



Amazon EMR Serverless ユーザーガイド

Amazon EMR



Amazon EMR: Amazon EMR Serverless ユーザーガイド

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

Amazon の商標およびトレードドレスは Amazon 以外の製品およびサービスに使用することはできません。また、お客様に誤解を与える可能性がある形式で、または Amazon の信用を損なう形式で使用することもできません。Amazon が所有していないその他のすべての商標は Amazon との提携、関連、支援関係の有無にかかわらず、それら該当する所有者の資産です。

Table of Contents

Amazon EMR Serverless とは	1
概念	1
リリースバージョン	1
アプリケーション	2
ジョブ実行	2
ワーカー	3
事前初期化された容量	3
EMR Studio	3
使用を開始するための前提条件	5
にサインアップする AWS アカウント	5
管理アクセスを持つユーザーを作成する	6
許可を付与する	7
プログラマチックアクセス権を付与する	9
のセットアップ AWS CLI	11
コンソールを開きます。	12
開始方法	13
アクセス許可	13
Storage	13
インタラクティブワークロード	13
ジョブランタイムロールを作成する	14
コンソールからの開始方法	20
ステップ 1: アプリケーションを作成する	20
ステップ 2: ジョブ実行またはインタラクティブワークロードを送信する	21
ステップ 3: アプリケーション UI とログを表示する	24
ステップ 4: クリーンアップする	25
の開始方法 AWS CLI	25
ステップ 1: アプリケーションを作成する	25
ステップ 2: ジョブ実行を送信する	26
ステップ 3: 出力を確認する	29
ステップ 4: クリーンアップする	30
EMR Serverless アプリケーションを操作して設定する	32
アプリケーションの状態	32
EMR Studio コンソールの使用	33
アプリケーションの作成	34

EMR Studio コンソールからアプリケーションを一覧表示する	35
EMR Studio コンソールからアプリケーションを管理する	35
の使用 AWS CLI	36
アプリケーションの設定	37
アプリケーションの動作	37
EMR Serverless でアプリケーションを操作するための事前初期化された容量	39
デフォルトのアプリケーション設定	42
イメージのカスタマイズ	49
前提条件	38
ステップ 1: EMR Serverless ベースイメージからカスタムイメージを作成する	50
ステップ 2: イメージをローカルで検証する	51
ステップ 3: イメージを Amazon ECR リポジトリにアップロードする	52
ステップ 4: カスタムイメージを使用してアプリケーションを作成または更新する	52
ステップ 5: EMR Serverless がカスタムイメージリポジトリにアクセスできるようにする	54
考慮事項と制限事項	55
データに接続するための EMR Serverless アプリケーションの VPC アクセスの設定	55
アプリケーションを作成	56
アプリケーションの設定	59
サブネット計画の作成に関するベストプラクティス	60
アーキテクチャオプション	61
x86_64 アーキテクチャの使用	62
arm64 アーキテクチャの使用 (Graviton)	62
Graviton で新しいアプリケーションを起動する	62
既存のアプリケーションを Graviton に変換する	63
考慮事項	64
ジョブの同時実行とキューイング	64
同時実行とキューイングの主な利点	64
同時実行とキューイングの開始方法	65
同時実行とキューイングに関する考慮事項	66
データのアップロード	67
前提条件	67
S3 Express One Zone の使用を開始する	68
ジョブの実行	70
ジョブ実行状態	70
猶予期間を含むジョブ実行のキャンセル	72

バッチジョブの猶予期間	72
ストリーミングジョブの猶予期間	74
考慮事項	55
EMR Studio コンソールの使用	77
ジョブを送信する	77
ジョブ実行へのアクセス	79
の使用 AWS CLI	80
実行 IAM ポリシー。	81
開始方法	81
CLI コマンドの例	82
重要な注意事項	83
ポリシーの交差	84
シャッフル最適化ディスクの使用	87
主な利点	87
開始方法	87
サーバーレスストレージの使用	91
主な利点	91
開始方法	92
考慮事項と制限事項	94
サポートされている AWS リージョン	94
継続的にストリーミングされたデータを処理するためのストリーミングジョブ	94
考慮事項と制限事項	96
開始方法	97
ストリーミングコネクタ	97
ログ管理	100
EMR Serverless ジョブ実行時の Spark 設定の使用	101
Spark パラメータ	101
Spark のプロパティ	104
リソース設定のベストプラクティス	110
Spark の例	111
EMR Serverless ジョブ実行時の Hive 設定の使用	112
Hive パラメータ	112
Hive プロパティ	114
Hive の例	127
ジョブの耐障害性	128
再試行ポリシーを使用してジョブをモニタリングする	131

再試行ポリシーを使用したログ記録	131
EMR Serverless のメタストア設定	132
AWS Glue データカタログをメタストアとして使用する	132
外部の Hive メタストアの使用	137
EMR Serverless AWS での Glue マルチカタログ階層の使用	142
外部メタストアを使用する場合の考慮事項	144
クロスアカウント S3 アクセス	144
前提条件	144
S3 バケットポリシーを使用する	145
引き受けたロールを使用する	146
引き受けたロールの例	149
エラーのトラブルシューティング	153
エラー: アカウントが同時に使用できる最大 vCPU のサービス制限に達したため、ジョブは失敗しました。	154
エラー: アプリケーションが maximumCapacity 設定を超えたため、ジョブが失敗しました。	154
エラー: アプリケーションが maximumCapacity を超え、ワーカーを割り当てることができなかったため、ジョブが失敗しました。	154
エラー: S3 アクセスが拒否されました。必要な S3 リソースのジョブランタイムロールの S3 アクセス許可を確認してください。	154
エラー: ModuleNotFoundError: <module> という名前のモジュールがありません。EMR Serverless で Python ライブラリを使用する方法については、ユーザーガイドを参照してください。	154
エラー: 実行ロール <role name> が存在しないか、必要な信頼関係で設定されていないため、引き受けることができませんでした。	155
ジョブレベルのコスト配分	155
デフォルトの 動作	155
機能を有効または無効にする方法	155
考慮事項と制約事項	156
インタラクティブワークロードの実行	157
概要:	157
前提条件	157
アクセス許可	158
設定	159
考慮事項	159
Apache Livy エンドポイントを介したインタラクティブワークロードの実行	161

前提条件	161
必要なアクセス許可	161
開始方法	163
考慮事項	170
ログ記録とモニタリング	172
ログの保存	172
マネージドストレージ	173
Amazon S3	174
Amazon CloudWatch	176
ログのローテーション	179
ログの暗号化	181
マネージドストレージ	181
Amazon S3 バケット	181
Amazon CloudWatch	181
必要なアクセス許可	182
Log4j2 の設定	186
Log4j2 と Spark	186
モニタリング	190
アプリケーションとジョブ	191
Spark エンジンメトリクス	199
使用状況メトリクス	204
EventBridge による自動化	205
EMR Serverless EventBridge のイベント例	206
リソースのタグ付け	210
タグとは	210
リソースのタグ付け	210
タグ付けに関する制限事項	211
タグの操作	212
チュートリアル	214
Java 17 の使用	214
JAVA_HOME	214
spark-defaults	215
Hudi の使用	216
Iceberg の使用	217
Python ライブラリの使用	218
ネイティブ Python 機能の使用	218

Python 仮想環境の構築	219
Python ライブラリを使用するように PySpark ジョブを設定する	220
さまざまな Python バージョンの使用	221
Delta Lake OSS の使用	223
Amazon EMR バージョン 6.9.0 以降	223
Amazon EMR バージョン 6.8.0 以前	224
Airflow からのジョブの送信	225
Hive ユーザー定義関数の使用	227
カスタムイメージの使用	229
カスタム Python バージョンを使用する	230
カスタム Java バージョンを使用する	230
データサイエンスイメージを構築する	231
Apache Sedona による地理空間データの処理	231
カスタムイメージを使用するためのライセンス情報	232
Amazon Redshift での Spark の使用	232
Spark アプリケーションの起動	233
Amazon Redshift の認証	234
Amazon Redshift に対する読み書き	237
考慮事項	239
DynamoDB への接続	240
ステップ 1: Amazon S3 にアップロードする	240
ステップ 2: Hive テーブルを作成する	241
ステップ 3: DynamoDB にコピーする	242
ステップ 4: DynamoDB からクエリする	244
クロスアカウントアクセスのセットアップ	245
考慮事項	247
セキュリティ	250
セキュリティのベストプラクティス	251
最小特権の原則を適用する	251
信頼できないアプリケーションコードを分離する	251
ロールベースのアクセスコントロール (RBAC) の許可	251
データ保護	251
保管中の暗号化	252
転送中の暗号化	255
Identity and Access Management (IAM)	255
オーディエンス	256

アイデンティティを使用した認証	256
ポリシーを使用したアクセスの管理	258
EMR Serverless が IAM と連携する仕組み	259
サービスにリンクされたロールの使用	265
Amazon EMR Serverless のジョブランタイムロール	270
ユーザーアクセスポリシー	273
タグベースのアクセスコントロールのポリシー	278
アイデンティティベースのポリシー	282
ポリシーの更新	284
トラブルシューティング	285
信頼できる ID の伝播	288
概要:	288
機能とメリット	288
仕組み	289
Trusted-Identity の伝播の開始方法	289
インタラクティブワークロードの信頼できる ID の伝播	293
ユーザーバックグラウンドセッション	294
EMR Serverless Trusted-Identity-Propagation 統合に関する考慮事項	298
EMR Serverless での Lake Formation の使用	299
機能を利用できるリージョン	299
EMR Serverless の Lake Formation フルテーブルアクセス	300
FGAC のための Lake Formation	316
ワーカー間の暗号化	347
EMR Serverless での相互 TLS 暗号化の有効化	347
KMS CMK によるディスク暗号化	348
暗号化コンテキストの使用	348
カスターマネージドキーを使用したディスク暗号化の設定	349
ディスク暗号化に必要なアクセス許可	351
キー使用状況のモニタリング	354
詳細はこちら	356
データ保護のための Secrets Manager	356
シークレットの仕組み	357
シークレットを作成する	357
シークレトリファレンスを指定する	357
シークレットにアクセス権を付与する	360
シークレットをローテーションする	362

データアクセスコントロールの S3 Access Grants	362
概要:	362
アプリケーションの起動	363
考慮事項	364
ログ記録用 CloudTrail	364
CloudTrail での EMR Serverless の情報	365
EMR Serverless のログファイルエントリの理解	366
コンプライアンス検証	367
耐障害性	368
インフラストラクチャセキュリティ	369
設定と脆弱性の分析	369
エンドポイントとクォータ	370
サービスエンドポイント	370
Service Quotas	375
API の制限	376
その他の考慮事項	55
リリースバージョン	380
AWS runtime for Apache Spark (emr-spark-8.0-preview)	381
EMR Serverless 7.12.0	382
EMR Serverless 7.11.0	383
EMR Serverless 7.10.0	384
EMR Serverless 7.9.0	384
EMR Serverless 7.8.0	385
EMR Serverless 7.7.0	385
EMR Serverless 7.6.0	385
EMR Serverless 7.5.0	386
EMR Serverless 7.4.0	386
EMR Serverless 7.3.0	386
EMR Serverless 7.2.0	387
EMR Serverless 7.1.0	388
EMR Serverless 7.0.0	388
EMR Serverless 6.15.0	388
EMR Serverless 6.14.0	389
EMR Serverless 6.13.0	389
EMR Serverless 6.12.0	389
EMR Serverless 6.11.0	390

EMR Serverless 6.10.0	390
EMR Serverless 6.9.0	391
EMR Serverless 6.8.0	392
EMR Serverless 6.7.0	392
エンジン固有の変更	392
EMR Serverless 6.6.0	393
ドキュメント履歴	395
.....	CCCXCVii

Amazon EMR Serverless とは

Amazon EMR Serverless は、サーバーレスランタイム環境を備えた Amazon EMR のデプロイオプションです。これにより、Apache Spark や Apache Hive などの最新のオープンソースフレームワークを使用する分析アプリケーションの運用が簡素化されます。EMR Serverless では、これらのフレームワークでアプリケーションを実行するためにクラスターを設定、最適化、保護、または操作する必要はありません。

EMR Serverless は、データ処理ジョブのリソースの過剰プロビジョニングや過少プロビジョニングを回避するのにサポートします。EMR Serverless は、アプリケーションが必要とするリソースを自動的に判別し、ジョブを処理するためにこれらのリソースを取得して、ジョブが終了するとリソースを解放します。インタラクティブデータ分析など、アプリケーションが数秒以内にレスポンスを必要とするユースケースでは、アプリケーションの作成時にアプリケーションが必要とするリソースを事前に初期化できます。

EMR Serverless を使用することにより、オープンソースの互換性、同時実行性、一般的なフレームワークのランタイムパフォーマンスの最適化など、Amazon EMR のメリットを引き続き享受できます。

EMR Serverless は、オープンソースフレームワークを使用してアプリケーションを簡単に操作することを必要とするお客様に適しています。ジョブの迅速な起動、容量の自動管理、容易なコスト管理を実現します。

概念

このセクションでは、EMR Serverless ユーザーガイド全般に記載されている EMR Serverless の用語と概念について説明します。

リリースバージョン

Amazon EMR リリースは、ビッグデータエコシステムの一連のオープンソースアプリケーションです。各リリースには、EMR Serverless がアプリケーションを実行できるようにデプロイおよび設定するために選択するさまざまなビッグデータアプリケーション、コンポーネント、および機能が含まれています。アプリケーションを作成するときにリリースバージョンを指定します。アプリケーションで使用する Amazon EMR リリースバージョンとオープンソースフレームワークバージョンを選択します。リリースバージョンの詳細については、「[Amazon EMR Serverless のリリースバージョン](#)」を参照してください。

