

- Lambda sends logs only to destinations that are inside the execution environment.
- Lambda retries the attempt to send the logs (with backoff) if there is no listener, or if the POST or PUT request results in an error. If the log subscriber crashes, it continues to receive logs after Lambda restarts the execution environment.
- Lambda reserves port 9001. There are no other port number restrictions or recommendations.

Buffering configuration

Lambda can buffer logs and deliver them to the subscriber. You can configure this behavior in the subscription request by specifying the following optional fields. Note that Lambda uses the default value for any field that you do not specify.

- **timeoutMs** – The maximum time (in milliseconds) to buffer a batch. Default: 1,000. Minimum: 25. Maximum: 30,000.
- **maxBytes** – The maximum size (in bytes) of the logs to buffer in memory. Default: 262,144. Minimum: 262,144. Maximum: 1,048,576.
- **maxItems** – The maximum number of events to buffer in memory. Default: 10,000. Minimum: 1,000. Maximum: 10,000.

During buffering configuration, note the following points:

- Lambda flushes the logs if any of the input streams are closed, for example, if the runtime crashes.
- Each subscriber can specify a different buffering configuration in their subscription request.
- Consider the buffer size that you need for reading the data. Expect to receive payloads as large as `2*maxBytes+metadata`, where `maxBytes` is configured in the subscribe request. For example, Lambda adds the following metadata bytes to each record:

```
{
"time": "2020-08-20T12:31:32.123Z",
"type": "function",
"record": "Hello World"
}
```

- If the subscriber cannot process incoming logs quickly enough, Lambda might drop logs to keep memory utilization bounded. To indicate the number of dropped records, Lambda sends a `platform.logsDropped` log. For more information, see the section called “Lambda: Not all of my function's logs appear”.

Example subscription

The following example shows a request to subscribe to the platform and function logs.

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs HTTP/1.1
```

```
{ "schemaVersion": "2020-08-15",
  "types": [
      "platform",
      "function"
    ],
  "buffering": {
      "maxItems": 1000,
      "maxBytes": 262144,
      "timeoutMs": 100
    },
  "destination": {
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080/lambda_logs"
  }
}
```

If the request succeeds, the subscriber receives an HTTP 200 success response.

```
HTTP/1.1 200 OK
"OK"
```

Sample code for Logs API

For sample code showing how to send logs to a custom destination, see Using AWS Lambda extensions to send logs to custom destinations on the AWS Compute Blog.

For Python and Go code examples showing how to develop a basic Lambda extension and subscribe to the Logs API, see AWS Lambda Extensions on the AWS Samples GitHub repository. For more information about building a Lambda extension, see the section called “Extensions API”.

Logs API reference

You can retrieve the Logs API endpoint from the `AWS_LAMBDA_RUNTIME_API` environment variable. To send an API request, use the prefix `2020-08-15/` before the API path. For example:

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

The OpenAPI specification for the Logs API version **2020-08-15** is available here: logs-api-request.zip

Subscribe

To subscribe to one or more of the log streams available in the Lambda execution environment, extensions send a Subscribe API request.

Path – `/logs`

Method – **PUT**

Body parameters

`destination` – See the section called "Destination protocols". Required: yes. Type: strings.

`buffering` – See the section called "Buffering configuration". Required: no. Type: strings.

`types` – An array of the types of logs to receive. Required: yes. Type: array of strings. Valid values: "platform", "function", "extension".

`schemaVersion` – Required: no. Default value: "2020-08-15". Set to "2021-03-18" for the extension to receive `platform.runtimeDone` messages.

Response parameters

The OpenAPI specifications for the subscription responses version **2020-08-15** are available for the HTTP and TCP protocols:

- HTTP: logs-api-http-response.zip
- TCP: logs-api-tcp-response.zip

Response codes

- 200 – Request completed successfully
- 202 – Request accepted. Response to a subscription request during local testing.
- 4XX – Bad Request
- 500 – Service error

If the request succeeds, the subscriber receives an HTTP 200 success response.

```
HTTP/1.1 200 OK
"OK"
```

If the request fails, the subscriber receives an error response. For example:

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

Log messages

The Logs API allows extensions to subscribe to three different logs streams:

- Function – Logs that the Lambda function generates and writes to `stdout` or `stderr`.
- Extension – Logs that extension code generates.
- Platform – Logs that the runtime platform generates, which record events and errors related to invocations and extensions.

Topics

- Function logs
- Extension logs
- Platform logs

Function logs

The Lambda function and internal extensions generate function logs and write them to `stdout` or `stderr`.

The following example shows the format of a function log message. { "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": "ERROR encountered. Stack trace:\n\my-function (line 10)\n" }

Extension logs

Extensions can generate extension logs. The log format is the same as for a function log.

Platform logs

Lambda generates log messages for platform events such as `platform.start`, `platform.end`, and `platform.fault`.

Optionally, you can subscribe to the **2021-03-18** version of the Logs API schema, which includes the `platform.runtimeDone` log message.

Example platform log messages

The following example shows the platform start and platform end logs. These logs indicate the invocation start time and invocation end time for the invocation that the requestId specifies.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.start",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.end",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
```

The **platform.initRuntimeDone** log message shows the status of the `Runtime init` sub-phase, which is part of the Init lifecyle phase. When `Runtime init` is successful, the runtime sends a `/next` runtime API request (for the `on-demand` and `provisioned-concurrency` initialization types) or `restore/next` (for the `snap-start` initialization type). The following example shows a successful **platform.initRuntimeDone** log message for the `snap-start` initialization type.

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

The **platform.initReport** log message shows how long the `Init` phase lasted and how many milliseconds you were billed for during this phase. When the initialization type is `provisioned-concurrency`, Lambda sends this message during invocation. When the initialization type is `snap-start`, Lambda sends this message after restoring the snapshot. The following example shows a **platform.initReport** log message for the `snap-start` initialization type.

```
{
```

```
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
          "durationMs":731.79,
          "billedDurationMs":732
          }
  }
}
```

The platform report log includes metrics about the invocation that the requestId specifies. The `initDurationMs` field is included in the log only if the invocation included a cold start. If AWS X-Ray tracing is active, the log includes X-Ray metadata. The following example shows a platform report log for an invocation that included a cold start.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.report",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56",
        "metrics": {"durationMs": 101.51,
            "billedDurationMs": 300,
            "memorySizeMB": 512,
            "maxMemoryUsedMB": 33,
            "initDurationMs": 116.67
        }
    }
}
```

The platform fault log captures runtime or execution environment errors. The following example shows a platform fault log message.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
}
```

> **Note**
>
> AWS is currently implementing changes to the Lambda service. Due to these changes, you may see minor differences between the structure and content of system log messages and trace segments emitted by different Lambda functions in your AWS account.
> One of the log outputs affected by this change is the platform fault log `"record"` field. The following examples show illustrative `"record"` fields in the old and new formats. The new style of fault log contains a more concise message
> These changes will be implemented during the coming weeks, and all functions in all AWS Regions except the China and GovCloud regions will transition to use the new-format log messages and trace segments.

Example platform fault log record (old style)

```
"record":"RequestId: ...\tError: Runtime exited with error: exit status
 255\nRuntime.ExitError"
```

Example platform fault log record (new style)

```
"record":"RequestId: ... Status: error\tErrorType: Runtime.ExitError"
```

Lambda generates a platform extension log when an extension registers with the extensions API. The following example shows a platform extension message.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
     }
}
```

Lambda generates a platform logs subscription log when an extension subscribes to the logs API. The following example shows a logs subscription message.

```
{
```

```
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

Lambda generates a platform logs dropped log when an extension is not able to process the number of logs that it is receiving. The following example shows a `platform.logsDropped` log message.

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsDropped",
    "record": {"reason": "Consumer seems to have fallen behind as it has not
 acknowledged receipt of logs.",
        "droppedRecords": 123,
        "droppedBytes" 12345
    }
}
```

The **platform.restoreStart** log message shows the time that the `Restore` phase started (`snap-start` initialization type only). Example:

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

The **platform.restoreReport** log message shows how long the `Restore` phase lasted and how many milliseconds you were billed for during this phase (`snap-start` initialization type only). Example:

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
```

```
            "billedDurationMs":13
        }
    }
}
```

Platform `runtimeDone` messages

If you set the schema version to "2021-03-18" in the subscribe request, Lambda sends a `platform.runtimeDone` message after the function invocation completes either successfully or with an error. The extension can use this message to stop all the telemetry collection for this function invocation.

The OpenAPI specification for the Log event type in schema version **2021-03-18** is available here: schema-2021-03-18.zip

Lambda generates the `platform.runtimeDone` log message when the runtime sends a `Next` or `Error` runtime API request. The `platform.runtimeDone` log informs consumers of the Logs API that the function invocation completes. Extensions can use this information to decide when to send all the telemetry collected during that invocation.

Examples

Lambda sends the `platform.runtimeDone` message after the runtime sends the NEXT request when the function invocation completes. The following examples show messages for each of the status values: success, failure, and timeout.

Example success message

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
}
```

Example failure message

```
{
   "time": "2021-02-04T20:00:05.123Z",
```

```
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
   }
}
```

Example timeout message

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "timeout"
  }
}
```

Example platform.restoreRuntimeDone message (snap-start initialization type only)

The **platform.restoreRuntimeDone** log message shows whether or not the `Restore` phase was successful. Lambda sends this message when the runtime sends a `restore/next` runtime API request. There are three possible statuses: success, failure, and timeout. The following example shows a successful **platform.restoreRuntimeDone** log message.

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

Troubleshooting issues in Lambda

The following topics provide troubleshooting advice for errors and issues that you might encounter when using the Lambda API, console, or tools. If you find an issue that is not listed here, you can use the **Feedback** button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the AWS Knowledge Center.

For more information about debugging and troubleshooting Lambda applications, see Debugging in Serverless Land.

Topics

- Troubleshoot configuration issues in Lambda
- Troubleshoot deployment issues in Lambda
- Troubleshoot invocation issues in Lambda
- Troubleshoot execution issues in Lambda
- Troubleshoot event source mapping issues in Lambda
- Troubleshoot networking issues in Lambda

Troubleshoot configuration issues in Lambda

Your function configuration settings can have an impact on the overall performance and behavior of your Lambda function. These may not cause actual function errors, but can cause unexpected timeouts and results.

The following topics provide troubleshooting advice for common issues that you might encounter related to Lambda function configuration settings.

Topics

- Memory configurations
- CPU-bound configurations
- Timeouts
- Memory leakage between invocations
- Asynchronous results returned to a later invocation

Memory configurations

You can configure a Lambda function to use between 128 MB and 10,240 MB of memory. By default, any function created in the console is assigned the smallest amount of memory. Many Lambda functions are performant at this lowest setting. However, if you are importing large code libraries or completing memory intensive tasks, 128 MB is not sufficient.

If your functions are running much slower than expected, the first step is to increase the memory setting. For memory-bound functions, this will resolve the bottleneck and may improve the performance of your function.

CPU-bound configurations

For compute-intensive operations, if your function experiences slower-than-expected performance, this may be due to your function being CPU-bound. In this case, the computational capacity of the function cannot keep pace with the work.

While Lambda doesn't allow you to modify CPU configuration directly, CPU is indirectly controlled via the memory settings. The Lambda service proportionally allocates more virtual CPU as you allocate more memory. At 1.8 GB memory, a Lambda function has an entire vCPU allocated, and above this level it has access to more than one vCPU core. At 10,240MB, it has 6 vCPUs available. In other words, you can improve performance by increasing the memory allocation, even if the function doesn’t use all of the memory.

Timeouts

Timeouts for Lambda functions can be set between 1 and 900 seconds (15 minutes). By default, the Lambda console sets this to 3 seconds. The timeout value is a safety valve that ensures functions do not run indefinitely. After the timeout value is reached, Lambda stops the function invocation.

If a timeout value is set close to the average duration of a function, this increases the risk that the function will time out unexpectedly. The duration of a function can vary based on the amount of data transfer and processing, and the latency of any services the function interacts with. Common causes of timeout include:

- When downloading data from S3 buckets or other data stores, the download is larger or takes longer than average.

- A function makes a request to another service, which takes longer to respond.
- The parameters provided to a function require more computational complexity in the function, which causes the invocation to take longer.

When testing your application, ensure that your tests accurately reflect the size and quantity of data, and realistic parameter values. Importantly, use datasets at the upper bounds of what is reasonably expected for your workload.

Additionally, implement upper-bound limits in your workload wherever practical. In this example, the application could use a maximum size limit for each file type. You can then test the performance of your application for a range of expected file sizes, up to and including the maximum limits.

Memory leakage between invocations

Global variables and objects stored in the INIT phase of a Lambda invocation retain their state between warm invocations. They are completely reset only when the execution environment is run for the first time (also known as a “cold start”). Any variables stored in the handler are destroyed when the handler exits. It’s best practice to use the INIT phase to set up database connections, load libraries, create caches, and load immutable assets.

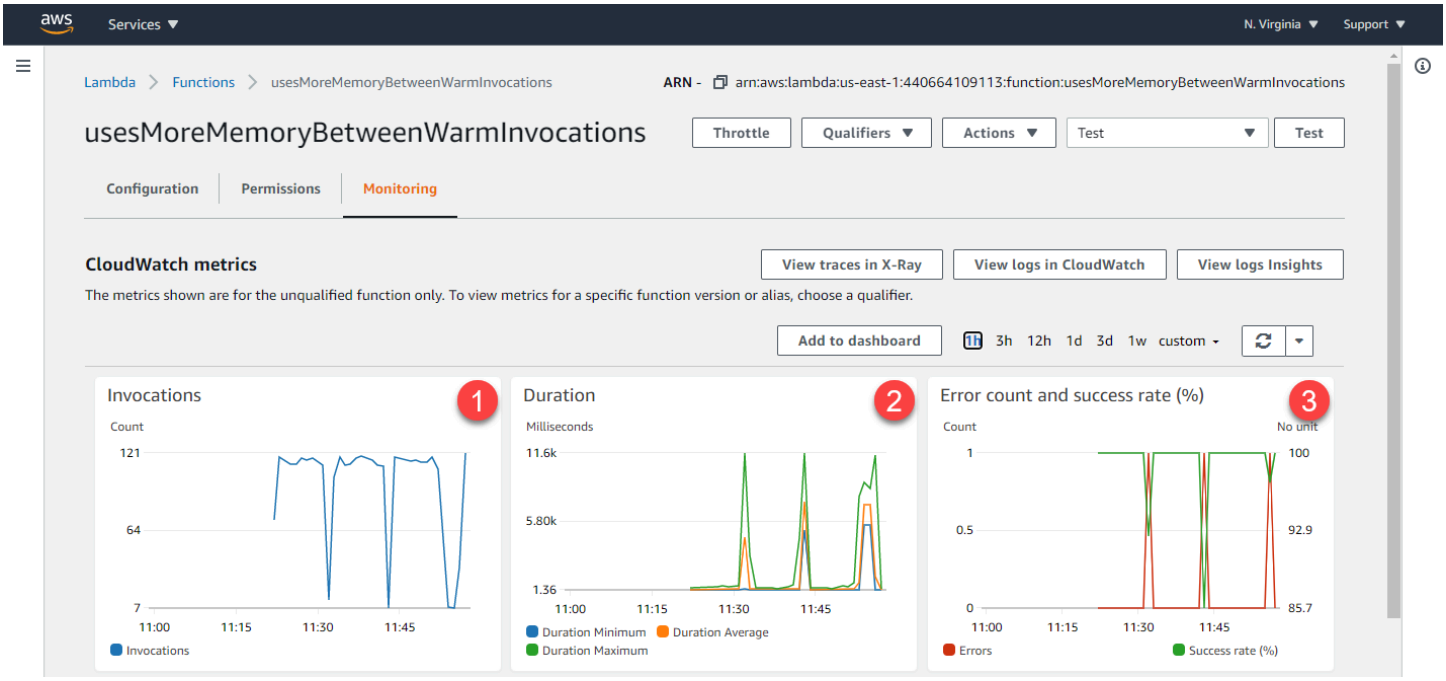
When you use third-party libraries across multiple invocations in the same execution environment, check their documentation for usage in a serverless compute environment. Some database connection and logging libraries may save intermediate invocation results and other data. This causes the memory usage of these libraries to grow with subsequent warm invocations. If this is the case, you may find the Lambda function runs out of memory, even if your custom code is disposing of variables correctly.

This issue affects invocations occurring in warm execution environments. For example, the following code creates a memory leak between invocations. The Lambda function consumes additional memory with each invocation by increasing the size of a global array:

```
let a = []

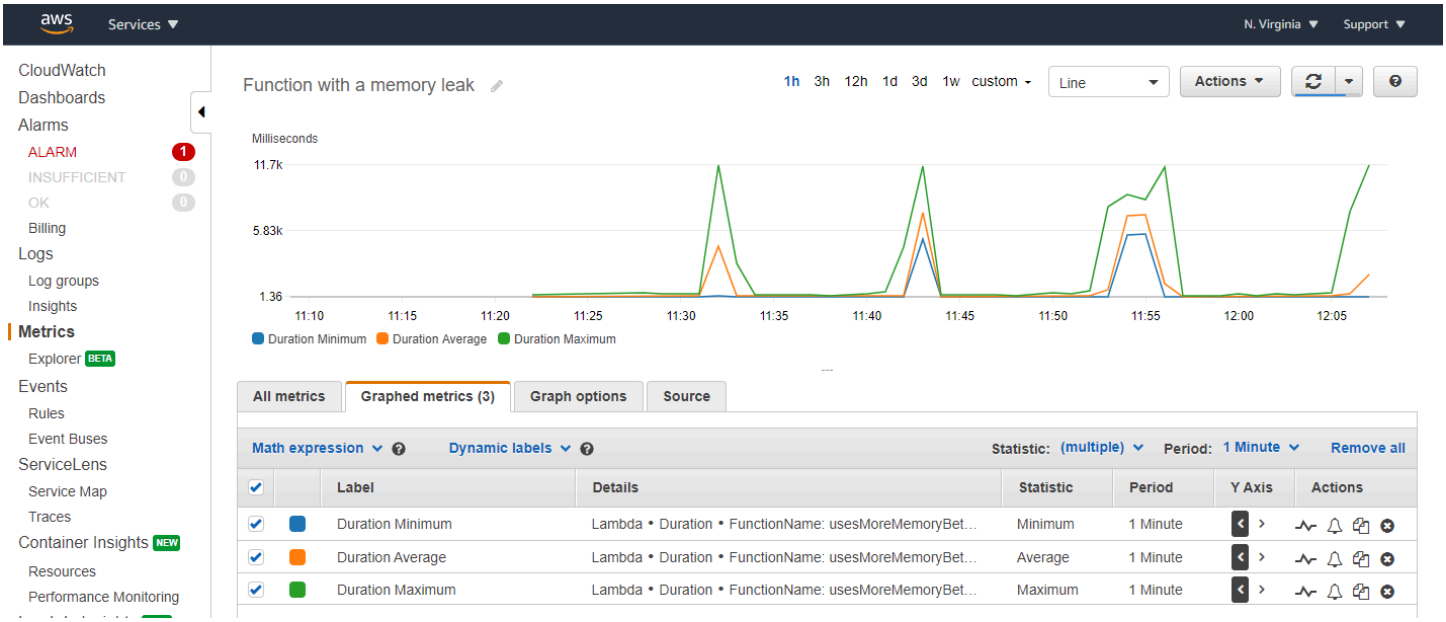
exports.handler = async (event) => {
    a.push(Array(100000).fill(1))
}
```

Configured with 128 MB of memory, after invoking this function 1000 times, the **Monitoring** tab of the Lambda function shows the typical changes in invocations, duration, and error counts when a memory leak occurs:



1. **Invocations** – A steady transaction rate is interrupted periodically as the invocations take longer to complete. During the steady state, the memory leak is not consuming all of the function's allocated memory. As performance degrades, the operating system is paging local storage to accommodate the growing memory required by the function, which results in fewer transactions being completed.
2. **Duration** – Before the function runs out of memory, it finishes invocations at a steady double-digit millisecond rate. As paging occurs, the duration takes an order of magnitude longer.
3. **Error count** – As the memory leak exceeds allocated memory, eventually the function errors due to the computation exceeding the timeout, or the execution environment stops the function.

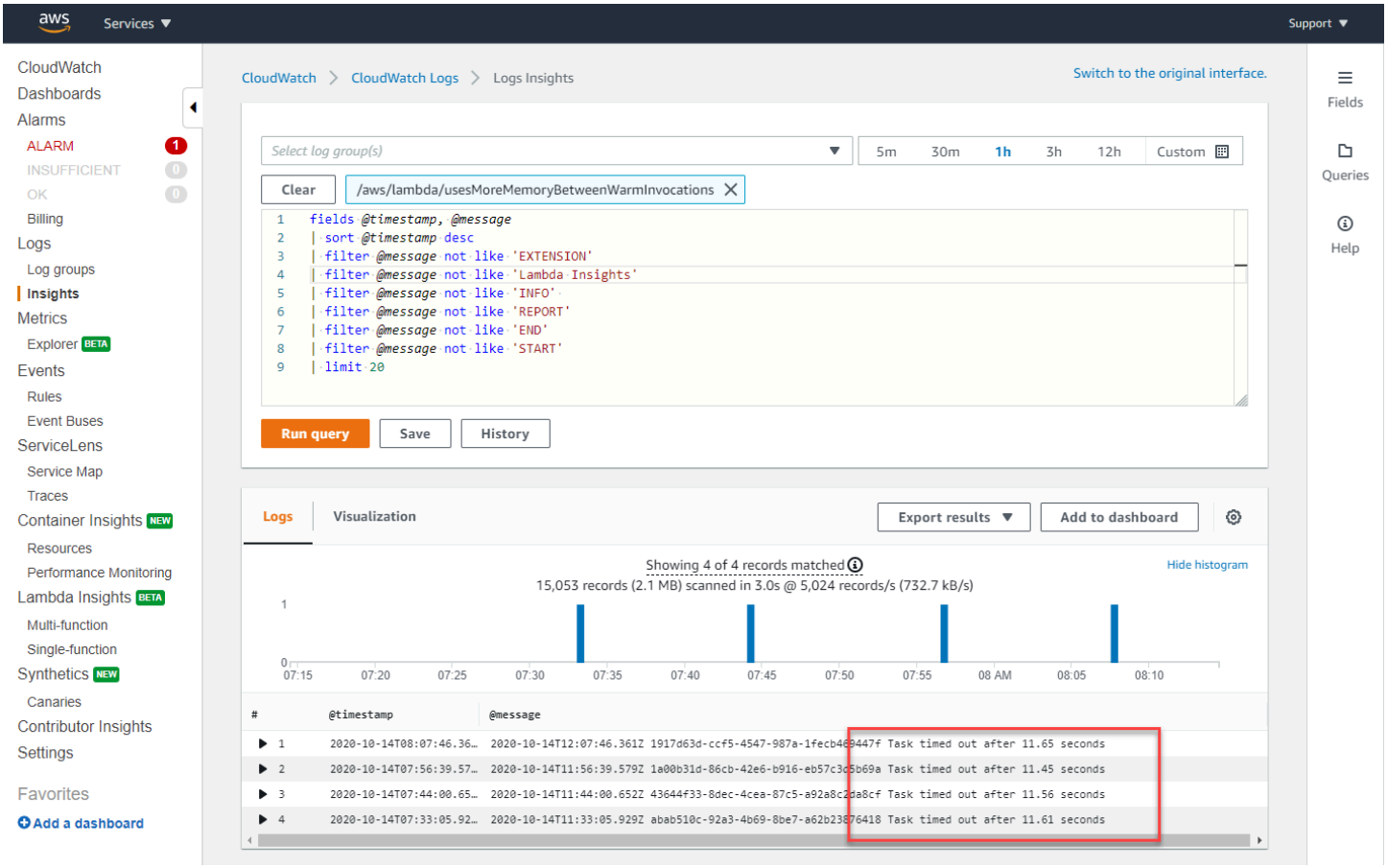
After the error, Lambda restarts the execution environment, which explains why all three graphs show a return to the original state. Expanding the CloudWatch metrics for duration provides more detail for the minimum, maximum and average duration statistics:



To find the errors generated across the 1000 invocations, you can use the CloudWatch Insights query language. The following query excludes informational logs to report only the errors:

```
fields @timestamp, @message
| sort @timestamp desc
| filter @message not like 'EXTENSION'
| filter @message not like 'Lambda Insights'
| filter @message not like 'INFO'
| filter @message not like 'REPORT'
| filter @message not like 'END'
| filter @message not like 'START'
```

When run against the log group for this function, this shows that timeouts were responsible for the periodic errors:



Asynchronous results returned to a later invocation

For function code that uses asynchronous patterns, it's possible for the callback results from one invocation to be returned in a future invocation. This example uses Node.js, but the same logic can apply to other runtimes using asynchronous patterns. The function uses the traditional callback syntax in JavaScript. It calls an asynchronous function with an incremental counter that tracks the number of invocations:

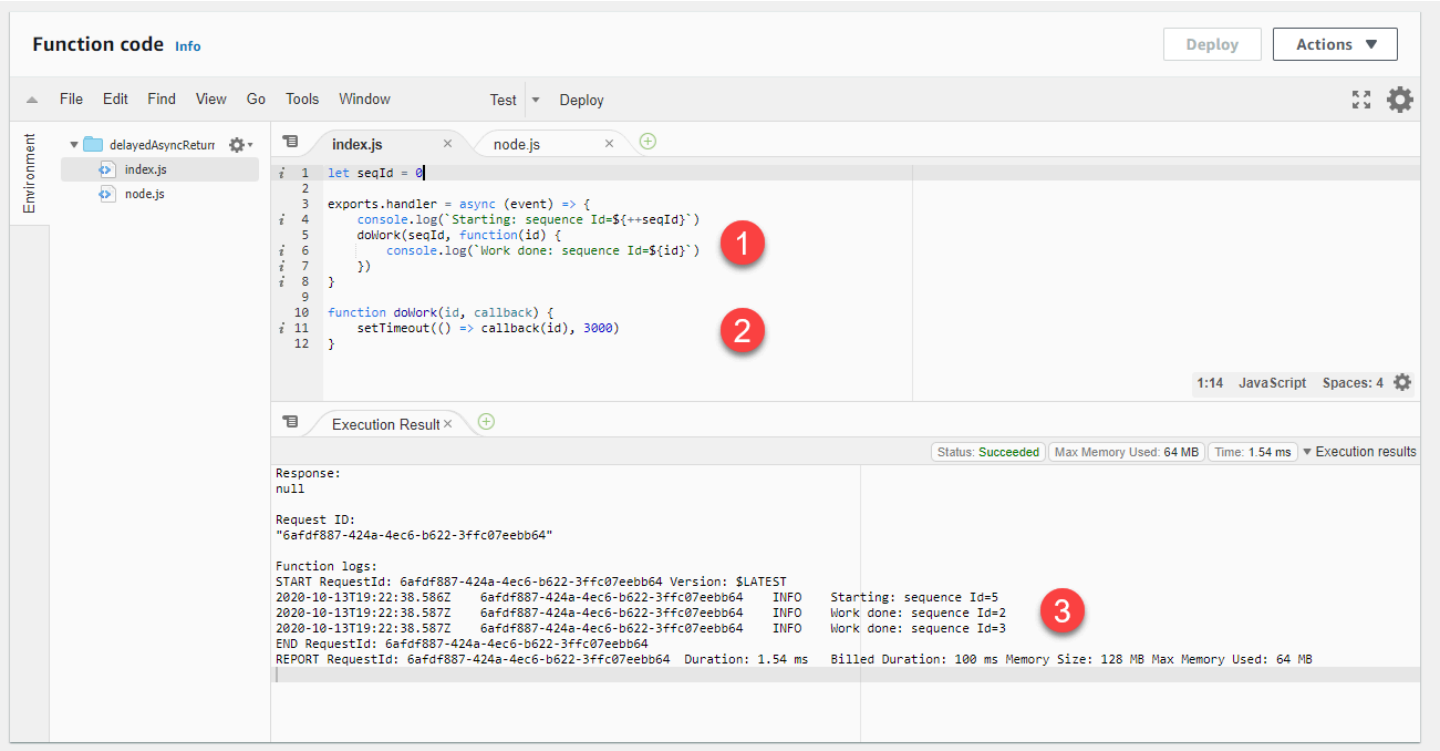
```
let seqId = 0

exports.handler = async (event, context) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    doWork(seqId, function(id) {
        console.log(`Work done: sequence Id=${id}`)
    })
}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)
```

```
}
```

When invoked several times in succession, the results of the callbacks occur in subsequent invocations:



1. The code calls the `doWork` function, providing a callback function as the last parameter.
2. The `doWork` function takes some period of time to complete before invoking the callback.
3. The function's logging indicates that the invocation is ending before the `doWork` function finishes execution. Additionally, after starting an iteration, callbacks from previous iterations are being processed, as shown in the logs.

In JavaScript, asynchronous callbacks are handled with an event loop. Other runtimes use different mechanisms to handle concurrency. When the function's execution environment ends, Lambda freezes the environment until the next invocation. After it resumes, JavaScript continues processing the event loop, which in this case includes an asynchronous callback from a previous invocation. Without this context, it can appear that the function is running code for no reason, and returning arbitrary data. In fact, it is really an artifact of how runtime concurrency and the execution environments interact.

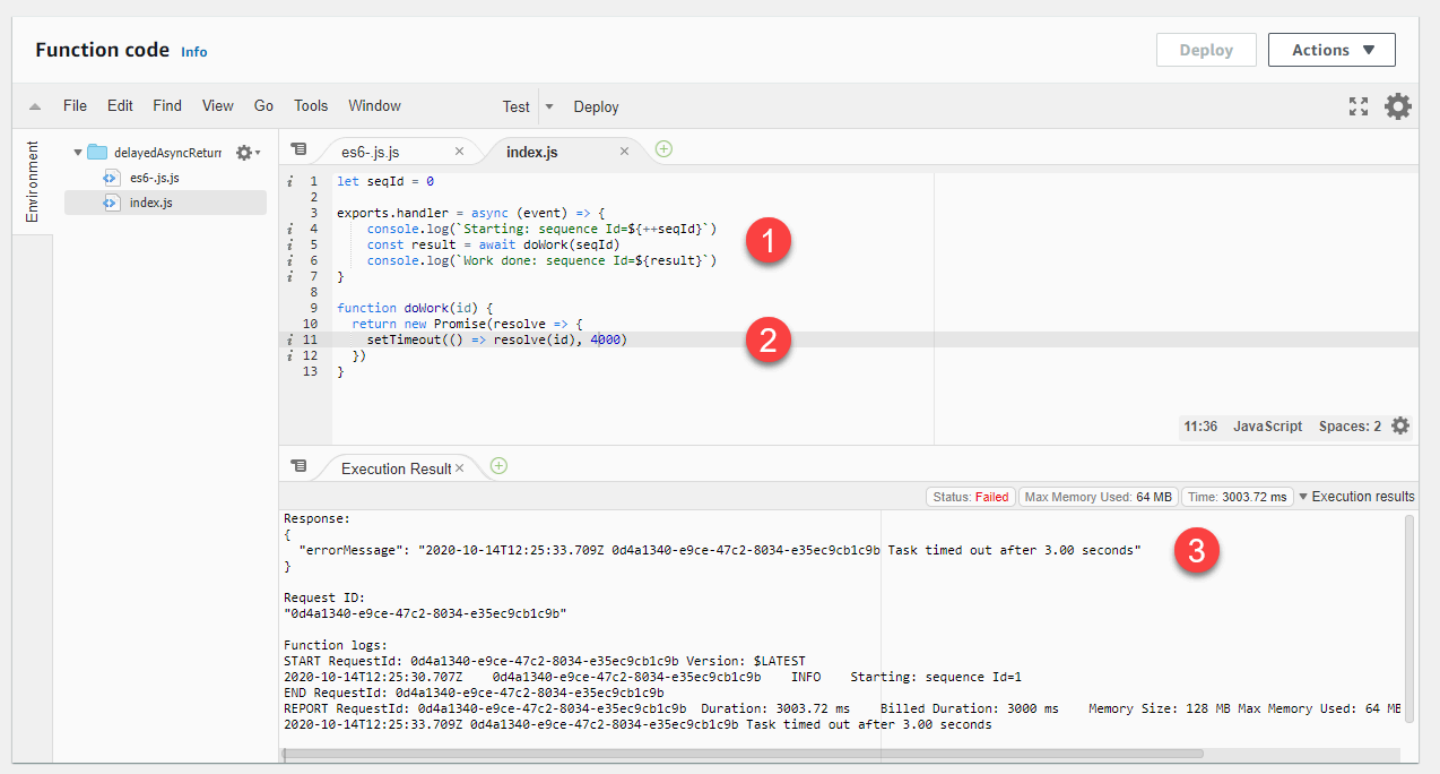
This creates the potential for private data from a previous invocation to appear in a subsequent invocation. There are two ways to prevent or detect this behavior. First, JavaScript provides the

async and await keywords to simplify asynchronous development and also force code execution to wait for an asynchronous call to complete. The function above can be rewritten using this approach as follows:

```
let seqId = 0
exports.handler = async (event) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    const result = await doWork(seqId)
    console.log(`Work done: sequence Id=${result}`)
}

function doWork(id) {
  return new Promise(resolve => {
    setTimeout(() => resolve(id), 4000)
  })
}
```

Using this syntax prevents the handler from exiting before the asynchronous function is finished. In this case, if the callback takes longer than the Lambda function's timeout, the function will throw an error, instead of returning the callback result in a later invocation:



1. The code calls the asynchronous `doWork` function using the await keyword in the handler.

2. The `doWork` function takes some period of time to complete before resolving the promise.
3. The function times out because `doWork` takes longer than the timeout limit allows and the callback result is not returned in a later invocation.

Generally, you should make sure any background processes or callbacks in the code are complete before the code exits. If this is not possible in your use case, you can use an identifier to ensure that the callback belongs to the current invocation. To do this, you can use the *awsRequestId* provided by the context object. By passing this value to the asynchronous callback, you can compare the passed value with the current value to detect if the callback originated from another invocation:

```
let currentContext

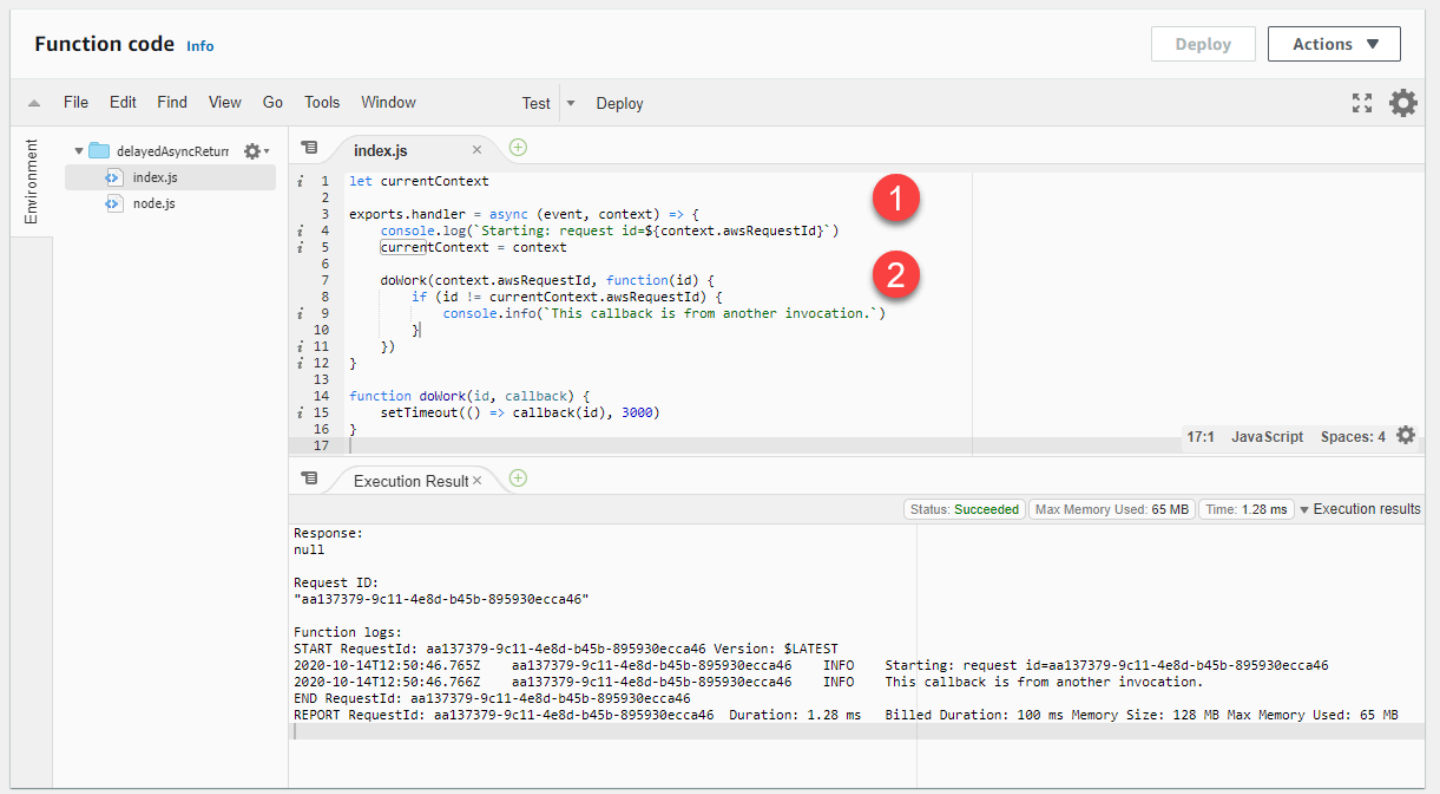
exports.handler = async (event, context) => {
    console.log(`Starting: request id=$\{context.awsRequestId}`)
    currentContext = context

    doWork(context.awsRequestId, function(id) {
        if (id != currentContext.awsRequestId) {
            console.info(`This callback is from another invocation.`)
        }
    })

}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)

}
```



1. The Lambda function handler takes the context parameter, which provides access to a unique invocation request ID.
2. The `awsRequestId` is passed to the doWork function. In the callback, the ID is compared with the `awsRequestId` of the current invocation. If these values are different, the code can take action accordingly.

Troubleshoot deployment issues in Lambda

When you update your function, Lambda deploys the change by launching new instances of the function with the updated code or settings. Deployment errors prevent the new version from being used and can arise from issues with your deployment package, code, permissions, or tools.

When you deploy updates to your function directly with the Lambda API or with a client such as the AWS CLI, you can see errors from Lambda directly in the output. If you use services like AWS CloudFormation, AWS CodeDeploy, or AWS CodePipeline, look for the response from Lambda in the logs or event stream for that service.

The following topics provide troubleshooting advice for errors and issues that you might encounter when using the Lambda API, console, or tools. If you find an issue that is not listed here, you can use the **Feedback** button on this page to report it.

For more troubleshooting advice and answers to common support questions, visit the AWS Knowledge Center.

For more information about debugging and troubleshooting Lambda applications, see Debugging in Serverless Land.

Topics

- General: Permission is denied / Cannot load such file
- General: Error occurs when calling the UpdateFunctionCode
- Amazon S3: Error Code PermanentRedirect.
- General: Cannot find, cannot load, unable to import, class not found, no such file or directory
- General: Undefined method handler
- General: Lambda code storage limit exceeded
- Lambda: Layer conversion failed
- Lambda: InvalidParameterValueException or RequestEntityTooLargeException
- Lambda: InvalidParameterValueException
- Lambda: Concurrency and memory quotas
- Lambda: Invalid alias configuration for provisioned concurrency

General: Permission is denied / Cannot load such file

Error: *EACCES: permission denied, open '/var/task/index.js'*

Error: *cannot load such file -- function*

Error: *[Errno 13] Permission denied: '/var/task/function.py'*

The Lambda runtime needs permission to read the files in your deployment package. In Linux permissions octal notation, Lambda needs 644 permissions for non-executable files (rw-r--r--) and 755 permissions (rwxr-xr-x) for directories and executable files.

In Linux and MacOS, use the chmod command to change file permissions on files and directories in your deployment package. For example, to give a non-executable file the correct permissions, run the following command.

```
chmod 644 <filepath>
```

To change file permissions in Windows, see Set, View, Change, or Remove Permissions on an Object in the Microsoft Windows documentation.

> **Note**
>
> If you don't grant Lambda the permissions it needs to access directories in your deployment package, Lambda sets the permissions for those directories to 755 (rwxr-xr-x).

General: Error occurs when calling the UpdateFunctionCode

Error: *An error occurred (RequestEntityTooLargeException) when calling the UpdateFunctionCode operation*

When you upload a deployment package or layer archive directly to Lambda, the size of the ZIP file is limited to 50 MB. To upload a larger file, store it in Amazon S3 and use the S3Bucket and S3Key parameters.

> **Note**
>
> When you upload a file directly with the AWS CLI, AWS SDK, or otherwise, the binary ZIP file is converted to base64, which increases its size by about 30%. To allow for this, and the size of other parameters in the request, the actual request size limit that Lambda applies is larger. Due to this, the 50 MB limit is approximate.

Amazon S3: Error Code PermanentRedirect.

Error: *Error occurred while GetObject. S3 Error Code: PermanentRedirect. S3 Error Message: The bucket is in this region: us-east-2. Please use this region to retry the request*

When you upload a function's deployment package from an Amazon S3 bucket, the bucket must be in the same Region as the function. This issue can occur when you specify an Amazon S3 object in a

call to UpdateFunctionCode, or use the package and deploy commands in the AWS CLI or AWS SAM CLI. Create a deployment artifact bucket for each Region where you develop applications.

General: Cannot find, cannot load, unable to import, class not found, no such file or directory

Error: *Cannot find module 'function'*

Error: *cannot load such file -- function*

Error: *Unable to import module 'function'*

Error: *Class not found: function.Handler*

Error: *fork/exec /var/task/function: no such file or directory*

Error: *Unable to load type 'Function.Handler' from assembly 'Function'.*

The name of the file or class in your function's handler configuration doesn't match your code. See the following section for more information.

General: Undefined method handler

Error: *index.handler is undefined or not exported*

Error: *Handler 'handler' missing on module 'function'*

Error: *undefined method `handler' for #<LambdaHandler:0x000055b76ccebf98>*

Error: *No public method named handleRequest with appropriate method signature found on class function.Handler*

Error: *Unable to find method 'handleRequest' in type 'Function.Handler' from assembly 'Function'*

The name of the handler method in your function's handler configuration doesn't match your code. Each runtime defines a naming convention for handlers, such as *`filename.methodname`*. The handler is the method in your function's code that the runtime runs when your function is invoked.

For some languages, Lambda provides a library with an interface that expects a handler method to have a specific name. For details about handler naming for each language, see the following topics.

- Building Lambda functions with Node.js

- Building Lambda functions with Python
- Building Lambda functions with Ruby
- Building Lambda functions with Java
- Building Lambda functions with Go
- Building Lambda functions with C#
- Building Lambda functions with PowerShell

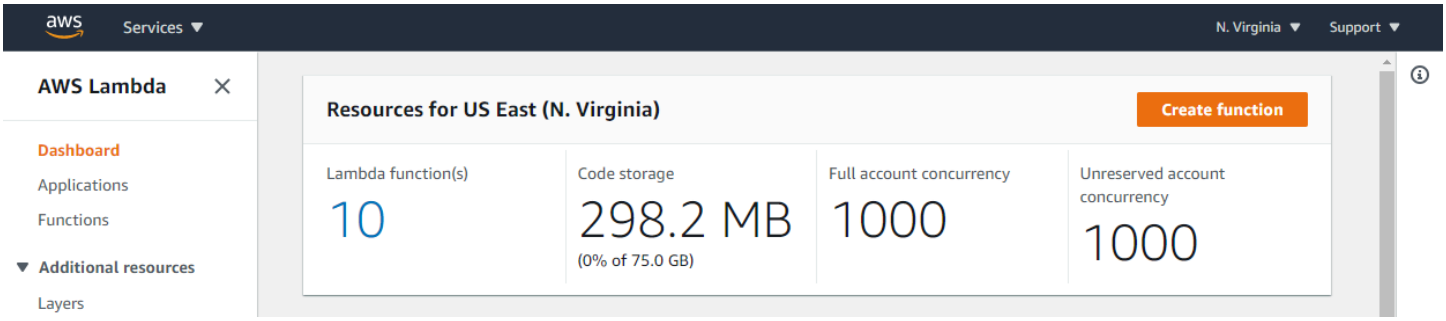
General: Lambda code storage limit exceeded

Error: *Code storage limit exceeded.*

Lambda stores your function code in an internal S3 bucket that's private to your account. Each AWS account is allocated 75 GB of storage in each Region. Code storage includes the total storage used by both Lambda functions and layers. If you reach the quota, you receive a *CodeStorageExceededException* when you attempt to deploy new functions.

Manage the storage space available by cleaning up old versions of functions, removing unused code, or using Lambda layers. In addition, it's good practice to use separate AWS accounts for separate workloads to help manage storage quotas.

You can view your total storage usage in the Lambda console, under the **Dashboard** submenu:



Lambda: Layer conversion failed

Error: *Lambda layer conversion failed. For advice on resolving this issue, see the Troubleshoot deployment issues in Lambda page in the Lambda User Guide.*

When you configure a Lambda function with a layer, Lambda merges the layer with your function code. If this process fails to complete, Lambda returns this error. If you encounter this error, take the following steps:

- Delete any unused files from your layer
- Delete any symbolic links in your layer
- Rename any files that have the same name as a directory in any of your function's layers

Lambda: InvalidParameterValueException or RequestEntityTooLargeException

Error: *InvalidParameterValueException: Lambda was unable to configure your environment variables because the environment variables you have provided exceeded the 4KB limit. String measured: {"A1":"uSFeY5cyPiPn7AtnX5BsM...*

Error: *RequestEntityTooLargeException: Request must be smaller than 5120 bytes for the UpdateFunctionConfiguration operation*

The maximum size of the variables object that is stored in the function's configuration must not exceed 4096 bytes. This includes key names, values, quotes, commas, and brackets. The total size of the HTTP request body is also limited.