アプリケーション

EMR Serverless を使用すると、オープンソース分析フレームワークを使用する 1 つ以上の EMR Serverless アプリケーションを作成できます。アプリケーションを作成するには、次の属性を指定します。

- 使用するオープンソースフレームワークバージョンの Amazon EMR リリースバージョン。リリースバージョンを確認するには、「[Amazon EMR Serverless のリリースバージョン](#)」を参照してください。
- Apache Spark や Apache Hive など、アプリケーションで使用する特定のランタイム。

アプリケーションを作成することにより、データ処理ジョブまたは対話型リクエストをアプリケーションに送信します。

各 EMR Serverless アプリケーションは、他のアプリケーションとは厳密に分離された状態で、安全な Amazon Virtual Private Cloud (VPC) で実行されます。さらに、AWS Identity and Access Management (IAM) ポリシーを使用して、アプリケーションにアクセスできるユーザーとロールを定義します。また、アプリケーションによって発生した使用コストを制御および追跡するための制限を指定することもできます。

以下のことを行う必要がある場合は、複数のアプリケーションを作成することを検討してください:

- さまざまなオープンソースフレームワークを使用する
- さまざまなユースケースに異なるバージョンのオープンソースフレームワークを使用する
- あるバージョンから別のバージョンにアップグレードするときに A/B テストを実施する
- テストシナリオと本番シナリオ用に個別の論理環境を維持する
- 独立したコスト管理と使用状況の追跡により、異なるチームに個別の論理環境を用意する
- 異なる基幹業務アプリケーションを分離する

EMR Serverless は、リージョン内の複数のアベイラビリティゾーン間でワークロードを実行する方法を簡素化するリージョンサービスです。EMR Serverless でアプリケーションを使用する方法の詳細については、「[EMR Serverless アプリケーションを操作して設定する](#)」を参照してください。

ジョブ実行

ジョブ実行は、EMR Serverless アプリケーションに送信されるリクエストであり、アプリケーションはそれらのリクエストを非同期的に実行し、完了まで追跡します。ジョブの例としては、Apache

Hive アプリケーションに送信する HiveQL クエリ、または Apache Spark アプリケーションに送信する PySpark データ処理スクリプトなどがあります。ジョブを送信するときは、ジョブが Amazon S3 オブジェクトなどの AWS リソースにアクセスするために使用する IAM で作成されたランタイムロールを指定する必要があります。アプリケーションに複数のジョブ実行リクエストを送信でき、ジョブ実行ごとに異なるランタイムロールを使用して AWS リソースにアクセスできます。EMR Serverless アプリケーションは、ジョブを受信したら直ちにジョブの実行を開始し、複数のジョブリクエストを同時に実行します。EMR Serverless がジョブを実行する方法の詳細については、「[ジョブの実行](#)」を参照してください。

ワーカー

EMR Serverless アプリケーションは、ワーカーを内部的に使用してワークロードを実行します。これらのワーカーのデフォルトサイズは、アプリケーションタイプと Amazon EMR リリースバージョンに基づいています。ジョブの実行をスケジュールすることにより、これらのサイズをオーバーライドします。

ジョブを送信すると、EMR Serverless はアプリケーションがジョブに必要なとするリソースを計算し、ワーカーをスケジュールします。EMR Serverless は、ワークロードをタスクに分割し、イメージをダウンロードして、ワーカーのプロビジョニングとセットアップを行い、ジョブが終了したら廃止します。EMR Serverless は、ジョブのすべての段階で必要なワークロードと並列処理に基づいて、ワーカーを自動的にスケールアップまたはスケールダウンします。この自動スケーリングにより、アプリケーションがワークロードを実行するために必要なワーカーの数をユーザーが見積もる必要がなくなります。

事前初期化された容量

EMR Serverless は、ワーカーが初期化され、数秒で応答できる状態に保つための事前初期化された容量機能を備えています。この容量によって、アプリケーションのワーカーのウォームプールが効果的に作成されます。各アプリケーションにこの機能を設定するには、アプリケーションの `initial-capacity` パラメータを設定します。事前初期化された容量を設定すると、ジョブが直ちに開始され、反復アプリケーションと時間的制約のあるジョブを実装できます。事前初期化されたワーカーの詳細については、「[EMR Serverless を使用する際のアプリケーションの設定](#)」を参照してください。

EMR Studio

EMR Studio は、EMR Serverless アプリケーションの管理のためのユーザーコンソールです。最初の EMR Serverless アプリケーションの作成時に EMR Studio がアカウントに存在しない場合は、

自動的に作成されます。Amazon EMR コンソールから EMR Studio にアクセスするか、IAM または IAM Identity Center を介して ID プロバイダー (IdP) からのフェデレーティッドアクセスを有効にします。これを行うと、ユーザーは Studio にアクセスし、Amazon EMR コンソールに直接アクセスすることなく EMR Serverless アプリケーションを管理できます。EMR Serverless アプリケーションと EMR Studio の連携の詳細については、「[EMR Studio コンソールからの EMR Serverless アプリケーションの作成](#)」および「[EMR Studio コンソールからのジョブの実行](#)」を参照してください。

EMR Serverless の使用を開始するための前提条件

このセクションでは、EMR Serverless を実行するための管理上の前提条件について説明します。これには、アカウント設定とアクセス許可の管理が含まれます。

トピック

- [にサインアップする AWS アカウント](#)
- [管理アクセスを持つユーザーを作成する](#)
- [許可を付与する](#)
- [のインストールと設定 AWS CLI](#)
- [コンソールを開きます。](#)

にサインアップする AWS アカウント

がない場合は AWS アカウント、次の手順を実行して作成します。

にサインアップするには AWS アカウント

1. <https://portal.aws.amazon.com/billing/signup> を開きます。
2. オンラインの手順に従います。

サインアップ手順の一環として、電話またはテキストメッセージを受け取り、電話キーパッドで検証コードを入力します。

にサインアップすると AWS アカウント、AWS アカウントのルートユーザー が作成されます。ルートユーザーには、アカウントのすべての AWS のサービス とリソースへのアクセス権があります。セキュリティベストプラクティスとして、ユーザーに管理アクセス権を割り当て、[ルートユーザーアクセスが必要なタスク](#)の実行にはルートユーザーのみを使用するようにしてください。

AWS サインアッププロセスが完了すると、 から確認メールが送信されます。<https://aws.amazon.com/> の [マイアカウント] をクリックして、いつでもアカウントの現在のアクティビティを表示し、アカウントを管理することができます。

管理アクセスを持つユーザーを作成する

にサインアップしたら AWS アカウント、日常的なタスクにルートユーザーを使用しないように AWS アカウントのルートユーザー、を保護し AWS IAM アイデンティティセンター、を有効にして管理ユーザーを作成します。

を保護する AWS アカウントのルートユーザー

1. ルートユーザーを選択し、AWS アカウント E メールアドレスを入力して、アカウント所有者[AWS マネジメントコンソール](#)としてサインインします。次のページでパスワードを入力します。

ルートユーザーを使用してサインインする方法については、「AWS サインイン ユーザーガイド」の「[ルートユーザーとしてサインインする](#)」を参照してください。

2. ルートユーザーの多要素認証 (MFA) を有効にします。

手順については、IAM [ユーザーガイドの AWS アカウント「ルートユーザー \(コンソール\) の仮想 MFA デバイス](#)を有効にする」を参照してください。

管理アクセスを持つユーザーを作成する

1. IAM アイデンティティセンターを有効にします。

手順については、「AWS IAM アイデンティティセンター ユーザーガイド」の「[AWS IAM アイデンティティセンターの有効化](#)」を参照してください。

2. IAM アイデンティティセンターで、ユーザーに管理アクセスを付与します。

を ID ソース IAM アイデンティティセンターディレクトリとして使用方法のチュートリアルについては、AWS IAM アイデンティティセンター「ユーザーガイド」の「[デフォルトを使用してユーザーアクセスを設定する IAM アイデンティティセンターディレクトリ](#)」を参照してください。

管理アクセス権を持つユーザーとしてサインインする

- IAM アイデンティティセンターのユーザーとしてサインインするには、IAM アイデンティティセンターのユーザーの作成時に E メールアドレスに送信されたサインイン URL を使用します。

IAM Identity Center ユーザーを使用してサインインする方法については、AWS サインイン「ユーザーガイド」の [AWS「アクセスポータルにサインインする」](#) を参照してください。

追加のユーザーにアクセス権を割り当てる

1. IAM アイデンティティセンターで、最小特権のアクセス許可を適用するというベストプラクティスに従ったアクセス許可セットを作成します。

手順については、「AWS IAM アイデンティティセンター ユーザーガイド」の「[アクセス許可セットを作成する](#)」を参照してください。

2. グループにユーザーを割り当て、そのグループにシングルサインオンアクセス権を割り当てます。

手順については、「AWS IAM アイデンティティセンター ユーザーガイド」の「[グループを追加する](#)」を参照してください。

許可を付与する

本番環境では、よりきめ細かなポリシーを使用することを提案します。そのようなポリシーの例については、「[EMR Serverless のユーザーアクセスポリシーの例](#)」を参照してください。アクセス管理の詳細については、IAM ユーザーガイドの [AWS「リソースのアクセス管理」](#) を参照してください。

サンドボックス環境で EMR Serverless の使用を開始する必要があるユーザーには、次のようなポリシーを使用します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EMRStudioCreate",
      "Effect": "Allow",
      "Action": [
        "elasticmapreduce:CreateStudioPresignedUrl",
        "elasticmapreduce:DescribeStudio",
        "elasticmapreduce:CreateStudio",
        "elasticmapreduce:ListStudios"
      ]
    }
  ]
}
```

```

    ],
    "Resource": [
        "*"
    ]
},
{
    "Sid": "EMRServerlessFullAccess",
    "Effect": "Allow",
    "Action": [
        "emr-serverless:*"
    ],
    "Resource": [
        "*"
    ]
},
{
    "Sid": "AllowEC2ENICreationWithEMRTags",
    "Effect": "Allow",
    "Action": [
        "ec2:CreateNetworkInterface"
    ],
    "Resource": [
        "arn:aws:ec2:*:*:network-interface/*"
    ],
    "Condition": {
        "StringEquals": {
            "aws:CalledViaLast": "ops.emr-serverless.amazonaws.com"
        }
    }
},
{
    "Sid": "AllowEMRServerlessServiceLinkedRoleCreation",
    "Effect": "Allow",
    "Action": [
        "iam:CreateServiceLinkedRole"
    ],
    "Resource": [
        "arn:aws:iam:*:*:role/aws-service-role/*"
    ]
}
]
}

```

アクセスを提供するには、ユーザー、グループ、またはロールにアクセス許可を追加します。

- 以下のユーザーとグループ AWS IAM アイデンティティセンター:

アクセス許可セットを作成します。「AWS IAM アイデンティティセンター ユーザーガイド」の「[アクセス許可セットを作成する](#)」の手順に従ってください。

- IAM 内で、ID プロバイダーによって管理されているユーザー:

ID フェデレーションのロールを作成します。詳細については IAM ユーザーガイド の [サードパーティー ID プロバイダー \(フェデレーション\) 用のロールを作成する](#) を参照してください。

- IAM ユーザー:

- ユーザーが担当できるロールを作成します。手順については IAM ユーザーガイド の [IAM ユーザーのロールの作成](#) を参照してください。
- (お奨めできない方法) ポリシーをユーザーに直接アタッチするか、ユーザーをユーザーグループに追加します。IAM ユーザーガイド の [ユーザー \(コンソール\) へのアクセス許可の追加](#) の指示に従います。

プログラマチックアクセス権を付与する

ユーザーが の AWS 外部とやり取りする場合は、プログラムによるアクセスが必要です AWS マネジメントコンソール。プログラムによるアクセスを許可する方法は、 がアクセスするユーザーのタイプによって異なります AWS。

ユーザーにプログラムによるアクセス権を付与するには、以下のいずれかのオプションを選択します。

プログラムによるアクセス権を必要とするユーザー	目的	方法
IAM	(推奨) コンソール認証情報を一時的な認証情報として使用して AWS CLI、AWS SDKs、または AWS APIs。	使用するインターフェイスの指示に従ってください。 • については AWS CLI、AWS Command Line Interface「ユーザーガイド」の AWS「ローカル開発用ログイン」 を参照してください。

プログラムによるアクセス権を必要とするユーザー	目的	方法
		• AWS SDKs 「SDK およびツールリファレンスガイド」の「Login for AWS local development」を参照してください。AWS SDKs
ワークフォースアイデンティティ (IAM アイデンティティセンターで管理されているユーザー)	一時的な認証情報を使用して AWS CLI、AWS SDKs、または AWS APIs。	使用するインターフェイスの指示に従ってください。 • については AWS CLI、「AWS Command Line Interface ユーザーガイド」の「使用する AWS CLI ように AWS IAM アイデンティティセンターを設定する」を参照してください。 • AWS SDKs、ツール、API については、AWS APIs 「SDK およびツールリファレンスガイド」の「IAM アイデンティティセンター認証」を参照してください。AWS SDKs
IAM	一時的な認証情報を使用して AWS CLI、AWS SDKs、または AWS APIs。	「IAM ユーザーガイド」の「AWS リソースで一時的な認証情報の使用」の手順に従います。

プログラムによるアクセス権を必要とするユーザー	目的	方法
IAM	(非推奨) 長期認証情報を使用して、AWS CLI、AWS SDKs、または AWS APIs。	使用するインターフェイスの指示に従ってください。 • については AWS CLI、「AWS Command Line Interface ユーザーガイド」の「IAM ユーザー認証情報を使用した認証」を参照してください。 • AWS SDKs 「SDK とツールリファレンスガイド」の「長期認証情報を使用した認証」を参照してください。 AWS SDKs • API AWS APIs 「IAM ユーザーのアクセスキーの管理」を参照してください。

のインストールと設定 AWS CLI

EMR Serverless API を使用する場合は、AWS Command Line Interface (AWS CLI) の最新バージョンをインストールします。EMR Studio コンソールから EMR Serverless を使用し、「」の手順に従って CLI を使用せずに開始 AWS CLI するには、必要ありません[コンソールから EMR Serverless の使用を開始する](#)。

を設定するには AWS CLI

1. macOS、Linux、または Windows AWS CLI 用の の最新バージョンをインストールするには、「[の最新バージョンのインストールまたは更新 AWS CLI](#)」を参照してください。
2. EMR Serverless を含む へのアクセスの AWS CLI と安全なセットアップを設定するには AWS のサービス、「[を使用したクイック設定aws configure](#)」を参照してください。
3. 設定を確認するには、コマンドプロンプトで次の DataBrew コマンドを入力します。