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    ...
}
```

In this example, the object is 39 characters and takes up 39 bytes when it's stored (without white space) as the string `{"BUCKET":"amzn-s3-demo-bucket","KEY":"file.txt"}`. Standard ASCII characters in environment variable values use one byte each. Extended ASCII and Unicode characters can use between 2 bytes and 4 bytes per character.

Lambda: InvalidParameterValueException

Error: *InvalidParameterValueException: Lambda was unable to configure your environment variables because the environment variables you have provided contains reserved keys that are currently not supported for modification.*

Lambda reserves some environment variable keys for internal use. For example, `AWS_REGION` is used by the runtime to determine the current Region and cannot be overridden. Other variables, like `PATH`, are used by the runtime but can be extended in your function configuration. For a full list, see Defined runtime environment variables.

Lambda: Concurrency and memory quotas

Error: *Specified ConcurrentExecutions for function decreases account's UnreservedConcurrentExecution below its minimum value*

Error: *'MemorySize' value failed to satisfy constraint: Member must have value less than or equal to 3008*

These errors occur when you exceed the concurrency or memory quotas for your account. New AWS accounts have reduced concurrency and memory quotas. To resolve errors related to concurrency, you can request a quota increase. You cannot request memory quota increases.

- **Concurrency:** You might get an error if you try to create a function using reserved or provisioned concurrency, or if your per-function concurrency request (PutFunctionConcurrency) exceeds your account's concurrency quota.
- **Memory:** Errors occur if the amount of memory allocated to the function exceeds your account's memory quota.

Lambda: Invalid alias configuration for provisioned concurrency

Error: *Invalid alias configuration for provisioned concurrency*

This error occurs when you try to update a Lambda function's code or configuration while an alias with provisioned concurrency is pointing to a version that has issues. Lambda pre-initializes execution environments for provisioned concurrency, and if these environments can't be properly initialized due to code errors, resource constraints, or affected stack and alias, the deployment fails. If you encounter this issue, take the following steps:

1. **Roll back the alias:** Temporarily update the alias to point to the previously working version.

```
aws lambda update-alias \
 --function-name <function-name> \
 --name <alias-name> \
 --function-version <known-good-version>
```

2. **Fix Lambda initialization code:** Ensure the initialization code that runs outside the handler doesn't have any uncaught exceptions and initialize the clients and connections.
3. **Redeploy safety:** Deploy fixed code and publish a new version. Then, update alias to point to the fixed version. Optionally, re-enable provisioned concurrency, if necessary.

If using AWS CloudFormation, update stack definition `FunctionVersion:!GetAtt version.Version` so that the alias points to the working version:

```
alias:
 Type: AWS::Lambda::Alias
 Properties:
 FunctionName: !Ref function
FunctionVersion: !GetAtt version.Version
 Name: BLUE
 ProvisionedConcurrencyConfig:
 ProvisionedConcurrentExecutions: 1
```

Troubleshoot invocation issues in Lambda

When you invoke a Lambda function, Lambda validates the request and checks for scaling capacity before sending the event to your function or, for asynchronous invocation, to the event queue. Invocation errors can be caused by issues with request parameters, event structure, function settings, user permissions, resource permissions, or limits.

If you invoke your function directly, you see any invocation errors in the response from Lambda. If you invoke your function asynchronously with an event source mapping or through another service, you might find errors in logs, a dead-letter queue, or a failed-event destination. Error handling options and retry behavior vary depending on how you invoke your function and on the type of error.

For a list of error types that the `Invoke` operation can return, see Invoke.

Topics

- Lambda: Function times out during Init phase (Sandbox.Timedout)
- IAM: lambda:InvokeFunction not authorized
- Lambda: Couldn't find valid bootstrap (Runtime.InvalidEntrypoint)
- Lambda: Operation cannot be performed ResourceConflictException
- Lambda: Function is stuck in Pending
- Lambda: One function is using all concurrency
- General: Cannot invoke function with other accounts or services
- General: Function invocation is looping
- Lambda: Alias routing with provisioned concurrency
- Lambda: Cold starts with provisioned concurrency
- Lambda: Cold starts with new versions
- Lambda: Unexpected Node.js exit in runtime (Runtime.NodejsExit)
- EFS: Function could not mount the EFS file system
- EFS: Function could not connect to the EFS file system
- EFS: Function could not mount the EFS file system due to timeout
- Lambda: Lambda detected an IO process that was taking too long
- Container: CodeArtifactUserException errors
- Container: InvalidEntrypoint errors

Lambda: Function times out during Init phase (Sandbox.Timedout)

Error: *Task timed out after 3.00 seconds*

When the Init phase times out, Lambda initializes the execution environment again by re-running the `Init` phase when the next invoke request arrives. This is called a suppressed init. However, if your function is configured with a short timeout duration (generally around 3 seconds), the suppressed init might not complete during the allocated timeout duration, causing the `Init` phase to time out again. Alternatively, the suppressed init completes but does not leave enough time for the Invoke phase to complete, causing the `Invoke` phase to time out.

To reduce timeout errors, use one or more of the following strategies:

- **Increase the function timeout duration**: Extend the timeout to give the `Init` and `Invoke` phases time to complete successfully.
- **Increase the function memory allocation**: More memory also means more proportional CPU allocation, which can speed up both the `Init` and `Invoke` phases.
- **Optimize the function initialization code**: Reduce the time needed for initialization to ensure that the the `Init` and `Invoke` phase can complete within the configured timeout.

IAM: lambda:InvokeFunction not authorized

Error: *User: arn:aws:iam::123456789012:user/developer is not authorized to perform: lambda:InvokeFunction on resource: my-function*

Your user, or the role that you assume, must have permission to invoke a function. This requirement also applies to Lambda functions and other compute resources that invoke functions. Add the AWS managed policy **AWSLambdaRole** to your user, or add a custom policy that allows the `lambda:InvokeFunction` action on the target function.

> **Note**
>
> The name of the IAM action (`lambda:InvokeFunction`) refers to the `Invoke` Lambda API operation.

For more information, see Managing permissions in AWS Lambda .

Lambda: Couldn't find valid bootstrap (Runtime.InvalidEntrypoint)

Error: *Couldn't find valid bootstrap(s): [/var/task/bootstrap /opt/bootstrap]*

This error typically occurs when the root of your deployment package doesn't contain an executable file named `bootstrap`. For example, if you're deploying a `provided.al2023` function with a .zip file, the `bootstrap` file must be at the root of the .zip file, not in a directory.

Lambda: Operation cannot be performed ResourceConflictException

Error: *ResourceConflictException: The operation cannot be performed at this time. The function is currently in the following state: Pending*

When you connect a function to a virtual private cloud (VPC) at the time of creation, the function enters a `Pending` state while Lambda creates elastic network interfaces. During this time, you can't invoke or modify your function. If you connect your function to a VPC after creation, you can invoke it while the update is pending, but you can't modify its code or configuration.

For more information, see Lambda function states .

Lambda: Function is stuck in Pending

Error: *A function is stuck in the `Pending` state for several minutes.*

If a function is stuck in the `Pending` state for more than six minutes, call one of the following API operations to unblock it:

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda cancels the pending operation and puts the function into the `Failed` state. You can then attempt another update.

Lambda: One function is using all concurrency

Issue: *One function is using all of the available concurrency, causing other functions to be throttled.*

To divide your AWS account's available concurrency in an AWS Region into pools, use reserved concurrency. Reserved concurrency ensures that a function can always scale to its assigned concurrency, and that it doesn't scale beyond its assigned concurrency.

General: Cannot invoke function with other accounts or services

Issue: *You can invoke your function directly, but it doesn't run when another service or account invokes it.*

You grant other services and accounts permission to invoke a function in the function's resource-based policy. If the invoker is in another account, that user must also have permission to invoke functions.

General: Function invocation is looping

Issue: *Function is invoked continuously in a loop.*

This typically occurs when your function manages resources in the same AWS service that triggers it. For example, it's possible to create a function that stores an object in an Amazon Simple Storage Service (Amazon S3) bucket that's configured with a notification that invokes the function again. To stop the function from running, reduce the available concurrency to zero, which throttles all future invocations. Then, identify the code path or configuration error that caused the recursive invocation. Lambda automatically detects and stops recursive loops for some AWS services and SDKs. For more information, see the section called “Recursive loop detection”.

Lambda: Alias routing with provisioned concurrency

Issue: *Provisioned concurrency spillover invocations during alias routing.*

Lambda uses a simple probabilistic model to distribute the traffic between the two function versions. At low traffic levels, you might see a high variance between the configured and actual percentage of traffic on each version. If your function uses provisioned concurrency, you can avoid spillover invocations by configuring a higher number of provisioned concurrency instances during the time that alias routing is active.

Lambda: Cold starts with provisioned concurrency

Issue: *You see cold starts after enabling provisioned concurrency.*

When the number of concurrent executions on a function is less than or equal to the configured level of provisioned concurrency, there shouldn't be any cold starts. To help you confirm if provisioned concurrency is operating normally, do the following:

- Check that provisioned concurrency is enabled on the function version or alias.

  > **Note**
  >
  > Provisioned concurrency is not configurable on the unpublished version of the function ($LATEST).

- Ensure that your triggers invoke the correct function version or alias. For example, if you're using Amazon API Gateway, check that API Gateway invokes the function version or alias with provisioned concurrency, not $LATEST. To confirm that provisioned concurrency is being used,

you can check the ProvisionedConcurrencyInvocations Amazon CloudWatch metric. A non-zero value indicates that the function is processing invocations on initialized execution environments.

- Determine whether your function concurrency exceeds the configured level of provisioned concurrency by checking the ProvisionedConcurrencySpilloverInvocations CloudWatch metric. A non-zero value indicates that all provisioned concurrency is in use and some invocation occurred with a cold start.
- Check your invocation frequency (requests per second). Functions with provisioned concurrency have a maximum rate of 10 requests per second per provisioned concurrency. For example, a function configured with 100 provisioned concurrency can handle 1,000 requests per second. If the invocation rate exceeds 1,000 requests per second, some cold starts can occur.

Lambda: Cold starts with new versions

Issue: *You see cold starts while deploying new versions of your function.*

When you update a function alias, Lambda automatically shifts provisioned concurrency to the new version based on the weights configured on the alias.

Error: *KMSDisabledException: Lambda was unable to decrypt the environment variables because the KMS key used is disabled. Please check the function's KMS key settings.*

This error can occur if your AWS Key Management Service (AWS KMS) key is disabled, or if the grant that allows Lambda to use the key is revoked. If the grant is missing, configure the function to use a different key. Then, reassign the custom key to recreate the grant.

Lambda: Unexpected Node.js exit in runtime (Runtime.NodejsExit)

Issue: *Lambda runtime client detected an unexpected Node.js exit code.*

This error occurs when your function exits before all Promises are settled, for example due to a code bug. It can also occur when Node.js detects a deadlock that prevents Promises from being settled. This error affects only async style handlers, not callback-style handlers.

Affected runtimes: Node.js 18 and later.

To resolve this issue:

1. Check your function code for unsettled promises in async handlers.
2. Ensure all promises are properly settled (resolved or rejected) before the function completes.

3. Review your code for potential race conditions in asynchronous operations.

For more information about Node.js exit codes and process termination, see the Node.js documentation.

EFS: Function could not mount the EFS file system

Error: *EFSMountFailureException: The function could not mount the EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd.*

The mount request to the function's file system was rejected. Check the function's permissions, and confirm that its file system and access point exist and are ready for use.

EFS: Function could not connect to the EFS file system

Error: *EFSMountConnectivityException: The function couldn't connect to the Amazon EFS file system with access point arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd. Check your network configuration and try again.*

The function couldn't establish a connection to the function's file system with the NFS protocol (TCP port 2049). Check the security group and routing configuration for the VPC's subnets.

If you get these errors after updating your function's VPC configuration settings, try unmounting and remounting the file system.

EFS: Function could not mount the EFS file system due to timeout

Error: *EFSMountTimeoutException: The function could not mount the EFS file system with access point {arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd} due to mount time out.*

The function could connect to the function's file system, but the mount operation timed out. Try again after a short time and consider limiting the function's concurrency to reduce load on the file system.

Lambda: Lambda detected an IO process that was taking too long

EFSIOException: This function instance was stopped because Lambda detected an IO process that was taking too long.

A previous invocation timed out and Lambda couldn't terminate the function handler. This issue can occur when an attached file system runs out of burst credits and the baseline throughput is insufficient. To increase throughput, you can increase the size of the file system or use provisioned throughput.

Container: CodeArtifactUserException errors

Error: *CodeArtifactUserPendingException error message*

The CodeArtifact is pending optimization. The function transitions to the Active state when Lambda completes the optimization. HTTP response code 409.

Error: *CodeArtifactUserDeletedException error message*

The CodeArtifact is scheduled to be deleted. HTTP response code 409.

Error: *CodeArtifactUserFailedException error message*

Lambda failed to optimize the code. You need to correct the code and upload it again. HTTP response code 409.

Container: InvalidEntrypoint errors

Error: *Runtime.ExitError or "errorType": "Runtime.InvalidEntrypoint"*

Verify that the ENTRYPOINT to your container image includes the absolute path as the location. Also verify that the image does not contain a symlink as the ENTRYPOINT.

Error: *You are using an CloudFormation template, and your container ENTRYPOINT is being overridden with a null or empty value.*

Review the ImageConfig resource in the CloudFormation template. If you declare an `ImageConfig` resource in your template, you must provide non-empty values for all three of the properties.

Troubleshoot execution issues in Lambda

When the Lambda runtime runs your function code, the event might be processed on an instance of the function that's been processing events for some time, or it might require a new instance to be initialized. Errors can occur during function initialization, when your handler code processes the event, or when your function returns (or fails to return) a response.

Function execution errors can be caused by issues with your code, function configuration, downstream resources, or permissions. If you invoke your function directly, you see function errors in the response from Lambda. If you invoke your function asynchronously, with an event source mapping, or through another service, you might find errors in logs, a dead-letter queue, or an on-failure destination. Error handling options and retry behavior vary depending on how you invoke your function and on the type of error.

When your function code or the Lambda runtime return an error, the status code in the response from Lambda is 200 OK. The presence of an error in the response is indicated by a header named `X-Amz-Function-Error`. 400 and 500-series status codes are reserved for invocation errors.

Topics

- Lambda: Remote debugging with Visual Studio Code
- Lambda: Execution takes too long
- Lambda: Unexpected event payload
- Lambda: Unexpectedly large payload sizes
- Lambda: JSON encoding and decoding errors
- Lambda: Logs or traces don't appear
- Lambda: Not all of my function's logs appear
- Lambda: The function returns before execution finishes
- Lambda: Running an unintended function version or alias
- Lambda: Detecting infinite loops
- General: Downstream service unavailability
- AWS SDK: Versions and updates
- Python: Libraries load incorrectly
- Java: Your function takes longer to process events after updating to Java 17 from Java 11
- Kafka: Error handling and retry configuration issues

Lambda: Remote debugging with Visual Studio Code

Issue: *Difficulty troubleshooting complex Lambda function behavior in the actual AWS environment*

Lambda provides a remote debugging feature through the AWS Toolkit for Visual Studio Code. For set up and general instructions, see Remotely debug Lambda functions with Visual Studio Code.

For detailed instructions on troubleshooting, advanced use cases, and region availability, see Remote debugging Lambda functions in the AWS Toolkit for Visual Studio Code User Guide.

Lambda: Execution takes too long

Issue: *Function execution takes too long.*

If your code takes much longer to run in Lambda than on your local machine, it may be constrained by the memory or processing power available to the function. Configure the function with additional memory to increase both memory and CPU.

Lambda: Unexpected event payload

Issue: *Function errors related to malformed JSON or inadequate data validation.*

All Lambda functions receive an event payload in the first parameter of the handler. The event payload is a JSON structure that may contain arrays and nested elements.

Malformed JSON can occur when provided by upstream services that do not use a robust process for checking JSON structures. This occurs when services concatenate text strings or embed user input that has not been sanitized. JSON is also frequently serialized for passing between services. Always parse JSON structures both as the producer and consumer of JSON to ensure that the structure is valid.

Similarly, failing to check for ranges of values in the event payload can result in errors. This example shows a function that calculates a tax withholding:

```
exports.handler = async (event) => {
    let pct = event.taxPct
    let salary = event.salary

    // Calculate % of paycheck for taxes
    return (salary * pct)
}
```

This function uses a salary and tax rate from the event payload to perform the calculation. However, the code fails to check if the attributes are present. It also fails to check data types, or ensure boundaries, such as ensuring that the tax percentage is between 0 and 1. As a result, values outside of these bounds produce nonsensical results. An incorrect type or missing attribute causes a runtime error.

Create tests to ensure that your function handles larger payload sizes. The maximum size for a Lambda event payload is 1 MB. Depending upon the content, larger payloads may mean more items passed to the function or more binary data embedded in a JSON attribute. In both cases, this can result in more processing for a Lambda function.

Larger payloads can also cause timeouts. For example, a Lambda function processes one record per 100 ms and has a timeout of 3 seconds. Processing is successful for 0-29 items in the payload. However, once the payload contains more than 30 items, the function times out and throws an error. To avoid this, ensure that timeouts are set to handle the additional processing time for the maximum number of items expected.

Lambda: Unexpectedly large payload sizes

Issue: *Functions are timing out or causing errors due to large payloads.*

Larger payloads can cause timeouts and errors. We recommend creating tests to ensure that your function handles your largest expected payloads, and ensuring the function timeout is properly set.

In addition, certain event payloads can contain pointers to other resources. For example, a Lambda function with 128 MB of memory may perform image processing on a JPG file stored as an object in S3. The function works as expected with smaller image files. However, when a larger JPG file is provided as input, the Lambda function throws an error due to running out of memory. To avoid this, the test cases should include examples from the upper bounds of expected data sizes. The code should also validate payload sizes.

Lambda: JSON encoding and decoding errors

Issue: *NoSuchKey exception when parsing JSON inputs.*

Check to ensure you are processing JSON attributes correctly. For example, for events generated by S3, the `s3.object.key` attribute contains a URL encoded object key name. Many functions process this attribute as text to load the referenced S3 object:

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: event.Records[0].s3.object.key
}).promise()
```

This code works with the key name `james.jpg` but throws a `NoSuchKey` error when the name is `james beswick.jpg`. Since URL encoding converts spaces and other characters in a key name, you must ensure that functions decode keys before using this data:

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " "))
}).promise()
```

Lambda: Logs or traces don't appear

Issue: *Logs don't appear in CloudWatch Logs.*

Issue: *Traces don't appear in AWS X-Ray.*

Your function needs permission to call CloudWatch Logs and X-Ray. Update its execution role to grant it permission. Add the following managed policies to enable logs and tracing.

- **AWSLambdaBasicExecutionRole**
- **AWSXRayDaemonWriteAccess**

When you add permissions to your function, perform a trivial update to its code or configuration as well. This forces running instances of your function, which have outdated credentials, to stop and be replaced.

> **Note**
>
> It may take 5 to 10 minutes for logs to show up after a function invocation.

Lambda: Not all of my function's logs appear

Issue: *Function logs are missing in CloudWatch Logs, even though my permissions are correct*

If your AWS account reaches its CloudWatch Logs quota limits, CloudWatch throttles function logging. When this happens, some of the logs output by your functions may not appear in CloudWatch Logs.

If your function outputs logs at too high a rate for Lambda to process them, this can also cause log outputs not to appear in CloudWatch Logs. When Lambda can't send logs to CloudWatch at the rate your function produces them, it drops logs to prevent the execution of your function from slowing down. Expect to consistently observe dropped logs when your log throughput exceeds 2 MB/s for a single log stream.

If your function is configured to use JSON formatted logs, Lambda tries to send a `logsDropped` event to CloudWatch Logs when it drops logs. However, when CloudWatch throttles your function's logging, this event might not reach CloudWatch Logs, so you won't always see a record when Lambda drops logs.

To check if your AWS account has reached its CloudWatch Logs quota limits, do the following:

1. Open the Service Quotas console.
2. In the navigation pane, choose **AWS services**.
3. From the **AWS services** list, search for Amazon CloudWatch Logs.
4. In the **Service quotas** list, choose the `CreateLogGroup throttle limit in transactions per second`, `CreateLogStream throttle limit in transactions per second` and `PutLogEvents throttle limit in transactions per second` quotas to view your utilization.

You can also set CloudWatch alarms to alert you when your account utilization exceeds a limit you specify for these quotas. See Create a CloudWatch alarm based on a static threshold to learn more.

If the default quota limits for CloudWatch Logs aren't enough for your use case, you can request a quota increase.

Lambda: The function returns before execution finishes

Issue: (Node.js) *Function returns before code finishes executing*

Many libraries, including the AWS SDK, operate asynchronously. When you make a network call or perform another operation that requires waiting for a response, libraries return an object called a promise that tracks the progress of the operation in the background.

To wait for the promise to resolve into a response, use the `await` keyword. This blocks your handler code from executing until the promise is resolved into an object that contains the

response. If you don't need to use the data from the response in your code, you can return the promise directly to the runtime.

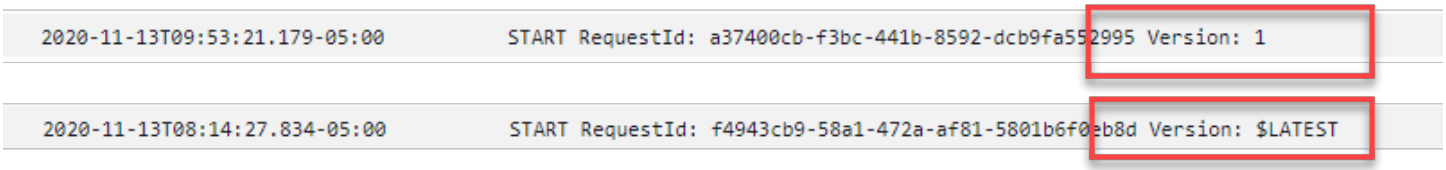
Some libraries don't return promises but can be wrapped in code that does. For more information, see Define Lambda function handler in Node.js.

Lambda: Running an unintended function version or alias

Issue: *Function version or alias not invoked*

When you publish new Lambda functions in the console or using AWS SAM, the latest code version is represented by $LATEST. By default, invocations that don't specify a version or alias automatically targets the $LATEST version of your function code.

If you use specific function versions or aliases, these are immutable published versions of a function in addition to $LATEST. When troubleshooting these functions, first determine that the caller has invoked the intended version or alias. You can do this by checking your function logs. The version of the function that was invoked is always shown in the START log line:

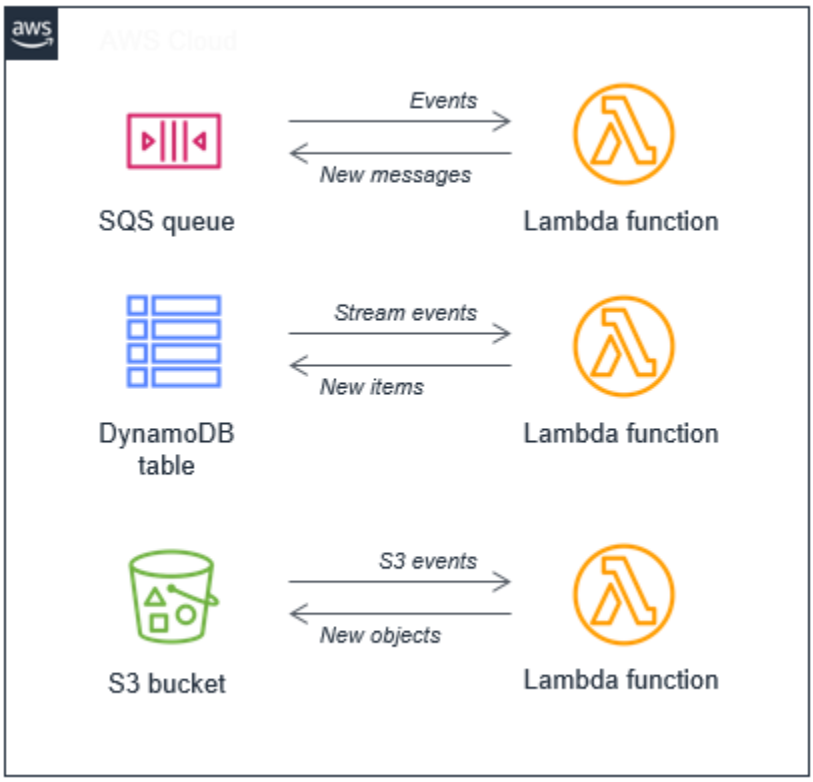


Lambda: Detecting infinite loops

Issue: *Infinite loop patterns related to Lambda functions*

There are two types of infinite loops in Lambda functions. The first is within the function itself, caused by a loop that never exits. The invocation ends only when the function times out. You can identify these by monitoring timeouts, and then fixing the looping behavior.

The second type of loop is between Lambda functions and other AWS resources. These occur when an event from a resource like an S3 bucket invokes a Lambda function, which then interacts with the same source resource to trigger another event. This invokes the function again, which creates another interaction with the same S3 bucket, and so on. These types of loops can be caused by a number of different AWS event sources, including Amazon SQS queues and DynamoDB tables. You can use recursive loop detection to identify these patterns.



You can avoid these loops by ensuring that Lambda functions write to resources that are not the same as the consuming resource. If you must publish data back to the consuming resource, ensure that the new data doesn't trigger the same event. Alternatively, use event filtering. For example, here are two proposed solutions to infinite loops with S3 and DynamoDB resources:

- If you write back to the same S3 bucket, use a different prefix or suffix from the event trigger.
- If you write items to the same DynamoDB table, include an attribute that a consuming Lambda function can filter on. If Lambda finds the attribute, it will not result in another invocation.

General: Downstream service unavailability

Issue: *Downstream services that your Lambda function relies on are unavailable*

For Lambda functions that call out to third-party endpoints or other downstream resources, ensure that they can handle service errors and timeouts. These downstream resources can have variable response times, or become unavailable due to service disruptions. Depending upon the implementation, these downstream errors may appear as Lambda timeouts or exceptions if the service's error response is not handled within the function code.

Anytime a function depends on a downstream service, such as an API call, implement appropriate error handling and retry logic. For critical services, the Lambda function should publish metrics or logs to CloudWatch. For example, if a third-party payment API becomes unavailable, your Lambda function can log this information. You can then set up CloudWatch alarms to send notifications related to these errors.

Since Lambda can scale quickly, non-serverless downstream services may struggle to handle spikes in traffic. There are three common approaches to handling this:

- **Caching** – Consider caching the result of values returned by third-party services if they don't change frequently. You can store these values in global variable in your function, or another service. For example, the results for a product list query from an Amazon RDS instance could be saved for a period of time within the function to prevent redundant queries.
- **Queuing** – When saving or updating data, add an Amazon SQS queue between the Lambda function and the resource. The queue durably persists data while the downstream service processes messages.
- **Proxies** – Where long-lived connections are typically used, such as for Amazon RDS instances, use a proxy layer to pool and reuse those connections. For relational databases, Amazon RDS Proxy is a service designed to help improve scalability and resiliency in Lambda-based applications.

AWS SDK: Versions and updates

Issue: *The AWS SDK included on the runtime is not the latest version*

Issue: *The AWS SDK included on the runtime updates automatically*

Runtimes for interpreted languages include a version of the AWS SDK. Lambda periodically updates these runtimes to use the latest SDK version. To find the version of the SDK that's included in your runtime, see the following sections:

- Runtime included SDK versions (Node.js)
- Runtime included SDK versions (Python)
- Runtime included SDK versions (Ruby)

To use a newer version of the AWS SDK, or to lock your functions to a specific version, you can bundle the library with your function code, or create a Lambda layer. For details on creating a deployment package with dependencies, see the following topics:

Node.js

- Deploy Node.js Lambda functions with .zip file archives

Python

- Working with .zip file archives for Python Lambda functions

Ruby

- Deploy Ruby Lambda functions with .zip file archives

Java

- Deploy Java Lambda functions with .zip or JAR file archives

Go

- Deploy Go Lambda functions with .zip file archives

C#

- Build and deploy C# Lambda functions with .zip file archives

PowerShell

- Deploy PowerShell Lambda functions with .zip file archives

Python: Libraries load incorrectly

Issue: (Python) *Some libraries don't load correctly from the deployment package*

Libraries with extension modules written in C or C++ must be compiled in an environment with the same processor architecture as Lambda (Amazon Linux). For more information, see Working with .zip file archives for Python Lambda functions.

Java: Your function takes longer to process events after updating to Java 17 from Java 11

Issue: (Java) *Your function takes longer to process events after updating to Java 17 from Java 11*

Tune your compiler using the `JAVA_TOOL_OPTIONS` parameter. Lambda runtimes for Java 17 and later Java versions change the default compiler options. The change improves cold start times

for short-lived functions, but the previous behavior is better suited to computationally intensive, longer-running functions. Set `JAVA_TOOL_OPTIONS` to `-XX:-TieredCompilation` to revert to the Java 11 behavior. For more information about the `JAVA_TOOL_OPTIONS` parameter, see the section called "Understanding the `JAVA_TOOL_OPTIONS` environment variable".

Kafka: Error handling and retry configuration issues

Issue: *Kafka event source mapping fails to configure retry settings or on-failure destinations*

Kafka retry configurations and on-failure destinations are only available for event source mappings with provisioned mode enabled. Ensure that you have configured `MinimumPollers` in your `ProvisionedPollerConfig` before attempting to set retry configurations.

Common configuration errors:

- **Infinite retries with bisect batch** – You cannot enable `BisectBatchOnFunctionError` when `MaximumRetryAttempts` is set to -1 (infinite). Set a finite retry limit or disable bisect batch.
- **Same topic recursion** – The Kafka on-failure destination topic cannot be the same as any of your source topics. Choose a different topic name for your dead letter topic.
- **Invalid Kafka destination format** – Use the `kafka://<topic-name>` format when specifying a Kafka topic as an on-failure destination.
- **kafka:WriteData permission issues** – Ensure your execution role has `kafka-cluster:WriteData` permissions for the destination topic. Topic doesn't exist timeout exceptions or write API throttling issues may require increasing the account limits.

Troubleshoot event source mapping issues in Lambda

Issues in Lambda that relate to an event source mapping can be more complex because they involve debugging across multiple services. Moreover, event source behavior can differ based on the exact event source used. This section lists common issues that involve event source mappings, and provides guidance on how to identify and troubleshoot them.

> **Note**
>
> This section uses an Amazon SQS event source for illustration, but the principles apply to other event source mappings that queue messages for Lambda functions.

Identifying and managing throttling

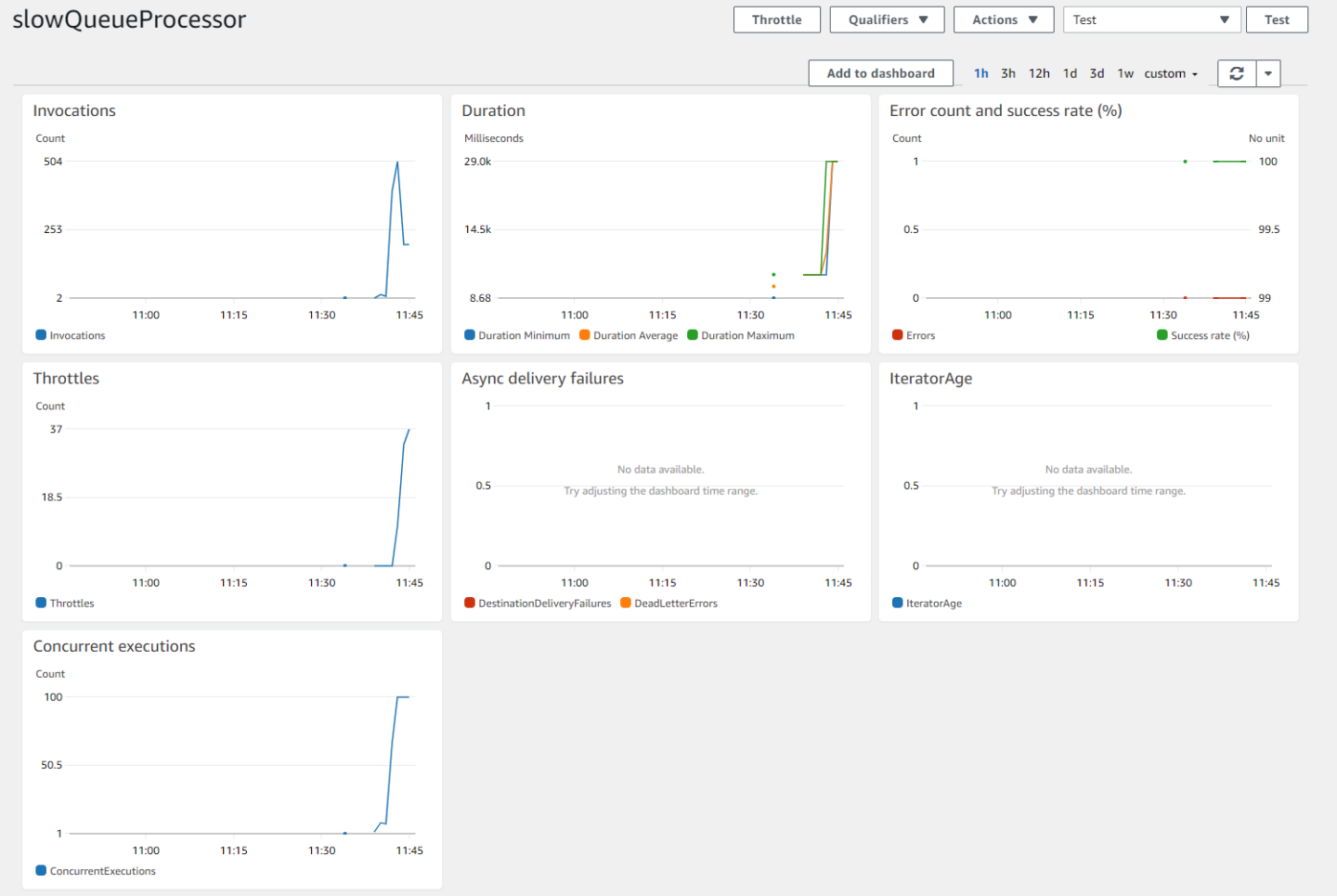
In Lambda, throttling occurs when you reach your function's or account's concurrency limit. Consider the following example, where there is a Lambda function that reads messages from an Amazon SQS queue. This Lambda function simulates 30 second invocations, and has a batch size of 1. This means that the function processes only 1 message every 30 seconds:

```
const doWork = (ms) => new Promise(resolve => setTimeout(resolve, ms))

exports.handler = async (event) => {
    await doWork(30000)