```
aws emr-serverless help
```

AWS CLI コマンドは、パラメータまたはプロファイルで設定しない限り、設定 AWS リージョンのデフォルトを使用します。パラメータ AWS リージョン を使用して を設定するには、各コマンドに `--region` パラメータを追加します。

プロファイル AWS リージョン で を設定するには、まず ファイルまたは `~/.aws/config` ファイル `%UserProfile%/.aws/config` (Microsoft Windows の場合) に名前付きプロファイルを追加します。「[AWS CLI の名前付きプロファイル](#)」のステップに従います。次に、次の例のようなコマンドを使用して、AWS リージョン およびその他の設定を行います。

```
[profile emr-serverless]
aws_access_key_id = ACCESS-KEY-ID-OF-IAM-USER
aws_secret_access_key = SECRET-ACCESS-KEY-ID-OF-IAM-USER
region = us-east-1
output = text
```

コンソールを開きます。

このセクションに記載されているコンソール指向のトピックのほとんどは、[Amazon EMR コンソール](#)を起点としています。にまだサインインしていない場合は AWS アカウント、サインインして [Amazon EMR コンソール](#)を開き、次のセクションに進み、Amazon EMR の使用を開始します。

Amazon EMR Serverless の使用を開始する

このチュートリアルは、サンプルの Spark または Hive ワークロードをデプロイするときに EMR Serverless の使用を開始するために活用できます。独自のアプリケーションを作成、実行、デバッグします。このチュートリアルのほとんどの部分でデフォルトのオプションを示します。

EMR Serverless アプリケーションを起動する前に、以下のタスクを完了してください。

トピック

- [EMR Serverless を使用するためのアクセス許可を付与する](#)
- [EMR Serverless 用のストレージを準備する](#)
- [EMR Studio を作成してインタラクティブワークロードを実行する](#)
- [ジョブランタイムロールを作成する](#)
- [コンソールから EMR Serverless の使用を開始する](#)
- [の開始方法 AWS CLI](#)

EMR Serverless を使用するためのアクセス許可を付与する

EMR Serverless を使用するには、EMR Serverless のアクセス許可を付与するポリシーがアタッチされたユーザーまたは IAM ロールが必要です。ユーザーを作成し、作成したユーザーに適切なポリシーをアタッチするには、「[許可を付与する](#)」の手順に従います。

EMR Serverless 用のストレージを準備する

このチュートリアルでは、S3 バケットを使用して、EMR Serverless アプリケーションを使用して実行するサンプルの Spark または Hive ワークロードからの出力ファイルとログを保存します。バケットを作成するには、「Amazon Simple Storage Service コンソールユーザーガイド」の「[バケットの作成](#)」の手順に従います。*amzn-s3-demo-bucket* への追加の参照は、新しく作成されたバケットの名前に置き換えます。

EMR Studio を作成してインタラクティブワークロードを実行する

EMR Serverless を使用して EMR Studio でホストされているノートブックを介してインタラクティブクエリを実行する場合は、ワークスペースを作成するには、S3 バケットと [EMR Serverless の最小サービスロール](#)を指定する必要があります。セットアップの手順については、「Amazon EMR 管

理ガイド」の「[EMR Studio の設定](#)」を参照してください。インタラクティブワークロードの詳細については、「[EMR Studio を使用して EMR Serverless でインタラクティブワークロードを実行する](#)」を参照してください。

ジョブランタイムロールを作成する

EMR Serverless で実行されるジョブは、実行時に特定の AWS のサービス およびリソースに詳細なアクセス許可を提供するランタイムロールを使用します。このチュートリアルでは、パブリック S3 バケットがデータとスクリプトをホストします。バケット `amzn-s3-demo-bucket` は出力を保存します。

ジョブランタイムロールを設定するには、まず信頼ポリシーを使用してランタイムロールを作成し、EMR Serverless が新しいロールを使用できるようにします。次に、必要な S3 アクセスポリシーを対象のロールにアタッチします。次の手順ではこのプロセスについて説明します。

Console

1. <https://console.aws.amazon.com/iam/> で IAM コンソールに移動します。
2. 左のナビゲーションペインの [ポリシー] を選択します。
3. [ポリシーを作成] を選択します。
4. 新しいタブで [ポリシーの作成] ページが開きます。ポリシーエディタを Json として選択し、以下のポリシー JSON を貼り付けます。

Important

以下のポリシーの `amzn-s3-demo-bucket` を [EMR Serverless 用のストレージを準備する](#) で作成された実際のバケット名に置き換えます。これは S3 アクセスの基本ポリシーです。ジョブランタイムロールのその他の例については、「[Amazon EMR Serverless のジョブランタイムロール](#)」を参照してください。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ReadAccessForEMRSamples",
```

```

    "Effect": "Allow",
    "Action": [
        "s3:GetObject",
        "s3:ListBucket"
    ],
    "Resource": [
        "arn:aws:s3::*.elasticmapreduce",
        "arn:aws:s3::*.elasticmapreduce/*"
    ]
},
{
    "Sid": "FullAccessToOutputBucket",
    "Effect": "Allow",
    "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:DeleteObject"
    ],
    "Resource": [
        "arn:aws:s3:::amzn-s3-demo-bucket",
        "arn:aws:s3:::amzn-s3-demo-bucket/*"
    ]
},
{
    "Sid": "GlueCreateAndReadDataCatalog",
    "Effect": "Allow",
    "Action": [
        "glue:GetDatabase",
        "glue:CreateDatabase",
        "glue:GetDataBases",
        "glue:CreateTable",
        "glue:GetTable",
        "glue:UpdateTable",
        "glue:DeleteTable",
        "glue:GetTables",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:CreatePartition",
        "glue:BatchCreatePartition",
        "glue:GetUserDefinedFunctions"
    ],
    "Resource": [
        "*"
    ]
}

```

```
    ]
  }
]
}
```

5. [次へ] を選択して、「EMRServerlessS3AndGlueAccessPolicy」や「Create policy」などのポリシーの名前を入力します
6. IAM コンソールのナビゲーションペインで [ロール] をクリックします。
7. [ロールの作成] を選択してください。
8. ロールタイプには、[カスタム信頼ポリシー] を選択し、以下の信頼ポリシーを貼り付けます。これにより、Amazon EMR Serverless アプリケーションに送信されたジョブが AWS のサービス ユーザーに代わって他の にアクセスできるようになります。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sts:AssumeRole"
      ],
      "Resource": "arn:aws:iam::123456789012:role/EMRServerlessExecutionRole",
      "Sid": "AllowSTSAssumerole"
    }
  ]
}
```

9. [次へ] を選択して [アクセス許可の追加] ページに移動し、[EMRServerlessS3AndGlueAccessPolicy] を選択します。
10. [名前、確認、および作成] ページで、[ロール名] にロールの名前 (例: EMRServerlessS3RuntimeRole) を入力します。この IAM ロールを作成するには、[ロールの作成] を選択します。

CLI

1. IAM ロールに使用する信頼ポリシーが含まれている `emr-serverless-trust-policy.json` という名前のファイルを作成します。ファイルには次のポリシーが含まれます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EMRServerlessTrustPolicy",
      "Action": [
        "sts:AssumeRole"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:iam::123456789012:role/EMRServerlessExecutionRole"
    }
  ]
}
```

2. IAM ロール (`EMRServerlessS3RuntimeRole`) を作成します。前のステップで作成した信頼ポリシーを使用します。

```
aws iam create-role \
  --role-name EMRServerlessS3RuntimeRole \
  --assume-role-policy-document file://emr-serverless-trust-policy.json
```

出力の ARN に注目してください。ジョブの送信中に新しいロールの ARN を使用します。この後、*job-role-arn* と呼ばれます。

3. ワークロードの IAM ポリシーを定義する `emr-sample-access-policy.json` という名前のファイルを作成します。これにより、パブリック S3 バケットに保存されているスクリプトとデータへの読み取りアクセス権と、*amzn-s3-demo-bucket* への読み取り/書き込みアクセス権が付与されます。

⚠ Important

以下のポリシーの *amzn-s3-demo-bucket* を [EMR Serverless 用のストレージを準備する](#) で作成された実際のバケット名に置き換えます。これは Glue AWS および S3 アクセスの基本ポリシーです。ジョブランタイムロールのその他の例については、「[Amazon EMR Serverless のジョブランタイムロール](#)」を参照してください。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ReadAccessForEMRSamples",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3::*.elasticmapreduce",
        "arn:aws:s3::*.elasticmapreduce/*"
      ]
    },
    {
      "Sid": "FullAccessToOutputBucket",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:DeleteObject"
      ],
      "Resource": [
        "arn:aws:s3::amzn-s3-demo-bucket",
        "arn:aws:s3::amzn-s3-demo-bucket/*"
      ]
    }
  ]
}
```

```

    "Sid": "GlueCreateAndReadDataCatalog",
    "Effect": "Allow",
    "Action": [
        "glue:GetDatabase",
        "glue:CreateDatabase",
        "glue:GetDataBases",
        "glue:CreateTable",
        "glue:GetTable",
        "glue:UpdateTable",
        "glue:DeleteTable",
        "glue:GetTables",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:CreatePartition",
        "glue:BatchCreatePartition",
        "glue:GetUserDefinedFunctions"
    ],
    "Resource": [
        "*"
    ]
  }
]
}

```

4. ステップ 3 で作成したポリシーファイルを使用し、EMRServerlessS3AndGlueAccessPolicy という名前の IAM ポリシーを作成します。次のステップで新しいポリシーの ARN を使用するため、出力の ARN を書き留めます。

```

aws iam create-policy \
  --policy-name EMRServerlessS3AndGlueAccessPolicy \
  --policy-document file://emr-sample-access-policy.json

```

出力の新しいポリシーの ARN を書き留めます。次のステップで *policy-arn* に置き換えます。

5. IAM ポリシー EMRServerlessS3AndGlueAccessPolicy をジョブランタイムロール EMRServerlessS3RuntimeRole にアタッチします。

```

aws iam attach-role-policy \
  --role-name EMRServerlessS3RuntimeRole \
  --policy-arn policy-arn

```

コンソールから EMR Serverless の使用を開始する

このセクションでは、EMR Studio の作成など、EMR Serverless の操作について説明します。また、ジョブ実行を送信し、ログを表示する方法についても説明します。

完了すべきステップ

- [ステップ 1: EMR Serverless アプリケーションを作成する](#)
- [ステップ 2: ジョブ実行またはインタラクティブワークロードを送信する](#)
- [ステップ 3: アプリケーション UI とログを表示する](#)
- [ステップ 4: クリーンアップする](#)

ステップ 1: EMR Serverless アプリケーションを作成する

EMR Serverless を使用して、次のように新しいアプリケーションを作成します。

1. にサインイン AWS マネジメントコンソール し、<https://console.aws.amazon.com/emr> で Amazon EMR コンソールを開きます。
 2. 左側のナビゲーションペインで、[EMR Serverless] を選択して EMR Serverless ランディングページに移動します。
 3. EMR Serverless アプリケーションを作成または管理するには、EMR Studio UI が必要です。
 - アプリケーション AWS リージョン を作成する に既に EMR Studio がある場合は、アプリケーションの管理を選択して EMR Studio に移動するか、使用するスタジオを選択します。
 - アプリケーション AWS リージョン を作成する に EMR Studio がない場合は、「Get started」を選択し、「Create and launch Studio」を選択します。EMR Serverless は、アプリケーションを作成および管理できるように EMR Studio を作成します。
 4. 新しいタブで開いた [Studio を作成] UI に、アプリケーションの名前、タイプ、リリースバージョンを入力します。バッチジョブのみを実行する場合は、[バッチジョブのデフォルト設定のみを使用する] を選択します。インタラクティブワークロードの場合は、[インタラクティブワークロードのデフォルト設定を使用する] を選択します。このオプションを使用すると、インタラクティブ対応アプリケーションでバッチジョブを実行することもできます。必要な場合は、これらの設定を後で変更できます。
- 詳細については、「[Studio を作成する](#)」を参照してください。
5. [アプリケーションの作成] を選択して、最初のアプリケーションを作成します。

次のセクション [ステップ 2: ジョブ実行またはインタラクティブワークロードを送信する](#) に進み、ジョブ実行またはインタラクティブワークロードを送信します。

ステップ 2: ジョブ実行またはインタラクティブワークロードを送信する

Spark job run

このチュートリアルでは、PySpark スクリプトを使用して、複数のテキストファイルにわたる一意の単語の出現数を計算します。パブリックで読み取り専用の S3 バケットには、スクリプトとデータセットの両方が保存されます。

Spark ジョブを実行するには

1. 次のコマンドを使用して、サンプルスクリプト `wordcount.py` を新しいバケットにアップロードします。