}
```

With such a long invocation time, messages begin arriving in the queue more rapidly than they are processed. If your account's unreserved concurrency is 100, Lambda scales up to 100 concurrent executions, and then throttling occurs. You can see this pattern in the CloudWatch metrics for the function:



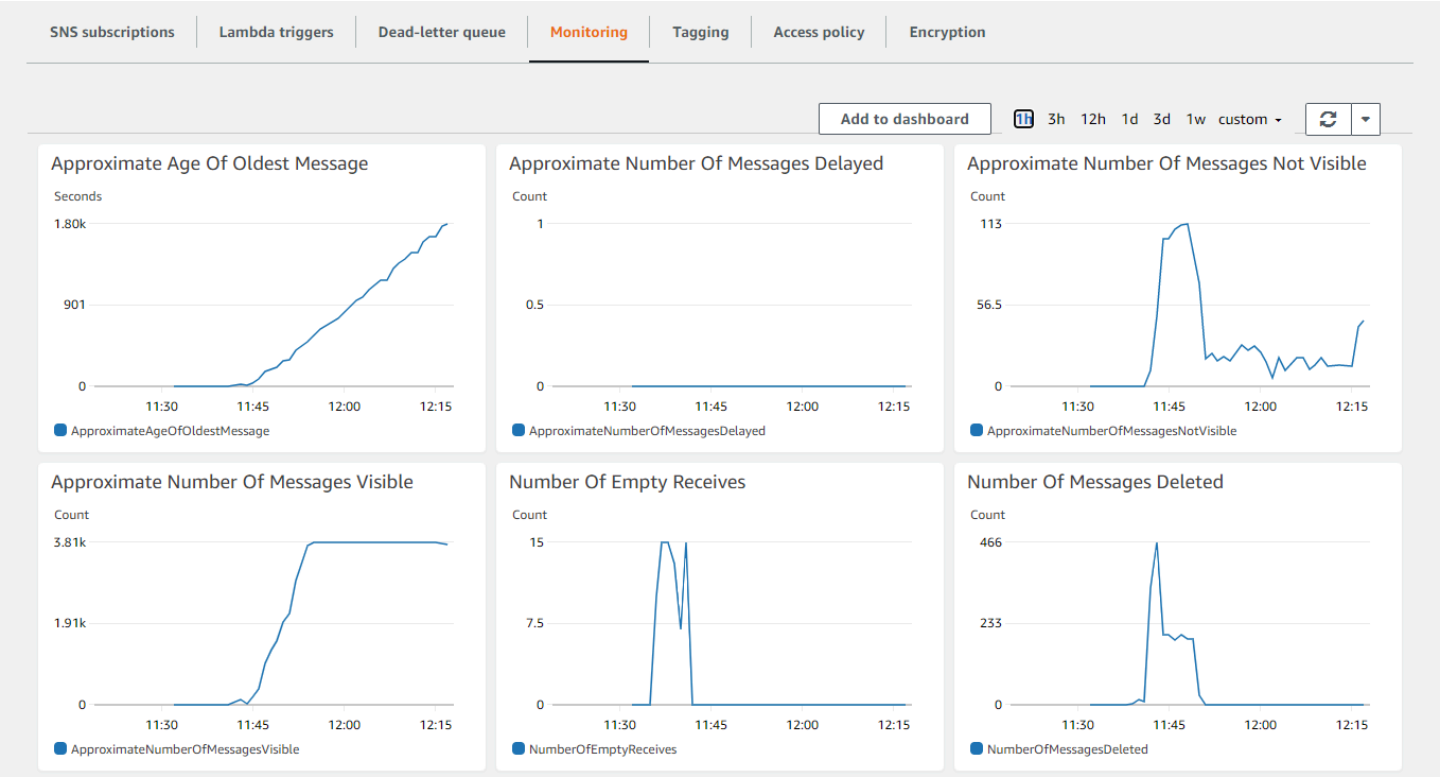
CloudWatch metrics for the function show no errors, but the **Concurrent executions** chart shows that the maximum concurrency of 100 is reached. As a result, the **Throttles** chart shows the throttling in place.

You can detect throttling with CloudWatch alarms, and setting an alarm anytime the throttling metric for a function is greater than 0. After you've identified the throttling issue, you have a few options for resolution:

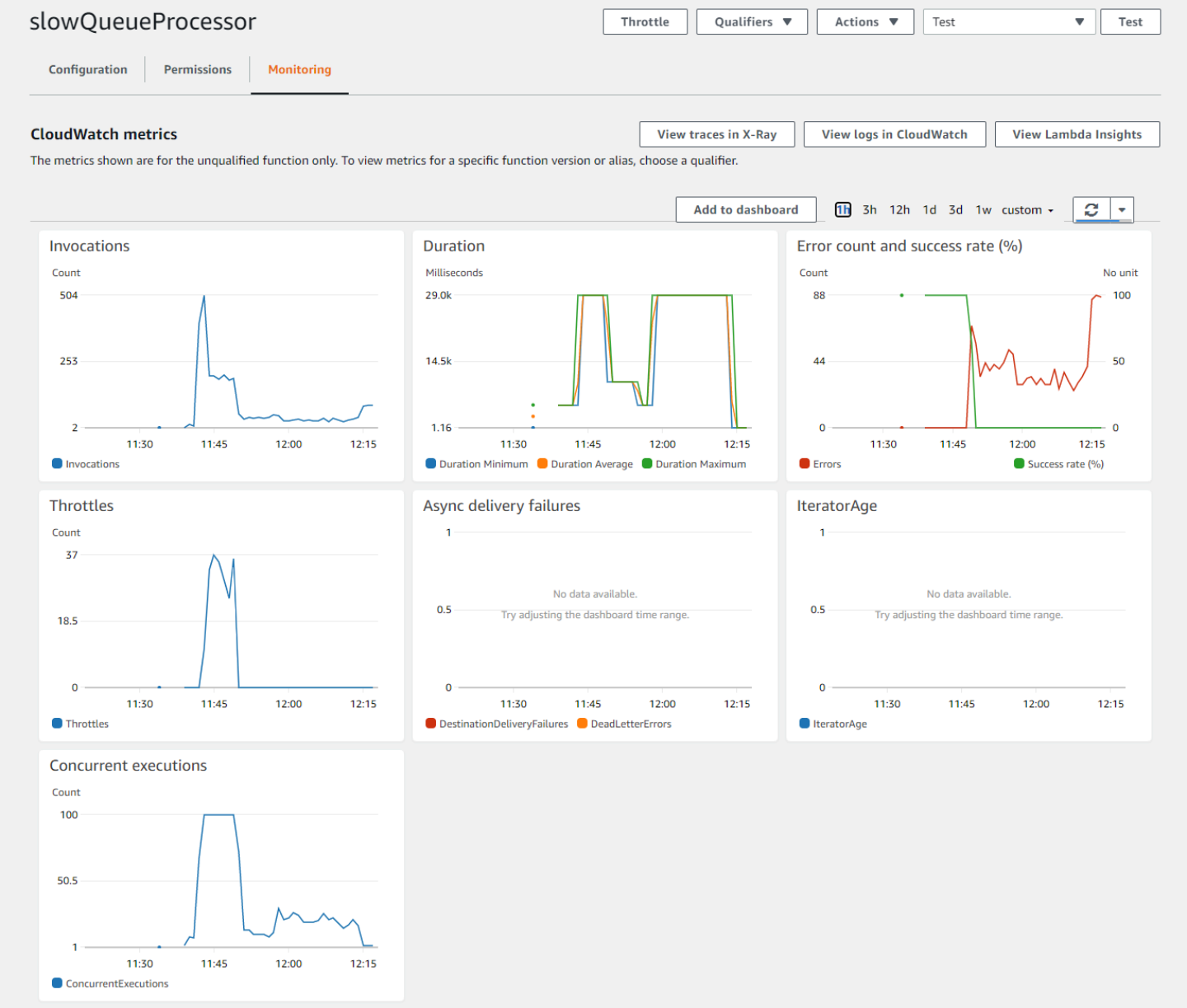
- Request a concurrency increase from AWS Support in this Region.
- Identify performance issues in the function to improve the speed of processing and therefore improve throughput.
- Increase the batch size of the function, so more messages are processed by each invocation.

Errors in the processing function

If the processing function throws errors, Lambda returns the messages to the SQS queue. Lambda prevents your function from scaling to prevent errors at scale. The following SQS metrics in CloudWatch indicate an issue with queue processing:

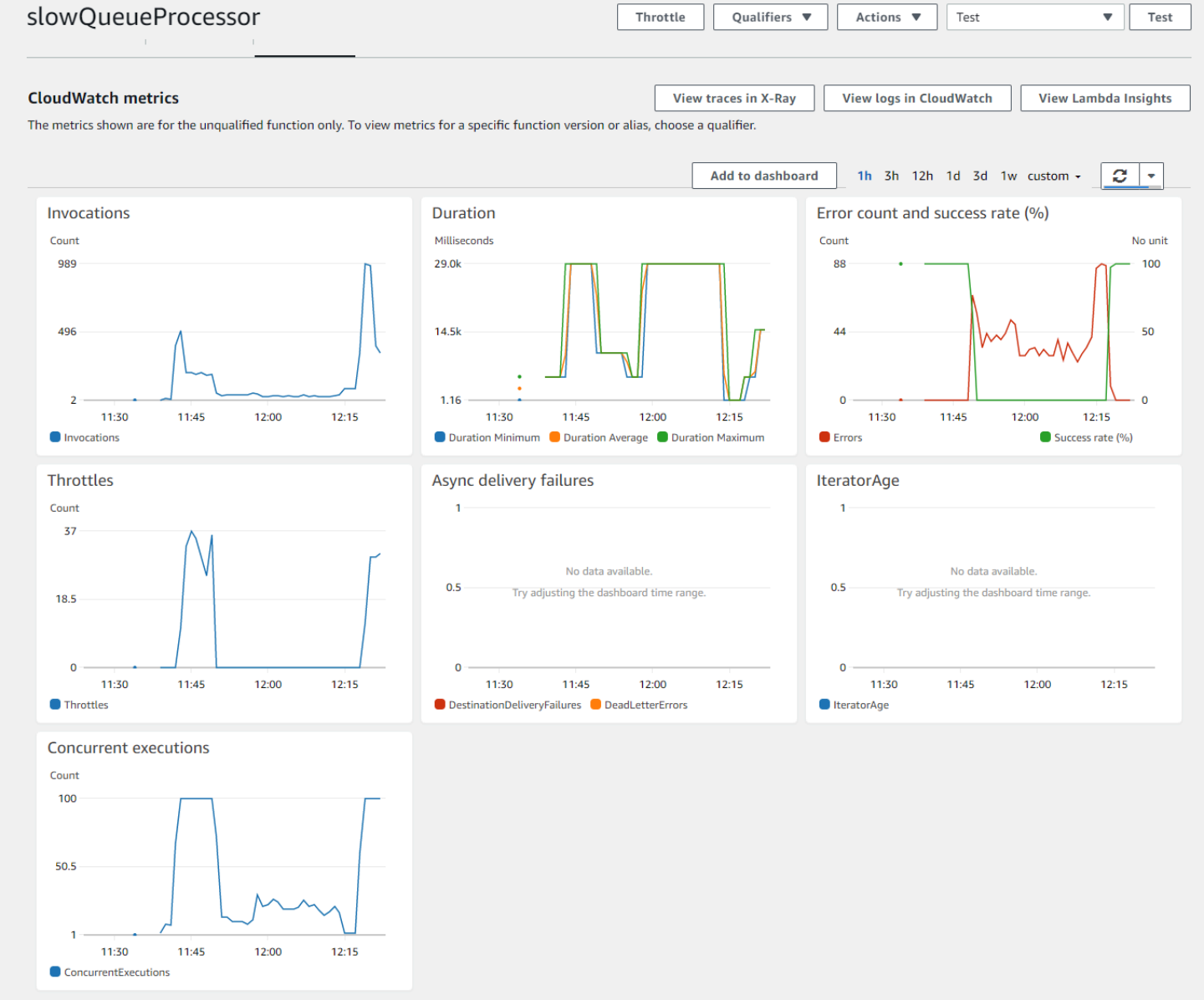


In particular, both the age of the oldest message and the number of messages visible are increasing, while no messages are deleted. The queue continues to grow but messages are not being processed. The CloudWatch metrics for the processing Lambda function also indicate that there is a problem:



The **Error count** metric is non-zero and growing, while **Concurrent executions** have reduced and throttling has stopped. This shows that Lambda has stopped scaling up your function due to errors. The CloudWatch logs for the function provide details of the type of error.

You can resolve this issue by identifying the function causing the error, then finding and resolving the error. After you fix the error and deploy the new function code, the CloudWatch metrics should show the processing recover:



Here, the **Error count** metric drops to zero and the **Success rate** metric returns to 100%. Lambda starts scaling up the function again, as shown in the **Concurrent executions** graph.

Identifying and handling backpressure

If an event producer consistently generates messages for an SQS queue faster than a Lambda function can process them, backpressure occurs. In this case, SQS monitoring should show the age of the oldest message growing linearly, along with the approximate number of messages visible. You can detect backpressure in queues using CloudWatch alarms.

The steps to resolve backpressure depend on your workload. If the primary goal is to increase processing capability and throughput by the Lambda function, you have a few options:

- Request a concurrency increase in the specific Region from AWS Support.
- Increase the batch size of the function, so more messages are processed by each invocation.

Troubleshoot networking issues in Lambda

By default, Lambda runs your functions in an internal virtual private cloud (VPC) with connectivity to AWS services and the internet. To access local network resources, you can configure your function to connect to a VPC in your account. When you use this feature, you manage the function's internet access and network connectivity with Amazon Virtual Private Cloud (Amazon VPC) resources.

Network connectivity errors can result from issues with your VPC's routing configuration, security group rules, AWS Identity and Access Management (IAM) role permissions, or network address translation (NAT), or from the availability of resources such as IP addresses or network interfaces. Depending on the issue, you might see a specific error or timeout if a request can't reach its destination.

Topics

- VPC: Function loses internet access or times out
- VPC: TCP or UDP connection intermittently fails
- VPC: Function needs access to AWS services without using the internet
- VPC: Elastic network interface limit reached
- EC2: Elastic network interface with type of "lambda"
- DNS: Fail to connect to hosts with UNKNOWNHOSTEXCEPTION

VPC: Function loses internet access or times out

Issue: *Your Lambda function loses internet access after connecting to a VPC.*

Error: *Error: connect ETIMEDOUT 176.32.98.189:443*

Error: *Error: Task timed out after 10.00 seconds*

Error: *ReadTimeoutError: Read timed out. (read timeout=15)*

When you connect a function to a VPC, all outbound requests go through the VPC. To connect to the internet, configure your VPC to send outbound traffic from the function's subnet to a NAT

gateway in a public subnet. For more information and sample VPC configurations, see the section called “Internet access for VPC functions”.

If some of your TCP connections are timing out, see the section called “VPC: TCP or UDP connection intermittently fails” if your subnet is using a network access control list (NACL). Otherwise, this is likely due to packet fragmentation. Lambda functions cannot handle incoming fragmented TCP requests, since Lambda does not support IP fragmentation for TCP or ICMP.

VPC: TCP or UDP connection intermittently fails

> **Note**
>
> This issue applies only if your subnet uses a network access control list (ACL). Network ACLs aren't required for Lambda to connect to your subnets.

Issue: *Lambda intermittently loses connection to your VPC subnets, which you have configured a network access control list (ACL) for.*

For VPC-enabled Lambda functions, AWS creates hyperplane ENIs in the customer's account, and uses ephemeral ports 1024 to 65535 to connect Lambda to the customer's VPC. If you use network ACLs in the target subnet, you must allow the port range 1024 to 65535 for both TCP and UDP. Not allowing this full port range can cause intermittent connection failures.

VPC: Function needs access to AWS services without using the internet

Issue: *Your Lambda function needs access to AWS services without using the internet.*

To connect a function to AWS services from a private subnet with no internet access, use VPC endpoints.

VPC: Elastic network interface limit reached

Error: *ENILimitReachedException: The elastic network interface limit was reached for the function's VPC.*

When you connect a Lambda function to a VPC, Lambda creates an elastic network interface for each combination of subnet and security group attached to the function. The default service quota is 250 network interfaces per VPC. To request a quota increase, use the Service Quotas console.

EC2: Elastic network interface with type of "lambda"

Error Code: *Client.OperationNotPermitted*

Error message: *The security group can not be modified for this type of interface*

You will receive this error if you attempt to modify an elastic network interface (ENI) that is managed by Lambda. The `ModifyNetworkInterfaceAttribute` is not included in the Lambda API for update operations on elastic network interfaces created by Lambda.

DNS: Fail to connect to hosts with UNKNOWNHOSTEXCEPTION

Error Message: *UNKNOWNHOSTEXCEPTION*

Lambda functions support a maximum of 20 concurrent TCP connections for DNS resolution. Your function may be exhausting that limit. Most common DNS requests are done over UDP. If your function is only making UDP DNS connections, this is unlikely to be your issue. This error is commonly thrown due to misconfiguration or degraded infrastructure, so before examining your DNS traffic in depth, confirm that your DNS infrastructure is properly configured and healthy and that your Lambda function is referring to a host specified in DNS.

If you diagnose your issue as related to the TCP connection maximum, note that you cannot request an increase to this limit. If your Lambda function is falling back to TCP DNS because of large DNS payloads, confirm that your solution is using libraries that support EDNS. For more information about EDNS, see the RFC 6891 standard. If your DNS payloads consistently exceed EDNS max sizes, your solution may still exhaust the TCP DNS limit.

Lambda sample applications

The GitHub repository for this guide includes sample applications that demonstrate the use of various languages and AWS services. Each sample application includes scripts for easy deployment and cleanup and supporting resources.

Node.js

Sample Lambda applications in Node.js

- blank-nodejs – A Node.js function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- nodejs-apig – A function with a public API endpoint that processes an event from API Gateway and returns an HTTP response.

Python

Sample Lambda applications in Python

- blank-python – A Python function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.

Ruby

Sample Lambda applications in Ruby

- blank-ruby – A Ruby function that shows the use of logging, environment variables, AWS X-Ray tracing, layers, unit tests and the AWS SDK.
- Ruby Code Samples for AWS Lambda – Code samples written in Ruby that demonstrate how to interact with AWS Lambda.

Java

Sample Lambda applications in Java

- example-java – A Java function that demonstrates how you can use Lambda to process orders. This function illustrates how to define and deserialize a custom input event object, use the AWS SDK, and output logging.

- java-basic – A collection of minimal Java functions with unit tests and variable logging configuration.
- java-events – A collection of Java functions that contain skeleton code for how to handle events from various services such as Amazon API Gateway, Amazon SQS, and Amazon Kinesis. These functions use the latest version of the aws-lambda-java-events library (3.0.0 and newer). These examples do not require the AWS SDK as a dependency.
- s3-java – A Java function that processes notification events from Amazon S3 and uses the Java Class Library (JCL) to create thumbnails from uploaded image files.
- layer-java – A Java function that illustrates how to use a Lambda layer to package dependencies separate from your core function code.

Running popular Java frameworks on Lambda

- spring-cloud-function-samples – An example from Spring that shows how to use the Spring Cloud Function framework to create AWS Lambda functions.
- Serverless Spring Boot Application Demo – An example that shows how to set up a typical Spring Boot application in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime.
- Serverless Micronaut Application Demo – An example that shows how to use Micronaut in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime. Learn more in the Micronaut/Lambda guides.
- Serverless Quarkus Application Demo – An example that shows how to use Quarkus in a managed Java runtime with and without SnapStart, or as a GraalVM native image with a custom runtime. Learn more in the Quarkus/Lambda guide and Quarkus/SnapStart guide.

Go

Lambda provides the following sample applications for the Go runtime:

Sample Lambda applications in Go

- go-al2 – A hello world function that returns the public IP address. This app uses the `provided.al2` custom runtime.
- blank-go – A Go function that shows the use of Lambda's Go libraries, logging, environment variables, and the AWS SDK. This app uses the `go1.x` runtime.

C#

Sample Lambda applications in C#

- blank-csharp – A C# function that shows the use of Lambda's .NET libraries, logging, environment variables, AWS X-Ray tracing, unit tests, and the AWS SDK.
- blank-csharp-with-layer – A C# function that uses the .NET CLI to create a layer that packages the function's dependencies.
- ec2-spot – A function that manages spot instance requests in Amazon EC2.

PowerShell

Lambda provides the following sample applications for PowerShell:

- blank-powershell – A PowerShell function that shows the use of logging, environment variables, and the AWS SDK.

To deploy a sample application, follow the instructions in its README file.

Using Lambda with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS CLI	AWS CLI code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS Tools for PowerShell	AWS Tools for PowerShell code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

For examples specific to Lambda, see Code examples for Lambda using AWS SDKs.

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Code examples for Lambda using AWS SDKs

The following code examples show how to use Lambda with an AWS software development kit (SDK).

Basics are code examples that show you how to perform the essential operations within a service.

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

Scenarios are code examples that show you how to accomplish specific tasks by calling multiple functions within a service or combined with other AWS services.

AWS community contributions are examples that were created and are maintained by multiple teams across AWS. To provide feedback, use the mechanism provided in the linked repositories.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples

- Basic examples for Lambda using AWS SDKs
  - Hello Lambda
  - Learn the basics of Lambda with an AWS SDK
  - Actions for Lambda using AWS SDKs
    - Use CreateAlias with a CLI
    - Use CreateFunction with an AWS SDK or CLI
    - Use DeleteAlias with a CLI
    - Use DeleteFunction with an AWS SDK or CLI
    - Use DeleteFunctionConcurrency with a CLI
    - Use DeleteProvisionedConcurrencyConfig with a CLI
    - Use GetAccountSettings with a CLI
    - Use GetAlias with a CLI
    - Use GetFunction with an AWS SDK or CLI
    - Use GetFunctionConcurrency with a CLI

- Use GetFunctionConfiguration with a CLI
- Use GetPolicy with a CLI
- Use GetProvisionedConcurrencyConfig with a CLI
- Use Invoke with an AWS SDK or CLI
- Use ListFunctions with an AWS SDK or CLI
- Use ListProvisionedConcurrencyConfigs with a CLI
- Use ListTags with a CLI
- Use ListVersionsByFunction with a CLI
- Use PublishVersion with a CLI
- Use PutFunctionConcurrency with a CLI
- Use PutProvisionedConcurrencyConfig with a CLI
- Use RemovePermission with a CLI
- Use TagResource with a CLI
- Use UntagResource with a CLI
- Use UpdateAlias with a CLI
- Use UpdateFunctionCode with an AWS SDK or CLI
- Use UpdateFunctionConfiguration with an AWS SDK or CLI

- Scenarios for Lambda using AWS SDKs
  - Automatically confirm known Amazon Cognito users with a Lambda function using an AWS SDK
  - Automatically migrate known Amazon Cognito users with a Lambda function using an AWS SDK
  - Create an API Gateway REST API to track COVID-19 data
  - Create a lending library REST API
  - Create a messenger application with Step Functions
  - Create a photo asset management application that lets users manage photos using labels
  - Create a websocket chat application with API Gateway
  - Create an application that analyzes customer feedback and synthesizes audio
  - Invoke a Lambda function from a browser
  - Transform data for your application with S3 Object Lambda
  - Use API Gateway to invoke a Lambda function

- Use Step Functions to invoke Lambda functions
- Use scheduled events to invoke a Lambda function
- Use the Amazon Neptune API to develop a Lambda function that queries graph data
- Write custom activity data with a Lambda function after Amazon Cognito user authentication using an AWS SDK

- Serverless examples for Lambda
  - Connecting to an Amazon RDS database in a Lambda function
  - Invoke a Lambda function from a Kinesis trigger
  - Invoke a Lambda function from a DynamoDB trigger
  - Invoke a Lambda function from a Amazon DocumentDB trigger
  - Invoke a Lambda function from an Amazon MSK trigger
  - Invoke a Lambda function from an Amazon S3 trigger
  - Invoke a Lambda function from an Amazon SNS trigger
  - Invoke a Lambda function from an Amazon SQS trigger
  - Reporting batch item failures for Lambda functions with a Kinesis trigger
  - Reporting batch item failures for Lambda functions with a DynamoDB trigger
  - Reporting batch item failures for Lambda functions with an Amazon SQS trigger
- AWS community contributions for Lambda
  - Build and test a serverless application

Basic examples for Lambda using AWS SDKs

The following code examples show how to use the basics of AWS Lambda with AWS SDKs.

Examples

- Hello Lambda
- Learn the basics of Lambda with an AWS SDK
- Actions for Lambda using AWS SDKs
  - Use CreateAlias with a CLI
  - Use CreateFunction with an AWS SDK or CLI
  - Use DeleteAlias with a CLI

- Use DeleteFunction with an AWS SDK or CLI
- Use DeleteFunctionConcurrency with a CLI
- Use DeleteProvisionedConcurrencyConfig with a CLI
- Use GetAccountSettings with a CLI
- Use GetAlias with a CLI
- Use GetFunction with an AWS SDK or CLI
- Use GetFunctionConcurrency with a CLI
- Use GetFunctionConfiguration with a CLI
- Use GetPolicy with a CLI
- Use GetProvisionedConcurrencyConfig with a CLI
- Use Invoke with an AWS SDK or CLI
- Use ListFunctions with an AWS SDK or CLI
- Use ListProvisionedConcurrencyConfigs with a CLI
- Use ListTags with a CLI
- Use ListVersionsByFunction with a CLI
- Use PublishVersion with a CLI
- Use PutFunctionConcurrency with a CLI
- Use PutProvisionedConcurrencyConfig with a CLI
- Use RemovePermission with a CLI
- Use TagResource with a CLI
- Use UntagResource with a CLI
- Use UpdateAlias with a CLI
- Use UpdateFunctionCode with an AWS SDK or CLI
- Use UpdateFunctionConfiguration with an AWS SDK or CLI

Hello Lambda

The following code examples show how to get started using Lambda.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{
    static async Task Main(string[] args)
    {
        var lambdaClient = new AmazonLambdaClient();

        Console.WriteLine("Hello AWS Lambda");
        Console.WriteLine("Let's get started with AWS Lambda by listing your
 existing Lambda functions:");

        var response = await lambdaClient.ListFunctionsAsync();
        response.Functions.ForEach(function =>
        {

 Console.WriteLine($"{function.FunctionName}\t{function.Description}");
        });
    }
}
```

- For API details, see ListFunctions in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Code for the CMakeLists.txt CMake file.

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)

# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.
```

```
    # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                    # and set the proper subdirectory to the
 executables' location.

    AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

Code for the hello_lambda.cpp source file.

```
#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
#include <iostream>

/*
 *  A "Hello Lambda" starter application which initializes an AWS Lambda (Lambda)
 client and lists the Lambda functions.
 *
 *  main function
 *
 *  Usage: 'hello_lambda'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";
```

```
        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome =
lambdaClient.ListFunctions(
                    request);

            if (outcome.IsSuccess()) {
                const Aws::Lambda::Model::ListFunctionsResult
&listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size()
                          << " lambda functions were retrieved." << std::endl;

                for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: listFunctionsResult.GetFunctions()) {
                    functions.push_back(functionConfiguration.GetFunctionName());
                    std::cout << functions.size() << "  "
                              << functionConfiguration.GetDescription() <<
std::endl;
                    std::cout << "   "
                              <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                      functionConfiguration.GetRuntime()) << ": "
                              << functionConfiguration.GetHandler()
                              << std::endl;
                }
                marker = listFunctionsResult.GetNextMarker();
            } else {
                std::cerr << "Error with Lambda::ListFunctions. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                result = 1;
                break;
            }
        } while (!marker.empty());
    }
```

```
    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- For API details, see ListFunctions in *AWS SDK for C++ API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
)

// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up
 to 10
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
```

```
  fmt.Println(err)
  return
 }
 lambdaClient := lambda.NewFromConfig(sdkConfig)

 maxItems := 10
 fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
 result, err := lambdaClient.ListFunctions(ctx, &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 if err != nil {
  fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
  return
 }
 if len(result.Functions) == 0 {
  fmt.Println("You don't have any functions!")
 } else {
  for _, function := range result.Functions {
   fmt.Printf("\t%v\n", *function.FunctionName)
  }
 }
}
```

- For API details, see ListFunctions in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
```

```
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
 Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
 config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- For API details, see ListFunctions in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
```

```
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
};
```

- For API details, see ListFunctions in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import boto3


def main():
    """
    List the Lambda functions in your AWS account.
    """
    # Create the Lambda client
    lambda_client = boto3.client("lambda")

    # Use the paginator to list the functions
    paginator = lambda_client.get_paginator("list_functions")
    response_iterator = paginator.paginate()

    print("Here are the Lambda functions in your account:")
    for page in response_iterator:
        for function in page["Functions"]:
            print(f"  {function['FunctionName']}")
```

```
if __name__ == "__main__":
    main()
```

- For API details, see ListFunctions in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
require 'aws-sdk-lambda'

# Creates an AWS Lambda client using the default credentials and configuration
def lambda_client
  Aws::Lambda::Client.new
end

# Lists the Lambda functions in your AWS account, paginating the results if
 necessary
def list_lambda_functions
  lambda = lambda_client

  # Use a pagination iterator to list all functions
  functions = []
  lambda.list_functions.each_page do |page|
    functions.concat(page.functions)
  end

  # Print the name and ARN of each function
  functions.each do |function|
    puts "Function name: #{function.function_name}"
    puts "Function ARN: #{function.function_arn}"
    puts
  end
```

```
  puts "Total functions: #{functions.count}"
end

list_lambda_functions if __FILE__ == $PROGRAM_NAME
```

- For API details, see ListFunctions in *AWS SDK for Ruby API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Learn the basics of Lambda with an AWS SDK

The following code examples show how to:

- Create an IAM role and Lambda function, then upload handler code.
- Invoke the function with a single parameter and get results.
- Update the function code and configure with an environment variable.
- Invoke the function with new parameters and get results. Display the returned execution log.
- List the functions for your account, then clean up resources.

For more information, see Create a Lambda function with the console.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create methods that perform Lambda actions.

```
namespace LambdaActions;
```

```
using Amazon.Lambda;
using Amazon.Lambda.Model;

/// <summary>
/// A class that implements AWS Lambda methods.
/// </summary>
public class LambdaWrapper
{
    private readonly IAmazonLambda _lambdaService;

    /// <summary>
    /// Constructor for the LambdaWrapper class.
    /// </summary>
    /// <param name="lambdaService">An initialized Lambda service client.</param>
    public LambdaWrapper(IAmazonLambda lambdaService)
    {
        _lambdaService = lambdaService;
    }

    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
```

```
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
 _lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }


    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }
```

```
    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }


    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }
```

```
    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
    }


    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };
```

```
        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }


    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
 actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
 variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
 environmentVariables },
        };

        var response = await
 _lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }


}
```

Create a function that runs the scenario.

```
global using System.Threading.Tasks;
global using Amazon.IdentityManagement;
global using Amazon.Lambda;
global using LambdaActions;
global using LambdaScenarioCommon;
global using Microsoft.Extensions.DependencyInjection;
global using Microsoft.Extensions.Hosting;
global using Microsoft.Extensions.Logging;
global using Microsoft.Extensions.Logging.Console;
global using Microsoft.Extensions.Logging.Debug;


using Amazon.Lambda.Model;
using Microsoft.Extensions.Configuration;

namespace LambdaBasics;

public class LambdaBasics
{
    private static ILogger logger = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                    .AddFilter<DebugLoggerProvider>("Microsoft",
 LogLevel.Information)
                    .AddFilter<ConsoleLoggerProvider>("Microsoft",
 LogLevel.Trace))
            .ConfigureServices((_, services) =>
            services.AddAWSService<IAmazonLambda>()
            .AddAWSService<IAmazonIdentityManagementService>()
            .AddTransient<LambdaWrapper>()
            .AddTransient<LambdaRoleWrapper>()
            .AddTransient<UIWrapper>()
        )
        .Build();

        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("settings.json") // Load test settings from .json file.
```

```
            .AddJsonFile("settings.local.json",
            true) // Optionally load local settings.
        .Build();


        logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
            .CreateLogger<LambdaBasics>();

        var lambdaWrapper = host.Services.GetRequiredService<LambdaWrapper>();
        var lambdaRoleWrapper =
host.Services.GetRequiredService<LambdaRoleWrapper>();
        var uiWrapper = host.Services.GetRequiredService<UIWrapper>();

        string functionName = configuration["FunctionName"]!;
        string roleName = configuration["RoleName"]!;
        string policyDocument = "{" +
            " \"Version\": \"2012-10-17\"," +
            " \"Statement\": [ " +
            "    {" +
            "        \"Effect\": \"Allow\"," +
            "        \"Principal\": {" +
            "            \"Service\": \"lambda.amazonaws.com\" " +
            "    }," +
            "        \"Action\": \"sts:AssumeRole\" " +
            "    }" +
            "]" +
        "}";

        var incrementHandler = configuration["IncrementHandler"];
        var calculatorHandler = configuration["CalculatorHandler"];
        var bucketName = configuration["BucketName"];
        var incrementKey = configuration["IncrementKey"];
        var calculatorKey = configuration["CalculatorKey"];
        var policyArn = configuration["PolicyArn"];

        uiWrapper.DisplayLambdaBasicsOverview();

        // Create the policy to use with the AWS Lambda functions and then attach
the
        // policy to a new role.
        var roleArn = await lambdaRoleWrapper.CreateLambdaRoleAsync(roleName,
policyDocument);

        Console.WriteLine("Waiting for role to become active.");
```

```
        uiWrapper.WaitABit(15, "Wait until the role is active before trying to
use it.");

        // Attach the appropriate AWS Identity and Access Management (IAM) role
policy to the new role.
        var success = await
lambdaRoleWrapper.AttachLambdaRolePolicyAsync(policyArn, roleName);
        uiWrapper.WaitABit(10, "Allow time for the IAM policy to be attached to
the role.");

        // Create the Lambda function using a zip file stored in an Amazon Simple
Storage Service
        // (Amazon S3) bucket.
        uiWrapper.DisplayTitle("Create Lambda Function");
        Console.WriteLine($"Creating the AWS Lambda function: {functionName}.");
        var lambdaArn = await lambdaWrapper.CreateLambdaFunctionAsync(
            functionName,
            bucketName,
            incrementKey,
            roleArn,
            incrementHandler);

        Console.WriteLine("Waiting for the new function to be available.");
        Console.WriteLine($"The AWS Lambda ARN is {lambdaArn}");

        // Get the Lambda function.
        Console.WriteLine($"Getting the {functionName} AWS Lambda function.");
        FunctionConfiguration config;
        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.State != State.Active);

        Console.WriteLine($"\nThe function, {functionName} has been created.");
        Console.WriteLine($"The runtime of this Lambda function is
{config.Runtime}.");

        uiWrapper.PressEnter();

        // List the Lambda functions.
        uiWrapper.DisplayTitle("Listing all Lambda functions.");
        var functions = await lambdaWrapper.ListFunctionsAsync();
```

```
        DisplayFunctionList(functions);

        uiWrapper.DisplayTitle("Invoke increment function");
        Console.WriteLine("Now that it has been created, invoke the Lambda
increment function.");
        string? value;
        do
        {
            Console.Write("Enter a value to increment: ");
            value = Console.ReadLine();
        }
        while (string.IsNullOrEmpty(value));

        string functionParameters = "{" +
            "\"action\": \"increment\", " +
            "\"x\": \"" + value + "\"" +
        "}";
        var answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
        Console.WriteLine($"{value} + 1 = {answer}.");

        uiWrapper.DisplayTitle("Update function");
        Console.WriteLine("Now update the Lambda function code.");
        await lambdaWrapper.UpdateFunctionCodeAsync(functionName, bucketName,
calculatorKey);

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        await lambdaWrapper.UpdateFunctionConfigurationAsync(
            functionName,
            calculatorHandler,
            new Dictionary<string, string> { { "LOG_LEVEL", "DEBUG" } });

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);
```

```
        uiWrapper.DisplayTitle("Call updated function");
        Console.WriteLine("Now call the updated function...");

        bool done = false;

        do
        {
            string? opSelected;

            Console.WriteLine("Select the operation to perform:");
            Console.WriteLine("\t1. add");
            Console.WriteLine("\t2. subtract");
            Console.WriteLine("\t3. multiply");
            Console.WriteLine("\t4. divide");
            Console.WriteLine("\tOr enter \"q\" to quit.");
            Console.WriteLine("Enter the number (1, 2, 3, 4, or q) of the
operation you want to perform: ");
            do
            {
                Console.Write("Your choice? ");
                opSelected = Console.ReadLine();
            }
            while (opSelected == string.Empty);

            var operation = (opSelected) switch
            {
                "1" => "add",
                "2" => "subtract",
                "3" => "multiply",
                "4" => "divide",
                "q" => "quit",
                _ => "add",
            };

            if (operation == "quit")
            {
                done = true;
            }
            else
            {
                // Get two numbers and an action from the user.
                value = string.Empty;
                do
```

```
                {
                    Console.Write("Enter the first value: ");
                    value = Console.ReadLine();
                }
                while (value == string.Empty);

                string? value2;
                do
                {
                    Console.Write("Enter a second value: ");
                    value2 = Console.ReadLine();
                }
                while (value2 == string.Empty);

                functionParameters = "{" +
                    "\"action\": \"" + operation + "\", " +
                    "\"x\": \"" + value + "\"," +
                    "\"y\": \"" + value2 + "\"" +
                "}";

                answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
                Console.WriteLine($"The answer when we {operation} the two
numbers is: {answer}.");
            }

            uiWrapper.PressEnter();
        } while (!done);

        // Delete the function created earlier.

        uiWrapper.DisplayTitle("Clean up resources");
        // Detach the IAM policy from the IAM role.
        Console.WriteLine("First detach the IAM policy from the role.");
        success = await lambdaRoleWrapper.DetachLambdaRolePolicyAsync(policyArn,
roleName);
        uiWrapper.WaitABit(15, "Let's wait for the policy to be fully detached
from the role.");

        Console.WriteLine("Delete the AWS Lambda function.");
        success = await lambdaWrapper.DeleteFunctionAsync(functionName);
        if (success)
        {
            Console.WriteLine($"The {functionName} function was deleted.");
```

```
        }
        else
        {
            Console.WriteLine($"Could not remove the function {functionName}");
        }

        // Now delete the IAM role created for use with the functions
        // created by the application.
        Console.WriteLine("Now we can delete the role that we created.");
        success = await lambdaRoleWrapper.DeleteLambdaRoleAsync(roleName);
        if (success)
        {
            Console.WriteLine("The role has been successfully removed.");
        }
        else
        {
            Console.WriteLine("Couldn't delete the role.");
        }

        Console.WriteLine("The Lambda Scenario is now complete.");
        uiWrapper.PressEnter();

        // Displays a formatted list of existing functions returned by the
        // LambdaMethods.ListFunctions.
        void DisplayFunctionList(List<FunctionConfiguration> functions)
        {
            functions.ForEach(functionConfig =>
            {

 Console.WriteLine($"{functionConfig.FunctionName}\t{functionConfig.Description}");
            });
        }
    }
}


namespace LambdaActions;

using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

public class LambdaRoleWrapper
{
    private readonly IAmazonIdentityManagementService _lambdaRoleService;
```

```
    public LambdaRoleWrapper(IAmazonIdentityManagementService lambdaRoleService)
    {
        _lambdaRoleService = lambdaRoleService;
    }

    /// <summary>
    /// Attach an AWS Identity and Access Management (IAM) role policy to the
    /// IAM role to be assumed by the AWS Lambda functions created for the
 scenario.
    /// </summary>
    /// <param name="policyArn">The Amazon Resource Name (ARN) of the IAM
 policy.</param>
    /// <param name="roleName">The name of the IAM role to attach the IAM policy
 to.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> AttachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.AttachRolePolicyAsync(new
 AttachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    /// <summary>
    /// Create a new IAM role.
    /// </summary>
    /// <param name="roleName">The name of the IAM role to create.</param>
    /// <param name="policyDocument">The policy document for the new IAM role.</
param>
    /// <returns>A string representing the ARN for newly created role.</returns>
    public async Task<string> CreateLambdaRoleAsync(string roleName, string
 policyDocument)
    {
        var request = new CreateRoleRequest
        {
            AssumeRolePolicyDocument = policyDocument,
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.CreateRoleAsync(request);
        return response.Role.Arn;
    }
```

```
    /// <summary>
    /// Deletes an IAM role.
    /// </summary>
    /// <param name="roleName">The name of the role to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteLambdaRoleAsync(string roleName)
    {
        var request = new DeleteRoleRequest
        {
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.DeleteRoleAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    public async Task<bool> DetachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.DetachRolePolicyAsync(new
 DetachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}


namespace LambdaScenarioCommon;

public class UIWrapper
{
    public readonly string SepBar = new('-', Console.WindowWidth);

    /// <summary>
    /// Show information about the AWS Lambda Basics scenario.
    /// </summary>
    public void DisplayLambdaBasicsOverview()
    {
        Console.Clear();

        DisplayTitle("Welcome to AWS Lambda Basics");
        Console.WriteLine("This example application does the following:");
        Console.WriteLine("\t1. Creates an AWS Identity and Access Management
 (IAM) role that will be assumed by the functions we create.");
```

```
        Console.WriteLine("\t2. Attaches an IAM role policy that has Lambda
permissions.");
        Console.WriteLine("\t3. Creates a Lambda function that increments the
value passed to it.");
        Console.WriteLine("\t4. Calls the increment function and passes a
value.");
        Console.WriteLine("\t5. Updates the code so that the function is a simple
calculator.");
        Console.WriteLine("\t6. Calls the calculator function with the values
entered.");
        Console.WriteLine("\t7. Deletes the Lambda function.");
        Console.WriteLine("\t7. Detaches the IAM role policy.");
        Console.WriteLine("\t8. Deletes the IAM role.");
        PressEnter();
    }

    /// <summary>
    /// Display a message and wait until the user presses enter.
    /// </summary>
    public void PressEnter()
    {
        Console.Write("\nPress <Enter> to continue. ");
        _ = Console.ReadLine();
        Console.WriteLine();
    }

    /// <summary>
    /// Pad a string with spaces to center it on the console display.
    /// </summary>
    /// <param name="strToCenter">The string to be centered.</param>
    /// <returns>The padded string.</returns>
    public string CenterString(string strToCenter)
    {
        var padAmount = (Console.WindowWidth - strToCenter.Length) / 2;
        var leftPad = new string(' ', padAmount);
        return $"{leftPad}{strToCenter}";
    }

    /// <summary>
    /// Display a line of hyphens, the centered text of the title and another
    /// line of hyphens.
    /// </summary>
    /// <param name="strTitle">The string to be displayed.</param>
    public void DisplayTitle(string strTitle)
```

```
    {
        Console.WriteLine(SepBar);
        Console.WriteLine(CenterString(strTitle));
        Console.WriteLine(SepBar);
    }

    /// <summary>
    /// Display a countdown and wait for a number of seconds.
    /// </summary>
    /// <param name="numSeconds">The number of seconds to wait.</param>
    public void WaitABit(int numSeconds, string msg)
    {
        Console.WriteLine(msg);

        // Wait for the requested number of seconds.
        for (int i = numSeconds; i > 0; i--)
        {
            System.Threading.Thread.Sleep(1000);
            Console.Write($"{i}...");
        }

        PressEnter();
    }
}
```

Define a Lambda handler that increments a number.

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaIncrement;

public class Function
{

    /// <summary>
    /// A simple function increments the integer parameter.
```

```
    /// </summary>
    /// <param name="input">A JSON string containing an action, which must be
    /// "increment" and a string representing the value to increment.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the incremented value of the parameter.</
returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        if (input["action"] == "increment")
        {
            int inputValue = Convert.ToInt32(input["x"]);
            return inputValue + 1;
        }
        else
        {
            return 0;
        }
    }
}
```

Define a second Lambda handler that performs arithmetic operations.

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaCalculator;

public class Function
{

    /// <summary>
    /// A simple function that takes two number in string format and performs
    /// the requested arithmetic function.
    /// </summary>
```

```
    /// <param name="input">JSON data containing an action, and x and y values.
    /// Valid actions include: add, subtract, multiply, and divide.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the results of the calculation.</returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        var action = input["action"];
        int x = Convert.ToInt32(input["x"]);
        int y = Convert.ToInt32(input["y"]);
        int result;
        switch (action)
        {
            case "add":
                result = x + y;
                break;
            case "subtract":
                result = x - y;
                break;
            case "multiply":
                result = x * y;
                break;
            case "divide":
                if (y == 0)
                {
                    Console.Error.WriteLine("Divide by zero error.");
                    result = 0;
                }
                else
                    result = x / y;
                break;
            default:
                Console.Error.WriteLine($"{action} is not a valid operation.");
                result = 0;
                break;
        }
        return result;
    }
}
```

- For API details, see the following topics in *AWS SDK for .NET API Reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
//! Get started with functions scenario.
/*!
 \param clientConfig: AWS client configuration.
 \return bool: Successful completion.
 */
bool AwsDoc::Lambda::getStartedWithFunctionsScenario(
        const Aws::Client::ClientConfiguration &clientConfig) {

    Aws::Lambda::LambdaClient client(clientConfig);

    // 1. Create an AWS Identity and Access Management (IAM) role for Lambda
 function.
    Aws::String roleArn;
    if (!getIamRoleArn(roleArn, clientConfig)) {
        return false;
    }

    // 2. Create a Lambda function.
    int seconds = 0;
    do {
```

```
        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
```

```
        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
seconds
                      << " seconds elapsed." << std::endl;
            break;
        }
        else if (outcome.GetError().GetErrorType() ==
                 Aws::Lambda::LambdaErrors::INVALID_PARAMETER_VALUE &&
                 outcome.GetError().GetMessage().find("role") >= 0) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout
                        << "Waiting for the IAM role to become available as a
CreateFunction parameter. "
                        << seconds
                        << " seconds elapsed." << std::endl;

                std::cout << outcome.GetError().GetMessage() << std::endl;
            }
        }
        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (60 > seconds);

    std::cout << "The current Lambda function increments 1 by an input." <<
std::endl;

    // 3.  Invoke the Lambda function.
    {
```

```
        int increment = askQuestionForInt("Enter an increment integer: ");

        Aws::Lambda::Model::InvokeResult invokeResult;
        Aws::Utils::Json::JsonValue jsonPayload;
        jsonPayload.WithString("action", "increment");
        jsonPayload.WithInteger("number", increment);
        if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
                                 invokeResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(invokeResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
            if (iter != values.end() && iter->second.IsIntegerType()) {
                {
                    std::cout << INCREMENT_RESUlT_PREFIX
                              << iter->second.AsInteger() << std::endl;
                }
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
Aws::Utils::HashingUtils::Base64Decode(
                        invokeResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
            }
        }
    }

    std::cout
            << "The Lambda function will now be updated with new code. Press
return to continue, ";
    Aws::String answer;
    std::getline(std::cin, answer);

    // 4.  Update the Lambda function code.
    {
        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
```

```
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
 client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
 std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout
            << "This function uses an environment variable to control the logging
 level."
            << std::endl;
    std::cout
            << "UpdateFunctionConfiguration will be used to set the LOG_LEVEL to
 DEBUG."
            << std::endl;
```

```
    seconds = 0;

    // 5.  Update the Lambda function configuration.
    do {
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

            // RESOURCE_IN_USE: function code update not completed.
        else if (outcome.GetError().GetErrorType() !=
                 Aws::Lambda::LambdaErrors::RESOURCE_IN_USE) {
            if ((seconds % 10) == 0) { // Log status every 10 seconds.
                std::cout << "Lambda function update in progress . After " <<
seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }

    } while (0 < seconds);

    if (0 > seconds) {
        std::cerr << "Function failed to become active." << std::endl;
    }
    else {
        std::cout << "Updated function active after " << seconds << " seconds."
```

```
                  << std::endl;
    }

    std::cout
            << "\nThe new code applies an arithmetic operator to two variables, x
 an y."
            << std::endl;
    std::vector<Aws::String> operators = {"plus", "minus", "times", "divided-
by"};
    for (size_t i = 0; i < operators.size(); ++i) {
        std::cout << "   " << i + 1 << " " << operators[i] << std::endl;
    }

    // 6.  Invoke the updated Lambda function.
    do {
        int operatorIndex = askQuestionForIntRange("Select an operator index 1 -
 4 ", 1,
                                                   4);
        int x = askQuestionForInt("Enter an integer for the x value ");
        int y = askQuestionForInt("Enter an integer for the y value ");

        Aws::Utils::Json::JsonValue calculateJsonPayload;
        calculateJsonPayload.WithString("action", operators[operatorIndex - 1]);
        calculateJsonPayload.WithInteger("x", x);
        calculateJsonPayload.WithInteger("y", y);
        Aws::Lambda::Model::InvokeResult calculatedResult;
        if (invokeLambdaFunction(calculateJsonPayload,
                                 Aws::Lambda::Model::LogType::Tail,
                                 calculatedResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(calculatedResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
            if (iter != values.end() && iter->second.IsIntegerType()) {
                std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                          << operators[operatorIndex - 1] << " "
                          << y << " is " << iter->second.AsInteger() <<
 std::endl;
            }
            else if (iter != values.end() && iter->second.IsFloatingPointType())
 {
                std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                          << operators[operatorIndex - 1] << " "
                          << y << " is " << iter->second.AsDouble() << std::endl;
```

```
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
Aws::Utils::HashingUtils::Base64Decode(
                        calculatedResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
            }
        }

        answer = askQuestion("Would you like to try another operation? (y/n) ");
    } while (answer == "y");

    std::cout
            << "A list of the lambda functions will be retrieved. Press return to
continue, ";
    std::getline(std::cin, answer);

    // 7.  List the Lambda functions.

    std::vector<Aws::String> functions;
    Aws::String marker;

    do {
        Aws::Lambda::Model::ListFunctionsRequest request;
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
                request);

        if (outcome.IsSuccess()) {
            const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
            std::cout << result.GetFunctions().size()
                      << " lambda functions were retrieved." << std::endl;

            for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
                functions.push_back(functionConfiguration.GetFunctionName());
                std::cout << functions.size() << "  "
```

```
                        << functionConfiguration.GetDescription() << std::endl;
                std::cout << "   "
                        <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                functionConfiguration.GetRuntime()) << ": "
                        << functionConfiguration.GetHandler()
                        << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    } while (!marker.empty());

    // 8.  Get a Lambda function.
    if (!functions.empty()) {
        std::stringstream question;
        question << "Choose a function to retrieve between 1 and " <<
functions.size()
                 << " ";
        int functionIndex = askQuestionForIntRange(question.str(), 1,

static_cast<int>(functions.size()));

        Aws::String functionName = functions[functionIndex - 1];

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);

        Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                      << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                      << outcome.GetError().GetMessage()
```

```
                        << std::endl;
        }
    }

    std::cout << "The resources will be deleted. Press return to continue, ";
    std::getline(std::cin, answer);

    // 9.  Delete the Lambda function.
    bool result = deleteLambdaFunction(client);

    // 10. Delete the IAM role.
    return result && deleteIamRole(clientConfig);
}

//! Routine which invokes a Lambda function and returns the result.
/*!
 \param jsonPayload: Payload for invoke function.
 \param logType: Log type setting for invoke function.
 \param invokeResult: InvokeResult object to receive the result.
 \param client: Lambda client.
 \return bool: Successful completion.
 */
bool
AwsDoc::Lambda::invokeLambdaFunction(const Aws::Utils::Json::JsonValue
 &jsonPayload,
                                     Aws::Lambda::Model::LogType logType,
                                     Aws::Lambda::Model::InvokeResult
 &invokeResult,
                                     const Aws::Lambda::LambdaClient &client) {
    int seconds = 0;
    bool result = false;
    /*
     * In this example, the Invoke function can be called before recently created
 resources are
     * available.  The Invoke function is called repeatedly until the resources
 are
     * available.
     */
    do {
        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
 Aws::MakeShared<Aws::StringStream>(
```

```
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }

            // ACCESS_DENIED: because the role is not available yet.
            // RESOURCE_CONFLICT: because the Lambda function is being created or
 updated.
        else if ((outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::ACCESS_DENIED) ||
                 (outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::RESOURCE_CONFLICT)) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout << "Waiting for the invoke api to be available, status
 " <<
                          ((outcome.GetError().GetErrorType() ==
                            Aws::Lambda::LambdaErrors::ACCESS_DENIED ?
                            "ACCESS_DENIED" : "RESOURCE_CONFLICT")) << ". " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (seconds < 60);

    return result;
}
```

- For API details, see the following topics in *AWS SDK for C++ API Reference*.