```
aws s3 cp s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/scripts/wordcount.py s3://amzn-s3-demo-bucket/scripts/
```

2. [ステップ 1: EMR Serverless アプリケーションを作成する](#) を完了すると、EMR Studio の [アプリケーションの詳細] ページに移動します。そこで、[ジョブの送信] オプションを選択します。
3. [ジョブの送信] ページで、以下の内容を行います。
 - [名前] フィールドに、ジョブ実行を呼び出す名前を入力します。
 - [ランタイムロール] フィールドに、[ジョブランタイムロールを作成する](#) で作成したロールの名前を入力します。
 - [スクリプトの場所] フィールドに、S3 URI として `s3://amzn-s3-demo-bucket/scripts/wordcount.py` を入力します。
 - [スクリプト引数] フィールドに、「["`s3://amzn-s3-demo-bucket/emr-serverless-spark/output`"]」と入力します。
 - [Spark プロパティ] セクションで、[テキストとして編集] を選択し、次の設定を入力します。

```
--conf spark.executor.cores=1 --conf spark.executor.memory=4g --  
conf spark.driver.cores=1 --conf spark.driver.memory=4g --conf  
spark.executor.instances=1
```

4. [ジョブの送信] を選択してジョブの実行を開始します。

5. [ジョブ実行] タブには、新しいジョブ実行が [実行中] のステータスで表示されます。

Hive job run

チュートリアルこの部分では、テーブルを作成し、いくつかのレコードを挿入して、カウント集計クエリを実行します。Hive ジョブを実行するには、まず、単一ジョブの一部として実行するすべての Hive クエリを含むファイルを作成し、ファイルを S3 にアップロードして、Hive ジョブを開始するときこの S3 パスを指定します。

Hive ジョブを実行するには

1. Hive ジョブで実行するすべてのクエリを含む hive-query.sql というファイルを作成します。

```
create database if not exists emrserverless;
use emrserverless;
create table if not exists test_table(id int);
drop table if exists Values__Tmp__Table__1;
insert into test_table values (1),(2),(2),(3),(3),(3);
select id, count(id) from test_table group by id order by id desc;
```

2. 次のコマンドを使用して、hive-query.sql を S3 バケットにアップロードします。

```
aws s3 cp hive-query.sql s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-query.sql
```

3. [ステップ 1: EMR Serverless アプリケーションを作成する](#) を完了すると、EMR Studio の [アプリケーションの詳細] ページに移動します。そこで、[ジョブの送信] オプションを選択します。
4. [ジョブの送信] ページで、以下の内容を行います。
 - [名前] フィールドに、ジョブ実行を呼び出す名前を入力します。
 - [ランタイムロール] フィールドに、[ジョブランタイムロールを作成する](#) で作成したロールの名前を入力します。
 - [スクリプトの場所] フィールドに、S3 URI として s3://*amzn-s3-demo-bucket*/emr-serverless-hive/query/hive-query.sql を入力します。
 - [Hive プロパティ] セクションで、[テキストとして編集] を選択し、次の設定を入力します。

```
--hiveconf hive.log.explain.output=false
```

- [ジョブ設定] セクションで、[JSON として編集] を選択し、次の JSON を入力します。

```
{
  "applicationConfiguration":
  [{
    "classification": "hive-site",
    "properties": {
      "hive.exec.scratchdir": "s3://amzn-s3-demo-bucket/emr-
serverless-hive/hive/scratch",
      "hive.metastore.warehouse.dir": "s3://amzn-s3-demo-bucket/emr-
serverless-hive/hive/warehouse",
      "hive.driver.cores": "2",
      "hive.driver.memory": "4g",
      "hive.tez.container.size": "4096",
      "hive.tez.cpu.vcores": "1"
    }
  ]
}
```

5. [ジョブの送信] を選択してジョブの実行を開始します。
6. [ジョブ実行] タブには、新しいジョブ実行が [実行中] のステータスで表示されます。

Interactive workload

Amazon EMR 6.14.0 以降では、EMR Studio でホストされているノートブックを使用して、EMR Serverless で Spark のインタラクティブワークロードを実行できます。アクセス許可や前提条件などの詳細については、「[EMR Studio を使用して EMR Serverless でインタラクティブワークロードを実行する](#)」を参照してください。

アプリケーションを作成し、必要なアクセス許可を設定したら、次の手順を使用して EMR Studio でインタラクティブノートブックを実行します。

1. EMR Studio UI で [Workspaces] タブに移動します。Amazon S3 ストレージの場所と [EMR Studio サービスロール](#) を引き続き設定する必要がある場合は、画面上部のバナーで [スタジオを設定する] ボタンを選択します。
2. ノートブックにアクセスするには、Workspace を選択するか、新しい Workspace を作成します。クイック起動を使用して、新しいタブで Workspace を開きます。

3. 新しく開いたタブに移動します。左側のナビゲーションからコンピューティングアイコンを選択します。EMR Serverless を [コンピューティングタイプ] として選択します。
4. 前のセクションで作成したインタラクティブ対応アプリケーションを選択します。
5. [ランタイムロール] フィールドに、EMR Serverless アプリケーションがジョブ実行のために引き受けることができる IAM ロールの名前を入力します。ランタイムロールの詳細については、「Amazon EMR Serverless ユーザーガイド」の「[Job runtime roles](#)」を参照してください。
6. [アタッチ] を選択します。これには最大 1 分を要する場合があります。ページはアタッチされると更新されます。
7. カーネルを選択し、ノートブックを起動します。EMR Serverless でサンプルノートブックを参照し、Workspace にコピーすることもできます。サンプルノートブックにアクセスするには、左側のナビゲーションの [{...}] メニューに移動し、ノートブックファイル名に serverless があるノートブックを参照します。
8. ノートブックでは、ドライバーログリンクと、ジョブをモニタリングするためのメトリクスを備えたリアルタイムインターフェイスである Apache Spark UI へのリンクにアクセスできます。詳細については、「Amazon EMR Serverless ユーザーガイド」の「[EMR Serverless アプリケーションとジョブのモニタリング](#)」を参照してください。

Studio ワークスペースにアプリケーションをアタッチすると、まだ実行されていない場合、アプリケーションは自動的にトリガーを開始します。また、アプリケーションを事前に起動し、ワークスペースにアタッチする前に準備しておくこともできます。

ステップ 3: アプリケーション UI とログを表示する

アプリケーション UI を表示するには、まずジョブ実行を識別します。Spark UI または Hive Tez UI のオプションは、ジョブタイプに基づいて、対象のジョブ実行のオプションの最初の行で使用できます。適切なオプションを選択します。

Spark UI を選択した場合は、[エグゼキューター] タブを選択してドライバーとエグゼキューターのログを表示します。Hive Tez UI を選択した場合は、[すべてのタスク] タブを選択してログを表示します。

ジョブ実行ステータスが [成功] と表示されたら、S3 バケットでジョブの出力を表示できます。

ステップ 4: クリーンアップする

作成したアプリケーションは 15 分間非アクティブになった後に自動停止されますが、再度使用しないリソースをリリースすることをお勧めします。

アプリケーションを削除するには、[アプリケーションを一覧表示する] ページに移動します。作成したアプリケーションを選択し、[アクション] → [停止] を選択してアプリケーションを停止します。アプリケーションが STOPPED 状態になったら、同じアプリケーションを選択し、[アクション] → [削除] を選択します。

Spark ジョブと Hive ジョブの実行に関するその他の例については、「[EMR Serverless ジョブ実行時の Spark 設定の使用](#)」および「[EMR Serverless ジョブ実行時の Hive 設定の使用](#)」を参照してください。

の開始方法 AWS CLI

コマンドを使用して から EMR Serverless の使用を開始し、アプリケーション AWS CLI の作成、ジョブの実行、ジョブ実行出力の確認、リソースの削除を行います。

ステップ 1: EMR Serverless アプリケーションを作成する

最初の EMR Serverless アプリケーションを構築するには、[emr-serverless create-application](#) コマンドを使用します。使用するアプリケーションバージョンに関連付けられたアプリケーションタイプと Amazon EMR リリースラベルを指定する必要があります。アプリケーションの名前はオプションです。

Spark

Spark SQL アプリケーションを作成するには、次のコマンドを実行します。

```
aws emr-serverless create-application \
  --release-label emr-6.6.0 \
  --type "SPARK" \
  --name my-application
```

Hive

Hive アプリケーションを作成するには、次のコマンドを実行します。

```
aws emr-serverless create-application \  
  --release-label emr-6.6.0 \  
  --type "HIVE" \  
  --name my-application
```

出力で返されるアプリケーション ID を書き留めます。アプリケーションの起動とジョブの送信にこの ID を使用します。以後は *application-id* として参照します。

[ステップ 2: EMR Serverless アプリケーションにジョブ実行を送信する](#) に進む前に、アプリケーションが [get-application](#) API で CREATED 状態に達していることを確認してください。

```
aws emr-serverless get-application \  
  --application-id application-id
```

EMR Serverless は、リクエストされたジョブに対応するワーカーを作成します。デフォルトでは、これらはオンデマンドで作成されますが、アプリケーションの作成時に `initialCapacity` パラメータを設定することで、事前に初期化された容量を指定することもできます。`maximumCapacity` パラメータを使用して、アプリケーションが使用できる最大容量の合計を制限することもできます。これらのオプションの詳細については、「[EMR Serverless を使用する際のアプリケーションの設定](#)」を参照してください。

ステップ 2: EMR Serverless アプリケーションにジョブ実行を送信する

これで、EMR Serverless アプリケーションはジョブを実行する準備が整いました。

Spark

このステップでは、PySpark スクリプトを使用して、複数のテキストファイルにわたる一意の単語の出現数を計算します。パブリックで読み取り専用の S3 バケットには、スクリプトとデータセットの両方が保存されます。アプリケーションは、Spark ランタイムの出力ファイルとログデータを、作成した S3 バケットの `/output` および `/logs` ディレクトリに送信します。

Spark ジョブを実行するには

1. 次のコマンドを使用して、実行するサンプルスクリプトを新しいバケットにコピーします。

```
aws s3 cp s3://us-east-1.elasticmapreduce/emr-containers/samples/wordcount/  
scripts/wordcount.py s3://amzn-s3-demo-bucket/scripts/
```

2. 次のコマンドで、*application-id* をアプリケーション ID に置き換えます。*job-role-arn* を [ジョブランタイムロールを作成する](#) で作成したランタイムロール ARN に置き換えます。*job-run-name* をジョブ実行を呼び出す名前に置き換えます。すべての *amzn-s3-demo-bucket* の文字列を作成した Amazon S3 バケットに置き換え、/output をパスに追加します。これにより、バケットに新しいフォルダが作成され、EMR Serverless はアプリケーションの出力ファイルをコピーできます。

```
aws emr-serverless start-job-run \  
  --application-id application-id \  
  --execution-role-arn job-role-arn \  
  --name job-run-name \  
  --job-driver '{  
    "sparkSubmit": {  
      "entryPoint": "s3://amzn-s3-demo-bucket/scripts/wordcount.py",  
      "entryPointArguments": ["s3://amzn-s3-demo-bucket/emr-serverless-  
spark/output"],  
      "sparkSubmitParameters": "--conf spark.executor.cores=1  
--conf spark.executor.memory=4g --conf spark.driver.cores=1 --conf  
spark.driver.memory=4g --conf spark.executor.instances=1"  
    }  
  }'
```

3. 出力で返されるジョブ実行 ID を書き留めておいてください。次の手順で、*job-run-id* をこの ID に置き換えます。

Hive

このチュートリアルでは、テーブルを作成し、いくつかのレコードを挿入して、カウント集計クエリを実行します。Hive ジョブを実行するには、まず、単一のジョブの一部として実行するすべての Hive クエリを含むファイルを作成し、ファイルを S3 にアップロードして、Hive ジョブを開始するときこの S3 パスを指定します。

Hive ジョブを実行するには

1. Hive ジョブで実行するすべてのクエリを含む hive-query.q1 というファイルを作成します。

```
create database if not exists emrserverless;  
use emrserverless;  
create table if not exists test_table(id int);  
drop table if exists Values__Tmp__Table__1;
```

```
insert into test_table values (1),(2),(2),(3),(3),(3);
select id, count(id) from test_table group by id order by id desc;
```

2. 次のコマンドを使用して、hive-query.q1 を S3 バケットにアップロードします。

```
aws s3 cp hive-query.q1 s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-
query.q1
```

3. 次のコマンドで、*application-id* を独自のアプリケーション ID に置き換えます。*job-role-arn* を [ジョブランタイムロールを作成する](#) で作成したランタイムロール ARN に置き換えます。すべての *amzn-s3-demo-bucket* の文字列を作成した Amazon S3 バケットに置き換え、/output と /logs をパスに追加します。これにより、バケットに新しいフォルダが作成され、EMR Serverless はアプリケーションの出力ファイルとログファイルをコピーできます。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "hive": {
      "query": "s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-
query.q1",
      "parameters": "--hiveconf hive.log.explain.output=false"
    }
  }' \
  --configuration-overrides '{
    "applicationConfiguration": [{
      "classification": "hive-site",
      "properties": {
        "hive.exec.scratchdir": "s3://amzn-s3-demo-bucket/emr-serverless-
hive/hive/scratch",
        "hive.metastore.warehouse.dir": "s3://amzn-s3-demo-bucket/emr-
serverless-hive/hive/warehouse",
        "hive.driver.cores": "2",
        "hive.driver.memory": "4g",
        "hive.tez.container.size": "4096",
        "hive.tez.cpu.vcores": "1"
      }
    }],
    "monitoringConfiguration": {
      "s3MonitoringConfiguration": {
        "logUri": "s3://amzn-s3-demo-bucket/emr-serverless-hive/logs"
```

```
    }  
  }  
}'
```

- 出力で返されるジョブ実行 ID を書き留めておいてください。次の手順で、*job-run-id* をこの ID に置き換えます。

ステップ 3: ジョブ実行の出力を確認する

ジョブの実行には通常、完了までに 3~5 分を要します。

Spark

次のコマンドを使用して、Spark ジョブの状態を確認できます。

```
aws emr-serverless get-job-run \  
  --application-id application-id \  
  --job-run-id job-run-id
```

ログの送信先を `s3://amzn-s3-demo-bucket/emr-serverless-spark/logs` に設定すると、この特定のジョブ実行のログを `s3://amzn-s3-demo-bucket/emr-serverless-spark/logs/applications/application-id/jobs/job-run-id` で確認できます。

Spark アプリケーションの場合、EMR Serverless は 30 秒ごとにイベントログを S3 ログ送信先の `sparklogs` フォルダにプッシュします。ジョブが完了すると、ドライバーとエグゼキューターの Spark ランタイムログが、`driver` や `executor` などのワーカータイプによって適切に名前が付けられたフォルダにアップロードされます。PySpark ジョブの出力が `s3://amzn-s3-demo-bucket/output/` にアップロードされます。

Hive

以下のコマンドを使用して Hive ジョブの状態を確認できます。

```
aws emr-serverless get-job-run \  
  --application-id application-id \  
  --job-run-id job-run-id
```

ログの送信先を `s3://amzn-s3-demo-bucket/emr-serverless-hive/logs` に設定すると、この特定のジョブ実行のログを `s3://amzn-s3-demo-bucket/emr-serverless-hive/logs/applications/application-id/jobs/job-run-id` で確認できます。

Hive アプリケーションの場合、EMR Serverless は Hive ドライバーを S3 ログ送信先の HIVE_DRIVER フォルダに、Tez タスクログを TEZ_TASK フォルダに継続的にアップロードします。ジョブの実行が SUCCEEDED 状態に達すると、Hive クエリの出力は、configurationOverrides の monitoringConfiguration フィールドで指定した Amazon S3 ロケーションで使用可能になります。

ステップ 4: クリーンアップする

このチュートリアルの操作が完了したら、作成したリソースの削除を検討してください。再度使用しないリソースをリリースすることをお勧めします。

アプリケーションを削除する

アプリケーションを削除するには、次のコマンドを使用します。

```
aws emr-serverless delete-application \  
  --application-id application-id
```

S3 ログバケットを削除する

S3 ログ記録と出力バケットを削除するには、次のコマンドを使用します。*amzn-s3-demo-bucket* を [EMR Serverless 用のストレージを準備する](#) で作成した S3 バケットの実際の名前に置き換えます。

```
aws s3 rm s3://amzn-s3-demo-bucket --recursive  
aws s3api delete-bucket --bucket amzn-s3-demo-bucket
```

ジョブランタイムロールを削除する

ランタイムロールを削除するには、ロールからポリシーをデタッチします。その後、ロールとポリシーの両方を削除できます。

```
aws iam detach-role-policy \  
  --role-name EMRServerlessS3RuntimeRole \  
  --policy-arn policy-arn
```

ロールを削除するには、次のコマンドを使用します。

```
aws iam delete-role \  
  --role-name EMRServerlessS3RuntimeRole
```

```
--role-name EMRServerlessS3RuntimeRole
```

ロールにアタッチされているポリシーを削除するには、次のコマンドを使用します。

```
aws iam delete-policy \  
  --policy-arn policy-arn
```

Spark ジョブと Hive ジョブの実行に関するその他の例については、「[EMR Serverless ジョブ実行時の Spark 設定の使用](#)」および「[EMR Serverless ジョブ実行時の Hive 設定の使用](#)」を参照してください。

EMR Serverless アプリケーションを操作して設定する

このセクションでは、AWS CLIで Amazon EMR Serverless アプリケーションを操作する方法について説明します。また、Spark エンジンと Hive エンジンのアプリケーションの設定、カスタマイズの実行、およびデフォルトについても説明します。

トピック

- [アプリケーションの状態](#)
- [EMR Studio コンソールからの EMR Serverless アプリケーションの作成](#)
- [での EMR Serverless アプリケーションの操作 AWS CLI](#)
- [EMR Serverless を使用する際のアプリケーションの設定](#)
- [EMR Serverless イメージのカスタマイズ](#)
- [データに接続するための EMR Serverless アプリケーションの VPC アクセスの設定](#)
- [Amazon EMR Serverless アーキテクチャオプション](#)
- [EMR Serverless アプリケーションのジョブ同時実行とキューイング](#)

アプリケーションの状態

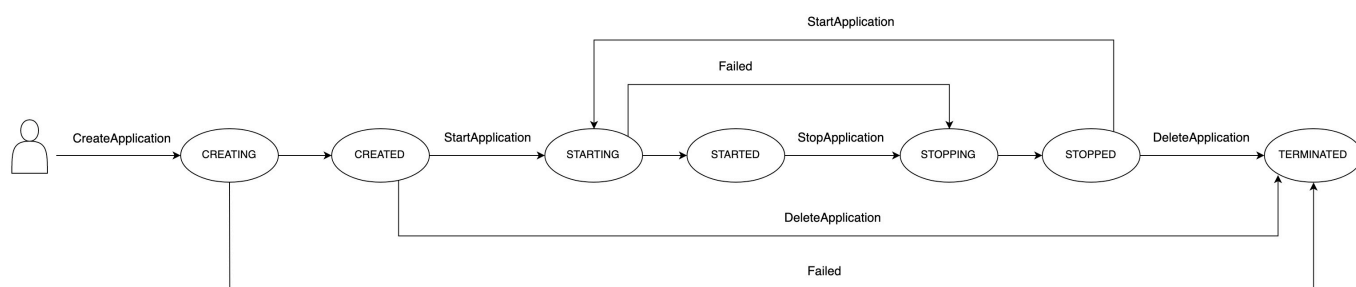
EMR Serverless でアプリケーションを作成すると、アプリケーションの実行は CREATING 状態になります。その後、以下の状態を経由して完了 (コード 0 で終了) または失敗 (0 以外のコードで終了) します。

アプリケーションには次の状態があります。

State	説明
[作成中]	アプリケーションは準備中であり、まだ使用する準備ができていません。
作成日	アプリケーションは作成されていますが、容量はまだプロビジョニングされていません。アプリケーションを変更して、初期容量設定を変更できます。

State	説明
スタート	アプリケーションが起動し、容量をプロビジョニングしています。
起動済み	アプリケーションは新しいジョブを受け入れる準備ができています。アプリケーションは、この状態にある場合にのみジョブを受け入れます。
停止中	すべてのジョブが完了し、アプリケーションは容量を解放しています。
停止	アプリケーションが停止しており、アプリケーションでリソースが実行されていません。アプリケーションを変更して、初期容量設定を変更できます。
終了済み	アプリケーションが終了しており、アプリケーションリストに表示されません。

次の図は、EMR Serverless アプリケーションの状態の軌跡を示しています。



EMR Studio コンソールからの EMR Serverless アプリケーションの作成

EMR Studio コンソールから、EMR Serverless アプリケーションを作成、表示、管理します。EMR Studio コンソールに移動するには、「[コンソールからの開始方法](#)」の手順に従ってください。

アプリケーションの作成

アプリケーションの作成ページでは、以下の手順に従って EMR Serverless アプリケーションを作成します。

1. [名前] フィールドに、アプリケーションを呼び出す名前を入力します。
2. [タイプ] フィールドで、アプリケーションのタイプとして Spark または Hive を選択します。
3. [リリースバージョン] フィールドで、EMR リリース番号を選択します。
4. [アーキテクチャ] オプションで、使用する指示セットアーキテクチャを選択します。詳細については、[Amazon EMR Serverless アーキテクチャオプション](#) を参照してください。
 - arm64 — 64-bit ARM アーキテクチャ。Graviton プロセッサを使用するためのアーキテクチャ
 - x86_64 — 64 ビット x86 アーキテクチャ。x86 ベースのプロセッサを使用するためのアーキテクチャ
5. 残りのフィールドには、デフォルト設定とカスタム設定の 2 つのアプリケーション設定オプションがあります。以下のフィールドはオプションです。

デフォルト設定 — デフォルト設定では、初期化済みの容量でアプリケーションをすばやく作成できます。これには、Spark 用のドライバー 1 つとエグゼキューター 1 つ、Hive 用のドライバー 1 つと Tez タスク 1 つが含まれます。デフォルト設定では VPC へのネットワーク接続は有効になりません。アプリケーションは、アイドル状態が 15 分間続くと停止するように設定され、ジョブの送信時に自動的に起動します。

カスタム設定 — カスタム設定では、次のプロパティを変更できます。

- 事前初期化された容量 — ドライバーとエグゼキューターまたは Hive Tez タスクワーカーの数、および各ワーカーのサイズ。
- アプリケーションの制限 — アプリケーションの最大容量。
- アプリケーションの動作 — アプリケーションの自動起動および自動停止の動作。
- ネットワーク接続 — VPC リソースへのネットワーク接続。
- タグ — アプリケーションに割り当てるカスタムタグ。

事前初期化された容量、アプリケーションの制限、およびアプリケーションの動作の詳細については、「[EMR Serverless を使用する際のアプリケーションの設定](#)」を参照してください。ネットワーク接続の詳細については、「[データに接続するための EMR Serverless アプリケーションの VPC アクセスの設定](#)」を参照してください。

6. アプリケーションを作成するには、[アプリケーションを作成] を選択します。

EMR Studio コンソールからアプリケーションを一覧表示する

既存の EMR Serverless アプリケーションはすべて、リストアプリケーションページからアクセスできます。アプリケーションの名前を選択して、対象のアプリケーションの詳細ページに移動できます。

EMR Studio コンソールからアプリケーションを管理する

アプリケーションに対して、[リストアプリケーション] ページまたは特定のアプリケーションの [詳細] ページから、次のアクションを実行できます。

アプリケーションの起動

アプリケーションを手動で起動するには、このオプションを選択します。

アプリケーションの停止

アプリケーションを手動で停止するには、このオプションを選択します。アプリケーションを停止するには、実行中のジョブがないことが必要です。アプリケーション状態の移行の詳細については、「[アプリケーションの状態](#)」を参照してください。

アプリケーションの設定

アプリケーションの設定ページからアプリケーションのオプション設定を編集します。ほとんどのアプリケーション設定を変更できます。例えば、アプリケーションのリリースラベルを変更して Amazon EMR の別のバージョンにアップグレードする、またはアーキテクチャを x86_64 から arm64 に切り替えます。その他のオプション設定は、アプリケーションの作成ページの [カスタム設定] セクションにある設定と同じです。アプリケーション設定の詳細については、「[アプリケーションの作成](#)」を参照してください。

アプリケーションを削除

アプリケーションを手動で削除するには、このオプションを選択します。削除するには、アプリケーションを停止する必要があります。アプリケーション状態の移行の詳細については、「[アプリケーションの状態](#)」を参照してください。

での EMR Serverless アプリケーションの操作 AWS CLI

から AWS CLI、個々のアプリケーションを作成、説明、削除します。また、すべてのアプリケーションを一覧表示して、それらを一度に確認します。このセクションでは、これらの手順の実施方法について説明します。アプリケーションの起動、停止、更新など、その他のアプリケーションオペレーションについては、「[EMR Serverless API Reference](#)」を参照してください。を使用して EMR Serverless API を使用する方法の例については AWS SDK for Java、GitHub リポジトリの [Java の例](#)を参照してください。を使用して EMR Serverless API を使用する方法の例については AWS SDK for Python (Boto)、GitHub リポジトリの [Python の例](#)を参照してください。

アプリケーションを作成するには、`create-application` を使用します。アプリケーションの `type` として SPARK または HIVE を指定する必要があります。このコマンドは、アプリケーションの ARN、名前、ID を返します。

```
aws emr-serverless create-application \  
--name my-application-name \  
--type 'application-type' \  
--release-label release-version
```

アプリケーションを記述するには、`get-application` を使用して `application-id` を指定します。このコマンドは、アプリケーションの状態と容量に関連する設定を返します。

```
aws emr-serverless get-application \  
--application-id application-id
```

すべてのアプリケーションを一覧表示するには、`list-applications` を呼び出します。このコマンドは `get-application` と同じプロパティを返しますが、すべてのアプリケーションが含まれます。

```
aws emr-serverless list-applications
```

アプリケーションを削除するには、`delete-application` を呼び出して `application-id` を指定します。

```
aws emr-serverless delete-application \  
--application-id application-id
```

EMR Serverless を使用する際のアプリケーションの設定

EMR Serverless では、使用するアプリケーションを設定します。例えば、アプリケーションがスケールアップできる最大容量を設定し、ドライバーとワーカーが応答できるように事前初期化された容量を設定して、アプリケーションレベルでランタイムとモニタリング設定の一般的なセットを指定します。次のページでは、EMR Serverless を使用する際にアプリケーションを設定する方法について説明します。

トピック

- [EMR Serverless でのアプリケーションの動作について](#)
- [EMR Serverless でアプリケーションを操作するための事前初期化された容量](#)
- [EMR Serverless のデフォルトのアプリケーション設定](#)

EMR Serverless でのアプリケーションの動作について

このセクションでは、ジョブを送信する際の動作、スケーリングの容量設定、EMR Serverless のワーカー設定について説明します。

アプリケーションのデフォルトの動作

自動起動 — デフォルトでは、アプリケーションはジョブの送信時に自動起動するように設定されています。この機能はオフにできます。

自動停止 — アプリケーションは、デフォルトで 15 分間アイドル状態になると自動停止するように設定されています。アプリケーションが STOPPED 状態になると、設定された事前初期化済みの容量が解放されます。アプリケーションが自動停止するまでのアイドル時間を変更できます。または、この機能をオフにすることもできます。