- CreateFunction
- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create an interactive scenario that shows you how to get started with Lambda functions.

```
import (
 "archive/zip"
 "bytes"
 "context"
 "encoding/base64"
 "encoding/json"
 "errors"
 "fmt"
 "log"
 "os"
 "strings"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/iam"
 iamtypes "github.com/aws/aws-sdk-go-v2/service/iam/types"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
```

```
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/lambda/actions"
)

// GetStartedFunctionsScenario shows you how to use AWS Lambda to perform the
 following
// actions:
//
//  1. Create an AWS Identity and Access Management (IAM) role and Lambda
 function, then upload handler code.
//  2. Invoke the function with a single parameter and get results.
//  3. Update the function code and configure with an environment variable.
//  4. Invoke the function with new parameters and get results. Display the
 returned execution log.
//  5. List the functions for your account, then clean up resources.
type GetStartedFunctionsScenario struct {
 sdkConfig       aws.Config
 functionWrapper actions.FunctionWrapper
 questioner      demotools.IQuestioner
 helper          IScenarioHelper
 isTestRun       bool
}

// NewGetStartedFunctionsScenario constructs a GetStartedFunctionsScenario
 instance from a configuration.
// It uses the specified config to get a Lambda client and create wrappers for
 the actions
// used in the scenario.
func NewGetStartedFunctionsScenario(sdkConfig aws.Config, questioner
 demotools.IQuestioner,
 helper IScenarioHelper) GetStartedFunctionsScenario {
 lambdaClient := lambda.NewFromConfig(sdkConfig)
 return GetStartedFunctionsScenario{
  sdkConfig:       sdkConfig,
  functionWrapper: actions.FunctionWrapper{LambdaClient: lambdaClient},
  questioner:      questioner,
  helper:          helper,
 }
}

// Run runs the interactive scenario.
func (scenario GetStartedFunctionsScenario) Run(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong with the demo.\n")
```

```
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the AWS Lambda get started with functions demo.")
 log.Println(strings.Repeat("-", 88))

 role := scenario.GetOrCreateRole(ctx)
 funcName := scenario.CreateFunction(ctx, role)
 scenario.InvokeIncrement(ctx, funcName)
 scenario.UpdateFunction(ctx, funcName)
 scenario.InvokeCalculator(ctx, funcName)
 scenario.ListFunctions(ctx)
 scenario.Cleanup(ctx, role, funcName)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}

// GetOrCreateRole checks whether the specified role exists and returns it if it
 does.
// Otherwise, a role is created that specifies Lambda as a trusted principal.
// The AWSLambdaBasicExecutionRole managed policy is attached to the role and the
 role
// is returned.
func (scenario GetStartedFunctionsScenario) GetOrCreateRole(ctx context.Context)
 *iamtypes.Role {
 var role *iamtypes.Role
 iamClient := iam.NewFromConfig(scenario.sdkConfig)
 log.Println("First, we need an IAM role that Lambda can assume.")
 roleName := scenario.questioner.Ask("Enter a name for the role:",
 demotools.NotEmpty{})
 getOutput, err := iamClient.GetRole(ctx, &iam.GetRoleInput{
  RoleName: aws.String(roleName)})
 if err != nil {
  var noSuch *iamtypes.NoSuchEntityException
  if errors.As(err, &noSuch) {
   log.Printf("Role %v doesn't exist. Creating it....\n", roleName)
  } else {
   log.Panicf("Couldn't check whether role %v exists. Here's why: %v\n",
    roleName, err)
  }
 } else {
```

```
  role = getOutput.Role
  log.Printf("Found role %v.\n", *role.RoleName)
 }
 if role == nil {
  trustPolicy := PolicyDocument{
   Version: "2012-10-17",
   Statement: []PolicyStatement{{
    Effect:    "Allow",
    Principal: map[string]string{"Service": "lambda.amazonaws.com"},
    Action:    []string{"sts:AssumeRole"},
   }},
  }
  policyArn := "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
  createOutput, err := iamClient.CreateRole(ctx, &iam.CreateRoleInput{
   AssumeRolePolicyDocument: aws.String(trustPolicy.String()),
   RoleName:                 aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't create role %v. Here's why: %v\n", roleName, err)
  }
  role = createOutput.Role
  _, err = iamClient.AttachRolePolicy(ctx, &iam.AttachRolePolicyInput{
   PolicyArn: aws.String(policyArn),
   RoleName:  aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't attach a policy to role %v. Here's why: %v\n", roleName,
 err)
  }
  log.Printf("Created role %v.\n", *role.RoleName)
  log.Println("Let's give AWS a few seconds to propagate resources...")
  scenario.helper.Pause(10)
 }
 log.Println(strings.Repeat("-", 88))
 return role
}

// CreateFunction creates a Lambda function and uploads a handler written in
 Python.
// The code for the Python handler is packaged as a []byte in .zip format.
func (scenario GetStartedFunctionsScenario) CreateFunction(ctx context.Context,
 role *iamtypes.Role) string {
 log.Println("Let's create a function that increments a number.\n" +
  "The function uses the 'lambda_handler_basic.py' script found in the \n" +
```

```
  "'handlers' directory of this project.")
 funcName := scenario.questioner.Ask("Enter a name for the Lambda function:",
 demotools.NotEmpty{})
 zipPackage := scenario.helper.CreateDeploymentPackage("lambda_handler_basic.py",
 fmt.Sprintf("%v.py", funcName))
 log.Printf("Creating function %v and waiting for it to be ready.", funcName)
 funcState := scenario.functionWrapper.CreateFunction(ctx, funcName,
 fmt.Sprintf("%v.lambda_handler", funcName),
  role.Arn, zipPackage)
 log.Printf("Your function is %v.", funcState)
 log.Println(strings.Repeat("-", 88))
 return funcName
}

// InvokeIncrement invokes a Lambda function that increments a number. The
 function
// parameters are contained in a Go struct that is used to serialize the
 parameters to
// a JSON payload that is passed to the function.
// The result payload is deserialized into a Go struct that contains an int
 value.
func (scenario GetStartedFunctionsScenario) InvokeIncrement(ctx context.Context,
 funcName string) {
 parameters := actions.IncrementParameters{Action: "increment"}
 log.Println("Let's invoke our function. This function increments a number.")
 parameters.Number = scenario.questioner.AskInt("Enter a number to increment:",
 demotools.NotEmpty{})
 log.Printf("Invoking %v with %v...\n", funcName, parameters.Number)
 invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, parameters,
 false)
 var payload actions.LambdaResultInt
 err := json.Unmarshal(invokeOutput.Payload, &payload)
 if err != nil {
  log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
   funcName, err)
 }
 log.Printf("Invoking %v with %v returned %v.\n", funcName, parameters.Number,
 payload)
 log.Println(strings.Repeat("-", 88))
}

// UpdateFunction updates the code for a Lambda function by uploading a simple
 arithmetic
// calculator written in Python. The code for the Python handler is packaged as a
```

```
// []byte in .zip format.
// After the code is updated, the configuration is also updated with a new log
// level that instructs the handler to log additional information.
func (scenario GetStartedFunctionsScenario) UpdateFunction(ctx context.Context,
 funcName string) {
 log.Println("Let's update the function to an arithmetic calculator.\n" +
  "The function uses the 'lambda_handler_calculator.py' script found in the \n" +
  "'handlers' directory of this project.")
 scenario.questioner.Ask("Press Enter when you're ready.")
 log.Println("Creating deployment package...")
 zipPackage :=
 scenario.helper.CreateDeploymentPackage("lambda_handler_calculator.py",
  fmt.Sprintf("%v.py", funcName))
 log.Println("...and updating the Lambda function and waiting for it to be
 ready.")
 funcState := scenario.functionWrapper.UpdateFunctionCode(ctx, funcName,
 zipPackage)
 log.Printf("Updated function %v. Its current state is %v.", funcName, funcState)
 log.Println("This function uses an environment variable to control logging
 level.")
 log.Println("Let's set it to DEBUG to get the most logging.")
 scenario.functionWrapper.UpdateFunctionConfiguration(ctx, funcName,
  map[string]string{"LOG_LEVEL": "DEBUG"})
 log.Println(strings.Repeat("-", 88))
}

// InvokeCalculator invokes the Lambda calculator function. The parameters are
 stored in a
// Go struct that is used to serialize the parameters to a JSON payload. That
 payload is then passed
// to the function.
// The result payload is deserialized to a Go struct that stores the result as
 either an
// int or float32, depending on the kind of operation that was specified.
func (scenario GetStartedFunctionsScenario) InvokeCalculator(ctx context.Context,
 funcName string) {
 wantInvoke := true
 choices := []string{"plus", "minus", "times", "divided-by"}
 for wantInvoke {
  choice := scenario.questioner.AskChoice("Select an arithmetic operation:\n",
 choices)
  x := scenario.questioner.AskInt("Enter a value for x:", demotools.NotEmpty{})
  y := scenario.questioner.AskInt("Enter a value for y:", demotools.NotEmpty{})
  log.Printf("Invoking %v %v %v...", x, choices[choice], y)
```

```
  calcParameters := actions.CalculatorParameters{
   Action: choices[choice],
   X:      x,
   Y:      y,
  }
  invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, calcParameters,
 true)
  var payload any
  if choice == 3 { // divide-by results in a float.
   payload = actions.LambdaResultFloat{}
  } else {
   payload = actions.LambdaResultInt{}
  }
  err := json.Unmarshal(invokeOutput.Payload, &payload)
  if err != nil {
   log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
    funcName, err)
  }
  log.Printf("Invoking %v with %v %v %v returned %v.\n", funcName,
   calcParameters.X, calcParameters.Action, calcParameters.Y, payload)
  scenario.questioner.Ask("Press Enter to see the logs from the call.")
  logRes, err := base64.StdEncoding.DecodeString(*invokeOutput.LogResult)
  if err != nil {
   log.Panicf("Couldn't decode log result. Here's why: %v\n", err)
  }
  log.Println(string(logRes))
  wantInvoke = scenario.questioner.AskBool("Do you want to calculate again? (y/
n)", "y")
 }
 log.Println(strings.Repeat("-", 88))
}

// ListFunctions lists up to the specified number of functions for your account.
func (scenario GetStartedFunctionsScenario) ListFunctions(ctx context.Context) {
 count := scenario.questioner.AskInt(
  "Let's list functions for your account. How many do you want to see?",
 demotools.NotEmpty{})
 functions := scenario.functionWrapper.ListFunctions(ctx, count)
 log.Printf("Found %v functions:", len(functions))
 for _, function := range functions {
  log.Printf("\t%v", *function.FunctionName)
 }
 log.Println(strings.Repeat("-", 88))
}
```

```
// Cleanup removes the IAM and Lambda resources created by the example.
func (scenario GetStartedFunctionsScenario) Cleanup(ctx context.Context, role
 *iamtypes.Role, funcName string) {
 if scenario.questioner.AskBool("Do you want to clean up resources created for
 this example? (y/n)",
  "y") {
  iamClient := iam.NewFromConfig(scenario.sdkConfig)
  policiesOutput, err := iamClient.ListAttachedRolePolicies(ctx,
   &iam.ListAttachedRolePoliciesInput{RoleName: role.RoleName})
  if err != nil {
   log.Panicf("Couldn't get policies attached to role %v. Here's why: %v\n",
    *role.RoleName, err)
  }
  for _, policy := range policiesOutput.AttachedPolicies {
   _, err = iamClient.DetachRolePolicy(ctx, &iam.DetachRolePolicyInput{
    PolicyArn: policy.PolicyArn, RoleName: role.RoleName,
   })
   if err != nil {
    log.Panicf("Couldn't detach policy %v from role %v. Here's why: %v\n",
     *policy.PolicyArn, *role.RoleName, err)
   }
  }
  _, err = iamClient.DeleteRole(ctx, &iam.DeleteRoleInput{RoleName:
 role.RoleName})
  if err != nil {
   log.Panicf("Couldn't delete role %v. Here's why: %v\n", *role.RoleName, err)
  }
  log.Printf("Deleted role %v.\n", *role.RoleName)

  scenario.functionWrapper.DeleteFunction(ctx, funcName)
  log.Printf("Deleted function %v.\n", funcName)
 } else {
  log.Println("Okay. Don't forget to delete the resources when you're done with
 them.")
 }
}

// IScenarioHelper abstracts I/O and wait functions from a scenario so that they
// can be mocked for unit testing.
type IScenarioHelper interface {
 Pause(secs int)
 CreateDeploymentPackage(sourceFile string, destinationFile string) *bytes.Buffer
}
```

```
// ScenarioHelper lets the caller specify the path to Lambda handler functions.
type ScenarioHelper struct {
 HandlerPath string
}

// Pause waits for the specified number of seconds.
func (helper *ScenarioHelper) Pause(secs int) {
 time.Sleep(time.Duration(secs) * time.Second)
}

// CreateDeploymentPackage creates an AWS Lambda deployment package from a source
 file. The
// deployment package is stored in .zip format in a bytes.Buffer. The buffer can
 be
// used to pass a []byte to Lambda when creating the function.
// The specified destinationFile is the name to give the file when it's deployed
 to Lambda.
func (helper *ScenarioHelper) CreateDeploymentPackage(sourceFile string,
 destinationFile string) *bytes.Buffer {
 var err error
 buffer := &bytes.Buffer{}
 writer := zip.NewWriter(buffer)
 zFile, err := writer.Create(destinationFile)
 if err != nil {
  log.Panicf("Couldn't create destination archive %v. Here's why: %v\n",
 destinationFile, err)
 }
 sourceBody, err := os.ReadFile(fmt.Sprintf("%v/%v", helper.HandlerPath,
 sourceFile))
 if err != nil {
  log.Panicf("Couldn't read handler source file %v. Here's why: %v\n",
   sourceFile, err)
 } else {
  _, err = zFile.Write(sourceBody)
  if err != nil {
   log.Panicf("Couldn't write handler %v to zip archive. Here's why: %v\n",
    sourceFile, err)
  }
 }
 err = writer.Close()
 if err != nil {
  log.Panicf("Couldn't close zip writer. Here's why: %v\n", err)
 }
```

```
 return buffer
}
```

Create a struct that wraps individual Lambda actions.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}


// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}
```

```
// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
```

```
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
```

```
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

```
// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}



// IncrementParameters is used to serialize parameters to the increment Lambda
 handler.
type IncrementParameters struct {
 Action string `json:"action"`
 Number int    `json:"number"`
}

// CalculatorParameters is used to serialize parameters to the calculator Lambda
 handler.
type CalculatorParameters struct {
 Action string `json:"action"`
 X      int    `json:"x"`
 Y      int    `json:"y"`
}
```

```
// LambdaResultInt is used to deserialize an int result from a Lambda handler.
type LambdaResultInt struct {
 Result int `json:"result"`
}

// LambdaResultFloat is used to deserialize a float32 result from a Lambda
 handler.
type LambdaResultFloat struct {
 Result float32 `json:"result"`
}
```

Define a Lambda handler that increments a number.

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
```

```
    return response
```

Define a second Lambda handler that performs arithmetic operations.

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
```

```
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/*
 *  Lambda function names appear as:
 *
```

```
 *  arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
 *
 *  To find this value, look at the function in the AWS Management Console.
 *
 *  Before running this Java code example, set up your development environment,
 including your credentials.
 *
 *  For more information, see this documentation topic:
 *
 *  https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 *  This example performs the following tasks:
 *
 * 1. Creates an AWS Lambda function.
 * 2. Gets a specific AWS Lambda function.
 * 3. Lists all Lambda functions.
 * 4. Invokes a Lambda function.
 * 5. Updates the Lambda function code and invokes it again.
 * 6. Updates a Lambda function's configuration value.
 * 7. Deletes a Lambda function.
 */

public class LambdaScenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws InterruptedException {
        final String usage = """

            Usage:
                <functionName> <role> <handler> <bucketName> <key>\s

            Where:
                functionName - The name of the Lambda function.\s
                role - The AWS Identity and Access Management (IAM) service role
 that has Lambda permissions.\s
                handler - The fully qualified method name (for example,
 example.Handler::handleRequest).\s
                bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
 name that contains the .zip or .jar used to update the Lambda function's code.\s
                key - The Amazon S3 key name that represents the .zip or .jar
 (for example, LambdaHello-1.0-SNAPSHOT.jar).
                """;
```

```
        if (args.length != 5) {
              System.out.println(usage);
              return;
        }

        String functionName = args[0];
        String role = args[1];
        String handler = args[2];
        String bucketName = args[3];
        String key = args[4];
        LambdaClient awsLambda = LambdaClient.builder()
            .build();

        System.out.println(DASHES);
        System.out.println("Welcome to the AWS Lambda Basics scenario.");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("1. Create an AWS Lambda function.");
        String funArn = createLambdaFunction(awsLambda, functionName, key,
bucketName, role, handler);
        System.out.println("The AWS Lambda ARN is " + funArn);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("2. Get the " + functionName + " AWS Lambda
function.");
        getFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. List all AWS Lambda functions.");
        listFunctions(awsLambda);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Invoke the Lambda function.");
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
```

```
        System.out.println("5. Update the Lambda function code and invoke it
again.");
        updateFunctionCode(awsLambda, functionName, bucketName, key);
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Update a Lambda function's configuration value.");
        updateFunctionConfiguration(awsLambda, functionName, handler);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Delete the AWS Lambda function.");
        LambdaScenario.deleteLambdaFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The AWS Lambda scenario completed successfully");
        System.out.println(DASHES);
        awsLambda.close();
    }

    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
                                               String functionName,
                                               String key,
                                               String bucketName,
                                               String role,
                                               String handler) {

        try {
```

```
            LambdaWaiter waiter = awsLambda.waiter();
            FunctionCode code = FunctionCode.builder()
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }

    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
```

```
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Invokes a specific AWS Lambda function.
     *
     * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
```

```
     * @param functionName the name of the AWS Lambda function to be invoked
     */
    public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
        InvokeResponse res;
        try {
            // Need a SdkBytes instance for the payload.
            JSONObject jsonObj = new JSONObject();
            jsonObj.put("inputValue", "2000");
            String json = jsonObj.toString();
            SdkBytes payload = SdkBytes.fromUtf8String(json);

            InvokeRequest request = InvokeRequest.builder()
                .functionName(functionName)
                .payload(payload)
                .build();

            res = awsLambda.invoke(request);
            String value = res.payload().asUtf8String();
            System.out.println(value);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Updates the code for an AWS Lambda function.
     *
     * @param awsLambda  the AWS Lambda client
     * @param functionName the name of the Lambda function to update
     * @param bucketName the name of the S3 bucket where the function code is
located
     * @param key the key (file name) of the function code in the S3 bucket
     * @throws LambdaException if there is an error updating the function code
     */
    public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
        try {
            LambdaWaiter waiter = awsLambda.waiter();
            UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                .functionName(functionName)
```

```
                .publish(true)
                .s3Bucket(bucketName)
                .s3Key(key)
                .build();

            UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
            GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Updates the configuration of an AWS Lambda function.
     *
     * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
     * @param functionName  the name of the AWS Lambda function to update
     * @param handler       the new handler for the AWS Lambda function
     *
     * @throws LambdaException if there is an error while updating the function
configuration
     */
    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .handler(handler)
                .runtime(Runtime.JAVA17)
```

```
                .build();

            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Deletes an AWS Lambda function.
     *
     * @param awsLambda     an instance of the {@link LambdaClient} class, which
 is used to interact with the AWS Lambda service
     * @param functionName  the name of the Lambda function to be deleted
     *
     * @throws LambdaException if an error occurs while deleting the Lambda
 function
     */
    public static void deleteLambdaFunction(LambdaClient awsLambda, String
 functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                .functionName(functionName)
                .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction

- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create an AWS Identity and Access Management (IAM) role that grants Lambda permission to write to logs.

```
    logger.log(`Creating role (${NAME_ROLE_LAMBDA})...`);
    const response = await createRole(NAME_ROLE_LAMBDA);

import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

Create a Lambda function and upload handler code.

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

Invoke the function with a single parameter and get results.

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

Update the function code and configure its Lambda environment with an environment variable.

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
```

```
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};

const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  const result = client.send(command);
  waitForFunctionUpdated({ FunctionName: funcName });
  return result;
};
```

List the functions for your account.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

Delete the IAM role and the Lambda function.

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
```

```
 *
 * @param {string} roleName
 */
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
  return client.send(command);
};

/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun main(args: Array<String>) {
```

```
   val usage = """
       Usage:
           <functionName> <role> <handler> <bucketName> <updatedBucketName>
<key>

       Where:
           functionName - The name of the AWS Lambda function.
           role - The AWS Identity and Access Management (IAM) service role that
has AWS Lambda permissions.
           handler - The fully qualified method name (for example,
example.Handler::handleRequest).
           bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
name that contains the ZIP or JAR used for the Lambda function's code.
           updatedBucketName - The Amazon S3 bucket name that contains the .zip
or .jar used to update the Lambda function's code.
           key - The Amazon S3 key name that represents the .zip or .jar file
(for example, LambdaHello-1.0-SNAPSHOT.jar).
           """

   if (args.size != 6) {
       println(usage)
       exitProcess(1)
   }

   val functionName = args[0]
   val role = args[1]
   val handler = args[2]
   val bucketName = args[3]
   val updatedBucketName = args[4]
   val key = args[5]

   println("Creating a Lambda function named $functionName.")
   val funArn = createScFunction(functionName, bucketName, key, handler, role)
   println("The AWS Lambda ARN is $funArn")

   // Get a specific Lambda function.
   println("Getting the $functionName AWS Lambda function.")
   getFunction(functionName)

   // List the Lambda functions.
   println("Listing all AWS Lambda functions.")
   listFunctionsSc()

   // Invoke the Lambda function.
```

```
    println("*** Invoke the Lambda function.")
    invokeFunctionSc(functionName)

    // Update the AWS Lambda function code.
    println("*** Update the Lambda function code.")
    updateFunctionCode(functionName, updatedBucketName, key)

    // println("*** Invoke the function again after updating the code.")
    invokeFunctionSc(functionName)

    // Update the AWS Lambda function configuration.
    println("Update the run time of the function.")
    updateFunctionConfiguration(functionName, handler)

    // Delete the AWS Lambda function.
    println("Delete the AWS Lambda function.")
    delFunction(functionName)
}

suspend fun createScFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java17
        }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-east-1" }.use { awsLambda ->
```

```
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn.toString()
    }
}

suspend fun getFunction(functionNameVal: String) {
    val functionRequest =
        GetFunctionRequest {
            functionName = functionNameVal
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.getFunction(functionRequest)
        println("The runtime of this Lambda function is
 ${response.configuration?.runtime}")
    }
}

suspend fun listFunctionsSc() {
    val request =
        ListFunctionsRequest {
            maxItems = 10
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.listFunctions(request)
        response.functions?.forEach { function ->
            println("The function name is ${function.functionName}")
        }
    }
}

suspend fun invokeFunctionSc(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            payload = byteArray
            logType = LogType.Tail
        }
```

```
    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("The function payload is
 ${res.payload?.toString(Charsets.UTF_8)}")
    }
}

suspend fun updateFunctionCode(
    functionNameVal: String?,
    bucketName: String?,
    key: String?,
) {
    val functionCodeRequest =
        UpdateFunctionCodeRequest {
            functionName = functionNameVal
            publish = true
            s3Bucket = bucketName
            s3Key = key
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.updateFunctionCode(functionCodeRequest)
        awsLambda.waitUntilFunctionUpdated {
            functionName = functionNameVal
        }
        println("The last modified value is " + response.lastModified)
    }
}

suspend fun updateFunctionConfiguration(
    functionNameVal: String?,
    handlerVal: String?,
) {
    val configurationRequest =
        UpdateFunctionConfigurationRequest {
            functionName = functionNameVal
            handler = handlerVal
            runtime = Runtime.Java17
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.updateFunctionConfiguration(configurationRequest)
    }
```

```
}

suspend fun delFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- For API details, see the following topics in *AWS SDK for Kotlin API reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
```

```
use Iam\IAMService;

class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("---------------------------------------\n");
        print("Welcome to the AWS Lambda getting started demo using PHP!\n");
        echo("---------------------------------------\n");

        $clientArgs = [
            'region' => 'us-west-2',
            'version' => 'latest',
            'profile' => 'default',
        ];
        $uniqid = uniqid();

        $iamService = new IAMService();
        $s3client = new S3Client($clientArgs);
        $lambdaService = new LambdaService();

        echo "First, let's create a role to run our Lambda code.\n";
        $roleName = "test-lambda-role-$uniqid";
        $rolePolicyDocument = "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
                {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"Service\": \"lambda.amazonaws.com\"
                    },
                    \"Action\": \"sts:AssumeRole\"
                }
            ]
        }";
        $role = $iamService->createRole($roleName, $rolePolicyDocument);
        echo "Created role {$role['RoleName']}.\n";

        $iamService->attachRolePolicy(
            $role['RoleName'],
            "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        );
```

```
        echo "Attached the AWSLambdaBasicExecutionRole to {$role['RoleName']}.
\n";

        echo "\nNow let's create an S3 bucket and upload our Lambda code there.
\n";
        $bucketName = "amzn-s3-demo-bucket-$uniqid";
        $s3client->createBucket([
            'Bucket' => $bucketName,
        ]);
        echo "Created bucket $bucketName.\n";

        $functionName = "doc_example_lambda_$uniqid";
        $codeBasic = __DIR__ . "/lambda_handler_basic.zip";
        $handler = "lambda_handler_basic";
        $file = file_get_contents($codeBasic);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "Uploaded the Lambda code.\n";

        $createLambdaFunction = $lambdaService->createFunction($functionName,
 $role, $bucketName, $handler);
        // Wait until the function has finished being created.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['State'] == "Pending");
        echo "Created Lambda function {$getLambdaFunction['Configuration']
['FunctionName']}.\n";

        sleep(1);

        echo "\nOk, let's invoke that Lambda code.\n";
        $basicParams = [
            'action' => 'increment',
            'number' => 3,
        ];
        /** @var Stream $invokeFunction */
        $invokeFunction = $lambdaService->invoke($functionName, $basicParams)
['Payload'];
        $result = json_decode($invokeFunction->getContents())->result;
```

```
        echo "After invoking the Lambda code with the input of
 {$basicParams['number']} we received $result.\n";

        echo "\nSince that's working, let's update the Lambda code.\n";
        $codeCalculator = "lambda_handler_calculator.zip";
        $handlerCalculator = "lambda_handler_calculator";
        echo "First, put the new code into the S3 bucket.\n";
        $file = file_get_contents($codeCalculator);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "New code uploaded.\n";

        $lambdaService->updateFunctionCode($functionName, $bucketName,
 $functionName);
        // Wait for the Lambda code to finish updating.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "New Lambda code uploaded.\n";

        $environment = [
            'Variable' => ['Variables' => ['LOG_LEVEL' => 'DEBUG']],
        ];
        $lambdaService->updateFunctionConfiguration($functionName,
 $handlerCalculator, $environment);
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "Lambda code updated with new handler and a LOG_LEVEL of DEBUG for
 more information.\n";

        echo "Invoke the new code with some new data.\n";
        $calculatorParams = [
            'action' => 'plus',
            'x' => 5,
            'y' => 4,
        ];
```

```
        $invokeFunction = $lambdaService->invoke($functionName,
$calculatorParams, "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "Indeed, {$calculatorParams['x']} + {$calculatorParams['y']} does
equal $result.\n";
        echo "Here's the extra debug info: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nBut what happens if you try to divide by zero?\n";
        $divZeroParams = [
            'action' => 'divide',
            'x' => 5,
            'y' => 0,
        ];
        $invokeFunction = $lambdaService->invoke($functionName, $divZeroParams,
"Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "You get a |$result| result.\n";
        echo "And an error message: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nHere's all the Lambda functions you have in this Region:\n";
        $listLambdaFunctions = $lambdaService->listFunctions(5);
        $allLambdaFunctions = $listLambdaFunctions['Functions'];
        $next = $listLambdaFunctions->get('NextMarker');
        while ($next != false) {
            $listLambdaFunctions = $lambdaService->listFunctions(5, $next);
            $next = $listLambdaFunctions->get('NextMarker');
            $allLambdaFunctions = array_merge($allLambdaFunctions,
$listLambdaFunctions['Functions']);
        }
        foreach ($allLambdaFunctions as $function) {
            echo "{$function['FunctionName']}\n";
        }

        echo "\n\nAnd don't forget to clean up your data!\n";

        $lambdaService->deleteFunction($functionName);
        echo "Deleted Lambda function.\n";
        $iamService->deleteRole($role['RoleName']);
        echo "Deleted Role.\n";
        $deleteObjects = $s3client->listObjectsV2([
            'Bucket' => $bucketName,
        ]);
```

```
        $deleteObjects = $s3client->deleteObjects([
            'Bucket' => $bucketName,
            'Delete' => [
                'Objects' => $deleteObjects['Contents'],
            ]
        ]);
        echo "Deleted all objects from the S3 bucket.\n";
        $s3client->deleteBucket(['Bucket' => $bucketName]);
        echo "Deleted the bucket.\n";
    }
}
```

- For API details, see the following topics in *AWS SDK for PHP API Reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Define a Lambda handler that increments a number.

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)
```

```
def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

Define a second Lambda handler that performs arithmetic operations.

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}
```

```
def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

Create functions that wrap Lambda actions.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
```

```
        self.iam_resource = iam_resource


    @staticmethod
    def create_deployment_package(source_file, destination_file):
        """
        Creates a Lambda deployment package in .zip format in an in-memory
buffer. This
        buffer can be passed directly to Lambda when creating the function.

        :param source_file: The name of the file that contains the Lambda handler
                            function.
        :param destination_file: The name to give the file when it's deployed to
Lambda.
        :return: The deployment package.
        """
        buffer = io.BytesIO()
        with zipfile.ZipFile(buffer, "w") as zipped:
            zipped.write(source_file, destination_file)
        buffer.seek(0)
        return buffer.read()

    def get_iam_role(self, iam_role_name):
        """
        Get an AWS Identity and Access Management (IAM) role.

        :param iam_role_name: The name of the role to retrieve.
        :return: The IAM role.
        """
        role = None
        try:
            temp_role = self.iam_resource.Role(iam_role_name)
            temp_role.load()
            role = temp_role
            logger.info("Got IAM role %s", role.name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "NoSuchEntity":
                logger.info("IAM role %s does not exist.", iam_role_name)
            else:
                logger.error(
                    "Couldn't get IAM role %s. Here's why: %s: %s",
                    iam_role_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
```

```
                )
                raise
        return role

    def create_iam_role_for_lambda(self, iam_role_name):
        """
        Creates an IAM role that grants the Lambda function basic permissions. If
 a
        role with the specified name already exists, it is used for the demo.

        :param iam_role_name: The name of the role to create.
        :return: The role and a value that indicates whether the role is newly
 created.
        """
        role = self.get_iam_role(iam_role_name)
        if role is not None:
            return role, False

        lambda_assume_role_policy = {
            "Version":"2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {"Service": "lambda.amazonaws.com"},
                    "Action": "sts:AssumeRole",
                }
            ],
        }
        policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

        try:
            role = self.iam_resource.create_role(
                RoleName=iam_role_name,
                AssumeRolePolicyDocument=json.dumps(lambda_assume_role_policy),
            )
            logger.info("Created role %s.", role.name)
            role.attach_policy(PolicyArn=policy_arn)
            logger.info("Attached basic execution policy to role %s.", role.name)
        except ClientError as error:
            if error.response["Error"]["Code"] == "EntityAlreadyExists":
                role = self.iam_resource.Role(iam_role_name)
                logger.warning("The role %s already exists. Using it.",
 iam_role_name)
```

```
            else:
                logger.exception(
                    "Couldn't create role %s or attach policy %s.",
                    iam_role_name,
                    policy_arn,
                )
                raise

        return role, True

    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
```

```
                                must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                     code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

```
   def invoke_function(self, function_name, function_params, get_log=False):
       """
       Invokes a Lambda function.

       :param function_name: The name of the function to invoke.
       :param function_params: The parameters of the function as a dict. This
dict
                               is serialized to JSON before it is sent to
Lambda.
       :param get_log: When true, the last 4 KB of the execution log are
included in
                       the response.
       :return: The response from the function invocation.
       """
       try:
           response = self.lambda_client.invoke(
               FunctionName=function_name,
               Payload=json.dumps(function_params),
               LogType="Tail" if get_log else "None",
           )
           logger.info("Invoked function %s.", function_name)
       except ClientError:
           logger.exception("Couldn't invoke function %s.", function_name)
           raise
       return response


   def update_function_code(self, function_name, deployment_package):
       """
       Updates the code for a Lambda function by submitting a .zip archive that
contains
       the code for the function.

       :param function_name: The name of the function to update.
       :param deployment_package: The function code to update, packaged as bytes
in
                                  .zip format.
       :return: Data about the update, including the status.
       """
       try:
           response = self.lambda_client.update_function_code(
               FunctionName=function_name, ZipFile=deployment_package
           )
```

```
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
```

```
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

Create a function that runs the scenario.

```
class UpdateFunctionWaiter(CustomWaiter):
    """A custom waiter that waits until a function is successfully updated."""

    def __init__(self, client):
        super().__init__(
            "UpdateSuccess",
            "GetFunction",
            "Configuration.LastUpdateStatus",
            {"Successful": WaitState.SUCCESS, "Failed": WaitState.FAILURE},
            client,
        )

    def wait(self, function_name):
        self._wait(FunctionName=function_name)


def run_scenario(lambda_client, iam_resource, basic_file, calculator_file,
 lambda_name):
    """
    Runs the scenario.

    :param lambda_client: A Boto3 Lambda client.
    :param iam_resource: A Boto3 IAM resource.
```

```
   :param basic_file: The name of the file that contains the basic Lambda
handler.
   :param calculator_file: The name of the file that contains the calculator
Lambda handler.
   :param lambda_name: The name to give resources created for the scenario, such
as the
                       IAM role and the Lambda function.
   """
   logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

   print("-" * 88)
   print("Welcome to the AWS Lambda getting started with functions demo.")
   print("-" * 88)

   wrapper = LambdaWrapper(lambda_client, iam_resource)

   print("Checking for IAM role for Lambda...")
   iam_role, should_wait = wrapper.create_iam_role_for_lambda(lambda_name)
   if should_wait:
       logger.info("Giving AWS time to create resources...")
       wait(10)

   print(f"Looking for function {lambda_name}...")
   function = wrapper.get_function(lambda_name)
   if function is None:
       print("Zipping the Python script into a deployment package...")
       deployment_package = wrapper.create_deployment_package(
           basic_file, f"{lambda_name}.py"
       )
       print(f"...and creating the {lambda_name} Lambda function.")
       wrapper.create_function(
           lambda_name, f"{lambda_name}.lambda_handler", iam_role,
deployment_package
       )
   else:
       print(f"Function {lambda_name} already exists.")
   print("-" * 88)

   print(f"Let's invoke {lambda_name}. This function increments a number.")
   action_params = {
       "action": "increment",
       "number": q.ask("Give me a number to increment: ", q.is_int),
   }
   print(f"Invoking {lambda_name}...")
```

```
    response = wrapper.invoke_function(lambda_name, action_params)
    print(
        f"Incrementing {action_params['number']} resulted in "
        f"{json.load(response['Payload'])}"
    )
    print("-" * 88)

    print(f"Let's update the function to an arithmetic calculator.")
    q.ask("Press Enter when you're ready.")
    print("Creating a new deployment package...")
    deployment_package = wrapper.create_deployment_package(
        calculator_file, f"{lambda_name}.py"
    )
    print(f"...and updating the {lambda_name} Lambda function.")
    update_waiter = UpdateFunctionWaiter(lambda_client)
    wrapper.update_function_code(lambda_name, deployment_package)
    update_waiter.wait(lambda_name)
    print(f"This function uses an environment variable to control logging
level.")
    print(f"Let's set it to DEBUG to get the most logging.")
    wrapper.update_function_configuration(
        lambda_name, {"LOG_LEVEL": logging.getLevelName(logging.DEBUG)}
    )

    actions = ["plus", "minus", "times", "divided-by"]
    want_invoke = True
    while want_invoke:
        print(f"Let's invoke {lambda_name}. You can invoke these actions:")
        for index, action in enumerate(actions):
            print(f"{index + 1}: {action}")
        action_params = {}
        action_index = q.ask(
            "Enter the number of the action you want to take: ",
            q.is_int,
            q.in_range(1, len(actions)),
        )
        action_params["action"] = actions[action_index - 1]
        print(f"You've chosen to invoke 'x {action_params['action']} y'.")
        action_params["x"] = q.ask("Enter a value for x: ", q.is_int)
        action_params["y"] = q.ask("Enter a value for y: ", q.is_int)
        print(f"Invoking {lambda_name}...")
        response = wrapper.invoke_function(lambda_name, action_params, True)
        print(
```

```
            f"Calculating {action_params['x']} {action_params['action']}
 {action_params['y']} "
            f"resulted in {json.load(response['Payload'])}"
        )
        q.ask("Press Enter to see the logs from the call.")
        print(base64.b64decode(response["LogResult"]).decode())
        want_invoke = q.ask("That was fun. Shall we do it again? (y/n) ",
 q.is_yesno)
    print("-" * 88)

    if q.ask(
        "Do you want to list all of the functions in your account? (y/n) ",
 q.is_yesno
    ):
        wrapper.list_functions()
    print("-" * 88)

    if q.ask("Ready to delete the function and role? (y/n) ", q.is_yesno):
        for policy in iam_role.attached_policies.all():
            policy.detach_role(RoleName=iam_role.name)
        iam_role.delete()
        print(f"Deleted role {lambda_name}.")
        wrapper.delete_function(lambda_name)
        print(f"Deleted function {lambda_name}.")

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        run_scenario(
            boto3.client("lambda"),
            boto3.resource("iam"),
            "lambda_handler_basic.py",
            "lambda_handler_calculator.py",
            "doc_example_lambda_calculator",
        )
    except Exception:
        logging.exception("Something went wrong with the demo!")
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.

- CreateFunction
- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Set up pre-requisite IAM permissions for a Lambda function capable of writing logs.

```
# Get an AWS Identity and Access Management (IAM) role.
#
# @param iam_role_name: The name of the role to retrieve.
# @param action: Whether to create or destroy the IAM apparatus.
# @return: The IAM role.
def manage_iam(iam_role_name, action)
  case action
  when 'create'
    create_iam_role(iam_role_name)
  when 'destroy'
    destroy_iam_role(iam_role_name)
  else
    raise "Incorrect action provided. Must provide 'create' or 'destroy'"
  end
end

private

def create_iam_role(iam_role_name)
```

```
    role_policy = {
      'Version': '2012-10-17',
      'Statement': [
        {
          'Effect': 'Allow',
          'Principal': { 'Service': 'lambda.amazonaws.com' },
          'Action': 'sts:AssumeRole'
        }
      ]
    }
    role = @iam_client.create_role(
      role_name: iam_role_name,
      assume_role_policy_document: role_policy.to_json
    )
    @iam_client.attach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    wait_for_role_to_exist(iam_role_name)
    @logger.debug("Successfully created IAM role: #{role['role']['arn']}")
    sleep(10)
    [role, role_policy.to_json]
  end

  def destroy_iam_role(iam_role_name)
    @iam_client.detach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    @iam_client.delete_role(role_name: iam_role_name)
    @logger.debug("Detached policy & deleted IAM role: #{iam_role_name}")
  end

  def wait_for_role_to_exist(iam_role_name)
    @iam_client.wait_until(:role_exists, { role_name: iam_role_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
```

```
  end
```

Define a Lambda handler that increments a number provided as an invocation parameter.

```
require 'logger'

# A function that increments a whole number by one (1) and logs the result.
# Requires a manually-provided runtime parameter, 'number', which must be Int
#
# @param event [Hash] Parameters sent when the function is invoked
# @param context [Hash] Methods and properties that provide information
# about the invocation, function, and execution environment.
# @return incremented_number [String] The incremented number.
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  log_level = ENV['LOG_LEVEL']
  logger.level = case log_level
                 when 'debug'
                   Logger::DEBUG
                 when 'info'
                   Logger::INFO
                 else
                   Logger::ERROR
                 end
  logger.debug('This is a debug log message.')
  logger.info('This is an info log message. Code executed successfully!')
  number = event['number'].to_i
  incremented_number = number + 1
  logger.info("You provided #{number.round} and it was incremented to
 #{incremented_number.round}")
  incremented_number.round.to_s
end
```

Zip your Lambda function into a deployment package.

```
  # Creates a Lambda deployment package in .zip format.
  #
  # @param source_file: The name of the object, without suffix, for the Lambda
 file and zip.
  # @return: The deployment package.
  def create_deployment_package(source_file)
```

```
    Dir.chdir(File.dirname(__FILE__))
    if File.exist?('lambda_function.zip')
      File.delete('lambda_function.zip')
      @logger.debug('Deleting old zip: lambda_function.zip')
    end
    Zip::File.open('lambda_function.zip', create: true) do |zipfile|
      zipfile.add('lambda_function.rb', "#{source_file}.rb")
    end
    @logger.debug("Zipping #{source_file}.rb into: lambda_function.zip.")
    File.read('lambda_function.zip').to_s
  rescue StandardError => e
    @logger.error("There was an error creating deployment package:\n
#{e.message}")
  end
```

Create a new Lambda function.

```
  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: 'ruby2.7',
                                                code: {
                                                  zip_file: deployment_package
                                                },
                                                environment: {
                                                  variables: {
                                                    'LOG_LEVEL' => 'info'
                                                  }
                                                }
                                              })
    @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
```

```
      w.max_attempts = 5
      w.delay = 5
    end
    response
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

Invoke your Lambda function with optional runtime parameters.

```
  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

Update your Lambda function's configuration to inject a new environment variable.

```
  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
                                                     }
                                                   }
                                                 })
```

```
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

Update your Lambda function's code with a different deployment package containing different code.

```
  # Updates the code for a Lambda function by submitting a .zip archive that
contains
  # the code for the function.
  #
  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

List all existing Lambda functions using the built-in paginator.

```
# Lists the Lambda functions for the current account.
def list_functions
  functions = []
  @lambda_client.list_functions.each do |response|
    response['functions'].each do |function|
      functions.append(function['function_name'])
    end
  end
  functions
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error listing functions:\n #{e.message}")
end
```

Delete a specific Lambda function.

```
# Deletes a Lambda function.
# @param function_name: The name of the function to delete.
def delete_function(function_name)
  print "Deleting function: #{function_name}..."
  @lambda_client.delete_function(
    function_name: function_name
  )
  print 'Done!'.green
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
end
```

- For API details, see the following topics in *AWS SDK for Ruby API Reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode

- UpdateFunctionConfiguration

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

The Cargo.toml with dependencies used in this scenario.

```
[package]
name = "lambda-code-examples"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[dependencies]
aws-config = { version = "1.0.1", features = ["behavior-version-latest"] }
aws-sdk-ec2 = { version = "1.3.0" }
aws-sdk-iam = { version = "1.3.0" }
aws-sdk-lambda = { version = "1.3.0" }
aws-sdk-s3 = { version = "1.4.0" }
aws-smithy-types = { version = "1.0.1" }
aws-types = { version = "1.0.1" }
clap = { version = "4.4", features = ["derive"] }
tokio = { version = "1.20.1", features = ["full"] }
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
tracing = "0.1.37"
serde_json = "1.0.94"
anyhow = "1.0.71"
uuid = { version = "1.3.3", features = ["v4"] }
lambda_runtime = "0.8.0"
serde = "1.0.164"
```

A collection of utilities that streamline calling Lambda for this scenario. This file is src/ations.rs in the crate.

```
use anyhow::anyhow;
use aws_sdk_iam::operation::{create_role::CreateRoleError,
 delete_role::DeleteRoleOutput};
use aws_sdk_lambda::{
    operation::{
        delete_function::DeleteFunctionOutput, get_function::GetFunctionOutput,
        invoke::InvokeOutput, list_functions::ListFunctionsOutput,
        update_function_code::UpdateFunctionCodeOutput,
        update_function_configuration::UpdateFunctionConfigurationOutput,
    },
    primitives::ByteStream,
    types::{Environment, FunctionCode, LastUpdateStatus, State},
};
use aws_sdk_s3::{
    error::ErrorMetadata,
    operation::{delete_bucket::DeleteBucketOutput,
 delete_object::DeleteObjectOutput},
    types::CreateBucketConfiguration,
};
use aws_smithy_types::Blob;
use serde::{ser::SerializeMap, Serialize};
use std::{fmt::Display, path::PathBuf, str::FromStr, time::Duration};
use tracing::{debug, info, warn};

/* Operation describes  */
#[derive(Clone, Copy, Debug, Serialize)]
pub enum Operation {
    #[serde(rename = "plus")]
    Plus,
    #[serde(rename = "minus")]
    Minus,
    #[serde(rename = "times")]
    Times,
    #[serde(rename = "divided-by")]
    DividedBy,
}

impl FromStr for Operation {
    type Err = anyhow::Error;
```

```
    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s {
            "plus" => Ok(Operation::Plus),
            "minus" => Ok(Operation::Minus),
            "times" => Ok(Operation::Times),
            "divided-by" => Ok(Operation::DividedBy),
            _ => Err(anyhow!("Unknown operation {s}")),
        }
    }
}

impl Display for Operation {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Operation::Plus => write!(f, "plus"),
            Operation::Minus => write!(f, "minus"),
            Operation::Times => write!(f, "times"),
            Operation::DividedBy => write!(f, "divided-by"),
        }
    }
}

/**
 * InvokeArgs will be serialized as JSON and sent to the AWS Lambda handler.
 */
#[derive(Debug)]
pub enum InvokeArgs {
    Increment(i32),
    Arithmetic(Operation, i32, i32),
}

impl Serialize for InvokeArgs {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match self {
            InvokeArgs::Increment(i) => serializer.serialize_i32(*i),
            InvokeArgs::Arithmetic(o, i, j) => {
                let mut map: S::SerializeMap =
 serializer.serialize_map(Some(3))?;
                map.serialize_key(&"op".to_string())?;
                map.serialize_value(&o.to_string())?;
```

```
                map.serialize_key(&"i".to_string())?;
                map.serialize_value(&i)?;
                map.serialize_key(&"j".to_string())?;
                map.serialize_value(&j)?;
                map.end()
            }
        }
    }
}

/** A policy document allowing Lambda to execute this function on the account's
 behalf. */
const ROLE_POLICY_DOCUMENT: &str = r#"{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": "sts:AssumeRole"
        }
    ]
}"#;

/**
 * A LambdaManager gathers all the resources necessary to run the Lambda example
 scenario.
 * This includes instantiated aws_sdk clients and details of resource names.
 */
pub struct LambdaManager {
    iam_client: aws_sdk_iam::Client,
    lambda_client: aws_sdk_lambda::Client,
    s3_client: aws_sdk_s3::Client,
    lambda_name: String,
    role_name: String,
    bucket: String,
    own_bucket: bool,
}

// These unit type structs provide nominal typing on top of String parameters for
 LambdaManager::new
pub struct LambdaName(pub String);
pub struct RoleName(pub String);
pub struct Bucket(pub String);
pub struct OwnBucket(pub bool);
```

```
impl LambdaManager {
    pub fn new(
        iam_client: aws_sdk_iam::Client,
        lambda_client: aws_sdk_lambda::Client,
        s3_client: aws_sdk_s3::Client,
        lambda_name: LambdaName,
        role_name: RoleName,
        bucket: Bucket,
        own_bucket: OwnBucket,
    ) -> Self {
        Self {
            iam_client,
            lambda_client,
            s3_client,
            lambda_name: lambda_name.0,
            role_name: role_name.0,
            bucket: bucket.0,
            own_bucket: own_bucket.0,
        }
    }

    /**
     * Load the AWS configuration from the environment.
     * Look up lambda_name and bucket if none are given, or generate a random
 name if not present in the environment.
     * If the bucket name is provided, the caller needs to have created the
 bucket.
     * If the bucket name is generated, it will be created.
     */
    pub async fn load_from_env(lambda_name: Option<String>, bucket:
 Option<String>) -> Self {
        let sdk_config = aws_config::load_from_env().await;
        let lambda_name = LambdaName(lambda_name.unwrap_or_else(|| {
            std::env::var("LAMBDA_NAME").unwrap_or_else(|_|
 "rust_lambda_example".to_string())
        }));
        let role_name = RoleName(format!("{}_role", lambda_name.0));
        let (bucket, own_bucket) =
            match bucket {
                Some(bucket) => (Bucket(bucket), false),
                None => (
                    Bucket(std::env::var("LAMBDA_BUCKET").unwrap_or_else(|_| {
                        format!("rust-lambda-example-{}", uuid::Uuid::new_v4())
```

```
                    })),
                    true,
                ),
            };

        let s3_client = aws_sdk_s3::Client::new(&sdk_config);

        if own_bucket {
            info!("Creating bucket for demo: {}", bucket.0);
            s3_client
                .create_bucket()
                .bucket(bucket.0.clone())
                .create_bucket_configuration(
                    CreateBucketConfiguration::builder()

 .location_constraint(aws_sdk_s3::types::BucketLocationConstraint::from(
                            sdk_config.region().unwrap().as_ref(),
                        ))
                        .build(),
                )
                .send()
                .await
                .unwrap();
        }

        Self::new(
            aws_sdk_iam::Client::new(&sdk_config),
            aws_sdk_lambda::Client::new(&sdk_config),
            s3_client,
            lambda_name,
            role_name,
            bucket,
            OwnBucket(own_bucket),
        )
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
```

```
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }

    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
```

```
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Create an IAM execution role for the managed Lambda function.
     * If the role already exists, use that instead.
     */
    async fn create_role(&self) -> Result<aws_sdk_iam::types::Role,
CreateRoleError> {
        info!("Creating execution role for function");
        let get_role = self
            .iam_client
            .get_role()
            .role_name(self.role_name.clone())
            .send()
            .await;
        if let Ok(get_role) = get_role {
            if let Some(role) = get_role.role {
                return Ok(role);
            }
        }

        let create_role = self
            .iam_client
            .create_role()
            .role_name(self.role_name.clone())
            .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
            .send()
            .await;
```

```
        match create_role {
            Ok(create_role) => match create_role.role {
                Some(role) => Ok(role),
                None => Err(CreateRoleError::generic(
                    ErrorMetadata::builder()
                        .message("CreateRole returned empty success")
                        .build(),
                )),
            },
            Err(err) => Err(err.into_service_error()),
        }
    }

    /**
     * Poll `is_function_ready` with a 1-second delay. It returns when the
function is ready or when there's an error checking the function's state.
     */
    pub async fn wait_for_function_ready(&self) -> Result<(), anyhow::Error> {
        info!("Waiting for function");
        while !self.is_function_ready(None).await? {
            info!("Function is not ready, sleeping 1s");
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
        Ok(())
    }

    /**
     * Check if a Lambda function is ready to be invoked.
     * A Lambda function is ready for this scenario when its state is active and
its LastUpdateStatus is Successful.
     * Additionally, if a sha256 is provided, the function must have that as its
current code hash.
     * Any missing properties or failed requests will be reported as an Err.
     */
    async fn is_function_ready(
        &self,
        expected_code_sha256: Option<&str>,
    ) -> Result<bool, anyhow::Error> {
        match self.get_function().await {
            Ok(func) => {
                if let Some(config) = func.configuration() {
                    if let Some(state) = config.state() {
                        info!(?state, "Checking if function is active");
                        if !matches!(state, State::Active) {
```

```
                            return Ok(false);
                        }
                    }
                    match config.last_update_status() {
                        Some(last_update_status) => {
                            info!(?last_update_status, "Checking if function is
ready");
                            match last_update_status {
                                LastUpdateStatus::Successful => {
                                    // continue
                                }
                                LastUpdateStatus::Failed |
LastUpdateStatus::InProgress => {
                                    return Ok(false);
                                }
                                unknown => {
                                    warn!(
                                        status_variant = unknown.as_str(),
                                        "LastUpdateStatus unknown"
                                    );
                                    return Err(anyhow!(
                                        "Unknown LastUpdateStatus, fn config is
{config:?}"
                                    ));
                                }
                            }
                        }
                        None => {
                            warn!("Missing last update status");
                            return Ok(false);
                        }
                    };
                    if expected_code_sha256.is_none() {
                        return Ok(true);
                    }
                    if let Some(code_sha256) = config.code_sha256() {
                        return Ok(code_sha256 ==
expected_code_sha256.unwrap_or_default());
                    }
                }
            }
            Err(e) => {
                warn!(?e, "Could not get function while waiting");
            }
```

```
        }
        Ok(false)
    }

    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
```

```
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
```

```
    }

    /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
    pub async fn delete_function(
        &self,
        location: Option<String>,
    ) -> (
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ) {
        info!("Deleting lambda function {}", self.lambda_name);
        let delete_function = self
            .lambda_client
            .delete_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
```

```
                None
            };

        (delete_function, delete_role, delete_object)
    }

    pub async fn cleanup(
        &self,
        location: Option<String>,
    ) -> (
        (
            Result<DeleteFunctionOutput, anyhow::Error>,
            Result<DeleteRoleOutput, anyhow::Error>,
            Option<Result<DeleteObjectOutput, anyhow::Error>>,
        ),
        Option<Result<DeleteBucketOutput, anyhow::Error>>,
    ) {
        let delete_function = self.delete_function(location).await;

        let delete_bucket = if self.own_bucket {
            info!("Deleting bucket {}", self.bucket);
            if delete_function.2.is_none() ||
 delete_function.2.as_ref().unwrap().is_ok() {
                Some(
                    self.s3_client
                        .delete_bucket()
                        .bucket(self.bucket.clone())
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                None
            }
        } else {
            info!("No bucket to clean up");
            None
        };

        (delete_function, delete_bucket)
    }
}

/**
```

```
 * Testing occurs primarily as an integration test running the `scenario` bin
 successfully.
 * Each action relies deeply on the internal workings and state of Amazon Simple
 Storage Service (Amazon S3), Lambda, and IAM working together.
 * It is therefore infeasible to mock the clients to test the individual actions.
 */
#[cfg(test)]
mod test {
    use super::{InvokeArgs, Operation};
    use serde_json::json;

    /** Make sure that the JSON output of serializing InvokeArgs is what's
 expected by the calculator. */
    #[test]
    fn test_serialize() {
        assert_eq!(json!(InvokeArgs::Increment(5)), 5);
        assert_eq!(
            json!(InvokeArgs::Arithmetic(Operation::Plus, 5, 7)).to_string(),
            r#"{"op":"plus","i":5,"j":7}"#.to_string(),
        );
    }
}
```