最大容量

アプリケーションがスケールアップ可能な最大容量を設定できます。CPU、メモリ (GB)、ディスク (GB) について最大容量を指定できます。

Note

ワーカー数にそれらのサイズを乗じることで、サポートされるワーカーサイズに比例するように最大容量を設定することがベストプラクティスです。例えば、アプリケーションを 2

vCPU、16 GB メモリ、20 GB のディスクを備えた 50 個のワーカーに制限する必要がある場合は、最大容量を 100 vCPU、800 GB メモリ、1,000 GB のディスクに設定します。

サポートされているワーカーの設定

次の表は、EMR Serverless に指定できるサポート対象のワーカーの設定とサイズを一覧示しています。ワークロードの必要性に応じて、ドライバーとエグゼキューターにさまざまなサイズを設定します。

ワーカーの設定とサイズ

CPU	メモリ	デフォルトのエフェメラルストレージ
1 vCPU	最小 2 GB、最大 8 GB、1 GB 単位	20 GB ~ 200 GB
2 vCPU	最小 4 GB、最大 16 GB、1 GB 単位	20 GB ~ 200 GB
4 vCPU	最小 8 GB、最大 30 GB、1 GB 単位	20 GB ~ 200 GB
8 vCPU	最小 16 GB、最大 60 GB、4 GB 単位	20 GB ~ 200 GB
16 vCPU	最小 32 GB、最大 120 GB、8 GB 単位	20 GB ~ 200 GB

CPU — 各ワーカーには、1、2、4、8、または 16 個の vCPU を配置できます。

メモリ — 各ワーカーには、前の表に記載されている制限内で GB 単位で指定されたメモリが搭載されています。Spark ジョブにはメモリオーバーヘッドがあります。つまり、使用するメモリが指定されたコンテナサイズを超えています。このオーバーヘッドは、プロパティ `spark.driver.memoryOverhead` と `spark.executor.memoryOverhead` で指定されます。オーバーヘッドのデフォルト値はコンテナメモリの 10% であり、最小値は 384 MB です。ワーカーサイズを選択する際は、このオーバーヘッドを考慮する必要があります。

例えば、ワーカーインスタンスに 4 個の vCPU を選択し、事前に初期化されたストレージ容量が 30 GB である場合は、Spark ジョブのエグゼキューターメモリとして約 27 GB の値を設定します。これにより、事前初期化された容量を最大限に活用できます。使用可能なメモリは 27 GB で、合計はこれに 27 GB の 10% (2.7 GB) を加えた 29.7 GB です。

ディスク — 各ワーカーには、最小サイズが 20 GB、最大サイズが 200 GB の一時ストレージディスクを設定できます。ワーカーごとに設定する 20 GB を超える追加ストレージに対してのみ料金が発生します。

EMR Serverless でアプリケーションを操作するための事前初期化された容量

EMR Serverless には、ドライバーとワーカーを事前に初期化され、数秒で応答できる状態に保つためのオプション機能が用意されています。これによって、アプリケーションのワーカーのウォームプールが効率的に作成されます。この機能は、事前初期化された容量と呼ばれます。この機能を設定するには、アプリケーションの `initialCapacity` パラメータを事前初期化するワーカーの数に設定します。ワーカー容量が初期化済みの場合、ジョブは直ちに開始されます。これは、反復アプリケーションと時間的制約のあるジョブを実装する場合に最適です。

事前初期化された容量により、ジョブとセッションが数秒で起動する準備ができた状態でワーカーのウォームプールが保持されます。プロビジョニングされた事前初期化済みのワーカーに対しては、アプリケーションがアイドル状態の場合でも料金が発生します。そのため、起動時間の短縮からメリットを得ることができるユースケースにはこれを有効にし、リソースを最適に活用できるようにサイズ設定することを提案します。EMR Serverless アプリケーションは、アイドル状態になると自動的にシャットダウンします。予期しない料金が発生しないように、事前に初期化されたワーカーを使用する場合は、この機能をオンにしておくことを提案します。

ジョブを送信する際に、`initialCapacity` のワーカーが利用可能な場合、ジョブはそれらのリソースを使用して実行を開始します。これらのワーカーが他のジョブで既に使用されている場合、またはジョブが `initialCapacity` から利用可能なリソースよりも多くのリソースを必要とする場合、アプリケーションは追加のワーカーをリクエストし、最大でアプリケーションに設定されている上限数のリソースを取得します。ジョブの実行が完了すると、使用したワーカーが解放され、アプリケーションで利用できるリソースの数は `initialCapacity` に戻ります。アプリケーションは、ジョブの実行が終了した後も、リソースの `initialCapacity` を維持します。アプリケーションは、ジョブの実行に必要ななくなった場合に、`initialCapacity` を超える余剰なリソースを解放します。

初期化済みの容量は、アプリケーションの起動が完了した時点で利用可能になり、直ちに使用できます。アプリケーションが停止すると、事前初期化された容量は非アクティブになります。アプ

リケーションは、リクエストされた事前初期化済みの容量が作成され、使用準備が整った場合のみ STARTED 状態に移行します。アプリケーションが STARTED 状態であるすべての期間、EMR Serverless は事前に初期化された容量をジョブやインタラクティブワークロードで使用可能または使用されている状態で保持します。この機能は、リリースされたコンテナまたは失敗したコンテナの容量を復元します。これにより、InitialCapacity パラメータが指定するワーカーの数が維持されます。事前に初期化された容量がないアプリケーションの状態は、直ちに CREATED から STARTED に変化する可能性があります。

一定期間使用されていない場合 (デフォルトは 15 分) に、事前に初期化された容量を解放するようにアプリケーションを設定できます。停止したアプリケーションは、新しいジョブを送信すると自動的に起動します。アプリケーションの作成時にこれらの自動開始および停止の設定を行うことができます。または、アプリケーションが CREATED または STOPPED の状態にある場合に変更します。

ワーカーごとに InitialCapacity カウントを変更し、CPU、メモリ、ディスクなどのコンピューティング設定を指定できます。部分的な変更を行うことができないため、値を変更するときにすべてのコンピューティング設定を指定します。アプリケーションが CREATED または STOPPED 状態にある場合にのみ、設定を変更できます。

Note

アプリケーションによるリソースの使用を最適化するには、コンテナサイズを事前に初期化された容量のワーカーのサイズに合わせることを提案します。例えば、Spark エグゼキューターのサイズを 2 CPU に設定し、メモリを 8 GB に設定したものの、事前初期化済みのキャパシティワーカーのサイズが 4 CPU でメモリが 16 GB の場合、Spark エグゼキューターはこのジョブに割り当てられたときにワーカーのリソースの半分のみを使用します。

Spark と Hive の事前初期化済み容量のカスタマイズ

特定のビッグデータフレームワークで実行されるワークロード用に、事前に初期化された容量をさらにカスタマイズできます。例えば、ワークロードが Apache Spark で実行されるときに、ドライバーとして起動するワーカーの数とエグゼキューターとして起動するワーカーの数を指定します。同様に、Apache Hive を使用する場合は、Hive ドライバーとして起動するワーカーの数と、Tez タスクを実行するワーカーの数を指定します。

Apache Hive を実行するアプリケーションを事前に初期化された容量で設定する

次の API リクエストは、Amazon EMR リリース emr-6.6.0 に基づいて Apache Hive を実行するアプリケーションを作成します。アプリケーションは、それぞれ 2 個の vCPU と 4 GB のメモリを搭載

した 5 つの事前初期化済み Hive ドライバー、およびそれぞれ 4 個の vCPU と 8 GB のメモリを搭載した 50 個の事前初期化済み Tez タスクワーカーで起動します。Hive クエリがこのアプリケーションで実行されると、まず事前初期化されたワーカーを使用し、直ちに実行を開始します。事前初期化済みのワーカーがすべてビジー状態であり、より多くの Hive ジョブが送信された場合、アプリケーションは合計 400 vCPU と 1,024 GB のメモリにスケールできます。必要に応じて、DRIVER ワーカーまたは TEZ_TASK ワーカーの容量を省略できます。

```
aws emr-serverless create-application \  
  --type "HIVE" \  
  --name my-application-name \  
  --release-label emr-6.6.0 \  
  --initial-capacity '{  
    "DRIVER": {  
      "workerCount": 5,  
      "workerConfiguration": {  
        "cpu": "2vCPU",  
        "memory": "4GB"  
      }  
    },  
    "TEZ_TASK": {  
      "workerCount": 50,  
      "workerConfiguration": {  
        "cpu": "4vCPU",  
        "memory": "8GB"  
      }  
    }  
  }' \  
  --maximum-capacity '{  
    "cpu": "400vCPU",  
    "memory": "1024GB"  
  }'
```

Apache Spark を実行しているアプリケーションを事前に初期化された容量で設定する

次の API リクエストは、Amazon EMR リリース 6.6.0 に基づいて Apache Spark 3.2.0 を実行するアプリケーションを作成します。アプリケーションは、それぞれ 2 個の vCPU と 4 GB のメモリを搭載した 5 つの事前初期化された Spark ドライバー、およびそれぞれ 4 vCPU と 8 GB のメモリを搭載した 50 個の事前初期化されたエグゼキューターで起動します。このアプリケーションで Spark ジョブを実行すると、まず事前初期化されたワーカーが使用され、直ちに実行を開始します。事前初期化済みのワーカーがすべてビジー状態であり、より多くの Spark ジョブが送信された場合、アプリ

ケーションは合計 400 vCPU と 1,024 GB のメモリにスケールできます。必要に応じて、DRIVER または EXECUTOR の容量を省略できます。

Note

Spark は、ドライバーとエグゼキューターにリクエストされたメモリに、デフォルト値の 10% の設定可能なメモリオーバーヘッドを追加します。ジョブが事前初期化されたワーカーを使用する場合、初期容量メモリ設定は、ジョブとオーバーヘッドがリクエストしたメモリよりも大きな値にする必要があります。

```
aws emr-serverless create-application \  
  --type "SPARK" \  
  --name my-application-name \  
  --release-label emr-6.6.0 \  
  --initial-capacity '{  
    "DRIVER": {  
      "workerCount": 5,  
      "workerConfiguration": {  
        "cpu": "2vCPU",  
        "memory": "4GB"  
      }  
    },  
    "EXECUTOR": {  
      "workerCount": 50,  
      "workerConfiguration": {  
        "cpu": "4vCPU",  
        "memory": "8GB"  
      }  
    }  
  }' \  
  --maximum-capacity '{  
    "cpu": "400vCPU",  
    "memory": "1024GB"  
  }'
```

EMR Serverless のデフォルトのアプリケーション設定

同じアプリケーションで送信するすべてのジョブに対して、アプリケーションレベルでランタイム設定とモニタリング設定の一般的なセットを指定できます。これにより、ジョブごとに同じ設定を送信する必要性に関連する追加のオーバーヘッドが軽減されます。

設定は、次の時点で変更できます。

- [ジョブの送信時にアプリケーションレベルの設定を宣言します。](#)
- [ジョブの実行中にデフォルト設定をオーバーライドします。](#)

次の各セクションでは、その他の構文の詳細と例を示します。

アプリケーションレベルでの設定の宣言

アプリケーションで送信するジョブのアプリケーションレベルのログ記録とランタイムの設定プロパティを指定できます。

monitoringConfiguration

アプリケーションで送信するジョブのログ設定を指定するには、[\[monitoringConfiguration\]](#) フィールドを使用します。EMR Serverless のログ記録について詳しくは、「[ログの保存](#)」を参照してください。

runtimeConfiguration

spark-defaults などのランタイム設定プロパティを指定するには、[\[runtimeConfiguration\]](#) フィールドに設定オブジェクトを指定します。これは、アプリケーションで送信するすべてのジョブのデフォルト設定に影響します。詳細については、[Hive 設定オーバーライドパラメータ](#) および [Spark 設定オーバーライドパラメータ](#) を参照してください。

特定の EMR Serverless リリースによって使用可能な設定分類は異なります。例えば、カスタム Log4j spark-driver-log4j2 と spark-executor-log4j2 の分類は、リリース 6.8.0 以降でのみ使用できます。アプリケーション固有のプロパティのリストについては、「[Spark ジョブのプロパティ](#)」および「[Hive ジョブプロパティ](#)」を参照してください。

アプリケーションレベルで [Apache Log4j2 プロパティ](#)、[データ保護のためのAWS Secrets Manager](#) および [Java 17 ランタイム](#)を設定することもできます。

アプリケーションレベルで Secrets Manager シークレットを渡すには、シークレットを使用して EMR Serverless アプリケーションを作成または更新する必要があるユーザーとロールに次のポリシーをアタッチします。

JSON

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

```
{
  "Sid": "SecretsManagerPolicy",
  "Effect": "Allow",
  "Action": [
    "secretsmanager:GetSecretValue",
    "secretsmanager:DescribeSecret"
  ],
  "Resource": [
    "arn:aws:secretsmanager:us-east-1:123456789012:secret:my-secret-name-123abc"
  ]
},
{
  "Sid": "KMSDecryptPolicy",
  "Effect": "Allow",
  "Action": [
    "kms:Decrypt"
  ],
  "Resource": [
    "arn:aws:kms:us-east-1:123456789012:key/12345678-1234-1234-1234-123456789012"
  ]
}
]
```

シークレットのカスタムポリシー作成の詳細については、「AWS Secrets Manager ユーザーガイド」の「[AWS Secrets Managerのアクセス許可ポリシーの例](#)」を参照してください。

 Note

アプリケーションレベルで指定した runtimeConfiguration は、[StartJobRun](#) API の applicationConfiguration にマッピングされます。

宣言の例

次の例は、`create-application` を使用してデフォルト設定を宣言する方法を示しています。

```
aws emr-serverless create-application \
  --release-label release-version \
```

```

--type SPARK \
--name my-application-name \
--runtime-configuration '[
    {
        "classification": "spark-defaults",
        "properties": {
            "spark.driver.cores": "4",
            "spark.executor.cores": "2",
            "spark.driver.memory": "8G",
            "spark.executor.memory": "8G",
            "spark.executor.instances": "2",

"spark.hadoop.javax.jdo.option.ConnectionDriverName": "org.mariadb.jdbc.Driver",
            "spark.hadoop.javax.jdo.option.ConnectionURL": "jdbc:mysql://db-host:db-
port/db-name",
            "spark.hadoop.javax.jdo.option.ConnectionUserName": "connection-user-
name",
            "spark.hadoop.javax.jdo.option.ConnectionPassword":
"EMR.secret@SecretID"
        }
    },
    {
        "classification": "spark-driver-log4j2",
        "properties": {
            "rootLogger.level": "error",
            "logger.IdentifierForClass.name": "classpathForSettingLogger",
            "logger.IdentifierForClass.level": "info"
        }
    }
]' \
--monitoring-configuration '{
    "s3MonitoringConfiguration": {
        "logUri": "s3://amzn-s3-demo-logging-bucket/logs/app-level"
    },
    "managedPersistenceMonitoringConfiguration": {
        "enabled": false
    }
}'

```


ジョブ実行中の設定のオーバーライド

[StartJobRun](#) API を使用して、アプリケーション設定とモニタリング設定の設定オーバーライドを指定できます。次に、EMR Serverless はアプリケーションレベルとジョブレベルで指定された設定をマージして、ジョブ実行の設定を決定します。

マージが発生する際の粒度レベルは次のとおりです。

- [ApplicationConfiguration](#) - 分類タイプ (例: spark-defaults)。
- [MonitoringConfiguration](#) - 設定タイプ (例: s3MonitoringConfiguration)。

Note

[StartJobRun](#) で指定した設定の優先度は、アプリケーションレベルで指定した設定よりも優先されます。

優先順位ランキングの詳細については、「[Hive 設定オーバーライドパラメータ](#)」および「[Spark 設定オーバーライドパラメータ](#)」を参照してください。

ジョブを開始するときに、特定の設定を指定しない場合、設定はアプリケーションから継承されます。ジョブレベルで設定を宣言する場合は、次の操作を実行できます。

- 既存の設定をオーバーライドする - オーバーライド値を使用して、StartJobRun リクエストに同じ設定パラメータを指定します。
- 設定を追加する - StartJobRun リクエストに、指定する値を含む新しい設定パラメータを追加します。
- 既存の設定を削除する - アプリケーションのランタイム設定を削除するには、削除する設定のキーを指定し、設定の空の宣言 {} を渡します。ジョブの実行に必要なパラメータを含む分類を削除することはお勧めしません。例えば、[Hive ジョブに必要なプロパティ](#) を削除しようとする、ジョブは失敗します。

アプリケーションのモニタリング設定を削除するには、関連する設定タイプに適した方法を使用します。

- **cloudWatchLoggingConfiguration** - cloudWatchLogging を削除するには、有効なフラグを false として渡します。

- **managedPersistenceMonitoringConfiguration** - マネージド永続化設定を削除してデフォルトの有効状態にフォールバックするには、設定の空の宣言 {} を渡します。
- **s3MonitoringConfiguration** - s3MonitoringConfiguration を削除するには、設定の空の宣言 {} を渡します。

オーバーライドの例

次の例は、start-job-run でのジョブ送信中に実行できるさまざまなオペレーションを示しています。

```
aws emr-serverless start-job-run \
  --application-id your-application-id \
  --execution-role-arn your-job-role-arn \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/
wordcount/scripts/wordcount.py",
      "entryPointArguments": ["s3://amzn-s3-demo-destination-bucket1/
wordcount_output"]
    }
  }' \
  --configuration-overrides '{
    "applicationConfiguration": [
      {
        // Override existing configuration for spark-defaults in the
application
        "classification": "spark-defaults",
        "properties": {
          "spark.driver.cores": "2",
          "spark.executor.cores": "1",
          "spark.driver.memory": "4G",
          "spark.executor.memory": "4G"
        }
      },
      {
        // Add configuration for spark-executor-log4j2
        "classification": "spark-executor-log4j2",
        "properties": {
          "rootLogger.level": "error",
          "logger.IdentifierForClass.name": "classpathForSettingLogger",
          "logger.IdentifierForClass.level": "info"
        }
      }
    ]
  }
```

```
    },
    {
      // Remove existing configuration for spark-driver-log4j2 from the
      application
      "classification": "spark-driver-log4j2",
      "properties": {}
    }
  ],
  "monitoringConfiguration": {
    "managedPersistenceMonitoringConfiguration": {
      // Override existing configuration for managed persistence
      "enabled": true
    },
    "s3MonitoringConfiguration": {
      // Remove configuration of S3 monitoring
    },
    "cloudWatchLoggingConfiguration": {
      // Add configuration for CloudWatch logging
      "enabled": true
    }
  }
}
```

ジョブの実行時に、「[Hive 設定オーバーライドパラメータ](#)」および「[Spark 設定オーバーライドパラメータ](#)」で説明されている優先度のオーバーライドのランキングに基づいて、次の分類と設定が適用されます。

- 分類 spark-defaults は、ジョブレベルで指定したプロパティで更新されます。この分類では、StartJobRun に含まれるプロパティのみが考慮されます。
- 分類 spark-executor-log4j2 は、既存の分類リストに追加されます。
- 分類 spark-driver-log4j2 は削除されます。
- managedPersistenceMonitoringConfiguration の設定は、ジョブレベルの設定で更新されます。
- s3MonitoringConfiguration の設定は削除されます。
- cloudWatchLoggingConfiguration の設定は、既存のモニタリング設定に追加されます。

EMR Serverless イメージのカスタマイズ

Amazon EMR 6.9.0 以降では、カスタムイメージを使用して、Amazon EMR Serverless を使用してアプリケーションの依存関係とランタイム環境を 1 つのコンテナにパッケージ化します。これにより、ワークロードの依存関係を管理する方法が簡単になり、パッケージの移植性が向上します。EMR Serverless イメージをカスタマイズすると、次の利点が得られます。

- ワークロードに最適化されたパッケージをインストールして設定します。これらのパッケージは、Amazon EMR ランタイム環境のパブリックディストリビューションでは広範囲に利用できません。
- EMR Serverless をローカルでの開発やテストなど、組織内で現在確立されているビルド、テスト、デプロイの各プロセスと統合します。
- 組織内のコンプライアンスとガバナンスに関する要件を満たす、確立されたセキュリティプロセス (イメージスキャンなど) を適用します。
- アプリケーションに独自のバージョンの JDK と Python を使用できます。

EMR Serverless には、独自のイメージを作成するときにベースとして使用するイメージが用意されています。ベースイメージは、イメージが EMR Serverless とやり取りするための必須のジャー、設定、ライブラリを備えています。ベースイメージは、[Amazon ECR Public Gallery](#) で取得できます。アプリケーションタイプ (Spark または Hive) とリリースバージョンに一致するイメージを使用します。例えば、Amazon EMR リリース 6.9.0 でアプリケーションを作成する場合は、次のイメージを使用します。

タイプ	Image
Spark	public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest
[Hive]	public.ecr.aws/emr-serverless/hive/emr-6.9.0:latest

前提条件

EMR Serverless カスタムイメージを作成する前に、以下の前提条件を満たすようにしてください。

1. EMR Serverless アプリケーションの起動に使用する AWS リージョン のと同じに Amazon ECR リポジトリを作成します。Amazon ECR プライベートリポジトリを作成するには、「[Creating a private repository](#)」を参照してください。
2. Amazon ECR リポジトリへのアクセス権をユーザーに付与するには、このリポジトリのイメージを使用して EMR Serverless アプリケーションを作成または更新するユーザーとロールに次のポリシーを追加します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ECRRepositoryListGetPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "ecr:DescribeImages"
      ],
      "Resource": [
        "arn:aws:ecr:*:123456789012:repository/my-repo"
      ]
    }
  ]
}
```

Amazon ECR アイデンティティベースのポリシーのその他の例については、「[Amazon Elastic Container Registry identity-based policy examples](#)」を参照してください。

ステップ 1: EMR Serverless ベースイメージからカスタムイメージを作成する

まず、任意のベースイメージを使用する FROM 指示で始まる [Dockerfile](#) を作成します。FROM 指示の後、イメージに加える必要がある変更を追加します。ベースイメージは、USER を自動的に `hadoop` に設定します。この設定には、追加するすべての変更に対するアクセス許可がありません。回避策として、USER を `root` に設定し、イメージを変更してから、USER を `hadoop:hadoop` に戻します。一般的なユースケースのサンプルを参照するには、「[EMR Serverless でのカスタムイメージの使用](#)」を参照してください。

```
# Dockerfile
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest

USER root
# MODIFICATIONS GO HERE

# EMRS runs the image as hadoop
USER hadoop:hadoop
```

Dockerfile を取得したら、次のコマンドを使用してイメージを構築します。

```
# build the docker image
docker build . -t aws-account-id.dkr.ecr.region.amazonaws.com/my-repository[:tag]or[@digest]
```

ステップ 2: イメージをローカルで検証する

EMR Serverless は、カスタムイメージを静的にチェックして、基本的なファイル、環境変数、および正しいイメージ設定を検証できるオフラインツールを備えています。ツールをインストールして実行する方法については、「[Amazon EMR Serverless Image CLI GitHub](#)」を参照してください。

ツールをインストールしたら、次のコマンドを実行してイメージを検証します。

```
amazon-emr-serverless-image \
validate-image -r emr-6.9.0 -t spark \
-i aws-account-id.dkr.ecr.region.amazonaws.com/my-repository:tag/@digest
```

以下のような出力が表示されます:

```
Amazon EMR Serverless - Image CLI
Version: 0.0.1
... Checking if docker cli is installed
... Checking Image Manifest
[INFO] Image ID: 9e2f4359cf5beb466a8a2ed047ab61c9d37786c555655ffc122272758f761b41a
[INFO] Created On: 2022-12-02T07:46:42.586249984Z
[INFO] Default User Set to hadoop:hadoop : PASS
[INFO] Working Directory Set to : PASS
[INFO] Entrypoint Set to /usr/bin/entrypoint.sh : PASS
[INFO] HADOOP_HOME is set with value: /usr/lib/hadoop : PASS
[INFO] HADOOP_LIBEXEC_DIR is set with value: /usr/lib/hadoop/libexec : PASS
```

```
[INFO] HADOOP_USER_HOME is set with value: /home/hadoop : PASS
[INFO] HADOOP_YARN_HOME is set with value: /usr/lib/hadoop-yarn : PASS
[INFO] HIVE_HOME is set with value: /usr/lib/hive : PASS
[INFO] JAVA_HOME is set with value: /etc/alternatives/jre : PASS
[INFO] TEZ_HOME is set with value: /usr/lib/tez : PASS
[INFO] YARN_HOME is set with value: /usr/lib/hadoop-yarn : PASS
[INFO] File Structure Test for hadoop-files in /usr/lib/hadoop: PASS
[INFO] File Structure Test for hadoop-jars in /usr/lib/hadoop/lib: PASS
[INFO] File Structure Test for hadoop-yarn-jars in /usr/lib/hadoop-yarn: PASS
[INFO] File Structure Test for hive-bin-files in /usr/bin: PASS
[INFO] File Structure Test for hive-jars in /usr/lib/hive/lib: PASS
[INFO] File Structure Test for java-bin in /etc/alternatives/jre/bin: PASS
[INFO] File Structure Test for tez-jars in /usr/lib/tez: PASS
-----
Overall Custom Image Validation Succeeded.
-----
```

ステップ 3: イメージを Amazon ECR リポジトリにアップロードする

以下のコマンドを使用して、Amazon ECR リポジトリに ECR イメージをプッシュします。イメージをリポジトリにプッシュするための正しい IAM アクセス許可があることを確認します。詳細については、「Amazon ECR ユーザーガイド」の「[Pushing an image](#)」を参照してください。

```
# login to ECR repo
aws ecr get-login-password --region region | docker login --username AWS --password-stdin aws-account-id.dkr.ecr.region.amazonaws.com

# push the docker image
docker push aws-account-id.dkr.ecr.region.amazonaws.com/my-repository:tag/@digest
```

ステップ 4: カスタムイメージを使用してアプリケーションを作成または更新する

アプリケーションを起動する方法に従って AWS CLI タブ AWS マネジメントコンソール を選択し、次の手順を実行します。

Console

1. <https://console.aws.amazon.com/emr> で EMR Studio コンソールにサインインします。アプリケーションに移動するか、「[アプリケーションの作成](#)」の手順で新しいアプリケーションを作成します。

2. EMR Serverless アプリケーションを作成または更新するときにカスタムイメージを指定するには、アプリケーション設定オプションで [カスタム設定] を選択します。
3. [カスタムイメージ設定] セクションで、[このアプリケーションでカスタムイメージを使用する] チェックボックスをオンにします。
4. Amazon ECR イメージ URI を [イメージ URI] フィールドに貼り付けます。EMR Serverless は、アプリケーションのすべてのワーカータイプにこのイメージを使用します。または、[異なるカスタムイメージ] を選択し、ワーカータイプごとに異なる Amazon ECR イメージ URI を貼り付けることができます。

CLI

- `image-configuration` パラメータを使用してアプリケーションを作成します。EMR Serverless は、この設定をすべてのワーカータイプに適用します。

```
aws emr-serverless create-application \  
--release-label emr-6.9.0 \  
--type SPARK \  
--image-configuration '{  
    "imageUri": "aws-account-id.dkr.ecr.region.amazonaws.com/my-repository:tag/  
@digest"  
}'
```

ワーカータイプごとに異なるイメージ設定を持つアプリケーションを作成するには、`worker-type-specifications` パラメータを使用します。