A binary to run the scenario from front to end, using command line flags to control some behavior. This file is src/bin/scenario.rs in the crate.

```

/*
## Service actions

Service actions wrap the SDK call, taking a client and any specific parameters
 necessary for the call.

* CreateFunction
* GetFunction
* ListFunctions
* Invoke
* UpdateFunctionCode
* UpdateFunctionConfiguration
* DeleteFunction

## Scenario
```

```
A scenario runs at a command prompt and prints output to the user on the result
 of each service action. A scenario can run in one of two ways: straight through,
 printing out progress as it goes, or as an interactive question/answer script.

## Getting started with functions

Use an SDK to manage AWS Lambda functions: create a function, invoke it, update
 its code, invoke it again, view its output and logs, and delete it.

This scenario uses two Lambda handlers:
_Note: Handlers don't use AWS SDK API calls._

The increment handler is straightforward:

1. It accepts a number, increments it, and returns the new value.
2. It performs simple logging of the result.

The arithmetic handler is more complex:
1. It accepts a set of actions ['plus', 'minus', 'times', 'divided-by'] and two
 numbers, and returns the result of the calculation.
2. It uses an environment variable to control log level (such as DEBUG, INFO,
 WARNING, ERROR).
It logs a few things at different levels, such as:
    * DEBUG: Full event data.
    * INFO: The calculation result.
    * WARN~ING~: When a divide by zero error occurs.
    * This will be the typical `RUST_LOG` variable.


The steps of the scenario are:

1. Create an AWS Identity and Access Management (IAM) role that meets the
 following requirements:
    * Has an assume_role policy that grants 'lambda.amazonaws.com' the
 'sts:AssumeRole' action.
    * Attaches the 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' managed role.
    * _You must wait for ~10 seconds after the role is created before you can use
 it!_
2. Create a function (CreateFunction) for the increment handler by packaging it
 as a zip and doing one of the following:
    * Adding it with CreateFunction Code.ZipFile.
    * --or--
```

```
    * Uploading it to Amazon Simple Storage Service (Amazon S3) and adding it
 with CreateFunction Code.S3Bucket/S3Key.
    * _Note: Zipping the file does not have to be done in code._
    * If you have a waiter, use it to wait until the function is active.
 Otherwise, call GetFunction until State is Active.
3. Invoke the function with a number and print the result.
4. Update the function (UpdateFunctionCode) to the arithmetic handler by
 packaging it as a zip and doing one of the following:
    * Adding it with UpdateFunctionCode ZipFile.
    * --or--
    * Uploading it to Amazon S3 and adding it with UpdateFunctionCode S3Bucket/
S3Key.
5. Call GetFunction until Configuration.LastUpdateStatus is 'Successful' (or
 'Failed').
6. Update the environment variable by calling UpdateFunctionConfiguration and
 pass it a log level, such as:
    * Environment={'Variables': {'RUST_LOG': 'TRACE'}}
7. Invoke the function with an action from the list and a couple of values.
 Include LogType='Tail' to get logs in the result. Print the result of the
 calculation and the log.
8. [Optional] Invoke the function to provoke a divide-by-zero error and show the
 log result.
9. List all functions for the account, using pagination (ListFunctions).
10. Delete the function (DeleteFunction).
11. Delete the role.

Each step should use the function created in Service Actions to abstract calling
 the SDK.
 */

use aws_sdk_lambda::{operation::invoke::InvokeOutput, types::Environment};
use clap::Parser;
use std::{collections::HashMap, path::PathBuf};
use tracing::{debug, info, warn};
use tracing_subscriber::EnvFilter;

use lambda_code_examples::actions::{
    InvokeArgs::{Arithmetic, Increment},
    LambdaManager, Operation,
};

#[derive(Debug, Parser)]
pub struct Opt {
    /// The AWS Region.
```

```
    #[structopt(short, long)]
    pub region: Option<String>,

    // The bucket to use for the FunctionCode.
    #[structopt(short, long)]
    pub bucket: Option<String>,

    // The name of the Lambda function.
    #[structopt(short, long)]
    pub lambda_name: Option<String>,

    // The number to increment.
    #[structopt(short, long, default_value = "12")]
    pub inc: i32,

    // The left operand.
    #[structopt(long, default_value = "19")]
    pub num_a: i32,

    // The right operand.
    #[structopt(long, default_value = "23")]
    pub num_b: i32,

    // The arithmetic operation.
    #[structopt(short, long, default_value = "plus")]
    pub operation: Operation,

    #[structopt(long)]
    pub cleanup: Option<bool>,

    #[structopt(long)]
    pub no_cleanup: Option<bool>,
}

fn code_path(lambda: &str) -> PathBuf {
    PathBuf::from(format!("../target/lambda/{lambda}/bootstrap.zip"))
}

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
```

```
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}

async fn main_block(
    opt: &Opt,
    manager: &LambdaManager,
    code_location: String,
) -> Result<(), anyhow::Error> {
    let invoke = manager.invoke(Increment(opt.inc)).await?;
    log_invoke_output(&invoke, "Invoked function configured as increment");

    let update_code = manager
        .update_function_code(code_path("arithmetic"), code_location.clone())
        .await?;

    let code_sha256 = update_code.code_sha256().unwrap_or("Unknown SHA");
    info!(?code_sha256, "Updated function code with arithmetic.zip");

    let arithmetic_args = Arithmetic(opt.operation, opt.num_a, opt.num_b);
    let invoke = manager.invoke(arithmetic_args).await?;
    log_invoke_output(&invoke, "Invoked function configured as arithmetic");

    let update = manager
        .update_function_configuration(
            Environment::builder()
                .set_variables(Some(HashMap::from([(
                    "RUST_LOG".to_string(),
                    "trace".to_string(),
                )])))
                .build(),
        )
        .await?;
    let updated_environment = update.environment();
    info!(?updated_environment, "Updated function configuration");

    let invoke = manager
        .invoke(Arithmetic(opt.operation, opt.num_a, opt.num_b))
        .await?;
    log_invoke_output(
```

```
        &invoke,
        "Invoked function configured as arithmetic with increased logging",
    );

    let invoke = manager
        .invoke(Arithmetic(Operation::DividedBy, opt.num_a, 0))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with divide by zero",
    );

    Ok::<(), anyhow::Error>(())
}

#[tokio::main]
async fn main() {
    tracing_subscriber::fmt()
        .without_time()
        .with_file(true)
        .with_line_number(true)
        .with_env_filter(EnvFilter::from_default_env())
        .init();

    let opt = Opt::parse();
    let manager = LambdaManager::load_from_env(opt.lambda_name.clone(),
 opt.bucket.clone()).await;

    let key = match manager.create_function(code_path("increment")).await {
        Ok(init) => {
            info!(?init, "Created function, initially with increment.zip");
            let run_block = main_block(&opt, &manager, init.clone()).await;
            info!(?run_block, "Finished running example, cleaning up");
            Some(init)
        }
        Err(err) => {
            warn!(?err, "Error happened when initializing function");
            None
        }
    };

    if Some(false) == opt.cleanup || Some(true) == opt.no_cleanup {
        info!("Skipping cleanup")
    } else {
```

```
        let delete = manager.cleanup(key).await;
        info!(?delete, "Deleted function & cleaned up resources");
    }
}
```

- For API details, see the following topics in *AWS SDK for Rust API reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   TRY.
       "Create an AWS Identity and Access Management (IAM) role that grants AWS
Lambda permission to write to logs."
       DATA(lv_policy_document) = `{` &&
           `"Version":"2012-10-17",       ` &&
                 `"Statement": [` &&
                   `{` &&
                     `"Effect": "Allow",` &&
                     `"Action": [` &&
                       `"sts:AssumeRole"` &&
                     `],` &&
                     `"Principal": {` &&
```

```
                        `"Service": [` &&
                          `"lambda.amazonaws.com"` &&
                        `]` &&
                      `}` &&
                    `}` &&
                  `]` &&
                `}`.
        TRY.
            DATA(lo_create_role_output) = lo_iam->createrole(
                    iv_rolename = iv_role_name
                    iv_assumerolepolicydocument = lv_policy_document
                    iv_description = 'Grant lambda permission to write to
 logs' ).
            DATA(lv_role_arn) = lo_create_role_output->get_role( )->get_arn( ).
            MESSAGE 'IAM role created.' TYPE 'I'.
            WAIT UP TO 10 SECONDS.            " Make sure that the IAM role is
 ready for use. "
          CATCH /aws1/cx_iamentityalrdyexex.
            DATA(lo_role) = lo_iam->getrole( iv_rolename = iv_role_name ).
            lv_role_arn = lo_role->get_role( )->get_arn( ).
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iammalformedplydocex.
            MESSAGE 'Policy document in the request is malformed.' TYPE 'E'.
        ENDTRY.

        TRY.
            lo_iam->attachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' ).
            MESSAGE 'Attached policy to the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.
```

```
        " Create a Lambda function and upload handler code. "
        " Lambda function performs 'increment' action on a number. "
        TRY.
            lo_lmd->createfunction(
                 iv_functionname = iv_function_name
                 iv_runtime = `python3.9`
                 iv_role = lv_role_arn
                 iv_handler = iv_handler
                 io_code = io_initial_zip_file
                 iv_description = 'AWS Lambda code example' ).
            MESSAGE 'Lambda function created.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Verify the function is in Active state "
        WHILE lo_lmd->getfunction( iv_functionname = iv_function_name )-
>get_configuration( )->ask_state( ) <> 'Active'.
          IF sy-index = 10.
            EXIT.               " Maximum 10 seconds. "
          ENDIF.
          WAIT UP TO 1 SECONDS.
        ENDWHILE.

        "Invoke the function with a single parameter and get results."
        TRY.
            DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "increment",`  &&
                `"number": 10` &&
              `}` ).
            DATA(lo_initial_invoke_output) = lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json ).
            ov_initial_invoke_payload = lo_initial_invoke_output->get_payload( ).
           " ov_initial_invoke_payload is returned for testing purposes. "
            DATA(lo_writer_json) = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_initial_invoke_payload RESULT
 XML lo_writer_json.
```

```
            DATA(lv_result) = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " Update the function code and configure its Lambda environment with an
 environment variable. "
        " Lambda function is updated to perform 'decrement' action also. "
        TRY.
            lo_lmd->updatefunctioncode(
                  iv_functionname = iv_function_name
                  iv_zipfile = io_updated_zip_file ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function code updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        TRY.
            DATA lt_variables TYPE /aws1/
cl_lmdenvironmentvaria00=>tt_environmentvariables.
            DATA ls_variable LIKE LINE OF lt_variables.
            ls_variable-key = 'LOG_LEVEL'.
            ls_variable-value = NEW /aws1/cl_lmdenvironmentvaria00( iv_value =
 'info' ).
            INSERT ls_variable INTO TABLE lt_variables.

            lo_lmd->updatefunctionconfiguration(
                  iv_functionname = iv_function_name
```

```
                  io_environment = NEW /aws1/cl_lmdenvironment( it_variables =
 lt_variables ) ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourceconflictex.
            MESSAGE 'Resource already exists or another operation is in
 progress.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        "Invoke the function with new parameters and get results. Display the
 execution log that's returned from the invocation."
        TRY.
            lv_json = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "decrement",`  &&
                `"number": 10` &&
              `}` ).
            DATA(lo_updated_invoke_output) = lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json ).
            ov_updated_invoke_payload = lo_updated_invoke_output->get_payload( ).
           " ov_updated_invoke_payload is returned for testing purposes. "
            lo_writer_json = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_updated_invoke_payload RESULT
 XML lo_writer_json.
            lv_result = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.
```

```
        " List the functions for your account. "
        TRY.
            DATA(lo_list_output) = lo_lmd->listfunctions( ).
            DATA(lt_functions) = lo_list_output->get_functions( ).
            MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
        ENDTRY.

        " Delete the Lambda function. "
        TRY.
            lo_lmd->deletefunction( iv_functionname = iv_function_name ).
            MESSAGE 'Lambda function deleted.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'W'.
        ENDTRY.

        " Detach role policy. "
        TRY.
            lo_iam->detachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' ).
            MESSAGE 'Detached policy from the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'W'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Delete the IAM role. "
        TRY.
            lo_iam->deleterole( iv_rolename = iv_role_name ).
            MESSAGE 'IAM role deleted.' TYPE 'I'.
          CATCH /aws1/cx_iamnosuchentityex.
```

```
            MESSAGE 'The requested resource entity does not exist.' TYPE 'W'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
modifiable.' TYPE 'E'.
        ENDTRY.

      CATCH /aws1/cx_rt_service_generic INTO lo_exception.
        DATA(lv_error) = lo_exception->get_longtext( ).
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- For API details, see the following topics in *AWS SDK for SAP ABAP API reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Define the first Lambda function, which simply increments the specified value.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.
```

```
import PackageDescription

let package = Package(
    name: "increment",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            branch: "main"),
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "increment",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
            ],
            path: "Sources"
        )
    ]
)

import Foundation
import AWSLambdaRuntime

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct Request: Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The number to act upon.
    let number: Int
}
```

```
/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}


/// The Lambda function body.
///
/// - Parameters:
///   - event: The `Request` describing the request made by the
///     client.
///   - context: A `LambdaContext` describing the context in
///     which the lambda function is running.
///
/// - Returns: A `Response` object that will be encoded to JSON and sent
///   to the client by the Lambda runtime.
let incrementLambdaRuntime = LambdaRuntime {
        (event: Request, context: LambdaContext) -> Response in
    let action = event.action
    var answer: Int?

    if action != "increment" {
        context.logger.error("Unrecognized operation: \"\(action)\". The only
 supported action is \"increment\".")
    } else {
        answer = event.number + 1
        context.logger.info("The calculated answer is \(answer!).")
    }

    let response = Response(answer: answer)
    return response
}

// Run the Lambda runtime code.

try await incrementLambdaRuntime.run()
```

Define the second Lambda function, which performs an arithmetic operation on two numbers.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "calculator",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            branch: "main"),
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "calculator",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
            ],
            path: "Sources"
        )
    ]
)

import Foundation
import AWSLambdaRuntime

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
```

```
struct Request: Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The first number to act upon.
    let x: Int
    /// The second number to act upon.
    let y: Int
}

/// A dictionary mapping operation names to closures that perform that
/// operation and return the result.
let actions = [
    "plus": { (x: Int, y: Int) -> Int in
        return x + y
    },
    "minus": { (x: Int, y: Int) -> Int in
        return x - y
    },
    "times": { (x: Int, y: Int) -> Int in
        return x * y
    },
    "divided-by": { (x: Int, y: Int) -> Int in
        return x / y
    }
]

/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}


/// The Lambda function's entry point. Called by the Lambda runtime.
///
/// - Parameters:
///   - event: The `Request` describing the request made by the
///     client.
///   - context: A `LambdaContext` describing the context in
///     which the lambda function is running.
///
/// - Returns: A `Response` object that will be encoded to JSON and sent
///   to the client by the Lambda runtime.
```

```
let calculatorLambdaRuntime = LambdaRuntime {
        (_ event: Request, context: LambdaContext) -> Response in
    let action = event.action
    var answer: Int?
    var actionFunc: ((Int, Int) -> Int)?

    // Get the closure to run to perform the calculation.

    actionFunc = await actions[action]

    guard let actionFunc else {
        context.logger.error("Unrecognized operation '\(action)\'")
        return Response(answer: nil)
    }

    // Perform the calculation and return the answer.

    answer = actionFunc(event.x, event.y)

    guard let answer else {
        context.logger.error("Error computing \(event.x) \(action) \(event.y)")
    }
    context.logger.info("\(event.x) \(action) \(event.y) = \(answer)")

    return Response(answer: answer)
}

try await calculatorLambdaRuntime.run()
```

Define the main program that will invoke the two Lambda functions.

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "lambda-basics",
```

```
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/awslabs/aws-sdk-swift",
            from: "1.0.0"),
        .package(
            url: "https://github.com/apple/swift-argument-parser.git",
            branch: "main"
        )
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "lambda-basics",
            dependencies: [
                .product(name: "AWSLambda", package: "aws-sdk-swift"),
                .product(name: "AWSIAM", package: "aws-sdk-swift"),
                .product(name: "ArgumentParser", package: "swift-argument-
parser")
            ],
            path: "Sources"
        )
    ]
)

//
/// An example demonstrating a variety of important AWS Lambda functions.

import ArgumentParser
import AWSIAM
import SmithyWaitersAPI
import AWSClientRuntime
import AWSLambda
import Foundation

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
```

```
/// converts an external representation into this type.
struct IncrementRequest: Encodable, Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The number to act upon.
    let number: Int
}

struct Response: Encodable, Decodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}

struct CalculatorRequest: Encodable, Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The first number to act upon.
    let x: Int
    /// The second number to act upon.
    let y: Int
}

let exampleName = "SwiftLambdaRoleExample"
let basicsFunctionName = "lambda-basics-function"

/// The ARN of the standard IAM policy for execution of Lambda functions.
let policyARN = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

struct ExampleCommand: ParsableCommand {
    // -MARK: Command arguments
    @Option(help: "Name of the IAM Role to use for the Lambda functions")
    var role = exampleName
    @Option(help: "Zip archive containing the 'increment' lambda function")
    var incpath: String
    @Option(help: "Zip archive containing the 'calculator' lambda function")
    var calcpath: String
    @Option(help: "Name of the Amazon S3 Region to use (default: us-east-1)")
    var region = "us-east-1"

    static var configuration = CommandConfiguration(
        commandName: "lambda-basics",
        abstract: """
        This example demonstrates several common operations using AWS Lambda.
```

```
        """,
        discussion: """
        """
    )

    /// Returns the specified IAM role object.
    ///
    /// - Parameters:
    ///   - iamClient: `IAMClient` to use when looking for the role.
    ///   - roleName: The name of the role to check.
    ///
    /// - Returns: The `IAMClientTypes.Role` representing the specified role.
    func getRole(iamClient: IAMClient, roleName: String) async throws
                -> IAMClientTypes.Role {
        do {
            let roleOutput = try await iamClient.getRole(
                input: GetRoleInput(
                    roleName: roleName
                )
            )

            guard let role = roleOutput.role else {
                throw ExampleError.roleNotFound
            }
            return role
        } catch {
            throw ExampleError.roleNotFound
        }
    }

    /// Create the AWS IAM role that will be used to access AWS Lambda.
    ///
    /// - Parameters:
    ///   - iamClient: The AWS `IAMClient` to use.
    ///   - roleName: The name of the AWS IAM role to use for Lambda.
    ///
    /// - Throws: `ExampleError.roleCreateError`
    ///
    /// - Returns: The `IAMClientTypes.Role` struct that describes the new role.
    func createRoleForLambda(iamClient: IAMClient, roleName: String) async throws
-> IAMClientTypes.Role {
        let output = try await iamClient.createRole(
            input: CreateRoleInput(
                assumeRolePolicyDocument:
```

```
                """
                {
                    "Version":"2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {"Service": "lambda.amazonaws.com"},
                            "Action": "sts:AssumeRole"
                        }
                    ]
                }
                """,
                roleName: roleName
            )
        )

        guard let role = output.role else {
            throw ExampleError.roleCreateError
        }

        // Wait for the role to be ready for use.

        _ = try await iamClient.waitUntilRoleExists(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetRoleInput(roleName: roleName)
        )

        return role
    }

    /// Detect whether or not the AWS Lambda function with the specified name
    /// exists, by requesting its function information.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - name: The name of the AWS Lambda function to find.
    ///
    /// - Returns: `true` if the Lambda function exists. Otherwise `false`.
    func doesLambdaFunctionExist(lambdaClient: LambdaClient, name: String) async
 -> Bool {
```

```
        do {
            _ = try await lambdaClient.getFunction(
                input: GetFunctionInput(functionName: name)
            )
        } catch {
            return false
        }

        return true
    }

    /// Create the specified AWS Lambda function.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to create.
    ///   - roleArn: The ARN of the role to apply to the function.
    ///   - path: The path of the Zip archive containing the function.
    ///
    /// - Returns: `true` if the AWS Lambda was successfully created; `false`
    ///   if it wasn't.
    func createFunction(lambdaClient: LambdaClient, functionName: String,
                                roleArn: String?, path: String) async throws ->
Bool {
        do {
            // Read the Zip archive containing the AWS Lambda function.

            let zipUrl = URL(fileURLWithPath: path)
            let zipData = try Data(contentsOf: zipUrl)

            // Create the AWS Lambda function that runs the specified code,
            // using the name given on the command line. The Lambda function
            // will run using the Amazon Linux 2 runtime.

            _ = try await lambdaClient.createFunction(
                input: CreateFunctionInput(
                    code: LambdaClientTypes.FunctionCode(zipFile: zipData),
                    functionName: functionName,
                    handler: "handle",
                    role: roleArn,
                    runtime: .providedal2
                )
            )
        } catch {
```

```
            print("*** Error creating Lambda function:")
            dump(error)
            return false
        }

        // Wait for a while to be sure the function is done being created.

        let output = try await lambdaClient.waitUntilFunctionActiveV2(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetFunctionInput(functionName: functionName)
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }

    /// Update the AWS Lambda function with new code to run when the function
    /// is invoked.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to update.
    ///   - path: The pathname of the Zip file containing the packaged Lambda
    ///     function.
    /// - Throws: `ExampleError.zipFileReadError`
    /// - Returns: `true` if the function's code is updated successfully.
    ///   Otherwise, returns `false`.
    func updateFunctionCode(lambdaClient: LambdaClient, functionName: String,
                            path: String) async throws -> Bool {
        let zipUrl = URL(fileURLWithPath: path)
        let zipData: Data

        // Read the function's Zip file.

        do {
            zipData = try Data(contentsOf: zipUrl)
```

```
        } catch {
            throw ExampleError.zipFileReadError
        }

        // Update the function's code and wait for the updated version to be
        // ready for use.

        do {
            _ = try await lambdaClient.updateFunctionCode(
                input: UpdateFunctionCodeInput(
                    functionName: functionName,
                    zipFile: zipData
                )
            )
        } catch {
            return false
        }

        let output = try await lambdaClient.waitUntilFunctionUpdatedV2(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetFunctionInput(
                functionName: functionName
            )
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }

    /// Tell the server-side component to log debug output by setting its
    /// environment's `LOG_LEVEL` to `DEBUG`.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to enable debug
    ///     logging for.
```

```
    ///
    /// - Throws: `ExampleError.environmentResponseMissingError`,
    ///   `ExampleError.updateFunctionConfigurationError`,
    ///   `ExampleError.environmentVariablesMissingError`,
    ///   `ExampleError.logLevelIncorrectError`,
    ///   `ExampleError.updateFunctionConfigurationError`
    func enableDebugLogging(lambdaClient: LambdaClient, functionName: String)
async throws {
        let envVariables = [
            "LOG_LEVEL": "DEBUG"
        ]
        let environment = LambdaClientTypes.Environment(variables: envVariables)

        do {
            let output = try await lambdaClient.updateFunctionConfiguration(
                input: UpdateFunctionConfigurationInput(
                    environment: environment,
                    functionName: functionName
                )
            )

            guard let response = output.environment else {
                throw ExampleError.environmentResponseMissingError
            }

            if response.error != nil {
                throw ExampleError.updateFunctionConfigurationError
            }

            guard let retVariables = response.variables else {
                throw ExampleError.environmentVariablesMissingError
            }

            for envVar in retVariables {
                if envVar.key == "LOG_LEVEL" && envVar.value != "DEBUG" {
                    print("*** Log level is not set to DEBUG!")
                    throw ExampleError.logLevelIncorrectError
                }
            }
        } catch {
            throw ExampleError.updateFunctionConfigurationError
        }
    }
```

```
    /// Returns an array containing the names of all AWS Lambda functions
    /// available to the user.
    ///
    /// - Parameter lambdaClient: The `IAMClient` to use.
    ///
    /// - Throws: `ExampleError.listFunctionsError`
    ///
    /// - Returns: An array of lambda function name strings.
    func getFunctionNames(lambdaClient: LambdaClient) async throws -> [String] {
        let pages = lambdaClient.listFunctionsPaginated(
            input: ListFunctionsInput()
        )

        var functionNames: [String] = []

        for try await page in pages {
            guard let functions = page.functions else {
                throw ExampleError.listFunctionsError
            }

            for function in functions {
                functionNames.append(function.functionName ?? "<unknown>")
            }
        }

        return functionNames
    }

    /// Invoke the Lambda function to increment a value.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - number: The number to increment.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: An integer number containing the incremented value.
    func invokeIncrement(lambdaClient: LambdaClient, number: Int) async throws ->
Int {
        do {
            let incRequest = IncrementRequest(action: "increment", number:
number)
            let incData = try! JSONEncoder().encode(incRequest)
```

```
            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: incData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }
    }

    /// Invoke the calculator Lambda function.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - action: Which arithmetic operation to perform: "plus", "minus",
    ///     "times", or "divided-by".
    ///   - x: The first number to use in the computation.
    ///   - y: The second number to use in the computation.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: The computed answer as an `Int`.
    func invokeCalculator(lambdaClient: LambdaClient, action: String, x: Int, y:
Int) async throws -> Int {
        do {
            let calcRequest = CalculatorRequest(action: action, x: x, y: y)
            let calcData = try! JSONEncoder().encode(calcRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
```

```
                    functionName: "lambda-basics-function",
                    payload: calcData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }

    }

    /// Perform the example's tasks.
    func basics() async throws {
        let iamClient = try await IAMClient(
            config: IAMClient.IAMClientConfiguration(region: region)
        )

        let lambdaClient = try await LambdaClient(
            config: LambdaClient.LambdaClientConfiguration(region: region)
        )

        /// The IAM role to use for the example.
        var iamRole: IAMClientTypes.Role

        // Look for the specified role. If it already exists, use it. If not,
        // create it and attach the desired policy to it.

        do {
            iamRole = try await getRole(iamClient: iamClient, roleName: role)
        } catch ExampleError.roleNotFound {
            // The role wasn't found, so create it and attach the needed
            // policy.

            iamRole = try await createRoleForLambda(iamClient: iamClient,
roleName: role)
```

```
            do {
                _ = try await iamClient.attachRolePolicy(
                    input: AttachRolePolicyInput(policyArn: policyARN, roleName:
role)
                )
            } catch {
                throw ExampleError.policyError
            }
        }

        // Give the policy time to attach to the role.

        sleep(5)

        // Look to see if the function already exists. If it does, throw an
        // error.

        if await doesLambdaFunctionExist(lambdaClient: lambdaClient, name:
basicsFunctionName) {
            throw ExampleError.functionAlreadyExists
        }

        // Create, then invoke, the "increment" version of the calculator
        // function.

        print("Creating the increment Lambda function...")
        if try await createFunction(lambdaClient: lambdaClient, functionName:
basicsFunctionName,
                                    roleArn: iamRole.arn, path: incpath) {
            print("Running increment function calls...")
            for number in 0...4 {
                do {
                    let answer = try await invokeIncrement(lambdaClient:
lambdaClient, number: number)
                    print("Increment \(number) = \(answer)")
                } catch {
                    print("Error incrementing \(number): ",
error.localizedDescription)
                }
            }
        } else {
            print("*** Failed to create the increment function.")
        }
```

```
        // Enable debug logging.

        print("\nEnabling debug logging...")
        try await enableDebugLogging(lambdaClient: lambdaClient, functionName:
basicsFunctionName)

        // Change it to a basic arithmetic calculator. Then invoke it a few
        // times.

        print("\nReplacing the Lambda function with a calculator...")

        if try await updateFunctionCode(lambdaClient: lambdaClient, functionName:
basicsFunctionName,
                                        path: calcpath) {
            print("Running calculator function calls...")
            for x in [6, 10] {
                for y in [2, 4] {
                    for action in ["plus", "minus", "times", "divided-by"] {
                        do {
                            let answer = try await invokeCalculator(lambdaClient:
lambdaClient, action: action, x: x, y: y)
                            print("\(x) \(action) \(y) = \(answer)")
                        } catch {
                            print("Error calculating \(x) \(action) \(y): ",
error.localizedDescription)
                        }
                    }
                }
            }
        }

        // List all lambda functions.

        let functionNames = try await getFunctionNames(lambdaClient:
lambdaClient)

        if functionNames.count > 0 {
            print("\nAWS Lambda functions available on your account:")
            for name in functionNames {
                print("  \(name)")
            }
        }

        // Delete the lambda function.
```

```
        print("Deleting lambda function...")

        do {
            _ = try await lambdaClient.deleteFunction(
                input: DeleteFunctionInput(
                    functionName: "lambda-basics-function"
                )
            )
        } catch {
            print("Error: Unable to delete the function.")
        }

        // Detach the role from the policy, then delete the role.

        print("Deleting the AWS IAM role...")

        do {
            _ = try await iamClient.detachRolePolicy(
                input: DetachRolePolicyInput(
                    policyArn: policyARN,
                    roleName: role
                )
            )
            _ = try await iamClient.deleteRole(
                input: DeleteRoleInput(
                    roleName: role
                )
            )
        } catch {
            throw ExampleError.deleteRoleError
        }
    }
}

// -MARK: - Entry point

/// The program's asynchronous entry point.
@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
```

```
            let command = try ExampleCommand.parse(args)
            try await command.basics()
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}


/// Errors thrown by the example's functions.
enum ExampleError: Error {
    /// An AWS Lambda function with the specified name already exists.
    case functionAlreadyExists
    /// The specified role doesn't exist.
    case roleNotFound
    /// Unable to create the role.
    case roleCreateError
    /// Unable to delete the role.
    case deleteRoleError
    /// Unable to attach a policy to the role.
    case policyError
    /// Unable to get the executable directory.
    case executableNotFound
    /// An error occurred creating a lambda function.
    case createLambdaError
    /// An error occurred invoking the lambda function.
    case invokeError
    /// No answer received from the invocation.
    case noAnswerReceived
    /// Unable to list the AWS Lambda functions.
    case listFunctionsError
    /// Unable to update the AWS Lambda function.
    case updateFunctionError
    /// Unable to update the function configuration.
    case updateFunctionConfigurationError
    /// The environment response is missing after an
    /// UpdateEnvironmentConfiguration attempt.
    case environmentResponseMissingError
    /// The environment variables are missing from the EnvironmentResponse and
    /// no errors occurred.
    case environmentVariablesMissingError
    /// The log level is incorrect after attempting to set it.
    case logLevelIncorrectError
    /// Unable to load the AWS Lambda function's Zip file.
```

```
    case zipFileReadError

    var errorDescription: String? {
        switch self {
        case .functionAlreadyExists:
            return "An AWS Lambda function with that name already exists."
        case .roleNotFound:
            return "The specified role doesn't exist."
        case .deleteRoleError:
            return "Unable to delete the AWS IAM role."
        case .roleCreateError:
            return "Unable to create the specified role."
        case .policyError:
            return "An error occurred attaching the policy to the role."
        case .executableNotFound:
            return "Unable to find the executable program directory."
        case .createLambdaError:
            return "An error occurred creating a lambda function."
        case .invokeError:
            return "An error occurred invoking a lambda function."
        case .noAnswerReceived:
            return "No answer received from the lambda function."
        case .listFunctionsError:
            return "Unable to list the AWS Lambda functions."
        case .updateFunctionError:
            return "Unable to update the AWS lambda function."
        case .updateFunctionConfigurationError:
            return "Unable to update the AWS lambda function configuration."
        case .environmentResponseMissingError:
            return "The environment is missing from the response after updating
 the function configuration."
        case .environmentVariablesMissingError:
            return "While no error occurred, no environment variables were
 returned following function configuration."
        case .logLevelIncorrectError:
            return "The log level is incorrect after attempting to set it to
 DEBUG."
        case .zipFileReadError:
            return "Unable to read the AWS Lambda function."
        }
    }
}
```

- For API details, see the following topics in *AWS SDK for Swift API reference*.
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Actions for Lambda using AWS SDKs

The following code examples demonstrate how to perform individual Lambda actions with AWS SDKs. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

These excerpts call the Lambda API and are code excerpts from larger programs that must be run in context. You can see actions in context in Scenarios for Lambda using AWS SDKs.

The following examples include only the most commonly used actions. For a complete list, see the AWS Lambda API Reference.

Examples

- Use CreateAlias with a CLI
- Use CreateFunction with an AWS SDK or CLI
- Use DeleteAlias with a CLI
- Use DeleteFunction with an AWS SDK or CLI
- Use DeleteFunctionConcurrency with a CLI
- Use DeleteProvisionedConcurrencyConfig with a CLI
- Use GetAccountSettings with a CLI
- Use GetAlias with a CLI
- Use GetFunction with an AWS SDK or CLI

- Use GetFunctionConcurrency with a CLI
- Use GetFunctionConfiguration with a CLI
- Use GetPolicy with a CLI
- Use GetProvisionedConcurrencyConfig with a CLI
- Use Invoke with an AWS SDK or CLI
- Use ListFunctions with an AWS SDK or CLI
- Use ListProvisionedConcurrencyConfigs with a CLI
- Use ListTags with a CLI
- Use ListVersionsByFunction with a CLI
- Use PublishVersion with a CLI
- Use PutFunctionConcurrency with a CLI
- Use PutProvisionedConcurrencyConfig with a CLI
- Use RemovePermission with a CLI
- Use TagResource with a CLI
- Use UntagResource with a CLI
- Use UpdateAlias with a CLI
- Use UpdateFunctionCode with an AWS SDK or CLI
- Use UpdateFunctionConfiguration with an AWS SDK or CLI

Use `CreateAlias` with a CLI

The following code examples show how to use `CreateAlias`.

CLI

AWS CLI

To create an alias for a Lambda function

The following `create-alias` example creates an alias named `LIVE` that points to version 1 of the `my-function` Lambda function.

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
```

```
    --function-version 1 \
    --name LIVE
```

Output:

```
{
    "FunctionVersion": "1",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "Description": "alias for live version of function"
}
```

For more information, see Configuring AWS Lambda Function Aliases in the *AWS Lambda Developer Guide*.

- For API details, see CreateAlias in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example creates a New Lambda Alias for specified version and routing configuration to specify the percentage of invocation requests that it receives.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- For API details, see CreateAlias in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example creates a New Lambda Alias for specified version and routing configuration to specify the percentage of invocation requests that it receives.

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- For API details, see CreateAlias in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateFunction` with an AWS SDK or CLI

The following code examples show how to use `CreateFunction`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/// <summary>
/// Creates a new Lambda function.
/// </summary>
/// <param name="functionName">The name of the function.</param>
/// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
/// bucket where the zip file containing the code is located.</param>
/// <param name="s3Key">The Amazon S3 key of the zip file.</param>
/// <param name="role">The Amazon Resource Name (ARN) of a role with the
/// appropriate Lambda permissions.</param>
/// <param name="handler">The name of the handler function.</param>
/// <returns>The Amazon Resource Name (ARN) of the newly created
/// Lambda function.</returns>
public async Task<string> CreateLambdaFunctionAsync(
    string functionName,
    string s3Bucket,
    string s3Key,
    string role,
    string handler)
```

```
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
_lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }
```

- For API details, see CreateFunction in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
```

```
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
```

```
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
seconds
                      << " seconds elapsed." << std::endl;
            break;
        }

        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
```

- For API details, see CreateFunction in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To create a Lambda function

The following `create-function` example creates a Lambda function named `my-function`.

```
aws lambda create-function \
```

```
    --function-name my-function \
    --runtime nodejs22.x \
    --zip-file fileb://my-function.zip \
    --handler my-function.handler \
    --role arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
tges6bf4
```

Contents of `my-function.zip`:

```
This file is a deployment package that contains your function code and any
 dependencies.
```

Output:

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "PFn4S+er27qk+UuZSTKEQfNKG/XNn7QJs90mJgq6oH8=",
    "FunctionName": "my-function",
    "CodeSize": 308,
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2025-10-14T22:26:11.234+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs22.x",
    "Description": ""
}
```

For more information, see Configure Lambda function memory in the *AWS Lambda Developer Guide*.

- For API details, see CreateFunction in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
```

```
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- For API details, see CreateFunction in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
                                               String functionName,
                                               String key,
                                               String bucketName,
                                               String role,
                                               String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            FunctionCode code = FunctionCode.builder()
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
```

```
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
 awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
 waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
```

- For API details, see CreateFunction in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
```

```
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- For API details, see CreateFunction in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun createNewFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String? {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
```

```
            role = myRole
            runtime = Runtime.Java17
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn
    }
}
```

- For API details, see CreateFunction in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function createFunction($functionName, $role, $bucketName, $handler)
    {
        //This assumes the Lambda function is in an S3 bucket.
        return $this->customWaiter(function () use ($functionName, $role,
$bucketName, $handler) {
            return $this->lambdaClient->createFunction([
                'Code' => [
                    'S3Bucket' => $bucketName,
                    'S3Key' => $functionName,
                ],
                'FunctionName' => $functionName,
                'Role' => $role['Arn'],
                'Runtime' => 'python3.9',
                'Handler' => "$handler.lambda_handler",
            ]);
        });
```

```
    }
```

- For API details, see CreateFunction in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example creates a new C# (dotnetcore1.0 runtime) function named MyFunction in AWS Lambda, providing the compiled binaries for the function from a zip file on the local file system (relative or absolute paths may be used). C# Lambda functions specify the handler for the function using the designation AssemblyName::Namespace.ClassName::MethodName. You should replace the assembly name (without .dll suffix), namespace, class name and method name parts of the handler spec appropriately. The new function will have environment variables 'envvar1' and 'envvar2' set up from the provided values.

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

Output:

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
```

```
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

Example 2: This example is similar to the previous one except the function binaries are first uploaded to an Amazon S3 bucket (which must be in the same region as the intended Lambda function) and the resulting S3 object is then referenced when creating the function.

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- For API details, see CreateFunction in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example creates a new C# (dotnetcore1.0 runtime) function named MyFunction in AWS Lambda, providing the compiled binaries for the function from a zip file on the local file system (relative or absolute paths may be used). C# Lambda functions specify the handler for the function using the designation AssemblyName::Namespace.ClassName::MethodName. You should replace the assembly name (without .dll suffix), namespace, class name and method name parts of the handler spec appropriately. The new function will have environment variables 'envvar1' and 'envvar2' set up from the provided values.

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

Output:

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

Example 2: This example is similar to the previous one except the function binaries are first uploaded to an Amazon S3 bucket (which must be in the same region as the intended Lambda function) and the resulting S3 object is then referenced when creating the function.

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- For API details, see CreateFunction in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
 This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
 function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
```

```
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn
```

- For API details, see CreateFunction in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
```

```
 # @param handler_name: The fully qualified name of the handler function.
 # @param role_arn: The IAM role to use for the function.
 # @param deployment_package: The deployment package that contains the function
code in .zip format.
 # @return: The Amazon Resource Name (ARN) of the newly created function.
 def create_function(function_name, handler_name, role_arn, deployment_package)
   response = @lambda_client.create_function({
                                               role: role_arn.to_s,
                                               function_name: function_name,
                                               handler: handler_name,
                                               runtime: 'ruby2.7',
                                               code: {
                                                 zip_file: deployment_package
                                               },
                                               environment: {
                                                 variables: {
                                                   'LOG_LEVEL' => 'info'
                                                 }
                                               }
                                             })
   @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
   response
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error creating #{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

- For API details, see CreateFunction in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
```

```
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- For API details, see CreateFunction in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   TRY.
       lo_lmd->createfunction(
           iv_functionname = iv_function_name
           iv_runtime = `python3.9`
           iv_role = iv_role_arn
           iv_handler = iv_handler
           io_code = io_zip_file
           iv_description = 'AWS Lambda code example' ).
       MESSAGE 'Lambda function created.' TYPE 'I'.
     CATCH /aws1/cx_lmdcodesigningcfgno00.
       MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodestorageexcdex.
       MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodeverification00.
       MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidcodesigex.
       MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- For API details, see CreateFunction in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        do {
            // Read the Zip archive containing the AWS Lambda function.

            let zipUrl = URL(fileURLWithPath: path)
            let zipData = try Data(contentsOf: zipUrl)

            // Create the AWS Lambda function that runs the specified code,
            // using the name given on the command line. The Lambda function
            // will run using the Amazon Linux 2 runtime.

            _ = try await lambdaClient.createFunction(
                input: CreateFunctionInput(
                    code: LambdaClientTypes.FunctionCode(zipFile: zipData),
                    functionName: functionName,
                    handler: "handle",
                    role: roleArn,
                    runtime: .providedal2
                )
            )
        } catch {
            print("*** Error creating Lambda function:")
            dump(error)
            return false
        }
```

- For API details, see CreateFunction in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteAlias` with a CLI

The following code examples show how to use `DeleteAlias`.

CLI

AWS CLI

To delete an alias of a Lambda function

The following `delete-alias` example deletes the alias named `LIVE` from the `my-function` Lambda function.

```
aws lambda delete-alias \
    --function-name my-function \
    --name LIVE
```

This command produces no output.

For more information, see Configuring AWS Lambda Function Aliases in the *AWS Lambda Developer Guide*.

- For API details, see DeleteAlias in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example deletes the Lambda function Alias mentioned in the command.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- For API details, see DeleteAlias in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example deletes the Lambda function Alias mentioned in the command.

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- For API details, see DeleteAlias in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteFunction` with an AWS SDK or CLI

The following code examples show how to use `DeleteFunction`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
```

```
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }
```

- For API details, see DeleteFunction in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    Aws::Lambda::Model::DeleteFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);

    Aws::Lambda::Model::DeleteFunctionOutcome outcome = client.DeleteFunction(
            request);

    if (outcome.IsSuccess()) {
```

```
        std::cout << "The lambda function was successfully deleted." <<
std::endl;
    }
    else {
        std::cerr << "Error with Lambda::DeleteFunction. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
```

- For API details, see DeleteFunction in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To delete a Lambda function by function name

The following `delete-function` example deletes the Lambda function named `my-function` by specifying the function's name.

```
aws lambda delete-function \
    --function-name my-function
```

This command produces no output.

Example 2: To delete a Lambda function by function ARN

The following `delete-function` example deletes the Lambda function named `my-function` by specifying the function's ARN.

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

This command produces no output.

Example 3: To delete a Lambda function by partial function ARN

The following `delete-function` example deletes the Lambda function named `my-function` by specifying the function's partial ARN.

```
aws lambda delete-function \
    --function-name 123456789012:function:my-function
```

This command produces no output.

For more information, see AWS Lambda Function Configuration in the *AWS Lambda Developer Guide*.

- For API details, see DeleteFunction in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}
```

```
// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

- For API details, see DeleteFunction in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /**
    * Deletes an AWS Lambda function.
    *
    * @param awsLambda     an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
    * @param functionName  the name of the Lambda function to be deleted
    *
    * @throws LambdaException if an error occurs while deleting the Lambda
function
    */
   public static void deleteLambdaFunction(LambdaClient awsLambda, String
functionName) {
       try {
           DeleteFunctionRequest request = DeleteFunctionRequest.builder()
               .functionName(functionName)
               .build();
```

```
            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- For API details, see DeleteFunction in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- For API details, see DeleteFunction in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun delLambdaFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- For API details, see DeleteFunction in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function deleteFunction($functionName)
    {
        return $this->lambdaClient->deleteFunction([
            'FunctionName' => $functionName,
        ]);
```

```
    }
```

- For API details, see DeleteFunction in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example deletes a specific version of a Lambda function

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- For API details, see DeleteFunction in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example deletes a specific version of a Lambda function

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- For API details, see DeleteFunction in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def delete_function(self, function_name):
        """
```

```
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

- For API details, see DeleteFunction in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
```

```
    print 'Done!'.green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- For API details, see DeleteFunction in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
    pub async fn delete_function(
        &self,
        location: Option<String>,
    ) -> (
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ) {
        info!("Deleting lambda function {}", self.lambda_name);
        let delete_function = self
            .lambda_client
            .delete_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
```

```
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
```

- For API details, see DeleteFunction in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        lo_lmd->deletefunction( iv_functionname = iv_function_name ).
        MESSAGE 'Lambda function deleted.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- For API details, see DeleteFunction in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        do {
            _ = try await lambdaClient.deleteFunction(
                input: DeleteFunctionInput(
                    functionName: "lambda-basics-function"
                )
            )
        } catch {
            print("Error: Unable to delete the function.")
        }
```

- For API details, see DeleteFunction in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteFunctionConcurrency` with a CLI

The following code examples show how to use `DeleteFunctionConcurrency`.

CLI

AWS CLI

To remove the reserved concurrent execution limit from a function

The following `delete-function-concurrency` example deletes the reserved concurrent execution limit from the `my-function` function.

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

This command produces no output.

For more information, see Reserving Concurrency for a Lambda Function in the *AWS Lambda Developer Guide*.

- For API details, see DeleteFunctionConcurrency in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This examples removes the Function Concurrency of the Lambda Function.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- For API details, see DeleteFunctionConcurrency in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This examples removes the Function Concurrency of the Lambda Function.

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- For API details, see DeleteFunctionConcurrency in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteProvisionedConcurrencyConfig` with a CLI

The following code examples show how to use `DeleteProvisionedConcurrencyConfig`.

CLI

AWS CLI

To delete a provisioned concurrency configuration

The following `delete-provisioned-concurrency-config` example deletes the provisioned concurrency configuration for the `GREEN` alias of the specified function.

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

- For API details, see DeleteProvisionedConcurrencyConfig in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example removes the Provisioned Concurrency Configuration for a specific Alias.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- For API details, see DeleteProvisionedConcurrencyConfig in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example removes the Provisioned Concurrency Configuration for a specific Alias.

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- For API details, see DeleteProvisionedConcurrencyConfig in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetAccountSettings` with a CLI

The following code examples show how to use `GetAccountSettings`.

CLI

AWS CLI

To retrieve details about your account in an AWS Region

The following `get-account-settings` example displays the Lambda limits and usage information for your account.

```
aws lambda get-account-settings
```

Output:

```
{
    "AccountLimit": {
        "CodeSizeUnzipped": 262144000,
        "UnreservedConcurrentExecutions": 1000,
        "ConcurrentExecutions": 1000,
        "CodeSizeZipped": 52428800,
        "TotalCodeSize": 80530636800
    },
```

```
    "AccountUsage": {
       "FunctionCount": 4,
       "TotalCodeSize": 9426
    }
}
```

For more information, see AWS Lambda Limits in the *AWS Lambda Developer Guide*.

- For API details, see GetAccountSettings in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This sample displays to compare the Account Limit and Account Usage

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

Output:

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- For API details, see GetAccountSettings in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This sample displays to compare the Account Limit and Account Usage

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

Output:

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
```

```
        80530636800          15078795
```

- For API details, see GetAccountSettings in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetAlias` with a CLI

The following code examples show how to use `GetAlias`.

CLI

AWS CLI

To retrieve details about a function alias

The following `get-alias` example displays details for the alias named `LIVE` on the `my-function` Lambda function.

```
aws lambda get-alias \
    --function-name my-function \
    --name LIVE
```

Output:

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

For more information, see Configuring AWS Lambda Function Aliases in the *AWS Lambda Developer Guide*.

- For API details, see GetAlias in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example retrieves the Routing Config weights for a specific Lambda Function Alias.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

Output:

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- For API details, see GetAlias in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example retrieves the Routing Config weights for a specific Lambda Function Alias.

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

Output:

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- For API details, see GetAlias in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetFunction` with an AWS SDK or CLI

The following code examples show how to use `GetFunction`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

- For API details, see GetFunction in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

       Aws::Lambda::Model::GetFunctionRequest request;
       request.SetFunctionName(functionName);

       Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

       if (outcome.IsSuccess()) {
           std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                     << std::endl;
       }
       else {
           std::cerr << "Error with Lambda::GetFunction. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
       }
```

- For API details, see GetFunction in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To retrieve information about a function

The following `get-function` example displays information about the `my-function` function.

```
aws lambda get-function \
    --function-name  my-function
```

Output:

```
{
    "Concurrency": {
        "ReservedConcurrentExecutions": 100
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-west-2-tasks.s3.us-
west-2.amazonaws.com/snapshots/123456789012/my-function..."
    },
    "Configuration": {
        "TracingConfig": {
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
        "FunctionName": "my-function",
        "VpcConfig": {
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
        "RevisionId": "28f0fb31-5c5c-43d3-8955-03e76c5c1075",
        "CodeSize": 304,
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
        "Handler": "index.handler",
        "Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-
role-uy3l9qyq",
```

```
        "Timeout": 3,
        "LastModified": "2025-09-24T18:20:35.054+0000",
        "Runtime": "nodejs22.x",
        "Description": ""
    }
}
```

For more information, see Configure Lambda function memory in the *AWS Lambda Developer Guide*.

- For API details, see GetFunction in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}
```

```
// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}
```

- For API details, see GetFunction in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
 is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
 information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
```

```
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
 awsLambda.getFunction(functionRequest);
            System.out.println("The runtime of this Lambda function is " +
 response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- For API details, see GetFunction in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- For API details, see GetFunction in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- For API details, see GetFunction in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def get_function(self, function_name):
        """
        Gets data about a Lambda function.
```

```
        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
 self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response
```

- For API details, see GetFunction in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
```

```
    @logger.level = Logger::WARN
  end

  # Gets data about a Lambda function.
  #
  # @param function_name: The name of the function.
  # @return response: The function data, or nil if no such function exists.
  def get_function(function_name)
    @lambda_client.get_function(
      {
        function_name: function_name
      }
    )
  rescue Aws::Lambda::Errors::ResourceNotFoundException => e
    @logger.debug("Could not find function: #{function_name}:\n #{e.message}")
    nil
  end
```

- For API details, see GetFunction in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- For API details, see GetFunction in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        oo_result = lo_lmd->getfunction( iv_functionname = iv_function_name ).
    " oo_result is returned for testing purposes. "
        MESSAGE 'Lambda function information retrieved.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
    TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- For API details, see GetFunction in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Detect whether or not the AWS Lambda function with the specified name
    /// exists, by requesting its function information.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - name: The name of the AWS Lambda function to find.
    ///
    /// - Returns: `true` if the Lambda function exists. Otherwise `false`.
    func doesLambdaFunctionExist(lambdaClient: LambdaClient, name: String) async
 -> Bool {
        do {
            _ = try await lambdaClient.getFunction(
                input: GetFunctionInput(functionName: name)
            )
        } catch {
            return false
        }

        return true
    }
```

- For API details, see GetFunction in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetFunctionConcurrency` with a CLI

The following code examples show how to use `GetFunctionConcurrency`.

CLI

AWS CLI

To view the reserved concurrency setting for a function

The following `get-function-concurrency` example retrieves the reserved concurrency setting for the specified function.

```
aws lambda get-function-concurrency \
    --function-name my-function
```

Output:

```
{
    "ReservedConcurrentExecutions": 250
}
```

- For API details, see GetFunctionConcurrency in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This examples gets the Reserved concurrency for the Lambda Function

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

Output:

```
ReservedConcurrentExecutions
----------------------------
100
```

- For API details, see GetFunctionConcurrency in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This examples gets the Reserved concurrency for the Lambda Function

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

Output:

```
ReservedConcurrentExecutions
----------------------------
```

```
100
```

- For API details, see GetFunctionConcurrency in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetFunctionConfiguration` with a CLI

The following code examples show how to use `GetFunctionConfiguration`.

CLI

AWS CLI

To retrieve the version-specific settings of a Lambda function

The following `get-function-configuration` example displays the settings for version 2 of the `my-function` function.

```
aws lambda get-function-configuration \
    --function-name  my-function:2
```

Output:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
```

```
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
    "Handler": "index.handler"
}
```

For more information, see AWS Lambda Function Configuration in the *AWS Lambda Developer Guide*.

- For API details, see GetFunctionConfiguration in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example returns the version specific configuration of a Lambda Function.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

Output:

```
CodeSha256                : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                  : 1426
DeadLetterConfig          : Amazon.Lambda.Model.DeadLetterConfig
Description               : Verson 3 to test Aliases
Environment               : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn               : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                            :PowershellAlias
FunctionName              : MylambdaFunction123
Handler                   : lambda_function.launch_instance
KMSKeyArn                 :
LastModified              : 2019-12-25T09:52:59.872+0000
LastUpdateStatus          : Successful
LastUpdateStatusReason    :
```

```
LastUpdateStatusReasonCode :
Layers                     : {}
MasterArn                  :
MemorySize                 : 128
RevisionId                 : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                       : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                    : python3.8
State                      : Active
StateReason                :
StateReasonCode            :
Timeout                    : 600
TracingConfig              : Amazon.Lambda.Model.TracingConfigResponse
Version                    : 4
VpcConfig                  : Amazon.Lambda.Model.VpcConfigDetail
```

- For API details, see GetFunctionConfiguration in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example returns the version specific configuration of a Lambda Function.

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

Output:

```
CodeSha256                 : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                   : 1426
DeadLetterConfig           : Amazon.Lambda.Model.DeadLetterConfig
Description                : Verson 3 to test Aliases
Environment                : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn                : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                             :PowershellAlias
FunctionName               : MylambdaFunction123
Handler                    : lambda_function.launch_instance
KMSKeyArn                  :
LastModified               : 2019-12-25T09:52:59.872+0000
LastUpdateStatus           : Successful
LastUpdateStatusReason     :
LastUpdateStatusReasonCode :
Layers                     : {}
```

```
MasterArn                  :
MemorySize                 : 128
RevisionId                 : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                       : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                    : python3.8
State                      : Active
StateReason                :
StateReasonCode            :
Timeout                    : 600
TracingConfig              : Amazon.Lambda.Model.TracingConfigResponse
Version                    : 4
VpcConfig                  : Amazon.Lambda.Model.VpcConfigDetail
```

- For API details, see GetFunctionConfiguration in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetPolicy` with a CLI

The following code examples show how to use `GetPolicy`.

CLI

AWS CLI

To retrieve the resource-based IAM policy for a function, version, or alias

The following `get-policy` example displays policy information about the `my-function` Lambda function.

```
aws lambda get-policy \
    --function-name my-function
```

Output:

```
{
    "Policy": {
        "Version":"2012-10-17",
        "Id":"default",
```

```
        "Statement":
        [
            {
                "Sid":"iot-events",
                "Effect":"Allow",
                "Principal": {"Service":"iotevents.amazonaws.com"},
                "Action":"lambda:InvokeFunction",
                "Resource":"arn:aws:lambda:us-west-2:123456789012:function:my-
function"
            }
        ]
    },
    "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668"
}
```

For more information, see Using Resource-based Policies for AWS Lambda in the *AWS Lambda Developer Guide*.

- For API details, see GetPolicy in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This sample displays the Function policy of the Lambda function

```
Get-LMPolicy -FunctionName test -Select Policy
```

Output:

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- For API details, see GetPolicy in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This sample displays the Function policy of the Lambda function

```
Get-LMPolicy -FunctionName test -Select Policy
```

Output:

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- For API details, see GetPolicy in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `GetProvisionedConcurrencyConfig` with a CLI

The following code examples show how to use `GetProvisionedConcurrencyConfig`.

CLI

AWS CLI

To view a provisioned concurrency configuration

The following `get-provisioned-concurrency-config` example displays details for the provisioned concurrency configuration for the BLUE alias of the specified function.

```
aws lambda get-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE
```

Output:

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
    "LastModified": "2019-12-31T20:28:49+0000"
}
```

- For API details, see GetProvisionedConcurrencyConfig in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example gets the provisioned Concurrency Configuration for the specified Alias of the Lambda Function.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

Output:

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- For API details, see GetProvisionedConcurrencyConfig in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example gets the provisioned Concurrency Configuration for the specified Alias of the Lambda Function.

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

Output:

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- For API details, see GetProvisionedConcurrencyConfig in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `Invoke` with an AWS SDK or CLI

The following code examples show how to use `Invoke`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };
```

```
        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }
```

- For API details, see Invoke in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
            invokeResult = std::move(outcome.GetResult());
```

```
            result = true;
            break;
        }

        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
```

- For API details, see Invoke in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

Example 1: To invoke a Lambda function synchronously

The following `invoke` example invokes the `my-function` function synchronously. The `cli-binary-format` option is required if you're using AWS CLI version 2. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide*.

```
aws lambda invoke \
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

Output:

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

For more information, see Invoke a Lambda function synchronously in the *AWS Lambda Developer Guide*.

Example 2: To invoke a Lambda function asynchronously

The following `invoke` example invokes the `my-function` function asynchronously. The `cli-binary-format` option is required if you're using AWS CLI version 2. For more information, see AWS CLI supported global command line options in the *AWS Command Line Interface User Guide*.

```
aws lambda invoke \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

Output:

```
{
    "StatusCode": 202
}
```

For more information, see Invoking a Lambda function asynchronously in the *AWS Lambda Developer Guide*.

- For API details, see Invoke in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
```

```
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}
```

- For API details, see Invoke in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /**
    * Invokes a specific AWS Lambda function.
    *
    * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
    * @param functionName the name of the AWS Lambda function to be invoked
    */
   public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
       InvokeResponse res;
       try {
           // Need a SdkBytes instance for the payload.
           JSONObject jsonObj = new JSONObject();
           jsonObj.put("inputValue", "2000");
           String json = jsonObj.toString();
           SdkBytes payload = SdkBytes.fromUtf8String(json);

           InvokeRequest request = InvokeRequest.builder()
               .functionName(functionName)
               .payload(payload)
               .build();

           res = awsLambda.invoke(request);
           String value = res.payload().asUtf8String();
           System.out.println(value);

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
```

```
    }
```

- For API details, see Invoke in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

- For API details, see Invoke in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
suspend fun invokeFunction(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            logType = LogType.Tail
            payload = byteArray
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("${res.payload?.toString(Charsets.UTF_8)}")
        println("The log result is ${res.logResult}")
    }
}
```

- For API details, see Invoke in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- For API details, see Invoke in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
 dict
                                is serialized to JSON before it is sent to
 Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
 included in
                        the response.
        :return: The response from the function invocation.
```

```
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response
```

- For API details, see Invoke in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
```

```
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
 #{e.message}")
  end
```

- For API details, see Invoke in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
 anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
```

```
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
```

- For API details, see Invoke in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    TRY.
        DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
          `{`  &&
            `"action": "increment",`  &&
            `"number": 10` &&
          `}` ).
        oo_result = lo_lmd->invoke(                 " oo_result is returned for
testing purposes. "
                iv_functionname = iv_function_name
                iv_payload = lv_json ).
        MESSAGE 'Lambda function invoked.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvrequestcontex.
        MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidzipfileex.
        MESSAGE 'The deployment package could not be unzipped.' TYPE 'E'.
      CATCH /aws1/cx_lmdrequesttoolargeex.
        MESSAGE 'Invoke request body JSON input limit was exceeded by the request
payload.' TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
      CATCH /aws1/cx_lmdunsuppedmediatyp00.
        MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
    ENDTRY.
```

- For API details, see Invoke in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Invoke the Lambda function to increment a value.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - number: The number to increment.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: An integer number containing the incremented value.
```

```
   func invokeIncrement(lambdaClient: LambdaClient, number: Int) async throws ->
Int {
       do {
           let incRequest = IncrementRequest(action: "increment", number:
number)
           let incData = try! JSONEncoder().encode(incRequest)

           // Invoke the lambda function.

           let invokeOutput = try await lambdaClient.invoke(
               input: InvokeInput(
                   functionName: "lambda-basics-function",
                   payload: incData
               )
           )

           let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

           guard let answer = response.answer else {
               throw ExampleError.noAnswerReceived
           }
           return answer

       } catch {
           throw ExampleError.invokeError
       }
   }
```

- For API details, see Invoke in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListFunctions` with an AWS SDK or CLI

The following code examples show how to use `ListFunctions`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
/// <summary>
/// Get a list of Lambda functions.
/// </summary>
/// <returns>A list of FunctionConfiguration objects.</returns>
public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
{
    var functionList = new List<FunctionConfiguration>();

    var functionPaginator =
        _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
    await foreach (var function in functionPaginator.Functions)
    {
        functionList.Add(function);
    }

    return functionList;
}
```

- For API details, see ListFunctions in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

   std::vector<Aws::String> functions;
   Aws::String marker;

   do {
       Aws::Lambda::Model::ListFunctionsRequest request;
       if (!marker.empty()) {
           request.SetMarker(marker);
       }

       Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
               request);

       if (outcome.IsSuccess()) {
           const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
           std::cout << result.GetFunctions().size()
                     << " lambda functions were retrieved." << std::endl;

           for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
               functions.push_back(functionConfiguration.GetFunctionName());
               std::cout << functions.size() << "  "
                         << functionConfiguration.GetDescription() << std::endl;
               std::cout << "   "
```

```
                            <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                    functionConfiguration.GetRuntime()) << ": "
                            << functionConfiguration.GetHandler()
                            << std::endl;
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    } while (!marker.empty());
```

- For API details, see ListFunctions in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To retrieve a list of Lambda functions

The following `list-functions` example displays a list of all of the functions for the current user.

```
aws lambda list-functions
```

Output:

```
{
    "Functions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "RevisionId": "1718e831-badf-4253-9518-d0644210af7b",
```

```
            "CodeSize": 294,
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:helloworld",
            "Handler": "helloworld.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-
role-zgur6bf4",
            "Timeout": 3,
            "LastModified": "2025-09-23T18:32:33.857+0000",
            "Runtime": "nodejs22.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2025-10-01T16:47:28.490+0000",
            "Runtime": "nodejs22.x",
            "Description": ""
        },
        {
            "Layers": [
                {
                    "CodeSize": 41784542,
                    "Arn": "arn:aws:lambda:us-
west-2:420165488524:layer:AWSLambda-Python37-SciPy1x:2"
                },
                {
```

```
                    "CodeSize": 4121,
                    "Arn": "arn:aws:lambda:us-
west-2:123456789012:layer:pythonLayer:1"
                }
            ],
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "ZQukCqxtkqFgyF2cU41Avj99TKQ/hNihPtDtRcc08mI=",
            "FunctionName": "my-python-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 128,
            "RevisionId": "80b4eabc-acf7-4ea8-919a-e874c213707d",
            "CodeSize": 299,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
python-function",
            "Handler": "lambda_function.lambda_handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/my-python-
function-role-z5g7dr6n",
            "Timeout": 3,
            "LastModified": "2025-10-01T19:40:41.643+0000",
            "Runtime": "python3.11",
            "Description": ""
        }
    ]
}
```

For more information, see Configure Lambda function memory in the *AWS Lambda Developer Guide*.

- For API details, see ListFunctions in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
```

```
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}
```

- For API details, see ListFunctions in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- For API details, see ListFunctions in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function listFunctions($maxItems = 50, $marker = null)
    {
        if (is_null($marker)) {
            return $this->lambdaClient->listFunctions([
                'MaxItems' => $maxItems,
            ]);
        }

        return $this->lambdaClient->listFunctions([
            'Marker' => $marker,
            'MaxItems' => $maxItems,
        ]);
    }
```

- For API details, see ListFunctions in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This sample displays all the Lambda functions with sorted code size

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

Output:

```
FunctionName                                                  Runtime   Timeout
 CodeSize
```

```
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- For API details, see ListFunctions in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This sample displays all the Lambda functions with sorted code size

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

Output:

```
FunctionName                                                    Runtime   Timeout
 CodeSize
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- For API details, see ListFunctions in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see ListFunctions in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Lists the Lambda functions for the current account.
  def list_functions
    functions = []
    @lambda_client.list_functions.each do |response|
      response['functions'].each do |function|
        functions.append(function['function_name'])
      end
    end
    functions
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error listing functions:\n #{e.message}")
  end
```

- For API details, see ListFunctions in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /** List all Lambda functions in the current Region. */
   pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
       info!("Listing lambda functions");
```

```
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- For API details, see ListFunctions in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   TRY.
       oo_result = lo_lmd->listfunctions( ).       " oo_result is returned for
testing purposes. "
       DATA(lt_functions) = oo_result->get_functions( ).
       MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
   ENDTRY.
```

- For API details, see ListFunctions in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Returns an array containing the names of all AWS Lambda functions
    /// available to the user.
    ///
    /// - Parameter lambdaClient: The `IAMClient` to use.
    ///
    /// - Throws: `ExampleError.listFunctionsError`
    ///
    /// - Returns: An array of lambda function name strings.
    func getFunctionNames(lambdaClient: LambdaClient) async throws -> [String] {
        let pages = lambdaClient.listFunctionsPaginated(
            input: ListFunctionsInput()
        )

        var functionNames: [String] = []

        for try await page in pages {
            guard let functions = page.functions else {
                throw ExampleError.listFunctionsError
            }

            for function in functions {
                functionNames.append(function.functionName ?? "<unknown>")
            }
        }

        return functionNames
    }
```

- For API details, see ListFunctions in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListProvisionedConcurrencyConfigs` with a CLI

The following code examples show how to use `ListProvisionedConcurrencyConfigs`.

CLI

AWS CLI

To get a list of provisioned concurrency configurations

The following `list-provisioned-concurrency-configs` example lists the provisioned concurrency configurations for the specified function.

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

Output:

```
{
    "ProvisionedConcurrencyConfigs": [
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:GREEN",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:29:00+0000"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:BLUE",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
```

```
            "Status": "READY",
            "LastModified": "2019-12-31T20:28:49+0000"
        }
    ]
}
```

- For API details, see ListProvisionedConcurrencyConfigs in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example retrieves the list of provisioned concurrency configurations for a Lambda function.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- For API details, see ListProvisionedConcurrencyConfigs in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example retrieves the list of provisioned concurrency configurations for a Lambda function.

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- For API details, see ListProvisionedConcurrencyConfigs in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListTags` with a CLI

The following code examples show how to use `ListTags`.

CLI

AWS CLI

To retrieve the list of tags for a Lambda function

The following `list-tags` example displays the tags attached to the `my-function` Lambda function.

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

Output:

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

For more information, see Tagging Lambda Functions in the *AWS Lambda Developer Guide*.

- For API details, see ListTags in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Retrieves the tags and their values currently set on the specified function.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

Output:

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
```

```
Washington Olympia
```

- For API details, see ListTags in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Retrieves the tags and their values currently set on the specified function.

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

Output:

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- For API details, see ListTags in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListVersionsByFunction` with a CLI

The following code examples show how to use `ListVersionsByFunction`.

CLI

AWS CLI

To retrieve a list of versions of a function

The following `list-versions-by-function` example displays the list of versions for the `my-function` Lambda function.

```
aws lambda list-versions-by-function \
    --function-name my-function
```

Output:


```
{
    "Versions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:$LATEST",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "1",
            "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "949c8914-012e-4795-998c-e467121951b1",
            "CodeSize": 304,
```

```
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:1",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-09-26T20:28:40.438+0000",
            "Runtime": "nodejs10.x",
            "Description": "new version"
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "2",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "cd669f21-0f3d-4e1c-9566-948837f2e2ea",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": "newer version"
        }
    ]
}
```

For more information, see Configuring AWS Lambda Function Aliases in the *AWS Lambda Developer Guide*.

- For API details, see ListVersionsByFunction in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example returns the list of version specific configurations for each version of the Lambda Function.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

Output:

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- For API details, see ListVersionsByFunction in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example returns the list of version specific configurations for each version of the Lambda Function.

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

Output:

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
```

```
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- For API details, see ListVersionsByFunction in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PublishVersion` with a CLI

The following code examples show how to use `PublishVersion`.

CLI

AWS CLI

To publish a new version of a function

The following `publish-version` example publishes a new version of the `my-function` Lambda function.

```
aws lambda publish-version \
    --function-name my-function
```

Output:

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
    "FunctionName": "my-function",
    "CodeSize": 294,
    "RevisionId": "f31d3d39-cc63-4520-97d4-43cd44c94c20",
    "MemorySize": 128,
```

```
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:3",
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2019-09-23T18:32:33.857+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs10.x",
    "Description": ""
}
```

For more information, see Configuring AWS Lambda Function Aliases in the *AWS Lambda Developer Guide*.

- For API details, see PublishVersion in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example creates a version for the existing snapshot of Lambda Function Code

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- For API details, see PublishVersion in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example creates a version for the existing snapshot of Lambda Function Code

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- For API details, see PublishVersion in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use PutFunctionConcurrency with a CLI

The following code examples show how to use PutFunctionConcurrency.

CLI

AWS CLI

To configure a reserved concurrency limit for a function

The following put-function-concurrency example configures 100 reserved concurrent executions for the my-function function.

```
aws lambda put-function-concurrency \
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

Output:

```
{
    "ReservedConcurrentExecutions": 100
}
```

For more information, see Reserving Concurrency for a Lambda Function in the *AWS Lambda Developer Guide*.

- For API details, see PutFunctionConcurrency in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example applies the concurrency settings for the Function as a whole.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- For API details, see PutFunctionConcurrency in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example applies the concurrency settings for the Function as a whole.

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- For API details, see PutFunctionConcurrency in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PutProvisionedConcurrencyConfig` with a CLI

The following code examples show how to use `PutProvisionedConcurrencyConfig`.

CLI

AWS CLI

To allocate provisioned concurrency

The following `put-provisioned-concurrency-config` example allocates 100 provisioned concurrency for the BLUE alias of the specified function.

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

Output:

```
{
    "Requested ProvisionedConcurrentExecutions": 100,
    "Allocated ProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

- For API details, see PutProvisionedConcurrencyConfig in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example adds a provisioned concurrency configuration to a Function's Alias

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- For API details, see PutProvisionedConcurrencyConfig in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example adds a provisioned concurrency configuration to a Function's Alias

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- For API details, see PutProvisionedConcurrencyConfig in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `RemovePermission` with a CLI

The following code examples show how to use `RemovePermission`.

CLI

AWS CLI

To remove permissions from an existing Lambda function

The following `remove-permission` example removes permission to invoke a function named `my-function`.

```
aws lambda remove-permission \
    --function-name my-function \
    --statement-id sns
```

This command produces no output.

For more information, see Using Resource-based Policies for AWS Lambda in the *AWS Lambda Developer Guide*.

- For API details, see RemovePermission in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example removes the function policy for the specified StatementId of a Lambda Function.

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- For API details, see RemovePermission in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example removes the function policy for the specified StatementId of a Lambda Function.

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- For API details, see RemovePermission in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `TagResource` with a CLI

The following code examples show how to use `TagResource`.

CLI

AWS CLI

To add tags to an existing Lambda function

The following `tag-resource` example adds a tag with the key name `DEPARTMENT` and a value of `Department A` to the specified Lambda function.

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

This command produces no output.

For more information, see Tagging Lambda Functions in the *AWS Lambda Developer Guide*.

- For API details, see TagResource in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Adds the three tags (Washington, Oregon and California) and their associated values to the specified function identified by its ARN.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- For API details, see TagResource in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Adds the three tags (Washington, Oregon and California) and their associated values to the specified function identified by its ARN.

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- For API details, see TagResource in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UntagResource` with a CLI

The following code examples show how to use `UntagResource`.

CLI

AWS CLI

To remove tags from an existing Lambda function

The following `untag-resource` example removes the tag with the key name `DEPARTMENT` tag from the `my-function` Lambda function.

```
aws lambda untag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

This command produces no output.

For more information, see Tagging Lambda Functions in the *AWS Lambda Developer Guide*.

- For API details, see UntagResource in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Removes the supplied tags from a function. The cmdlet will prompt for confirmation before proceeding unless the -Force switch is specified. A single call is made to the service to remove the tags.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

Example 2: Removes the supplied tags from a function. The cmdlet will prompt for confirmation before proceeding unless the -Force switch is specified. Once call to the service is made per supplied tag.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- For API details, see UntagResource in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Removes the supplied tags from a function. The cmdlet will prompt for confirmation before proceeding unless the -Force switch is specified. A single call is made to the service to remove the tags.

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

Example 2: Removes the supplied tags from a function. The cmdlet will prompt for confirmation before proceeding unless the -Force switch is specified. Once call to the service is made per supplied tag.

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- For API details, see UntagResource in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UpdateAlias` with a CLI

The following code examples show how to use `UpdateAlias`.

CLI

AWS CLI

To update a function alias

The following `update-alias` example updates the alias named `LIVE` to point to version 3 of the `my-function` Lambda function.

```
aws lambda update-alias \
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

Output:

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

For more information, see Configuring AWS Lambda Function Aliases in the *AWS Lambda Developer Guide*.

- For API details, see UpdateAlias in *AWS CLI Command Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example updates the Configuration of an existing Lambda function Alias. It updates the RoutingConfiguration value to shift 60% (0.6) of traffic to version 1

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- For API details, see UpdateAlias in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example updates the Configuration of an existing Lambda function Alias. It updates the RoutingConfiguration value to shift 60% (0.6) of traffic to version 1

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- For API details, see UpdateAlias in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UpdateFunctionCode` with an AWS SDK or CLI

The following code examples show how to use `UpdateFunctionCode`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
```

```
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
_lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
{response.LastModified}.");
    }
```

- For API details, see UpdateFunctionCode in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";
```

```
    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
 client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
 std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- For API details, see UpdateFunctionCode in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To update the code of a Lambda function

The following `update-function-code` example replaces the code of the unpublished ($LATEST) version of the `my-function` function with the contents of the specified zip file.

```
aws lambda update-function-code \
    --function-name  my-function \
    --zip-file fileb://my-function.zip
```

Output:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

For more information, see AWS Lambda Function Configuration in the *AWS Lambda Developer Guide*.

- For API details, see UpdateFunctionCode in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
```

```
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- For API details, see UpdateFunctionCode in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /**
     * Updates the code for an AWS Lambda function.
     *
     * @param awsLambda  the AWS Lambda client
```

```
    * @param functionName the name of the Lambda function to update
    * @param bucketName the name of the S3 bucket where the function code is
located
    * @param key the key (file name) of the function code in the S3 bucket
    * @throws LambdaException if there is an error updating the function code
    */
   public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
       try {
           LambdaWaiter waiter = awsLambda.waiter();
           UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                .functionName(functionName)
                .publish(true)
                .s3Bucket(bucketName)
                .s3Key(key)
                .build();

           UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
           GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .build();

           WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
           waiterResponse.matched().response().ifPresent(System.out::println);
           System.out.println("The last modified value is " +
response.lastModified());

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- For API details, see UpdateFunctionCode in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- For API details, see UpdateFunctionCode in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
    {
```

```
        return $this->lambdaClient->updateFunctionCode([
            'FunctionName' => $functionName,
            'S3Bucket' => $s3Bucket,
            'S3Key' => $s3Key,
        ]);
    }
```

- For API details, see UpdateFunctionCode in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: Updates the function named 'MyFunction' with new content contained in the specified zip file. For a C# .NET Core Lambda function the zip file should contain the compiled assembly.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

Example 2: This example is similar to the previous one but uses an Amazon S3 object containing the updated code to update the function.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- For API details, see UpdateFunctionCode in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: Updates the function named 'MyFunction' with new content contained in the specified zip file. For a C# .NET Core Lambda function the zip file should contain the compiled assembly.

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

Example 2: This example is similar to the previous one but uses an Amazon S3 object containing the updated code to update the function.

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- For API details, see UpdateFunctionCode in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
 contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
 in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
```

```
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- For API details, see UpdateFunctionCode in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the code for a Lambda function by submitting a .zip archive that
 contains
  # the code for the function.
  #
  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
```

```
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

- For API details, see UpdateFunctionCode in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
```

```
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- For API details, see UpdateFunctionCode in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   TRY.
       oo_result = lo_lmd->updatefunctioncode(     " oo_result is returned for
testing purposes. "
             iv_functionname = iv_function_name
             iv_zipfile = io_zip_file ).

       MESSAGE 'Lambda function code updated.' TYPE 'I'.
     CATCH /aws1/cx_lmdcodesigningcfgno00.
       MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodestorageexcdex.
       MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodeverification00.
       MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidcodesigex.
       MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
```

```
    ENDTRY.
```

- For API details, see UpdateFunctionCode in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSClientRuntime
import AWSLambda
import Foundation

        let zipUrl = URL(fileURLWithPath: path)
        let zipData: Data

        // Read the function's Zip file.

        do {
            zipData = try Data(contentsOf: zipUrl)
        } catch {
            throw ExampleError.zipFileReadError
        }

        // Update the function's code and wait for the updated version to be
        // ready for use.

        do {
            _ = try await lambdaClient.updateFunctionCode(
                input: UpdateFunctionCodeInput(
                    functionName: functionName,
                    zipFile: zipData
                )
            )
        } catch {
            return false
```

```
        }
```

- For API details, see UpdateFunctionCode in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `UpdateFunctionConfiguration` with an AWS SDK or CLI

The following code examples show how to use `UpdateFunctionConfiguration`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Learn the basics

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
```

```
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
environmentVariables },
        };

        var response = await
_lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for .NET API Reference*.

C++

SDK for C++

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
```

```
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for C++ API Reference*.

CLI

AWS CLI

To modify the configuration of a function

The following `update-function-configuration` example modifies the memory size to be 256 MB for the unpublished ($LATEST) version of the `my-function` function.

```
aws lambda update-function-configuration \
    --function-name  my-function \
    --memory-size 256
```

Output:

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
```

```
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

For more information, see AWS Lambda Function Configuration in the *AWS Lambda Developer Guide*.

- For API details, see UpdateFunctionConfiguration in *AWS CLI Command Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
```

```
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   /**
    * Updates the configuration of an AWS Lambda function.
    *
    * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
    * @param functionName  the name of the AWS Lambda function to update
    * @param handler       the new handler for the AWS Lambda function
    *
    * @throws LambdaException if there is an error while updating the function
configuration
    */
   public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
       try {
           UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
               .functionName(functionName)
               .handler(handler)
               .runtime(Runtime.JAVA17)
               .build();

           awsLambda.updateFunctionConfiguration(configurationRequest);

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  const result = client.send(command);
  waitForFunctionUpdated({ FunctionName: funcName });
  return result;
};
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    public function updateFunctionConfiguration($functionName, $handler,
 $environment = '')
    {
        return $this->lambdaClient->updateFunctionConfiguration([
            'FunctionName' => $functionName,
```

```
            'Handler' => "$handler.lambda_handler",
            'Environment' => $environment,
        ]);
    }
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for PHP API Reference*.

PowerShell

Tools for PowerShell V4

Example 1: This example updates the existing Lambda Function Configuration

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- For API details, see UpdateFunctionConfiguration in *AWS Tools for PowerShell Cmdlet Reference (V4)*.

Tools for PowerShell V5

Example 1: This example updates the existing Lambda Function Configuration

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- For API details, see UpdateFunctionConfiguration in *AWS Tools for PowerShell Cmdlet Reference (V5)*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for Python (Boto3) API Reference*.

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
                                                     }
                                                   }
                                                 })
    @lambda_client.wait_until(:function_updated_v2, { function_name:
 function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
```

```
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for Ruby API Reference*.

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;
```

```
        Ok(updated)
    }
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for Rust API reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
   TRY.
       oo_result = lo_lmd->updatefunctionconfiguration(     " oo_result is
returned for testing purposes. "
             iv_functionname = iv_function_name
             iv_runtime = iv_runtime
             iv_description  = 'Updated Lambda function'
             iv_memorysize  = iv_memory_size ).

       MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
     CATCH /aws1/cx_lmdcodesigningcfgno00.
       MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdcodeverification00.
       MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidcodesigex.
       MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for SAP ABAP API reference*.

Swift

SDK for Swift

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Tell the server-side component to log debug output by setting its
    /// environment's `LOG_LEVEL` to `DEBUG`.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to enable debug
    ///     logging for.
    ///
    /// - Throws: `ExampleError.environmentResponseMissingError`,
    ///   `ExampleError.updateFunctionConfigurationError`,
    ///   `ExampleError.environmentVariablesMissingError`,
    ///   `ExampleError.logLevelIncorrectError`,
    ///   `ExampleError.updateFunctionConfigurationError`
    func enableDebugLogging(lambdaClient: LambdaClient, functionName: String)
 async throws {
        let envVariables = [
            "LOG_LEVEL": "DEBUG"
        ]
        let environment = LambdaClientTypes.Environment(variables: envVariables)

        do {
```

```
            let output = try await lambdaClient.updateFunctionConfiguration(
                input: UpdateFunctionConfigurationInput(
                    environment: environment,
                    functionName: functionName
                )
            )

            guard let response = output.environment else {
                throw ExampleError.environmentResponseMissingError
            }

            if response.error != nil {
                throw ExampleError.updateFunctionConfigurationError
            }

            guard let retVariables = response.variables else {
                throw ExampleError.environmentVariablesMissingError
            }

            for envVar in retVariables {
                if envVar.key == "LOG_LEVEL" && envVar.value != "DEBUG" {
                    print("*** Log level is not set to DEBUG!")
                    throw ExampleError.logLevelIncorrectError
                }
            }
        } catch {
            throw ExampleError.updateFunctionConfigurationError
        }
    }
```

- For API details, see UpdateFunctionConfiguration in *AWS SDK for Swift API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Scenarios for Lambda using AWS SDKs

The following code examples show you how to implement common scenarios in Lambda with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions

within Lambda or combined with other AWS services. Each scenario includes a link to the complete source code, where you can find instructions on how to set up and run the code.

Scenarios target an intermediate level of experience to help you understand service actions in context.

Examples

- Automatically confirm known Amazon Cognito users with a Lambda function using an AWS SDK
- Automatically migrate known Amazon Cognito users with a Lambda function using an AWS SDK
- Create an API Gateway REST API to track COVID-19 data
- Create a lending library REST API
- Create a messenger application with Step Functions
- Create a photo asset management application that lets users manage photos using labels
- Create a websocket chat application with API Gateway
- Create an application that analyzes customer feedback and synthesizes audio
- Invoke a Lambda function from a browser
- Transform data for your application with S3 Object Lambda
- Use API Gateway to invoke a Lambda function
- Use Step Functions to invoke Lambda functions
- Use scheduled events to invoke a Lambda function
- Use the Amazon Neptune API to develop a Lambda function that queries graph data
- Write custom activity data with a Lambda function after Amazon Cognito user authentication using an AWS SDK

Automatically confirm known Amazon Cognito users with a Lambda function using an AWS SDK

The following code examples show how to automatically confirm known Amazon Cognito users with a Lambda function.

- Configure a user pool to call a Lambda function for the `PreSignUp` trigger.
- Sign up a user with Amazon Cognito.
- The Lambda function scans a DynamoDB table and automatically confirms known users.

- Sign in as the new user, then clean up resources.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario at a command prompt.

```
import (
 "context"
 "errors"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// AutoConfirm separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type AutoConfirm struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewAutoConfirm constructs a new auto confirm runner.
func NewAutoConfirm(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) AutoConfirm {
 scenario := AutoConfirm{
```

```
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddPreSignUpTrigger adds a Lambda handler as an invocation target for the
 PreSignUp trigger.
func (runner *AutoConfirm) AddPreSignUpTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the PreSignUp trigger from
 Cognito.\n" +
  "This trigger happens when a user signs up, and lets your function take action
 before the main Cognito\n" +
  "sign up processing occurs.\n")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PreSignUp, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the PreSignUp
 trigger.\n",
  functionArn, userPoolId)
}

// SignUpUser signs up a user from the known user table with a password you
 specify.
func (runner *AutoConfirm) SignUpUser(ctx context.Context, clientId string,
 usersTable string) (string, string) {
 log.Println("Let's sign up a user to your Cognito user pool. When the user's
 email matches an email in the\n" +
  "DynamoDB known users table, it is automatically verified and the user is
 confirmed.")

 knownUsers, err := runner.helper.GetKnownUsers(ctx, usersTable)
 if err != nil {
  panic(err)
 }
```

```
 userChoice := runner.questioner.AskChoice("Which user do you want to use?\n",
 knownUsers.UserNameList())
 user := knownUsers.Users[userChoice]

 var signedUp bool
 var userConfirmed bool
 password := runner.questioner.AskPassword("Enter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !signedUp {
  log.Printf("Signing up user '%v' with email '%v' to Cognito.\n", user.UserName,
 user.UserEmail)
  userConfirmed, err = runner.cognitoActor.SignUp(ctx, clientId, user.UserName,
 password, user.UserEmail)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("Enter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   signedUp = true
  }
 }
 log.Printf("User %v signed up, confirmed = %v.\n", user.UserName, userConfirmed)

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// SignInUser signs in a user.
func (runner *AutoConfirm) SignInUser(ctx context.Context, clientId string,
 userName string, password string) string {
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 log.Printf("Let's sign in as %v...\n", userName)
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 log.Println(strings.Repeat("-", 88))
```

```
 return *authResult.AccessToken
}

// Run runs the scenario.
func (runner *AutoConfirm) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])

 runner.AddPreSignUpTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["AutoConfirmFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers, actions.PreSignUp)
 userName, password := runner.SignUpUser(ctx, stackOutputs["UserPoolClientId"],
 stackOutputs["TableName"])
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["AutoConfirmFunction"])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
  runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password))

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Handle the `PreSignUp` trigger with a Lambda function.

```
import (
 "context"
 "log"
 "os"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 dynamodbtypes "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PreSignUp event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be confirmed and verified.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsPreSignup) (events.CognitoEventUserPoolsPreSignup,
 error) {
```

```
 log.Printf("Received presignup from %v for user '%v'", event.TriggerSource,
 event.UserName)
 if event.TriggerSource != "PreSignUp_SignUp" {
  // Other trigger sources, such as PreSignUp_AdminInitiateAuth, ignore the
 response from this handler.
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserEmail: event.Request.UserAttributes["email"],
 }
 log.Printf("Looking up email %v in table %v.\n", user.UserEmail, tableName)
 output, err := h.dynamoClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key:       user.GetKey(),
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Error looking up email %v.\n", user.UserEmail)
  return event, err
 }
 if output.Item == nil {
  log.Printf("Email %v not found. Email verification is required.\n",
 user.UserEmail)
  return event, err
 }

 err = attributevalue.UnmarshalMap(output.Item, &user)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB item. Here's why: %v\n", err)
  return event, err
 }

 if user.UserName != event.UserName {
  log.Printf("UserEmail %v found, but stored UserName '%v' does not match
 supplied UserName '%v'. Verification is required.\n",
   user.UserEmail, user.UserName, event.UserName)
 } else {
  log.Printf("UserEmail %v found with matching UserName %v. User is confirmed.
\n", user.UserEmail, user.UserName)
  event.Response.AutoConfirmUser = true
  event.Response.AutoVerifyEmail = true
 }

 return event, err
```

```
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

Create a struct that performs common tasks.

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
```

```
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
```

```
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
```

```
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Create a struct that wraps Amazon Cognito actions.

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)
```

```
type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}
```

```
// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
```

```
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
```

```
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
```

```
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

Create a struct that wraps DynamoDB actions.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
```

```
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
```

```
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
```

```
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

Create a struct that wraps CloudWatch Logs actions.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
```

```
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Create a struct that wraps CloudFormation actions.

```
import (
 "context"
 "log"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

Clean up resources.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)
```

```
// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
```

```
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Configure an interactive "Scenario" run. The JavaScript (v3) examples share a Scenario runner to streamline complex examples. The complete source code is on GitHub.

```
import { AutoConfirm } from "./scenario-auto-confirm.js";
```

```
/**
 * The context is passed to every scenario. Scenario steps
 * will modify the context.
 */
const context = {
  errors: [],
  users: [
    {
      UserName: "test_user_1",
      UserEmail: "test_email_1@example.com",
    },
    {
      UserName: "test_user_2",
      UserEmail: "test_email_2@example.com",
    },
    {
      UserName: "test_user_3",
      UserEmail: "test_email_3@example.com",
    },
  ],
};

/**
 * Three Scenarios are created for the workflow. A Scenario is an orchestration
 class
 * that simplifies running a series of steps.
 */
export const scenarios = {
  // Demonstrate automatically confirming known users in a database.
  "auto-confirm": AutoConfirm(context),
};

// Call function if run directly
import { fileURLToPath } from "node:url";
import { parseScenarioArgs } from "@aws-doc-sdk-examples/lib/scenario/index.js";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios, {
    name: "Cognito user pools and triggers",
    description:
      "Demonstrate how to use the AWS SDKs to customize Amazon Cognito
 authentication behavior.",
  });
}
```

This Scenario demonstrates auto-confirming a known user. It orchestrates the example steps.

```
import { wait } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";
import {
  Scenario,
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";

import {
  getStackOutputs,
  logCleanUpReminder,
  promptForStackName,
  promptForStackRegion,
  skipWhenErrors,
} from "./steps-common.js";
import { populateTable } from "./actions/dynamodb-actions.js";
import {
  addPreSignUpHandler,
  deleteUser,
  getUser,
  signIn,
  signUpUser,
} from "./actions/cognito-actions.js";
import {
  getLatestLogStreamForLambda,
  getLogEvents,
} from "./actions/cloudwatch-logs-actions.js";

/**
 * @typedef {{
 *   errors: Error[],
 *   password: string,
 *   users: { UserName: string, UserEmail: string }[],
 *   selectedUser?: string,
 *   stackName?: string,
 *   stackRegion?: string,
 *   token?: string,
 *   confirmDeleteSignedInUser?: boolean,
```

```
 *   TableName?: string,
 *   UserPoolClientId?: string,
 *   UserPoolId?: string,
 *   UserPoolArn?: string,
 *   AutoConfirmHandlerArn?: string,
 *   AutoConfirmHandlerName?: string
 * }} State
 */

const greeting = new ScenarioOutput(
  "greeting",
  (/** @type {State} */ state) => `This demo will populate some users into the \
database created as part of the "${state.stackName}" stack. \
Then the AutoConfirmHandler will be linked to the PreSignUp \
trigger from Cognito. Finally, you will choose a user to sign up.`,
  { skipWhen: skipWhenErrors },
);

const logPopulatingUsers = new ScenarioOutput(
  "logPopulatingUsers",
  "Populating the DynamoDB table with some users.",
  { skipWhenErrors: skipWhenErrors },
);

const logPopulatingUsersComplete = new ScenarioOutput(
  "logPopulatingUsersComplete",
  "Done populating users.",
  { skipWhen: skipWhenErrors },
);

const populateUsers = new ScenarioAction(
  "populateUsers",
  async (/** @type {State} */ state) => {
    const [_, err] = await populateTable({
      region: state.stackRegion,
      tableName: state.TableName,
      items: state.users,
    });
    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: skipWhenErrors,
```

```
  },
);

const logSetupSignUpTrigger = new ScenarioOutput(
  "logSetupSignUpTrigger",
  "Setting up the PreSignUp trigger for the Cognito User Pool.",
  { skipWhen: skipWhenErrors },
);

const setupSignUpTrigger = new ScenarioAction(
  "setupSignUpTrigger",
  async (/** @type {State} */ state) => {
    const [_, err] = await addPreSignUpHandler({
      region: state.stackRegion,
      userPoolId: state.UserPoolId,
      handlerArn: state.AutoConfirmHandlerArn,
    });
    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const logSetupSignUpTriggerComplete = new ScenarioOutput(
  "logSetupSignUpTriggerComplete",
  (
    /** @type {State} */ state,
  ) => `The lambda function "${state.AutoConfirmHandlerName}" \
has been configured as the PreSignUp trigger handler for the user pool
 "${state.UserPoolId}".`,
  { skipWhen: skipWhenErrors },
);

const selectUser = new ScenarioInput(
  "selectedUser",
  "Select a user to sign up.",
  {
    type: "select",
    choices: (/** @type {State} */ state) => state.users.map((u) => u.UserName),
    skipWhen: skipWhenErrors,
    default: (/** @type {State} */ state) => state.users[0].UserName,
```

```
  },
);

const checkIfUserAlreadyExists = new ScenarioAction(
  "checkIfUserAlreadyExists",
  async (/** @type {State} */ state) => {
    const [user, err] = await getUser({
      region: state.stackRegion,
      userPoolId: state.UserPoolId,
      username: state.selectedUser,
    });

    if (err?.name === "UserNotFoundException") {
      // Do nothing. We're not expecting the user to exist before
      // sign up is complete.
      return;
    }

    if (err) {
      state.errors.push(err);
      return;
    }

    if (user) {
      state.errors.push(
        new Error(
          `The user "${state.selectedUser}" already exists in the user pool
 "${state.UserPoolId}".`,
        ),
      );
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const createPassword = new ScenarioInput(
  "password",
  "Enter a password that has at least eight characters, uppercase, lowercase,
 numbers and symbols.",
  { type: "password", skipWhen: skipWhenErrors, default: "Abcd1234!" },
);
```

```
const logSignUpExistingUser = new ScenarioOutput(
  "logSignUpExistingUser",
  (/** @type {State} */ state) => `Signing up user "${state.selectedUser}".`,
  { skipWhen: skipWhenErrors },
);

const signUpExistingUser = new ScenarioAction(
  "signUpExistingUser",
  async (/** @type {State} */ state) => {
    const signUp = (password) =>
      signUpUser({
        region: state.stackRegion,
        userPoolClientId: state.UserPoolClientId,
        username: state.selectedUser,
        email: state.users.find((u) => u.UserName === state.selectedUser)
          .UserEmail,
        password,
      });

    let [_, err] = await signUp(state.password);

    while (err?.name === "InvalidPasswordException") {
      console.warn("The password you entered was invalid.");
      await createPassword.handle(state);
      [_, err] = await signUp(state.password);
    }

    if (err) {
      state.errors.push(err);
    }
  },
  { skipWhen: skipWhenErrors },
);

const logSignUpExistingUserComplete = new ScenarioOutput(
  "logSignUpExistingUserComplete",
  (/** @type {State} */ state) =>
    `"${state.selectedUser} was signed up successfully.`,
  { skipWhen: skipWhenErrors },
);

const logLambdaLogs = new ScenarioAction(
  "logLambdaLogs",
  async (/** @type {State} */ state) => {
```

```
    console.log(
      "Waiting a few seconds to let Lambda write to CloudWatch Logs...\n",
    );
    await wait(10);

    const [logStream, logStreamErr] = await getLatestLogStreamForLambda({
      functionName: state.AutoConfirmHandlerName,
      region: state.stackRegion,
    });
    if (logStreamErr) {
      state.errors.push(logStreamErr);
      return;
    }

    console.log(
      `Getting some recent events from log stream "${logStream.logStreamName}"`,
    );
    const [logEvents, logEventsErr] = await getLogEvents({
      functionName: state.AutoConfirmHandlerName,
      region: state.stackRegion,
      eventCount: 10,
      logStreamName: logStream.logStreamName,
    });
    if (logEventsErr) {
      state.errors.push(logEventsErr);
      return;
    }

    console.log(logEvents.map((ev) => `\t${ev.message}`).join(""));
  },
  { skipWhen: skipWhenErrors },
);

const logSignInUser = new ScenarioOutput(
  "logSignInUser",
  (/** @type {State} */ state) => `Let's sign in as ${state.selectedUser}`,
  { skipWhen: skipWhenErrors },
);

const signInUser = new ScenarioAction(
  "signInUser",
  async (/** @type {State} */ state) => {
    const [response, err] = await signIn({
      region: state.stackRegion,
```

```
      clientId: state.UserPoolClientId,
      username: state.selectedUser,
      password: state.password,
    });

    if (err?.name === "PasswordResetRequiredException") {
      state.errors.push(new Error("Please reset your password."));
      return;
    }

    if (err) {
      state.errors.push(err);
      return;
    }

    state.token = response?.AuthenticationResult?.AccessToken;
  },
  { skipWhen: skipWhenErrors },
);

const logSignInUserComplete = new ScenarioOutput(
  "logSignInUserComplete",
  (/** @type {State} */ state) =>
    `Successfully signed in. Your access token starts with:
 ${state.token.slice(0, 11)}`,
  { skipWhen: skipWhenErrors },
);

const confirmDeleteSignedInUser = new ScenarioInput(
  "confirmDeleteSignedInUser",
  "Do you want to delete the currently signed in user?",
  { type: "confirm", skipWhen: skipWhenErrors },
);

const deleteSignedInUser = new ScenarioAction(
  "deleteSignedInUser",
  async (/** @type {State} */ state) => {
    const [_, err] = await deleteUser({
      region: state.stackRegion,
      accessToken: state.token,
    });

    if (err) {
      state.errors.push(err);
```

```
    }
  },
  {
    skipWhen: (/** @type {State} */ state) =>
      skipWhenErrors(state) || !state.confirmDeleteSignedInUser,
  },
);

const logErrors = new ScenarioOutput(
  "logErrors",
  (/** @type {State}*/ state) => {
    const errorList = state.errors
      .map((err) => ` - ${err.name}: ${err.message}`)
      .join("\n");
    return `Scenario errors found:\n${errorList}`;
  },
  {
    // Don't log errors when there aren't any!
    skipWhen: (/** @type {State} */ state) => state.errors.length === 0,
  },
);

export const AutoConfirm = (context) =>
  new Scenario(
    "AutoConfirm",
    [
      promptForStackName,
      promptForStackRegion,
      getStackOutputs,
      greeting,
      logPopulatingUsers,
      populateUsers,
      logPopulatingUsersComplete,
      logSetupSignUpTrigger,
      setupSignUpTrigger,
      logSetupSignUpTriggerComplete,
      selectUser,
      checkIfUserAlreadyExists,
      createPassword,
      logSignUpExistingUser,
      signUpExistingUser,
      logSignUpExistingUserComplete,
      logLambdaLogs,
      logSignInUser,
```

```
      signInUser,
      logSignInUserComplete,
      confirmDeleteSignedInUser,
      deleteSignedInUser,
      logCleanUpReminder,
      logErrors,
    ],
    context,
  );
```

These are steps that are shared with other Scenarios.

```
import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";
import { getCfnOutputs } from "@aws-doc-sdk-examples/lib/sdk/cfn-outputs.js";

export const skipWhenErrors = (state) => state.errors.length > 0;

export const getStackOutputs = new ScenarioAction(
  "getStackOutputs",
  async (state) => {
    if (!state.stackName || !state.stackRegion) {
      state.errors.push(
        new Error(
          "No stack name or region provided. The stack name and \
region are required to fetch CFN outputs relevant to this example.",
        ),
      );
      return;
    }

    const outputs = await getCfnOutputs(state.stackName, state.stackRegion);
    Object.assign(state, outputs);
  },
);

export const promptForStackName = new ScenarioInput(
  "stackName",
  "Enter the name of the stack you deployed earlier.",
```

```
  { type: "input", default: "PoolsAndTriggersStack" },
);

export const promptForStackRegion = new ScenarioInput(
  "stackRegion",
  "Enter the region of the stack you deployed earlier.",
  { type: "input", default: "us-east-1" },
);

export const logCleanUpReminder = new ScenarioOutput(
  "logCleanUpReminder",
  "All done. Remember to run 'cdk destroy' to teardown the stack.",
  { skipWhen: skipWhenErrors },
);
```

A handler for the `PreSignUp` trigger with a Lambda function.

```
import type { PreSignUpTriggerEvent, Handler } from "aws-lambda";
import type { UserRepository } from "./user-repository";
import { DynamoDBUserRepository } from "./user-repository";

export class PreSignUpHandler {
  private userRepository: UserRepository;

  constructor(userRepository: UserRepository) {
    this.userRepository = userRepository;
  }

  private isPreSignUpTriggerSource(event: PreSignUpTriggerEvent): boolean {
    return event.triggerSource === "PreSignUp_SignUp";
  }

  private getEventUserEmail(event: PreSignUpTriggerEvent): string {
    return event.request.userAttributes.email;
  }

  async handlePreSignUpTriggerEvent(
    event: PreSignUpTriggerEvent,
  ): Promise<PreSignUpTriggerEvent> {
    console.log(
      `Received presignup from ${event.triggerSource} for user
 '${event.userName}'`,
```

```
    );

    if (!this.isPreSignUpTriggerSource(event)) {
      return event;
    }

    const eventEmail = this.getEventUserEmail(event);
    console.log(`Looking up email ${eventEmail}.`);
    const storedUserInfo =
      await this.userRepository.getUserInfoByEmail(eventEmail);

    if (!storedUserInfo) {
      console.log(
        `Email ${eventEmail} not found. Email verification is required.`,
      );
      return event;
    }

    if (storedUserInfo.UserName !== event.userName) {
      console.log(
        `UserEmail ${eventEmail} found, but stored UserName
 '${storedUserInfo.UserName}' does not match supplied UserName
 '${event.userName}'. Verification is required.`,
      );
    } else {
      console.log(
        `UserEmail ${eventEmail} found with matching UserName
 ${storedUserInfo.UserName}. User is confirmed.`,
      );
      event.response.autoConfirmUser = true;
      event.response.autoVerifyEmail = true;
    }
    return event;
  }
}

const createPreSignUpHandler = (): PreSignUpHandler => {
  const tableName = process.env.TABLE_NAME;
  if (!tableName) {
    throw new Error("TABLE_NAME environment variable is not set");
  }

  const userRepository = new DynamoDBUserRepository(tableName);
  return new PreSignUpHandler(userRepository);
```

```
};

export const handler: Handler = async (event: PreSignUpTriggerEvent) => {
  const preSignUpHandler = createPreSignUpHandler();
  return preSignUpHandler.handlePreSignUpTriggerEvent(event);
};
```

Module of CloudWatch Logs actions.

```
import {
  CloudWatchLogsClient,
  GetLogEventsCommand,
  OrderBy,
  paginateDescribeLogStreams,
} from "@aws-sdk/client-cloudwatch-logs";

/**
 * Get the latest log stream for a Lambda function.
 * @param {{ functionName: string, region: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cloudwatch-logs").LogStream | null,
 unknown]>}
 */
export const getLatestLogStreamForLambda = async ({ functionName, region }) => {
  try {
    const logGroupName = `/aws/lambda/${functionName}`;
    const cwlClient = new CloudWatchLogsClient({ region });
    const paginator = paginateDescribeLogStreams(
      { client: cwlClient },
      {
        descending: true,
        limit: 1,
        orderBy: OrderBy.LastEventTime,
        logGroupName,
      },
    );

    for await (const page of paginator) {
      return [page.logStreams[0], null];
    }
  } catch (err) {
    return [null, err];
```

```
  }
};

/**
 * Get the log events for a Lambda function's log stream.
 * @param {{
 *   functionName: string,
 *   logStreamName: string,
 *   eventCount: number,
 *   region: string
 * }} config
 * @returns {Promise<[import("@aws-sdk/client-cloudwatch-logs").OutputLogEvent[]
 | null, unknown]>}
 */
export const getLogEvents = async ({
  functionName,
  logStreamName,
  eventCount,
  region,
}) => {
  try {
    const cwlClient = new CloudWatchLogsClient({ region });
    const logGroupName = `/aws/lambda/${functionName}`;
    const response = await cwlClient.send(
      new GetLogEventsCommand({
        logStreamName: logStreamName,
        limit: eventCount,
        logGroupName: logGroupName,
      }),
    );

    return [response.events, null];
  } catch (err) {
    return [null, err];
  }
};
```

Module of Amazon Cognito actions.

```
import {
  AdminGetUserCommand,
```

```
  CognitoIdentityProviderClient,
  DeleteUserCommand,
  InitiateAuthCommand,
  SignUpCommand,
  UpdateUserPoolCommand,
} from "@aws-sdk/client-cognito-identity-provider";

/**
 * Connect a Lambda function to the PreSignUp trigger for a Cognito user pool
 * @param {{ region: string, userPoolId: string, handlerArn: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").UpdateUserPoolCommandOutput | null, unknown]>}
 */
export const addPreSignUpHandler = async ({
  region,
  userPoolId,
  handlerArn,
}) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({
      region,
    });

    const command = new UpdateUserPoolCommand({
      UserPoolId: userPoolId,
      LambdaConfig: {
        PreSignUp: handlerArn,
      },
    });

    const response = await cognitoClient.send(command);
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Attempt to register a user to a user pool with a given username and password.
 * @param {{
 *   region: string,
 *   userPoolClientId: string,
 *   username: string,
 *   email: string,
```

```
 *   password: string
 * }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").SignUpCommandOutput | null, unknown]>}
 */
export const signUpUser = async ({
  region,
  userPoolClientId,
  username,
  email,
  password,
}) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({
      region,
    });

    const response = await cognitoClient.send(
      new SignUpCommand({
        ClientId: userPoolClientId,
        Username: username,
        Password: password,
        UserAttributes: [{ Name: "email", Value: email }],
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Sign in a user to Amazon Cognito using a username and password authentication
 flow.
 * @param {{ region: string, clientId: string, username: string, password:
 string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").InitiateAuthCommandOutput | null, unknown]>}
 */
export const signIn = async ({ region, clientId, username, password }) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({ region });
    const response = await cognitoClient.send(
      new InitiateAuthCommand({
```

```
        AuthFlow: "USER_PASSWORD_AUTH",
        ClientId: clientId,
        AuthParameters: { USERNAME: username, PASSWORD: password },
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Retrieve an existing user from a user pool.
 * @param {{ region: string, userPoolId: string, username: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").AdminGetUserCommandOutput | null, unknown]>}
 */
export const getUser = async ({ region, userPoolId, username }) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({ region });
    const response = await cognitoClient.send(
      new AdminGetUserCommand({
        UserPoolId: userPoolId,
        Username: username,
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Delete the signed-in user. Useful for allowing a user to delete their
 * own profile.
 * @param {{ region: string, accessToken: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").DeleteUserCommandOutput | null, unknown]>}
 */
export const deleteUser = async ({ region, accessToken }) => {
  try {
    const client = new CognitoIdentityProviderClient({ region });
    const response = await client.send(
      new DeleteUserCommand({ AccessToken: accessToken }),
```

```
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};
```

Module of DynamoDB actions.

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

/**
 * Populate a DynamoDB table with provide items.
 * @param {{ region: string, tableName: string, items: Record<string,
 unknown>[] }} config
 * @returns {Promise<[import("@aws-sdk/lib-dynamodb").BatchWriteCommandOutput |
 null, unknown]>}
 */
export const populateTable = async ({ region, tableName, items }) => {
  try {
    const ddbClient = new DynamoDBClient({ region });
    const docClient = DynamoDBDocumentClient.from(ddbClient);
    const response = await docClient.send(
      new BatchWriteCommand({
        RequestItems: {
          [tableName]: items.map((item) => ({
            PutRequest: {
              Item: item,
            },
          })),
        },
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
```

```
};
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Automatically migrate known Amazon Cognito users with a Lambda function using an AWS SDK

The following code example shows how to automatically migrate known Amazon Cognito users with a Lambda function.

- Configure a user pool to call a Lambda function for the `MigrateUser` trigger.
- Sign in to Amazon Cognito with a username and email that is not in the user pool.
- The Lambda function scans a DynamoDB table and automatically migrates known users to the user pool.
- Perform the forgot password flow to reset the password for the migrated user.
- Sign in as the new user, then clean up resources.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario at a command prompt.

```
import (
 "context"
 "errors"
 "fmt"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// MigrateUser separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type MigrateUser struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewMigrateUser constructs a new migrate user runner.
func NewMigrateUser(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) MigrateUser {
 scenario := MigrateUser{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddMigrateUserTrigger adds a Lambda handler as an invocation target for the
 MigrateUser trigger.
```

```
func (runner *MigrateUser) AddMigrateUserTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the MigrateUser trigger from
 Cognito.\n" +
  "This trigger happens when an unknown user signs in, and lets your function
 take action before Cognito\n" +
  "rejects the user.\n\n")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.UserMigration, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the MigrateUser
 trigger.\n",
  functionArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser adds a new user to the known users table and signs that user in to
 Amazon Cognito.
func (runner *MigrateUser) SignInUser(ctx context.Context, usersTable string,
 clientId string) (bool, actions.User) {
 log.Println("Let's sign in a user to your Cognito user pool. When the username
 and email matches an entry in the\n" +
  "DynamoDB known users table, the email is automatically verified and the user
 is migrated to the Cognito user pool.")

 user := actions.User{}
 user.UserName = runner.questioner.Ask("\nEnter a username:")
 user.UserEmail = runner.questioner.Ask("\nEnter an email that you own. This
 email will be used to confirm user migration\n" +
  "during this example:")

 runner.helper.AddKnownUser(ctx, usersTable, user)

 var err error
 var resetRequired *types.PasswordResetRequiredException
 var authResult *types.AuthenticationResultType
 signedIn := false
 for !signedIn && resetRequired == nil {
```

```
  log.Printf("Signing in to Cognito as user '%v'. The expected result is a
 PasswordResetRequiredException.\n\n", user.UserName)
  authResult, err = runner.cognitoActor.SignIn(ctx, clientId, user.UserName, "_")
  if err != nil {
   if errors.As(err, &resetRequired) {
    log.Printf("\nUser '%v' is not in the Cognito user pool but was found in the
 DynamoDB known users table.\n"+
     "User migration is started and a password reset is required.",
 user.UserName)
   } else {
    panic(err)
   }
  } else {
   log.Printf("User '%v' successfully signed in. This is unexpected and probably
 means you have not\n"+
    "cleaned up a previous run of this scenario, so the user exist in the Cognito
 user pool.\n"+
    "You can continue this example and select to clean up resources, or manually
 remove\n"+
    "the user from your user pool and try again.", user.UserName)
   runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
   signedIn = true
  }
 }

 log.Println(strings.Repeat("-", 88))
 return resetRequired != nil, user
}

// ResetPassword starts a password recovery flow.
func (runner *MigrateUser) ResetPassword(ctx context.Context, clientId string,
 user actions.User) {
 wantCode := runner.questioner.AskBool(fmt.Sprintf("In order to migrate the user
 to Cognito, you must be able to receive a confirmation\n"+
  "code by email at %v. Do you want to send a code (y/n)?", user.UserEmail), "y")
 if !wantCode {
  log.Println("To complete this example and successfully migrate a user to
 Cognito, you must enter an email\n" +
   "you own that can receive a confirmation code.")
  return
 }
 codeDelivery, err := runner.cognitoActor.ForgotPassword(ctx, clientId,
 user.UserName)
```

```
 if err != nil {
  panic(err)
 }
 log.Printf("\nA confirmation code has been sent to %v.",
 *codeDelivery.Destination)
 code := runner.questioner.Ask("Check your email and enter it here:")

 confirmed := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !confirmed {
  log.Printf("\nConfirming password reset for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.ConfirmForgotPassword(ctx, clientId, code,
 user.UserName, password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   confirmed = true
  }
 }
 log.Printf("User '%v' successfully confirmed and migrated.\n", user.UserName)
 log.Println("Signing in with your username and password...")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, user.UserName,
 password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)

 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *MigrateUser) Run(ctx context.Context, stackName string) {
 defer func() {
```

```
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]

 runner.AddMigrateUserTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["MigrateUserFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers,
 actions.UserMigration)
 resetNeeded, user := runner.SignInUser(ctx, stackOutputs["TableName"],
 stackOutputs["UserPoolClientId"])
 if resetNeeded {
  runner.helper.ListRecentLogEvents(ctx, stackOutputs["MigrateUserFunction"])
  runner.ResetPassword(ctx, stackOutputs["UserPoolClientId"], user)
 }

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Handle the `MigrateUser` trigger with a Lambda function.

```
import (
 "context"
 "log"
```

```
 "os"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
)

const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the MigrateUser event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be migrated to the user pool.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsMigrateUser)
 (events.CognitoEventUserPoolsMigrateUser, error) {
 log.Printf("Received migrate trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 if event.TriggerSource != "UserMigration_Authentication" {
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName: event.UserName,
 }
 log.Printf("Looking up user '%v' in table %v.\n", user.UserName, tableName)
 filterEx := expression.Name("UserName").Equal(expression.Value(user.UserName))
 expr, err := expression.NewBuilder().WithFilter(filterEx).Build()
 if err != nil {
```

```
  log.Printf("Error building expression to query for user '%v'.\n",
 user.UserName)
  return event, err
 }
 output, err := h.dynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName:                 aws.String(tableName),
  FilterExpression:          expr.Filter(),
  ExpressionAttributeNames:  expr.Names(),
  ExpressionAttributeValues: expr.Values(),
 })
 if err != nil {
  log.Printf("Error looking up user '%v'.\n", user.UserName)
  return event, err
 }
 if len(output.Items) == 0 {
  log.Printf("User '%v' not found, not migrating user.\n", user.UserName)
  return event, err
 }

 var users []UserInfo
 err = attributevalue.UnmarshalListOfMaps(output.Items, &users)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB items. Here's why: %v\n", err)
  return event, err
 }

 user = users[0]
 log.Printf("UserName '%v' found with email %v. User is migrated and must reset
 password.\n", user.UserName, user.UserEmail)
 event.CognitoEventUserPoolsMigrateUserResponse.UserAttributes =
 map[string]string{
  "email":          user.UserEmail,
  "email_verified": "true", // email_verified is required for the forgot password
 flow.
 }
 event.CognitoEventUserPoolsMigrateUserResponse.FinalUserStatus =
 "RESET_REQUIRED"
 event.CognitoEventUserPoolsMigrateUserResponse.MessageAction = "SUPPRESS"

 return event, err
}

func main() {
 ctx := context.Background()
```

```
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

Create a struct that performs common tasks.

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
```

```
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
```

```
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
```

```
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Create a struct that wraps Amazon Cognito actions.

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
```

```
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
```

```
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
```

```
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
```

```
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}
```

```
// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

Create a struct that wraps DynamoDB actions.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)
```

```
// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
```

```
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
```

```
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

Create a struct that wraps CloudWatch Logs actions.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
```

```
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Create a struct that wraps CloudFormation actions.

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)
```

```
// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

Clean up resources.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
```

```
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
```

```
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.
  - ConfirmForgotPassword
  - DeleteUser
  - ForgotPassword
  - InitiateAuth
  - SignUp
  - UpdateUserPool

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create an API Gateway REST API to track COVID-19 data

The following code example shows how to create a REST API that simulates a system to track daily cases of COVID-19 in the United States, using fictional data.

Python

SDK for Python (Boto3)

Shows how to use AWS Chalice with the AWS SDK for Python (Boto3) to create a serverless REST API that uses Amazon API Gateway, AWS Lambda, and Amazon DynamoDB. The REST

API simulates a system that tracks daily cases of COVID-19 in the United States, using fictional data. Learn how to:

- Use AWS Chalice to define routes in Lambda functions that are called to handle REST requests that come through API Gateway.
- Use Lambda functions to retrieve and store data in a DynamoDB table to serve REST requests.
- Define table structure and security role resources in an AWS CloudFormation template.
- Use AWS Chalice and CloudFormation to package and deploy all necessary resources.
- Use CloudFormation to clean up all created resources.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- CloudFormation
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a lending library REST API

The following code example shows how to create a lending library where patrons can borrow and return books by using a REST API backed by an Amazon Aurora database.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with the Amazon Relational Database Service (Amazon RDS) API and AWS Chalice to create a REST API backed by an Amazon Aurora database. The web service is fully serverless and represents a simple lending library where patrons can borrow and return books. Learn how to:

- Create and manage a serverless Aurora database cluster.

- Use AWS Secrets Manager to manage database credentials.
- Implement a data storage layer that uses Amazon RDS to move data into and out of the database.
- Use AWS Chalice to deploy a serverless REST API to Amazon API Gateway and AWS Lambda.
- Use the Requests package to send requests to the web service.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- Aurora
- Lambda
- Secrets Manager

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a messenger application with Step Functions

The following code example shows how to create an AWS Step Functions messenger application that retrieves message records from a database table.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with AWS Step Functions to create a messenger application that retrieves message records from an Amazon DynamoDB table and sends them with Amazon Simple Queue Service (Amazon SQS). The state machine integrates with an AWS Lambda function to scan the database for unsent messages.

- Create a state machine that retrieves and updates message records from an Amazon DynamoDB table.
- Update the state machine definition to also send messages to Amazon Simple Queue Service (Amazon SQS).

- Start and stop state machine runs.
- Connect to Lambda, DynamoDB, and Amazon SQS from a state machine by using service integrations.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Lambda
- Amazon SQS
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a photo asset management application that lets users manage photos using labels

The following code examples show how to create a serverless application that lets users manage photos using labels.

.NET

SDK for .NET

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway

- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

SDK for C++

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

SDK for Java 2.x

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK for Kotlin

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

SDK for PHP

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Rust

SDK for Rust

Shows how to develop a photo asset management application that detects labels in images using Amazon Rekognition and stores them for later retrieval.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

For a deep dive into the origin of this example see the post on AWS Community.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create a websocket chat application with API Gateway

The following code example shows how to create a chat application that is served by a websocket API built on Amazon API Gateway.

Python

SDK for Python (Boto3)

Shows how to use the AWS SDK for Python (Boto3) with Amazon API Gateway V2 to create a websocket API that integrates with AWS Lambda and Amazon DynamoDB.

- Create a websocket API served by API Gateway.
- Define a Lambda handler that stores connections in DynamoDB and posts messages to other chat participants.
- Connect to the websocket chat application and send messages with the Websockets package.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Create an application that analyzes customer feedback and synthesizes audio

The following code examples show how to create an application that analyzes customer comment cards, translates them from their original language, determines their sentiment, and generates an audio file from the translated text.

.NET

SDK for .NET

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in GitHub.

Services used in this example

- Amazon Comprehend
- Lambda

- Amazon Polly
- Amazon Textract
- Amazon Translate

Java

SDK for Java 2.x

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in GitHub.

Services used in this example

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

SDK for JavaScript (v3)

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into

the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in GitHub. The following excerpts show how the AWS SDK for JavaScript is used inside of Lambda functions.

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string}} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });
```

```
  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
 eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
  // Each block also contains geometry of the detected text.
  // For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
  const { Blocks } = await textractClient.send(detectDocumentTextCommand);

  // For the purpose of this example, we are only interested in words.
  const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
    (b) => b.Text,
  );
```

```
  return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
 *
 * @param {{ bucket: string, translated_text: string, object: string}}
 sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
 * @param {{ extracted_text: string, source_language_code: string}}
 textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

Services used in this example

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

SDK for Ruby

This example application analyzes and stores customer feedback cards. Specifically, it fulfills the need of a fictitious hotel in New York City. The hotel receives feedback from guests in various languages in the form of physical comment cards. That feedback is uploaded into

the app through a web client. After an image of a comment card is uploaded, the following steps occur:

- Text is extracted from the image using Amazon Textract.
- Amazon Comprehend determines the sentiment of the extracted text and its language.
- The extracted text is translated to English using Amazon Translate.
- Amazon Polly synthesizes an audio file from the extracted text.

The full app can be deployed with the AWS CDK. For source code and deployment instructions, see the project in GitHub.

Services used in this example

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from a browser

The following code example shows how to invoke an AWS Lambda function from a browser.

JavaScript

SDK for JavaScript (v2)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB

- Lambda

SDK for JavaScript (v3)

You can create a browser-based application that uses an AWS Lambda function to update an Amazon DynamoDB table with user selections. This app uses AWS SDK for JavaScript v3.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Transform data for your application with S3 Object Lambda

The following code example shows how to transform data for your application with S3 Object Lambda.

.NET

SDK for .NET

Shows how to add custom code to standard S3 GET requests to modify the requested object retrieved from S3 so that the object suit the needs of the requesting client or application.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Lambda
- Amazon S3

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use API Gateway to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by Amazon API Gateway.

Java

SDK for Java 2.x

Shows how to create an AWS Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an AWS Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create a Lambda function invoked by Amazon API Gateway that scans an Amazon DynamoDB table for work anniversaries and uses Amazon Simple Notification Service (Amazon SNS) to send a text message to your employees that congratulates them at their one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

This example is also available in the AWS SDK for JavaScript v3 developer guide.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

This example shows how to create and use an Amazon API Gateway REST API that targets an AWS Lambda function. The Lambda handler demonstrates how to route based on HTTP methods; how to get data from the query string, header, and body; and how to return a JSON response.

- Deploy a Lambda function.
- Create an API Gateway REST API.
- Create a REST resource that targets the Lambda function.
- Grant permission to let API Gateway invoke the Lambda function.
- Use the Requests package to send requests to the REST API.
- Clean up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use Step Functions to invoke Lambda functions

The following code example shows how to create an AWS Step Functions state machine that invokes AWS Lambda functions in sequence.

Java

SDK for Java 2.x

Shows how to create an AWS serverless workflow by using AWS Step Functions and the AWS SDK for Java 2.x. Each workflow step is implemented using an AWS Lambda function.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- DynamoDB
- Lambda
- Amazon SES
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use scheduled events to invoke a Lambda function

The following code examples show how to create an AWS Lambda function invoked by an Amazon EventBridge scheduled event.

Java

SDK for Java 2.x

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda Java runtime API. This example invokes different AWS services to perform a specific use case.

This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

SDK for JavaScript (v3)

Shows how to create an Amazon EventBridge scheduled event that invokes an AWS Lambda function. Configure EventBridge to use a cron expression to schedule when the Lambda function is invoked. In this example, you create a Lambda function by using the Lambda JavaScript runtime API. This example invokes different AWS services to perform a specific use case. This example demonstrates how to create an app that sends a mobile text message to your employees that congratulates them at the one year anniversary date.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

This example is also available in the AWS SDK for JavaScript v3 developer guide.

Services used in this example

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

SDK for Python (Boto3)

This example shows how to register an AWS Lambda function as the target of a scheduled Amazon EventBridge event. The Lambda handler writes a friendly message and the full event data to Amazon CloudWatch Logs for later retrieval.

- Deploys a Lambda function.
- Creates an EventBridge scheduled event and makes the Lambda function the target.
- Grants permission to let EventBridge invoke the Lambda function.
- Prints the latest data from CloudWatch Logs to show the result of the scheduled invocations.
- Cleans up all resources created during the demo.

This example is best viewed on GitHub. For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use the Amazon Neptune API to develop a Lambda function that queries graph data

The following code example shows how to use the Neptune API to query graph data.

Java

SDK for Java 2.x

Shows how to use Amazon Neptune Java API to create a Lambda function that queries graph data within the VPC.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- Lambda
- Neptune

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Write custom activity data with a Lambda function after Amazon Cognito user authentication using an AWS SDK

The following code example shows how to write custom activity data with a Lambda function after Amazon Cognito user authentication.

- Use administrator functions to add a user to a user pool.
- Configure a user pool to call a Lambda function for the `PostAuthentication` trigger.
- Sign the new user in to Amazon Cognito.
- The Lambda function writes custom information to CloudWatch Logs and to an DynamoDB table.
- Get and display custom data from the DynamoDB table, then clean up resources.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Run an interactive scenario at a command prompt.

```
import (
 "context"
 "errors"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// ActivityLog separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type ActivityLog struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewActivityLog constructs a new activity log runner.
func NewActivityLog(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) ActivityLog {
 scenario := ActivityLog{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
```

```
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddUserToPool selects a user from the known users table and uses administrator
 credentials to add the user to the user pool.
func (runner *ActivityLog) AddUserToPool(ctx context.Context, userPoolId string,
 tableName string) (string, string) {
 log.Println("To facilitate this example, let's add a user to the user pool using
 administrator privileges.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 user := users.Users[0]
 log.Printf("Adding known user %v to the user pool.\n", user.UserName)
 err = runner.cognitoActor.AdminCreateUser(ctx, userPoolId, user.UserName,
 user.UserEmail)
 if err != nil {
  panic(err)
 }
 pwSet := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !pwSet {
  log.Printf("\nSetting password for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.AdminSetUserPassword(ctx, userPoolId, user.UserName,
 password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   pwSet = true
  }
 }
```

```
 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// AddActivityLogTrigger adds a Lambda handler as an invocation target for the
 PostAuthentication trigger.
func (runner *ActivityLog) AddActivityLogTrigger(ctx context.Context, userPoolId
 string, activityLogArn string) {
 log.Println("Let's add a Lambda function to handle the PostAuthentication
 trigger from Cognito.\n" +
  "This trigger happens after a user is authenticated, and lets your function
 take action, such as logging\n" +
  "the outcome.")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PostAuthentication, HandlerArn:
 aws.String(activityLogArn)})
 if err != nil {
  panic(err)
 }
 runner.resources.triggers = append(runner.resources.triggers,
 actions.PostAuthentication)
 log.Printf("Lambda function %v added to user pool %v to handle
 PostAuthentication Cognito trigger.\n",
  activityLogArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser signs in as the specified user.
func (runner *ActivityLog) SignInUser(ctx context.Context, clientId string,
 userName string, password string) {
 log.Printf("Now we'll sign in user %v and check the results in the logs and the
 DynamoDB table.", userName)
 runner.questioner.Ask("Press Enter when you're ready.")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Println("Sign in successful.",
  "The PostAuthentication Lambda handler writes custom information to CloudWatch
 Logs.")
```

```
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
}

// GetKnownUserLastLogin gets the login info for a user from the Amazon DynamoDB
 table and displays it.
func (runner *ActivityLog) GetKnownUserLastLogin(ctx context.Context, tableName
 string, userName string) {
 log.Println("The PostAuthentication handler also writes login data to the
 DynamoDB table.")
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 for _, user := range users.Users {
  if user.UserName == userName {
   log.Println("The last login info for the user in the known users table is:")
   log.Printf("\t%+v", *user.LastLogin)
  }
 }
 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *ActivityLog) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])
```

```
 userName, password := runner.AddUserToPool(ctx, stackOutputs["UserPoolId"],
 stackOutputs["TableName"])

 runner.AddActivityLogTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["ActivityLogFunctionArn"])
 runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password)
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["ActivityLogFunction"])
 runner.GetKnownUserLastLogin(ctx, stackOutputs["TableName"], userName)

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

Handle the `PostAuthentication` trigger with a Lambda function.

```
import (
 "context"
 "fmt"
 "log"
 "os"
 "time"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 dynamodbtypes "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

const TABLE_NAME = "TABLE_NAME"

// LoginInfo defines structured login data that can be marshalled to a DynamoDB
 format.
type LoginInfo struct {
 UserPoolId string `dynamodbav:"UserPoolId"`
```

```
 ClientId   string `dynamodbav:"ClientId"`
 Time       string `dynamodbav:"Time"`
}

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string    `dynamodbav:"UserName"`
 UserEmail string    `dynamodbav:"UserEmail"`
 LastLogin LoginInfo `dynamodbav:"LastLogin"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PostAuthentication event by writing custom data to
 the logs and
// to an Amazon DynamoDB table.
func (h *handler) HandleRequest(ctx context.Context,
 event events.CognitoEventUserPoolsPostAuthentication)
 (events.CognitoEventUserPoolsPostAuthentication, error) {
 log.Printf("Received post authentication trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName:  event.UserName,
  UserEmail: event.Request.UserAttributes["email"],
  LastLogin: LoginInfo{
   UserPoolId: event.UserPoolID,
   ClientId:   event.CallerContext.ClientID,
   Time:       time.Now().Format(time.UnixDate),
  },
 }
 // Write to CloudWatch Logs.
```

```
 fmt.Printf("%#v", user)

 // Also write to an external system. This examples uses DynamoDB to demonstrate.
 userMap, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshal to DynamoDB map. Here's why: %v\n", err)
 } else if len(userMap) == 0 {
  log.Printf("User info marshaled to an empty map.")
 } else {
  _, err := h.dynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
   Item:      userMap,
   TableName: aws.String(tableName),
  })
  if err != nil {
   log.Printf("Couldn't write to DynamoDB. Here's why: %v\n", err)
  } else {
   log.Printf("Wrote user info to DynamoDB table %v.\n", tableName)
  }
 }

 return event, nil
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

Create a struct that performs common tasks.

```
import (
 "context"
 "log"
```

```
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
```

```
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
```

```
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

Create a struct that wraps Amazon Cognito actions.

```
import (
 "context"
 "errors"
 "log"
```

```
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
```

```
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
```

```
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
```

```
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}
```

```
// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
```

```
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

Create a struct that wraps DynamoDB actions.

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
```

```
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}
```

```
// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

Create a struct that wraps CloudWatch Logs actions.

```
import (
 "context"
```

```
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
```

```
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

Create a struct that wraps CloudFormation actions.

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
```

```
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

Clean up resources.

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
```

```
    log.Printf("Something went wrong during cleanup.\n%v\n", r)
    log.Println("Use the AWS Management Console to remove any remaining resources
  \n" +
     "that were created for this scenario.")
   }
  }()

  wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
  resources that were created "+
   "during this demo (y/n)?", "y")
  if wantDelete {
   for _, accessToken := range resources.userAccessTokens {
    err := resources.cognitoActor.DeleteUser(ctx, accessToken)
    if err != nil {
     log.Println("Couldn't delete user during cleanup.")
     panic(err)
    }
    log.Println("Deleted user.")
   }
   triggerList := make([]actions.TriggerInfo, len(resources.triggers))
   for i := 0; i < len(resources.triggers); i++ {
    triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
  HandlerArn: nil}
   }
   err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
  triggerList...)
   if err != nil {
    log.Println("Couldn't update Cognito triggers during cleanup.")
    panic(err)
   }
   log.Println("Removed Cognito triggers from user pool.")
  } else {
   log.Println("Be sure to remove resources when you're done with them to avoid
  unexpected charges!")
  }
}
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.
  - AdminCreateUser
  - AdminSetUserPassword

- DeleteUser
- InitiateAuth
- UpdateUserPool

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Serverless examples for Lambda

The following code examples show how to use Lambda with AWS SDKs.

Examples

- Connecting to an Amazon RDS database in a Lambda function
- Invoke a Lambda function from a Kinesis trigger
- Invoke a Lambda function from a DynamoDB trigger
- Invoke a Lambda function from a Amazon DocumentDB trigger
- Invoke a Lambda function from an Amazon MSK trigger
- Invoke a Lambda function from an Amazon S3 trigger
- Invoke a Lambda function from an Amazon SNS trigger
- Invoke a Lambda function from an Amazon SQS trigger
- Reporting batch item failures for Lambda functions with a Kinesis trigger
- Reporting batch item failures for Lambda functions with a DynamoDB trigger
- Reporting batch item failures for Lambda functions with an Amazon SQS trigger

Connecting to an Amazon RDS database in a Lambda function

The following code examples show how to implement a Lambda function that connects to an RDS database. The function makes a simple database request and returns the result.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using .NET.

```
using System.Data;
using System.Text.Json;
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Core;
using MySql.Data.MySqlClient;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace aws_rds;

public class InputModel
{
    public string key1 { get; set; }
    public string key2 { get; set; }
}

public class Function
{
    /// <summary>
    // Handles the Lambda function execution for connecting to RDS using IAM
 authentication.
    /// </summary>
    /// <param name="input">The input event data passed to the Lambda function</
param>
    /// <param name="context">The Lambda execution context that provides runtime
 information</param>
    /// <returns>A response object containing the execution result</returns>
```

```
   public async Task<APIGatewayProxyResponse>
FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
   {
       // Sample Input: {"body": "{\"key1\":\"20\", \"key2\":\"25\"}"}
       var input = JsonSerializer.Deserialize<InputModel>(request.Body);

       /// Obtain authentication token
       var authToken = RDSAuthTokenGenerator.GenerateAuthToken(
           Environment.GetEnvironmentVariable("RDS_ENDPOINT"),
           Convert.ToInt32(Environment.GetEnvironmentVariable("RDS_PORT")),
           Environment.GetEnvironmentVariable("RDS_USERNAME")
       );

       /// Build the Connection String with the Token
       string connectionString =
$"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +

$"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

$"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                                  $"Pwd={authToken};";


       try
       {
           await using var connection = new MySqlConnection(connectionString);
           await connection.OpenAsync();

           const string sql = "SELECT @param1 + @param2 AS Sum";

           await using var command = new MySqlCommand(sql, connection);
           command.Parameters.AddWithValue("@param1", int.Parse(input.key1 ??
"0"));
           command.Parameters.AddWithValue("@param2", int.Parse(input.key2 ??
"0"));

           await using var reader = await command.ExecuteReaderAsync();
           if (await reader.ReadAsync())
           {
               int result = reader.GetInt32("Sum");

               //Sample Response: {"statusCode":200,"body":"{\"message\":\"The
sum is: 45\"}","isBase64Encoded":false}
               return new APIGatewayProxyResponse
```

```
                {
                    StatusCode = 200,
                    Body = JsonSerializer.Serialize(new { message = $"The sum is:
 {result}" })
                };
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        return new APIGatewayProxyResponse
        {
            StatusCode = 500,
            Body = JsonSerializer.Serialize(new { error = "Internal server
 error" })
        };
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Go.

```
/*
Golang v2 code here.
*/

package main

import (
```

```
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }
```

```
 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Java.

```
import com.amazonaws.services.lambda.runtime.Context;
```

```
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();

        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);
```

```
                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }

        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
```

```
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;

}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

Connecting to an Amazon RDS database in a Lambda function using TypeScript.

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!
```

```
async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
```

```
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using PHP.

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
```

```
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
            'region' => getenv('AWS_REGION'),
        ];

        // Create RDS AuthTokenGenerator object
        $generator = new
AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
```

```
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }

    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Python.

```
import json
import os
import boto3
import pymysql

# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result
```

```
    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Ruby.

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
```

```
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )

  begin
    conn = Mysql2::Client.new(
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Connecting to an Amazon RDS database in a Lambda function using Rust.

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;
```

```
    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
        db_user = db_username
    );

    let signable_request =
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
```

```
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from a Kinesis trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving records from a Kinesis stream. The function retrieves the Kinesis payload, decodes from Base64, and logs the record contents.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
```

```
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Consuming a Kinesis event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
```

```
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
```

```
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Kinesis event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
```

```
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Kinesis event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
```

```
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from a DynamoDB trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving records from a DynamoDB stream. The function retrieves the DynamoDB payload and logs the record contents.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
```

```
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
```

```
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

Consuming a DynamoDB event with Lambda using TypeScript.

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
```

```
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
```

```
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Python.

```
import json
```

```
def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Ruby.

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a DynamoDB event with Lambda using Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");
```

```
    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from a Amazon DocumentDB trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving records from a DocumentDB change stream. The function retrieves the DocumentDB payload and logs the record contents.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using .NET.

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaDocDb;

public class Function
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }
```

```
    private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
ILambdaContext context)
   {

       var eventData = record.Event;
       var operationType = eventData.OperationType;
       var databaseName = eventData.Ns.Db;
       var collectionName = eventData.Ns.Coll;
       var fullDocument = JsonSerializer.Serialize(eventData.FullDocument, new
JsonSerializerOptions { WriteIndented = true });

       context.Logger.LogLine($"Operation type: {operationType}");
       context.Logger.LogLine($"Database: {databaseName}");
       context.Logger.LogLine($"Collection: {collectionName}");
       context.Logger.LogLine($"Full document:\n{fullDocument}");
   }



   public class Event
   {
       [JsonPropertyName("eventSourceArn")]
       public string EventSourceArn { get; set; }

       [JsonPropertyName("events")]
       public List<DocumentDBEventRecord> Events { get; set; }

       [JsonPropertyName("eventSource")]
       public string EventSource { get; set; }
   }

   public class DocumentDBEventRecord
   {
       [JsonPropertyName("event")]
       public EventData Event { get; set; }
   }

   public class EventData
   {
       [JsonPropertyName("_id")]
       public IdData Id { get; set; }

       [JsonPropertyName("clusterTime")]
```

```
        public ClusterTime ClusterTime { get; set; }

        [JsonPropertyName("documentKey")]
        public DocumentKey DocumentKey { get; set; }

        [JsonPropertyName("fullDocument")]
        public Dictionary<string, object> FullDocument { get; set; }

        [JsonPropertyName("ns")]
        public Namespace Ns { get; set; }

        [JsonPropertyName("operationType")]
        public string OperationType { get; set; }
    }

    public class IdData
    {
        [JsonPropertyName("_data")]
        public string Data { get; set; }
    }

    public class ClusterTime
    {
        [JsonPropertyName("$timestamp")]
        public Timestamp Timestamp { get; set; }
    }

    public class Timestamp
    {
        [JsonPropertyName("t")]
        public long T { get; set; }

        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }

    public class Id
    {
```

```
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Go.

```
package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}
```

```
type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Java.

```
import java.util.List;
import java.util.Map;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Example implements RequestHandler<Map<String, Object>, String> {

    @SuppressWarnings("unchecked")
    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        List<Map<String, Object>> events = (List<Map<String, Object>>)
 event.get("events");
        for (Map<String, Object> record : events) {
            Map<String, Object> eventData = (Map<String, Object>)
 record.get("event");
            processEventData(eventData);
        }

        return "OK";
    }

    @SuppressWarnings("unchecked")
    private void processEventData(Map<String, Object> eventData) {
        String operationType = (String) eventData.get("operationType");
        System.out.println("operationType: %s".formatted(operationType));

        Map<String, Object> ns = (Map<String, Object>) eventData.get("ns");

        String db = (String) ns.get("db");
        System.out.println("db: %s".formatted(db));
        String coll = (String) ns.get("coll");
        System.out.println("coll: %s".formatted(coll));

        Map<String, Object> fullDocument = (Map<String, Object>)
 eventData.get("fullDocument");
        System.out.println("fullDocument: %s".formatted(fullDocument));
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using JavaScript.

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

Consuming a Amazon DocumentDB event with Lambda using TypeScript

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from 'aws-
lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
```

```
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using PHP.

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
```

```
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];

        // Log the event details

        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument, JSON_PRETTY_PRINT) .
 "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Python.

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'
```

```
def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Ruby.

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
```

```
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming a Amazon DocumentDB event with Lambda using Rust.

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
```

```
        return Ok(());
    }

    for record in records{
        log_document_db_event(record);
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from an Amazon MSK trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving records from an Amazon MSK cluster. The function retrieves the MSK payload and logs the record contents.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using .NET.

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {
```

```
        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);

                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Go.

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
```

```
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)

   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Java.

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
```

```
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
                System.out.println("Record: " + record);

                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using JavaScript.

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
```

```
      })
    }
}
```

Consuming an Amazon MSK event with Lambda using TypeScript.

```
import { MSKEvent, Context } from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "msk-handler-sample",
});

export const handler = async (
  event: MSKEvent,
  context: Context
): Promise<void> => {
  for (const [topic, topicRecords] of Object.entries(event.records)) {
    logger.info(`Processing key: ${topic}`);

    // Process each record in the partition
    for (const record of topicRecords) {
      try {
        // Decode the message value from base64
        const decodedMessage = Buffer.from(record.value, 'base64').toString();

        logger.info({
          message: decodedMessage
        });
      }
      catch (error) {
        logger.error('Error processing event', { error });
        throw error;
      }
    };
  }
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using PHP.

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();
```

```
        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Python.

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
```

```
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Ruby.

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an Amazon MSK event with Lambda using Rust.

```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// Pre-Requisites:
/// 1. Install Cargo Lambda - see https://www.cargo-lambda.info/guide/getting-
started.html
/// 2. Add packages tracing, tracing-subscriber, serde_json, base64
///
/// This is the main body for the function.
/// Write your code inside it.
/// There are some code example in the following URLs:
/// - https://github.com/awslabs/aws-lambda-rust-runtime/tree/main/examples
/// - https://github.com/aws-samples/serverless-rust-demo/

async fn function_handler(event: LambdaEvent<KafkaEvent>) -> Result<Value, Error>
 {

    let payload = event.payload.records;

    for (_name, records) in payload.iter() {

        for record in records {

         let record_text = record.value.as_ref().ok_or("Value is None")?;
         info!("Record: {}", &record_text);

         // perform Base64 decoding
         let record_bytes = BASE64_STANDARD.decode(record_text)?;
```

```
            let message = std::str::from_utf8(&record_bytes)?;

            info!("Message: {}", message);
          }

    }

    Ok(().into())
}

#[tokio::main]
async fn main() -> Result<(), Error> {

    // required to enable CloudWatch error logging by the runtime
    tracing::init_default_subscriber();
    info!("Setup CW subscriber!");

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from an Amazon S3 trigger

The following code examples show how to implement a Lambda function that receives an event triggered by uploading an object to an S3 bucket. The function retrieves the S3 bucket name and object key from the event parameter and calls the Amazon S3 API to retrieve and log the content type of the object.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");
```

```
                var objectResult = await _s3Client.GetObjectAsync(bucket, key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
```

```
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotifi

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey + " of
 type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket,
 String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
```

```
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using JavaScript.

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g,
 ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
```

```
        const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

Consuming an S3 event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
 '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure
 they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using PHP.

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
```

```
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as this
 function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
```

```
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and
your bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Ruby.

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
```

```
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']['key'],
 Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they exist
 and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an S3 event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();
```

```
    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request from
 SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket
 name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to
 exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
```

```
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from an Amazon SNS trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving messages from an SNS topic. The function retrieves the messages from the event parameter and logs the content of each message.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
```

```
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;

    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SNS event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
```

```
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
```

```
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here

    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
```

```
  raise
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SNS event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);
```

```
    // Implement your record handling code here.

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke a Lambda function from an Amazon SQS trigger

The following code examples show how to implement a Lambda function that receives an event triggered by receiving messages from an SQS queue. The function retrieves the messages from the event parameter and logs the content of each message.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
```

```
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
```

```
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using JavaScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

Consuming an SQS event with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";
```

```
export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
```

```
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
```

```
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Consuming an SQS event with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
```

```
        .init();

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Reporting batch item failures for Lambda functions with a Kinesis trigger

The following code examples show how to implement partial batch response for Lambda functions that receive events from a Kinesis stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
```

```
namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
```

```
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Javascript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

Reporting Kinesis batch item failures with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting Kinesis batch item failures with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Reporting batch item failures for Lambda functions with a DynamoDB trigger

The following code examples show how to implement partial batch response for Lambda functions that receive events from a DynamoDB stream. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;
```

```
// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
```

```
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }
```

```
 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {
```

```
    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using JavaScript.

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

Reporting DynamoDB batch item failures with Lambda using TypeScript.

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
```

```
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using PHP.

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
```

```
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Ruby.

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting DynamoDB batch item failures with Lambda using Rust.

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;
```

```
    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }
```

```
    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Reporting batch item failures for Lambda functions with an Amazon SQS trigger

The following code examples show how to implement partial batch response for Lambda functions that receive events from an SQS queue. The function reports the batch item failures in the response, signaling to Lambda to retry those messages later.

.NET

SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using .NET.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
```

```
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Go.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {
  if len(message.Body) > 0 {
   // Your message processing condition here
   fmt.Printf("Successfully processed message: %s\n", message.Body)
  } else {
   // Message processing failed
   fmt.Printf("Failed to process message %s\n", message.MessageId)
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
```

```
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Java.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {
         List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();
```

```
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(message.getMessageId()));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using JavaScript.

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
```

```
    }
    console.log(`Processed message: ${record.body}`);
}
```

Reporting SQS batch item failures with Lambda using TypeScript.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord }
 from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using PHP.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
```

```
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Python.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                print(f"Processed message: {record['body']}")
            except Exception as e:
```

```
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK for Ruby

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Ruby.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

SDK for Rust

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the Serverless examples repository.

Reporting SQS batch item failures with Lambda using Rust.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
```

```
    run(service_fn(function_handler)).await
}
```

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

AWS community contributions for Lambda

AWS community contributions are examples that were created and are maintained by multiple teams across AWS. To provide feedback, use the mechanism provided in the linked repositories.

Examples

- Build and test a serverless application

Build and test a serverless application

The following code examples show how to build and test a serverless application using API Gateway with Lambda and DynamoDB

.NET

SDK for .NET

Shows how to build and test a serverless application that consists of an API Gateway with Lambda and DynamoDB using the .NET SDK.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

Go

SDK for Go V2

Shows how to build and test a serverless application that consists of an API Gateway with Lambda and DynamoDB using the Go SDK.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

Java

SDK for Java 2.x

Shows how to build and test a serverless application that consists of an API Gateway with Lambda and DynamoDB using the Java SDK.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

Rust

SDK for Rust

Shows how to build and test a serverless application that consists of an API Gateway with Lambda and DynamoDB using the Rust SDK.

For complete source code and instructions on how to set up and run, see the full example on GitHub.

Services used in this example

- API Gateway
- DynamoDB
- Lambda

For a complete list of AWS SDK developer guides and code examples, see Using Lambda with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

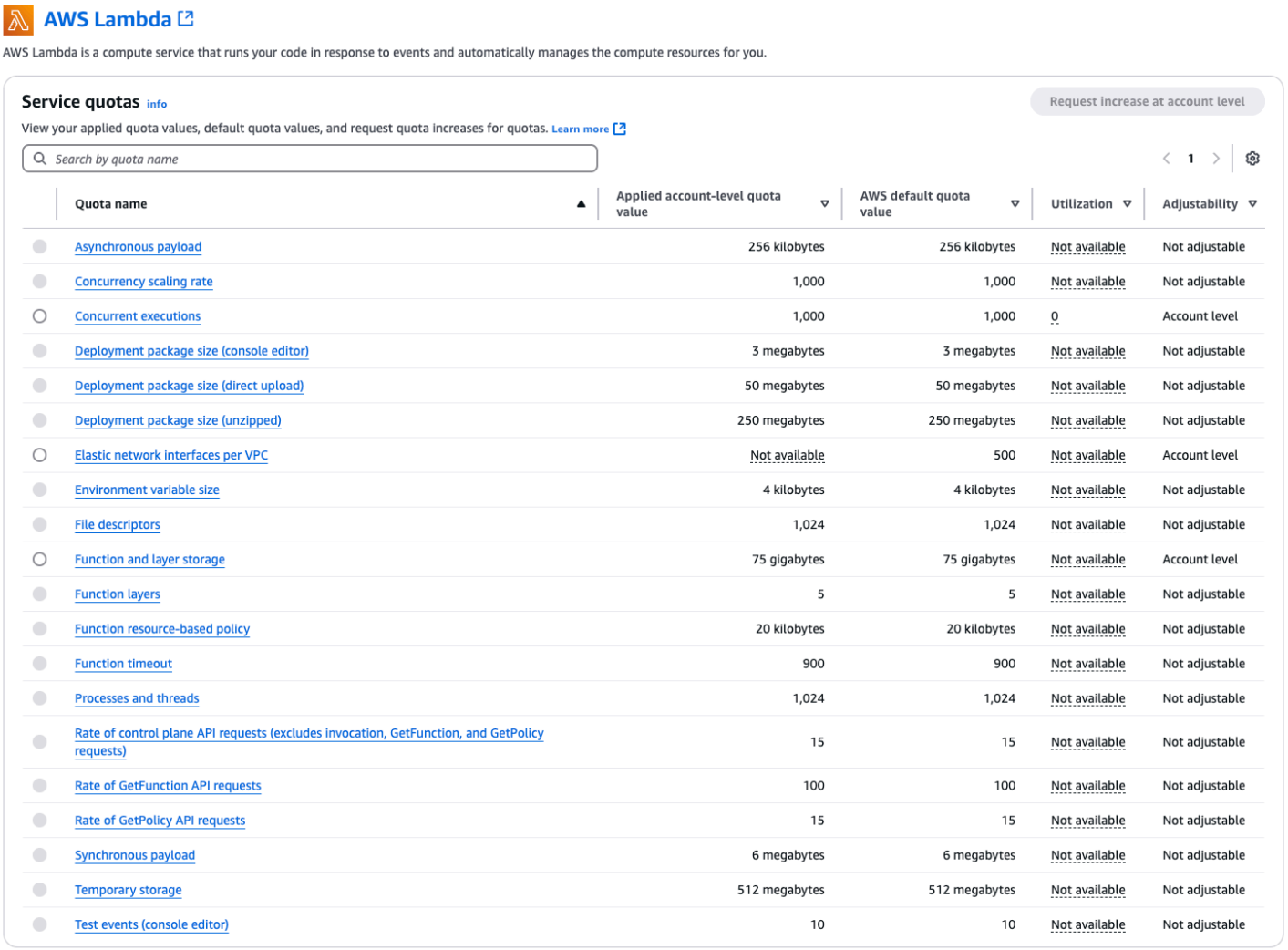
Lambda quotas

> **Important**
>
> New AWS accounts have reduced concurrency and memory quotas. AWS raises these quotas automatically based on your usage.

AWS Lambda is designed to scale rapidly to meet demand, allowing your functions to scale up to serve traffic in your application. Lambda is designed for short-lived compute tasks that do not retain or rely upon state between invocations. Code can run for up to 15 minutes in a single invocation and a single function can use up to 10,240 MB of memory.

It's important to understand the guardrails that are put in place to protect your account and the workloads of other customers. Service quotas exist in all AWS services and consist of hard limits, which you cannot change, and soft limits, which you can request increases for. By default, all new accounts are assigned a quota profile that allows exploration of AWS services.

To see the quotas that apply to your account, navigate to the Service Quotas dashboard. Here, you can view your service quotas, request a quota increase, and view current utilization. From here, you can drill down to a specific AWS service, such as Lambda:



AWS Lambda

AWS Lambda is a compute service that runs your code in response to events and automatically manages the compute resources for you.

Service quotas info

Request increase at account level

View your applied quota values, default quota values, and request quota increases for quotas. Learn more

Search by quota name

1

Quota name	Applied account-level quota value	AWS default quota value	Utilization	Adjustability
Asynchronous payload	256 kilobytes	256 kilobytes	Not available	Not adjustable
Concurrency scaling rate	1,000	1,000	Not available	Not adjustable
Concurrent executions	1,000	1,000	0	Account level
Deployment package size (console editor)	3 megabytes	3 megabytes	Not available	Not adjustable
Deployment package size (direct upload)	50 megabytes	50 megabytes	Not available	Not adjustable
Deployment package size (unzipped)	250 megabytes	250 megabytes	Not available	Not adjustable
Elastic network interfaces per VPC	Not available	500	Not available	Account level
Environment variable size	4 kilobytes	4 kilobytes	Not available	Not adjustable
File descriptors	1,024	1,024	Not available	Not adjustable
Function and layer storage	75 gigabytes	75 gigabytes	Not available	Account level
Function layers	5	5	Not available	Not adjustable
Function resource-based policy	20 kilobytes	20 kilobytes	Not available	Not adjustable
Function timeout	900	900	Not available	Not adjustable
Processes and threads	1,024	1,024	Not available	Not adjustable
Rate of control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)	15	15	Not available	Not adjustable
Rate of GetFunction API requests	100	100	Not available	Not adjustable
Rate of GetPolicy API requests	15	15	Not available	Not adjustable
Synchronous payload	6 megabytes	6 megabytes	Not available	Not adjustable
Temporary storage	512 megabytes	512 megabytes	Not available	Not adjustable
Test events (console editor)	10	10	Not available	Not adjustable

The following sections list default quotas and limits in Lambda by category.

Topics

- Compute and storage
- Function configuration, deployment, and execution
- Lambda API requests
- Other services

Compute and storage

Lambda sets quotas for the amount of compute and storage resources that you can use to run and store functions. Quotas for concurrent executions and storage apply per AWS Region. Elastic

network interface (ENI) quotas apply per virtual private cloud (VPC), regardless of Region. The following quotas can be increased from their default values. For more information, see Requesting a quota increase in the *Service Quotas User Guide*.

Resource	Default quota	Can be increased up to
Concurrent executions	1,000	Tens of thousands
Storage for uploaded functions (.zip file archives) and layers. Each function version and layer version consumes storage. For best practices on managing your code storage, see Monitoring Lambda code storage in Serverless Land.	75 GB	Terabytes
Storage for functions defined as container images. These images are stored in Amazon ECR.	See Amazon ECR service quotas.	
Elastic network interfaces per virtual private cloud (VPC) **Note** This quota is shared with other services, such as Amazon Elastic File System (Amazon EFS). See Amazon VPC quotas.	500	Thousands

For details on concurrency and how Lambda scales your function concurrency in response to traffic, see Understanding Lambda function scaling.

Function configuration, deployment, and execution

The following quotas apply to function configuration, deployment, and execution. Except as noted, they can't be changed.

> **Note**
>
> The Lambda documentation, log messages, and console use the abbreviation MB (rather than MiB) to refer to 1,024 KB.

Resource	Quota
Function memory allocation	128 MB to 10,240 MB, in 1-MB increments. **Note:** Lambda allocates CPU power in proportion to the amount of memory configured. You can increase or decrease the memory and CPU power allocated to your function using the **Memory (MB)** setting. At 1,769 MB, a function has the equivalent of one vCPU.
Function timeout	900 seconds (15 minutes)
Function environment variables	4 KB, for all environment variables associated with the function, in aggregate
Function resource-based policy	20 KB
Function layers	5 layers
Function concurrency scaling limit	For each function, 1,000 execution environments every 10 seconds

Resource	Quota
Invocation payload (request and response)	6 MB each for request and response (synchronous) 200 MB for each streamed response (synchronous) 1 MB (asynchronous) 1 MB for the total combined size of request line and header values
Bandwidth for streamed responses	Uncapped for the first 6 MB of your function's response For responses larger than 6 MB, 2MBps for the remainder of the response
Deployment package (.zip file archive) size	50 MB (zipped, when uploaded through the Lambda API or SDKs). Upload larger files with Amazon S3. 50 MB (when uploaded through the Lambda console) 250 MB The maximum size of the contents of a deployment package, including layers and custom runtimes. (unzipped)
Container image settings size	16 KB
Container image code package size	10 GB (maximum uncompressed image size, including all layers)
Test events (console editor)	10

Resource	Quota
/tmp directory storage	Between 512 MB and 10,240 MB, in 1-MB increments
File descriptors	1,024 **Note** Lambda Managed Instances use a higher file descriptor limit of 4,096. For more information, see Understanding the Lambda Managed Instances execution environment.
Execution processes/threads	1,024 **Note** Lambda Managed Instances use the default process and thread limits from Bottlerocket. For more information, see Understanding the Lambda Managed Instances execution environment.

Lambda API requests

The following quotas are associated with Lambda API requests.

Resource	Quota
Invocation requests per function per Region (synchronous)	Each instance of your execution environment can serve up to 10 requests per second. In other words, the total invocation limit is 10 times your concurrency limit. See Understanding Lambda function scaling.
Invocation requests per function per Region (asynchronous)	Each instance of your execution environment can serve an unlimited number of requests. In other words, the total invocation limit is based only on concurrency available to your function. See Understanding Lambda function scaling.
Invocation requests per function version or alias (requests per second)	10 x allocated provisioned concurrency **Note** This quota applies only to functions that use provisioned concurrency.
GetFunction API requests	100 requests per second. Cannot be increased.
GetPolicy API requests	15 requests per second. Cannot be increased.
Remainder of the control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)	15 requests per second across all APIs (not 15 requests per second per API). Cannot be increased.

Other services

Quotas for other services, such as AWS Identity and Access Management (IAM), Amazon CloudFront (Lambda@Edge), and Amazon Virtual Private Cloud (Amazon VPC), can impact your Lambda functions. For more information, see AWS service quotas in the *Amazon Web Services General Reference*, and Invoking Lambda with events from other AWS services.

Many applications involving Lambda use multiple AWS services. Because different services have different quotas for various features, it can be challenging to manage these quotas across your entire application. For example, API Gateway has a default throttle limit of 10,000 requests per second, whereas Lambda has a default concurrency limit of 1,000. Due to this mismatch, it's possible to have more incoming requests from API Gateway that Lambda can handle. You can resolve this by requesting a Lambda concurrency limit increase to match the expected level of traffic.

Load testing your application allows you to monitor the performance of your application end-to-end before deploying to production. During a load test, you can identify any quotas that may act as a limiting factor for the traffic levels you expect and take action accordingly.

Document history

The following table describes the important changes to the *AWS Lambda Developer Guide* since May 2018. For notification about updates to this documentation, subscribe to the RSS feed.

Change	Description	Date
AWS managed policy updates	Lambda has released a new managed policy `AWSLambdaBasicDurableExecutionRolePolicy` that provides write permissions to CloudWatch Logs and read/write permissions to durable execution APIs used by Lambda durable functions.	December 1, 2025
Service-linked role for Lambda	Lambda now supports a service-linked role (`AWSServiceRoleForLambda`) that allows Lambda to terminate instances managed as part of Lambda capacity providers.	November 30, 2025
Lambda Managed Instances	Lambda now supports Managed Instances, which provide dedicated compute capacity for your functions with enhanced performance, security, and control. For details, see Lambda Managed Instances.	November 30, 2025
AWS managed policy updates	Lambda added a new AWS managed policy (`AWSLambda`	November 30, 2025

	`BasicDurableExecutionRolePolicy`).	
AWS managed policy updates	Lambda updated the existing AWS managed policy `AWSLambda_FullAccess`) allow the `kms:DescribeKey` and `iam:CreateServiceLinkedRole` actions.	November 30, 2025
AWS managed policy update	Lambda added a new AWS managed policy (`AWSLambdaManagedEC2ResourceOperator`) to enable automated Amazon EC2 instance management for Lambda capacity providers . For details, see Lambda updates to AWS managed policies.	November 30, 2025
AWS managed policy update	Lambda added a new AWS managed policy (`AWSLambdaServiceRolePolicy`). For details, see Lambda updates to AWS managed policies.	November 30, 2025
AWS managed policy updates	Lambda updated two existing AWS managed policies (`AWSLambda_ReadOnlyAccess` and `AWSLambda_FullAccess`).	February 20, 2025
Node.js 22.x runtime	Lambda now supports Node.js 22 as a managed runtime and container base image (`nodejs22.x`).	November 22, 2024

SnapStart support for Python and .NET	Lambda SnapStart is now available for Python and .NET managed runtimes, beginning with `python3.12` and `dotnet8`.	November 18, 2024
Python 3.13 runtime	Lambda now supports Python 3.13 as a managed runtime and container base image.	November 13, 2024
Customer managed encryption for .zip deployment packages	Lambda now supports AWS KMS customer managed key encryption for .zip deployment packages.	November 8, 2024
Support for SnapStart in new Regions	Lambda SnapStart is now available in the following Regions: Europe (Spain), Europe (Zurich), Asia Pacific (Melbourne), Asia Pacific (Hyderabad), and Middle East (UAE).	January 12, 2024
AWS managed policy updates	Lambda updated an existing AWS managed policy (`AWSLambdaVPCAccessExecutionRole`).	January 5, 2024
Python 3.12 runtime	Lambda now supports Python 3.12 as a managed runtime and container base image. For more information, see Python 3.12 runtime now available in AWS Lambda on the AWS Compute Blog.	December 14, 2023

Java 21 runtime	Lambda now supports Java 21 as a managed runtime and container base image (`java21`).	November 16, 2023
Node.js 20.x runtime	Lambda now supports Node.js 20 as a managed runtime and container base image (`nodejs20.x`). For more information, see Node.js 20.x runtime now available in AWS Lambda on the AWS Compute Blog.	November 14, 2023
provided.al2023 runtime	Lambda now supports Amazon Linux 2023 as a managed runtime and container base image. For more information, see Introducing the Amazon Linux 2023 runtime for AWS Lambda on the AWS Compute Blog.	November 9, 2023
IPv6 support for dual-stack subnets	Lambda now supports outbound IPv6 traffic to dual-stack subnets. For more information, see IPv6 support.	October 12, 2023

Testing serverless functions and applications	Learn about techniques to debug and automate testing serverless functions in the cloud. There is now a testing chapter and resources included in the Python and Typescript language sections. For details, see Testing serverless functions and applications.	June 16, 2023
Ruby 3.2 runtime	Lambda now supports a new runtime for Ruby 3.2. For more information, see Building Lambda functions with Ruby.	June 7, 2023
Response streaming	Lambda now supports streaming responses from functions. For more informati on, see Configuring a Lambda function to stream responses.	April 6, 2023
Asynchronous invocation metrics	Lambda releases asynchron ous invocation metrics. For more information, see Asynchronous invocation metrics.	February 9, 2023

Runtime version controls	Lambda releases new runtime versions that include security updates, bug fixes, and new features. You can now control when your functions get updated to the new runtime versions. For more information, see Lambda runtime updates.	January 23, 2023
Lambda SnapStart	Use Lambda SnapStart to reduce startup time for Java functions without provisioning additional resources or implementing complex performance optimizations. For more information, see Improving startup performance with with Lambda SnapStart.	November 28, 2022
Node.js 18 runtime	Lambda now supports a new runtime for Node.js 18. Node.js 18 uses Amazon Linux 2. For details, see Building Lambda functions with Node.js.	November 18, 2022
lambda:SourceFunctionArn condition key	For an AWS resource, the `lambda:SourceFunctionArn` condition key filters access to the resource by the ARN of a Lambda function. For details, see Working with Lambda execution environment credentials.	July 1, 2022

Node.js 16 runtime	Lambda now supports a new runtime for Node.js 16. Node.js 16 uses Amazon Linux 2. For details, see Building Lambda functions with Node.js.	May 11, 2022
Lambda function URLs	Lambda now supports function URLs, which are dedicated HTTP(S) endpoints for Lambda functions. For details, see Lambda function URLs.	April 6, 2022
Shared test events in the AWS Lambda console	Lambda now supports sharing test events with other users in the same AWS account. For details, see Testing Lambda functions in the console.	March 16, 2022
PrincipalOrgId in resource-based policies	Lambda now supports granting permissions to an organization in AWS Organizations. For details, see Using resource-based policies for AWS Lambda.	March 11, 2022
.NET 6 runtime	Lambda now supports a new runtime for .NET 6. For details, see Lambda runtimes.	February 23, 2022
Event filtering for Kinesis, DynamoDB, and Amazon SQS event sources	Lambda now supports event filtering for Kinesis, DynamoDB, and Amazon SQS event sources. For details, see Lambda event filtering.	November 24, 2021

mTLS authentication for Amazon MSK and self-managed Apache Kafka event sources	Lambda now supports mTLS authentication for Amazon MSK and self-managed Apache Kafka event sources. For details, see Using Lambda with Amazon MSK.	November 19, 2021
Lambda on Graviton2	Lambda now supports Graviton2 for functions using arm64 architecture. For details, see Lambda instruction set architectures.	September 29, 2021
Python 3.9 runtime	Lambda now supports a new runtime for Python 3.9. For details, see Lambda runtimes.	August 16, 2021
New runtime versions for Node.js, Python, and Java	New runtime versions are available for Node.js, Python, and Java. For details, see Lambda runtimes.	July 21, 2021
Support for RabbitMQ as an event source on Lambda	Lambda now supports Amazon MQ for RabbitMQ as an event source. Amazon MQ is a managed message broker service for Apache ActiveMQ and RabbitMQ that makes it easy to set up and operate message brokers in the cloud. For details, see Using Lambda with Amazon MQ.	July 7, 2021

SASL/PLAIN authentication for self-managed Kafka on Lambda	SASL/PLAIN is now a supported authentication mechanism for self-managed Kafka event sources on Lambda Customers already using SASL/PLAIN on their self-managed Kafka cluster can now easily use Lambda to build consumer applications without having to modify the way they authenticate. For details, see Using Lambda with self-managed Apache Kafka.	June 29, 2021
Lambda Extensions API	General availability for Lambda extensions. Use extensions to augment your Lambda functions. You can use extensions provided by Lambda Partners, or you can create your own Lambda extensions. For details, see Lambda Extensions API.	May 24, 2021
New Lambda console experience	The Lambda console has been redesigned to improve performance and consistency.	March 2, 2021
Node.js 14 runtime	Lambda now supports a new runtime for Node.js 14. Node.js 14 uses Amazon Linux 2. For details, see Building Lambda functions with Node.js.	January 27, 2021

Lambda container images	Lambda now supports functions defined as container images. You can combine the flexibility of container tooling with the agility and operation al simplicity of Lambda to build applications. For details, see Using container images with Lambda.	December 1, 2020
Code signing for Lambda functions	Lambda now supports code signing. Administrators can configure Lambda functions to accept only signed code on deployment. Lambda checks the signatures to ensure that the code is not altered or tampered. Additionally, Lambda ensures that the code is signed by trusted developers before accepting the deployment. For details, see Configuring code signing for Lambda.	November 23, 2020
Preview: Lambda Runtime Logs API	Lambda now supports the Runtime Logs API. Lambda extensions can use the Logs API to subscribe to log streams in the execution environment. For details, see Lambda Runtime Logs API.	November 12, 2020

New event source to for Amazon MQ	Lambda now supports Amazon MQ as an event source. Use a Lambda function to process records from your Amazon MQ message broker. For details, see Using Lambda with Amazon MQ.	November 5, 2020
Preview: Lambda Extensions API	Use Lambda extensions to augment your Lambda functions. You can use extensions provided by Lambda Partners, or you can create your own Lambda extensions. For details, see Lambda Extensions API.	October 8, 2020
Support for Java 8 and custom runtimes on AL2	Lambda now supports Java 8 and custom runtimes on Amazon Linux 2. For details, see Lambda runtimes.	August 12, 2020
New event source for Amazon Managed Streaming for Apache Kafka	Lambda now supports Amazon MSK as an event source. Use a Lambda function with Amazon MSK to process records in a Kafka topic. For details, see Using Lambda with Amazon MSK.	August 11, 2020

IAM condition keys for Amazon VPC settings	You can now use Lambda-specific condition keys for VPC settings. For example, you can require that all functions in your organization are connected to a VPC. You can also specify the subnets and security groups that the function's users can and can't use. For details, see Configuring VPC for IAM functions.	August 10, 2020
Concurrency settings for Kinesis HTTP/2 stream consumers	You can now use the following concurrency settings for Kinesis consumers with enhanced fan-out (HTTP/2 streams): ParallelizationFactor, MaximumRetryAttempts, MaximumRecordAgeInSeconds, DestinationConfig, and BisectBatchOnFunctionError. For details, see Using AWS Lambda with Amazon Kinesis.	July 7, 2020
Batch window for Kinesis HTTP/2 stream consumers	You can now configure a batch window (MaximumBatchingWindowInSeconds) for HTTP/2 streams. Lambda reads records from the stream until it has gathered a full batch, or until the batch window expires. For details, see Using AWS Lambda with Amazon Kinesis.	June 18, 2020

Support for Amazon EFS file systems	You can now connect an Amazon EFS file system to your Lambda functions for shared network file access. For details, see Configuring file system access for Lambda functions.	June 16, 2020
AWS CDK sample applications in the Lambda console	The Lambda console now includes sample applicati ons that use the AWS Cloud Development Kit (AWS CDK) for TypeScript. The AWS CDK is a framework that enables you to define your applicati on resources in TypeScript, Python, Java, or .NET.	June 1, 2020
Support for .NET Core 3.1.0 runtime in AWS Lambda	AWS Lambda now supports the .NET Core 3.1.0 runtime. For details, see .NET Core CLI.	March 31, 2020
Support for API Gateway HTTP APIs	Updated and expanded documentation for using Lambda with API Gateway, including support for HTTP APIs. Added a sample application that creates an API and function with CloudFormation. For details, see Using Lambda with Amazon API Gateway.	March 23, 2020

Ruby 2.7	A new runtime is available for Ruby 2.7, ruby2.7, which is the first Ruby runtime to use Amazon Linux 2. For details, see Building Lambda functions with Ruby.	February 19, 2020
Concurrency metrics	Lambda now reports the `ConcurrentExecutions` metric for all functions, aliases, and versions. You can view a graph for this metric on the monitoring page for your function. Previously, `ConcurrentExecutions` was only reported at the account level and for functions that use reserved concurrency. For details, see AWS Lambda function metrics.	February 18, 2020

Update to function states	Function states are now enforced for all functions by default. When you connect a function to a VPC, Lambda creates shared elastic network interfaces. This enables your function to scale up without creating additiona l network interfaces. During this time, you can't perform additional operations on the function, including updating its configuration and publishing versions. In some cases, invocation is also impacted. Details about a function's current state are available from the Lambda API. This update is being released in phases. For details, see Updated Lambda states lifecycle for VPC networking on the AWS Compute Blog. For more information about states, see AWS Lambda function states.	January 24, 2020

Updates to function configuration API output	Added reason codes to StateReasonCode (InvalidSubnet, InvalidSecurityGroup) and LastUpdateStatusReasonCode (SubnetOutOfIPAddresses, InvalidSubnet, InvalidSecurityGroup) for functions that connect to a VPC. For more information about states, see AWS Lambda function states.	January 20, 2020
Provisioned concurrency	You can now allocate provisioned concurrency for a function version or alias. Provisioned concurrency enables a function to scale without fluctuations in latency. For details, see Managing concurrency for a Lambda function.	December 3, 2019
Create a database proxy	You can now use the Lambda console to create a database proxy for a Lambda function. A database proxy enables a function to reach high concurrency levels without exhausting database connections. For details, see Configuring database access for a Lambda function.	December 3, 2019
Percentiles support for the duration metric	You can now filter the duration metric based on percentiles. For details, see AWS Lambda metrics.	November 26, 2019

Increased concurrency for stream event sources	A new option for DynamoDB stream and Kinesis stream event source mappings enables you to process more than one batch at a time from each shard. When you increase the number of concurrent batches per shard, your function's concurrency can be up to 10 times the number of shards in your stream. For details, see Lambda event source mapping.	November 25, 2019
Function states	When you create or update a function, it enters a pending state while Lambda provisions resources to support it. If you connect your function to a VPC, Lambda can create a shared elastic network interface right away, instead of creating network interfaces when your function is invoked. This results in better performance for VPC-connected functions, but might require an update to your automation. For details, see AWS Lambda function states.	November 25, 2019

Error handling options for asynchronous invocation	New configuration options are available for asynchronous invocation. You can configure Lambda to limit retries and set a maximum event age. For details, see Configuring error handling for asynchronous invocation.	November 25, 2019
Error handling for stream event sources	New configuration options are available for event source mappings that read from streams. You can configure DynamoDB stream and Kinesis stream event source mappings to limit retries and set a maximum record age. When errors occur, you can configure the event source mapping to split batches before retrying, and to send invocation records for failed batches to a queue or topic. For details, see Lambda event source mapping.	November 25, 2019

Destinations for asynchronous invocation	You can now configure Lambda to send records of asynchronous invocations to another service. Invocation records contain details about the event, context, and function response. You can send invocation records to an SQS queue, SNS topic, Lambda function, or EventBridge event bus. For details, see Configuring destinations for asynchronous invocation.	November 25, 2019
New runtimes for Node.js, Python, and Java	New runtimes are available for Node.js 12, Python 3.8, and Java 11. For details, see Lambda runtimes.	November 18, 2019
FIFO queue support for Amazon SQS event sources	You can now create an event source mapping that reads from a first-in, first-out (FIFO) queue. Previously, only standard queues were supported. For details, see Using Lambda with Amazon SQS.	November 18, 2019
Create applications in the Lambda console	Application creation in the Lambda console is now generally available. For instructions, see Managing applications in the Lambda console.	October 31, 2019

Create applications in the Lambda console (beta)	You can now create a Lambda application with an integrated continuous delivery pipeline in the Lambda console. The console provides sample applications that you can use as a starting point for your own project. Choose between AWS CodeCommit and GitHub for source control. Each time you push changes to your repository, the included pipeline builds and deploys them automatically. For instructions, see Managing applications in the Lambda console.	October 3, 2019

Performance improvements for VPC-connected functions	Lambda now uses a new type of elastic network interface that is shared by all functions in a virtual private cloud (VPC) subnet. When you connect a function to a VPC, Lambda creates a network interface for each combinati on of security group and subnet that you choose. When the shared network interface s are available, the function no longer needs to create additional network interfaces as it scales up. This dramatica lly improves startup times. For details, see Configuring a Lambda function to access resources in a VPC.	September 3, 2019
Stream batch settings	You can now configure a batch window for Amazon DynamoDB and Amazon Kinesis event source mappings. Configure a batch window of up to five minutes to buffer incoming records until a full batch is available . This reduces the number of times that your function is invoked when the stream is less active.	August 29, 2019

CloudWatch Logs Insights integration	The monitoring page in the Lambda console now includes reports from Amazon CloudWatch Logs Insights.	June 18, 2019
Amazon Linux 2018.03	The Lambda execution environment is being updated to use Amazon Linux 2018.03. For details, see Execution environment.	May 21, 2019
Node.js 10	A new runtime is available for Node.js 10, nodejs10.x. This runtime uses Node.js 10.15 and will be updated with the latest point release of Node.js 10 periodically. Node.js 10 is also the first runtime to use Amazon Linux 2. For details, see Building Lambda functions with Node.js.	May 13, 2019
GetLayerVersionByArn API	Use the GetLayerVersionByArn API to download layer version information with the version ARN as input. Compared to GetLayerVersion, GetLayerVersionByArn lets you use the ARN directly instead of parsing it to get the layer name and version number.	April 25, 2019
Ruby	AWS Lambda now supports Ruby 2.5 with a new runtime. For details, see Building Lambda functions with Ruby.	November 29, 2018

Layers	With Lambda layers, you can package and deploy libraries , custom runtimes, and other dependencies separately from your function code. Share your layers with your other accounts or the whole world. For details, see Lambda layers.	November 29, 2018
Custom runtimes	Build a custom runtime to run Lambda functions in your favorite programming language. For details, see Custom Lambda runtimes.	November 29, 2018
Application Load Balancer triggers	Elastic Load Balancing now supports Lambda functions as a target for Application Load Balancers. For details, see Using Lambda with applicati on load balancers.	November 29, 2018
Use Kinesis HTTP/2 stream consumers as a trigger	You can use Kinesis HTTP/2 data stream consumers to send events to AWS Lambda. Stream consumers have dedicated read throughpu t from each shard in your data stream and use HTTP/2 to minimize latency. For details, see Using Lambda with Kinesis.	November 19, 2018

Python 3.7	AWS Lambda now supports Python 3.7 with a new runtime. For more information, see Building Lambda functions with Python.	November 19, 2018
Payload limit increase for asynchronous function invocation	The maximum payload size for asynchronous invocations increased from 128 KB to 256 KB, which matches the maximum message size from an Amazon SNS trigger. For details, see Lambda quotas.	November 16, 2018
AWS GovCloud (US-East) Region	AWS Lambda is now available in the AWS GovCloud (US-East) Region.	November 12, 2018
Moved AWS SAM topics to a separate Developer Guide	A number of topics were focused on building serverless applications using the AWS Serverless Application Model (AWS SAM). These topics have been moved to AWS Serverless Application Model developer guide.	October 25, 2018

View Lambda applications in the console	You can view the status of your Lambda applications on the Applications page in the Lambda console. This page shows the status of the CloudFormation stack. It includes links to pages where you can view more informati on about the resources in the stack. You can also view aggregate metrics for the application and create custom monitoring dashboards.	October 11, 2018
Function execution timeout limit	To allow for long-running functions, the maximum configurable execution timeout increased from 5 minutes to 15 minutes. For details, see Lambda limits.	October 10, 2018
Support for PowerShell Core language in AWS Lambda	AWS Lambda now supports the PowerShell Core language. For more informati on, see Programming model for authoring Lambda functions in PowerShell.	September 11, 2018
Support for .NET Core 2.1.0 runtime in AWS Lambda	AWS Lambda now supports the .NET Core 2.1.0 runtime. For more information, see .NET Core CLI.	July 9, 2018
Updates now available over RSS	You can now subscribe to an RSS feed to follow releases for this guide.	July 5, 2018

Support for Amazon SQS as event source	AWS Lambda now supports Amazon Simple Queue Service (Amazon SQS) as an event source. For more information, see Invoking Lambda functions.	June 28, 2018
China (Ningxia) Region	AWS Lambda is now available in the China (Ningxia) Region. For more information about Lambda Regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	June 28, 2018

Earlier updates

The following table describes the important changes in each release of the *AWS Lambda Developer Guide* before June 2018.

Change	Description	Date
Runtime support for Node.js runtime 8.10	AWS Lambda now supports Node.js runtime version 8.10. For more information, see Building Lambda functions with Node.js.	April 2, 2018
Function and alias revision IDs	AWS Lambda now supports revision IDs on your function versions and aliases. You can use these IDs to track and apply conditional updates when you are updating your function version or alias resources.	January 25, 2018
Runtime support for Go and .NET 2.0	AWS Lambda has added runtime support for Go and .NET 2.0. For more information, see Building Lambda functions with Go and Building Lambda functions with C#.	January 15, 2018
Console Redesign	AWS Lambda has introduced a new Lambda console to simplify your experience and added a Cloud9 Code Editor	November 30,2017

Change	Description	Date
	to enhance your ability debug and revise your function code.	
Setting Concurrency Limits on Individual Functions	AWS Lambda now supports setting concurrency limits on individual functions. For more information, see Configuring reserved concurrency for a function.	November 30,2017
Shifting Traffic with Aliases	AWS Lambda now supports shifting traffic with aliases. For more information, see Create a rolling deployment with weighted aliases.	November 28, 2017
Gradual Code Deployment	AWS Lambda now supports safely deploying new versions of your Lambda function by leveraging Code Deploy. For more information, see Gradual code deployment.	November 28, 2017
China (Beijing) Region	AWS Lambda is now available in the China (Beijing) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	November 9, 2017
Introducing SAM Local	AWS Lambda introduces SAM Local (now known as SAM CLI), a AWS CLI tool that provides an environment for you to develop, test, and analyze your serverless applications locally before uploading them to the Lambda runtime. For more information, see Testing and debugging serverless applications.	August 11, 2017
Canada (Central) Region	AWS Lambda is now available in the Canada (Central) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	June 22, 2017
South America (São Paulo) Region	AWS Lambda is now available in the South America (São Paulo) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	June 6, 2017

Change	Description	Date
AWS Lambda support for AWS X-Ray.	Lambda introduces support for X-Ray, which allows you to detect, analyze, and optimize performance issues with your Lambda applications. For more information, see Visualize Lambda function invocations using AWS X-Ray.	April 19, 2017
Asia Pacific (Mumbai) Region	AWS Lambda is now available in the Asia Pacific (Mumbai) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	March 28, 2017
AWS Lambda now supports Node.js runtime v6.10	AWS Lambda added support for Node.js runtime v6.10. For more information, see Building Lambda functions with Node.js.	March 22, 2017
Europe (London) Region	AWS Lambda is now available in the Europe (London) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	February 1, 2017
AWS Lambda support for the .NET runtime, Lambda@Edge (Preview), Dead Letter Queues and automated deployment of serverless applications.	AWS Lambda added support for C#. For more information, see Building Lambda functions with C#. Lambda@Edge allows you to run Lambda functions at the AWS Edge locations in response to CloudFront events. For more information, see Customize at the edge with Lambda@Edge.	December 3, 2016
AWS Lambda adds Amazon Lex as a supported event source.	Using Lambda and Amazon Lex, you can quickly build chat bots for various services like Slack and Facebook.	November 30, 2016

Change	Description	Date
US West (N. Californi a) Region	AWS Lambda is now available in the US West (N. Californi a) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	November 21, 2016
Introduced the AWS SAM for creating and deploying Lambda-ba sed applications and using environment variables for Lambda function configura tion settings.	AWS SAM: You can now use the AWS SAM to define the syntax for expressing resources within a serverless application. In order to deploy your application, simply specify the resources you need as part of your applicati on, along with their associated permissions policies in a AWS CloudFormation template file (written in either JSON or YAML), package your deployment artifacts, and deploy the template. Environment variables: You can use environment variables to specify configuration settings for your Lambda function outside of your function code. For more information, see Working with Lambda environment variables.	November 18, 2016
Asia Pacific (Seoul) Region	AWS Lambda is now available in the Asia Pacific (Seoul) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	August 29, 2016
Asia Pacific (Sydney) Region	Lambda is now available in the Asia Pacific (Sydney) Region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	June 23, 2016
Updates to the Lambda console	The Lambda console has been updated to simplify the role-creation process.	June 23, 2016
AWS Lambda now supports Node.js runtime v4.3	AWS Lambda added support for Node.js runtime v4.3. For more information, see Building Lambda functions with Node.js.	April 07, 2016

Change	Description	Date
Europe (Frankfurt) region	Lambda is now available in the Europe (Frankfurt) region. For more information about Lambda regions and endpoints, see Regions and endpoints in the *AWS General Reference*.	March 14, 2016
VPC support	You can now configure a Lambda function to access resources in your VPC. For more information, see Giving Lambda functions access to resources in an Amazon VPC.	February 11, 2016
Lambda runtime has been updated.	The execution environment has been updated.	November 4, 2015
Versioning support, Python for developing code for Lambdafunctions, scheduled events, and increase in execution time	You can now develop your Lambda function code using Python. For more information, see Building Lambda functions with Python. Versioning: You can maintain one or more versions of your Lambda function. Versioning allows you to control which Lambda function version is executed in different environments (for example, development, testing, or production). For more information, see Manage Lambda function versions. Scheduled events: You can also set up Lambda to invoke your code on a regular, scheduled basis using the Lambda console. You can specify a fixed rate (number of hours, days, or weeks) or you can specify a cron expression. For more information, see Invoke a Lambda function on a schedule. Increase in execution time: You can now set up your Lambda functions to run for up to five minutes allowing longer running functions such as large volume data ingestion and processing jobs.	October 08, 2015

Change	Description	Date
Support for DynamoDB Streams	DynamoDB Streams is now generally available and you can use it in all the regions where DynamoDB is available . You can enable DynamoDB Streams for your table and use a Lambda function as a trigger for the table. Triggers are custom actions you take in response to updates made to the DynamoDB table. For an example walkthrough, see Tutorial: Using AWS Lambda with Amazon DynamoDB streams.	July 14, 2015
Lambda now supports invoking Lambda functions with REST-compatible clients.	Until now, to invoke your Lambda function from your web, mobile, or IoT application you needed the AWS SDKs (for example, AWS SDK for Java, AWS SDK for Android, or AWS SDK for iOS). Now, Lambda supports invoking a Lambda function with REST-compatible clients through a customized API that you can create using Amazon API Gateway. You can send requests to your Lambda function endpoint URL. You can configure security on the endpoint to allow open access, leverage AWS Identity and Access Management (IAM) to authorize access, or use API keys to meter access to your Lambda functions by others. For an example Getting Started exercise, see Invoking a Lambda function using an Amazon API Gateway endpoint.	July 09, 2015
The Lambda console now provides blueprints to easily create Lambda functions and test them.	Lambda console provides a set of *blueprints*. Each blueprint provides a sample event source configuration and sample code for your Lambda function that you can use to easily create Lambda-based applications. All of the Lambda Getting Started exercises now use the blueprints.	July 09, 2015
Lambda now supports Java to author your Lambda functions.	You can now author Lambda code in Java. For more information, see Building Lambda functions with Java.	June 15, 2015

Change	Description	Date
Lambda now supports specifying an Amazon S3 object as the function .zip when creating or updating a Lambda function.	You can upload a Lambda function deployment package (.zip file) to an Amazon S3 bucket in the same region where you want to create a Lambda function. Then, you can specify the bucket name and object key name when you create or update a Lambda function.	May 28, 2015
Lambda now generally available with added support for mobile backends	Lambda is now generally available for production use. The release also introduces new features that make it easier to build mobile, tablet, and Internet of Things (IoT) backends using Lambda that scale automatically without provisioning or managing infrastructure. Lambda now supports both real-time (synchronous) and asynchronous events. Additional features include easier event source configuration and management. The permission model and the programming model have been simplified by the introduction of resource policies for your Lambda functions.	April 9, 2015
Preview release	Preview release of the *AWS Lambda Developer Guide*.	November 13, 2014