```
aws emr-serverless create-application \  
--release-label emr-6.9.0 \  
--type SPARK \  
--worker-type-specifications '{  
    "Driver": {  
        "imageConfiguration": {  
            "imageUri": "aws-account-id.dkr.ecr.region.amazonaws.com/my-  
repository:tag/@digest"  
        }  
    },  
    "Executor" : {  
        "imageConfiguration": {  
            "imageUri": "aws-account-id.dkr.ecr.region.amazonaws.com/my-  
repository:tag/@digest"  
        }  
    }  
}'
```



```
}
}'
```

アプリケーションを更新するには、`image-configuration` パラメータを使用します。EMR Serverless は、この設定をすべてのワーカータイプに適用します。

```
aws emr-serverless update-application \
--application-id application-id \
--image-configuration '{
    "imageUri": "aws-account-id.dkr.ecr.region.amazonaws.com/my-repository:tag/
@digest"
}'
```

ステップ 5: EMR Serverless がカスタムイメージリポジトリにアクセスできるようにする

Amazon ECR リポジトリに次のリソースポリシーを追加して、EMR Serverless サービスプリンシパルがこのリポジトリの `get`、`describe`、`download` のリクエストを使用できるようにします。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EmrServerlessCustomImageSupport",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:DescribeImages",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": "arn:aws:ecr:*:123456789012:repository/my-repo",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:emr-serverless:*:123456789012:/applications/
**
    }
  }
}
```

```
]
}
```

セキュリティのベストプラクティスとして、リポジトリポリシーに `aws:SourceArn` 条件キーを追加します。IAM グローバル条件キー `aws:SourceArn` は、EMR Serverless がアプリケーション ARN にのみリポジトリを使用するようにします。Amazon ECR リポジトリポリシーの詳細については、「[Creating a private repository](#)」を参照してください。

考慮事項と制限事項

カスタムイメージを使用して作業する場合は、次の点を考慮してください。

- アプリケーションのタイプ (Spark または Hive) とリリースラベル (`emr-6.9.0` など) に一致する正しいベースイメージを使用します。
- EMR Serverless は、Docker ファイル内の `[CMD]` または `[ENTRYPOINT]` の指示を無視します。`[COPY]`、`[RUN]`、`[WORKDIR]` など Docker ファイルで一般的な手順を使用します。
- カスタムイメージを作成するときに、環境変数 `JAVA_HOME`、`SPARK_HOME`、`HIVE_HOME`、`TEZ_HOME` を変更しないでください。
- カスタムイメージのサイズは 10 GB を超えることはできません。
- Amazon EMR ベースイメージのバイナリまたはジャーを変更することにより、アプリケーションまたはジョブの起動に失敗する可能性があります。
- Amazon ECR リポジトリは、EMR Serverless アプリケーションの起動に使用する AWS リージョンのと同じ 必要がある必要があります。

データに接続するための EMR Serverless アプリケーションの VPC アクセスの設定

Amazon Redshift クラスター、Amazon RDS データベース、VPC エンドポイントを持つ Amazon S3 バケットなど、VPC 内のデータストアに接続するように EMR Serverless アプリケーションを設定できます。EMR Serverless アプリケーションは、VPC 内のデータストアにアウトバウンド接続します。デフォルトでは、EMR Serverless はアプリケーションへのインバウンドアクセスおよびアウトバウンドインターネットアクセスの両方をブロックし、セキュリティを強化します。

Note

アプリケーションに外部 Hive メタストアデータベースを使用する場合は、VPC アクセスを設定する必要があります。外部 Hive メタストアを設定する方法については、「[Metastore configuration](#)」を参照してください。

アプリケーションを作成

[アプリケーションの作成] ページで、カスタム設定を選択し、EMR Serverless アプリケーションで利用できる VPC、サブネット、セキュリティグループを指定します。

VPC

データストアを含む仮想プライベートクラウド (VPC) の名前を選択します。[アプリケーションの作成] ページには、選択した AWS リージョンのすべての VPC が一覧表示されます。

サブネット

データストアを含む VPC 内のサブネットを選択します。[アプリケーションの作成] ページには、VPC 内のデータストアのすべてのサブネットが一覧表示されます。パブリックサブネットとプライベートサブネットの両方がサポートされています。プライベートサブネットまたはパブリックサブネットをアプリケーションに渡すことができます。パブリックサブネットとプライベートサブネットのどちらを持つかの選択には、いくつかの関連する考慮事項があります。

プライベートサブネットの場合:

- 関連付けられたルートテーブルにインターネットゲートウェイがあってはいません。
- インターネットへのアウトバウンド接続では、必要な場合、NAT Gateway を使用したアウトバウンドルートを設定します。NAT ゲートウェイを設定するには、「[NAT ゲートウェイ](#)」を参照してください。
- Amazon S3 接続の場合、NAT ゲートウェイまたは VPC エンドポイントを設定します。S3 VPC エンドポイントを設定するには、「[Create a gateway endpoint](#)」を参照してください。
- S3 VPC エンドポイントを設定し、アクセスを制御するエンドポイントポリシーをアタッチする場合は、「[Logging for EMR Serverless with managed storage](#)」の手順に従って、EMR Serverless がアプリケーションログを保存して処理するためのアクセス許可を提供します。

- Amazon DynamoDB など、VPC AWS のサービス 外の他の に接続するには、VPC エンドポイントまたは NAT ゲートウェイを設定します。AWS のサービスの VPC エンドポイントを設定するには、「[Work with VPC endpoints](#)」を参照してください。

Note

プライベートサブネットに Amazon EMR Serverless アプリケーションを設定するときには、Amazon S3 の VPC エンドポイントも設定することを提案します。EMR Serverless アプリケーションが Amazon S3 の VPC エンドポイントのないプライベートサブネットにある場合、S3 トラフィックに関連する追加の NAT ゲートウェイ料金が発生します。これは、VPC エンドポイントが設定されていない場合、EMR アプリケーションと Amazon S3 間のトラフィックが VPC 内にとどまらないためです。

パブリックサブネットの場合:

- これらには、インターネットゲートウェイへのルートがあります。
- アウトバウンドトラフィックを制御するには、適切なセキュリティグループ設定を確認しなければなりません。

ワーカーは、アウトバウンドトラフィックを介して VPC 内のデータストアに接続できます。デフォルトでは、EMR Serverless はワーカーへのインバウンドアクセスをブロックします。これはセキュリティ向上のためです。

を使用すると AWS Config、EMR Serverless はワーカーごとに Elastic Network Interface 項目レコードを作成します。このリソースに関連するコストを回避するには、 をオフにすることを検討AWS::EC2::NetworkInterfaceしてください AWS Config。

Note

複数のアベイラビリティーゾーンで複数のサブネットを選択することを提案します。これは、選択したサブネットによって、EMR Serverless アプリケーションを起動するために使用できるアベイラビリティーゾーンが決定されるためです。各ワーカーは、起動するサブネットで IP アドレスを使用します。指定したサブネットに、起動するワーカー数に対して十分な IP アドレスがあることを確認してください。サブネット計画の作成について詳しくは、

「[the section called “サブネット計画の作成に関するベストプラクティス”](#)」を参照してください。

サブネットの考慮事項と制約事項

- パブリックサブネットを持つ EMR Serverless は AWS Lake Formation をサポートしていません。
- インバウンドトラフィックは、パブリックサブネットではサポートされていません。

セキュリティグループ

データストアと通信できる 1 つ以上のセキュリティグループを選択します。[アプリケーションの作成] ページには、VPC 内のすべてのセキュリティグループが一覧表示されます。EMR Serverless は、VPC サブネットにアタッチされている Elastic Network Interface にこれらのセキュリティグループを関連付けます。

Note

EMR Serverless アプリケーション用に別のセキュリティグループを作成することを提案します。セキュリティグループで 0.0.0.0/0 または ::/0 の範囲にパブリックインターネットへのポートが開いている場合、EMR Serverless ではアプリケーションを作成/更新/開始できません。これにより、セキュリティ、分離が向上し、ネットワークルールの管理がより効率的になります。例えば、パブリック IP アドレスを持つワーカーへの予期しないトラフィックをブロックします。Amazon Redshift クラスターと通信するには、次のセクションの例に示すように、Redshift と EMR Serverless セキュリティグループ間のトラフィックルールを定義します。

Example例 — Amazon Redshift クラスターとの通信

1. EMR Serverless セキュリティグループの一つから Amazon Redshift セキュリティグループにインバウンドトラフィックのルールを追加します。

タイプ	プロトコル	ポート範囲	ソース
すべての TCP	TCP	5439	emr-serverless-security-group

2. EMR Serverless セキュリティグループの一つからのアウトバウンドトラフィックのルールを追加します。これは以下の 2 つの方法のいずれかで行います。まず、すべてのポートへのアウトバウンドトラフィックを開きます。

タイプ	プロトコル	ポート範囲	宛先
すべてのトラフィック	TCP	すべて	0.0.0.0/0

または、アウトバウンドトラフィックを Amazon Redshift クラスターに制限することもできます。これは、アプリケーションが Amazon Redshift クラスターとのみ通信する必要がある場合に限って有効です。

タイプ	プロトコル	ポート範囲	ソース
すべての TCP	TCP	5439	redshift-security-group

アプリケーションの設定

既存の EMR Serverless アプリケーションのネットワーク設定は、[アプリケーションの設定] ページから変更できます。

アクセスジョブ実行の詳細

[ジョブ実行の詳細] ページで、特定の実行のためにジョブが使用するサブネットにアクセスします。ジョブは、指定されたサブネットから選択された 1 つのサブネットでのみ実行されることに注意してください。

サブネット計画の作成に関するベストプラクティス

AWS リソースは、Amazon VPC で使用可能な IP アドレスのサブセットであるサブネットに作成されます。例えば、/16 ネットマスクを持つ VPC には最大 65,536 個の使用可能な IP アドレスがあり、サブネットマスクを使用して複数の小さなネットワークに分割できます。例えば、この範囲を 2 つのサブネットに分割し、それぞれに /17 マスクと 32,768 個の使用可能な IP アドレスを使用できます。サブネットはアベイラビリティゾーン内に存在し、複数のゾーンにわたって存在することはできません。

サブネットは、EMR Serverless アプリケーションのスケールリング制限を考慮して設計する必要があります。例えば、4 つの vCpu ワーカーをリクエストするアプリケーションがあり、4,000 vCpu までスケールアップできる場合、アプリケーションには最大 1,000 個のワーカーが必要になり、合計 1,000 個のネットワークインターフェイスが必要です。複数のアベイラビリティゾーンにサブネットを作成することを提案します。これにより、EMR Serverless は、アベイラビリティゾーンが失敗した場合に、万一、ジョブを再試行する、または別のアベイラビリティゾーンに事前初期化された容量をプロビジョニングすることができます。したがって、少なくとも 2 つのアベイラビリティゾーンの各サブネットには、1,000 個を超える使用可能な IP アドレスが必要です。

1,000 個のネットワークインターフェイスをプロビジョニングするには、マスクサイズが 22 以下のサブネットが必要です。22 を超えるマスクは要件を満たしません。例えば、/23 のサブネットマスクは 512 個の IP アドレスを提供し、/22 のマスクは 1,024 個を提供し、/21 のマスクは 2,048 個の IP アドレスを提供します。以下に示すのは、異なるアベイラビリティゾーンに割り当てることができる /16 ネットマスクの VPC に /22 マスクを持つ 4 つのサブネットの例です。各サブネットの最初の 4 つの IP アドレスと最後の IP アドレスは によって予約されるため、使用可能な IP アドレスと使用可能な IP アドレスには 5 つの違いがあります AWS。

サブネット ID	サブネットアドレス	サブネットマスク	IP アドレス範囲	利用可能な IP アドレス	使用可能な IP アドレス
1	10.0.0.0	255.255.252.0/22	10.0.0.0 ~ 10.0.3.255	1,024	1,019
2	10.0.4.0	255.255.252.0/22	10.0.4.0 ~ 10.0.7.255	1,024	1,019
3	10.0.8.0	255.255.252.0/22	10.0.8.0 - 10.0.11.255	1,024	1,019

サブネット ID	サブネットアドレス	サブネットマスク	IP アドレス範囲	利用可能な IP アドレス	使用可能な IP アドレス
4	10.0.12.0	255.255.252.0/22	10.0.12.0 ~ 10.0.15.255	1,024	1,019

ワークロードがより大きなワーカーサイズに適しているかどうかを評価する必要があります。より大きなワーカーサイズを使用すると、必要なネットワークインターフェイスが少なくなります。例えば、アプリケーションのスケール制限が 4,000 vCpu の 16 個の vCpu ワーカーを使用する場合、ネットワークインターフェイスをプロビジョニングするには、合計 250 個の使用可能な IP アドレスに対して最大 250 個のワーカーが必要です。250 個のネットワークインターフェイスをプロビジョニングするには、マスクサイズが 24 以下の複数のアベイラビリティゾーンにサブネットが必要です。24 を超えるマスクサイズは、250 個未満の IP アドレスを提供します。

複数のアプリケーション間でサブネットを共有する場合、各サブネットはすべてのアプリケーションの集合的なスケール制限を念頭に置いて設計する必要があります。例えば、4 つの vCpu ワーカーをリクエストする 3 つのアプリケーションがあり、それぞれが 12,000 の vCpu アカウントレベルのサービスベースのクォータで 4,000 vCpu までスケールアップできる場合、各サブネットには 3,000 個の使用可能な IP アドレスが必要です。使用する VPC に十分な数の IP アドレスがない場合は、使用可能な IP アドレスの数を増やしてみてください。この操作を行うには、VPC への追加の Classless Inter-Domain Routing (CIDR) ブロックの関連付けが必要になります。詳細については、「Amazon VPC ユーザーガイド」の「[追加の IPv4 CIDR ブロックと VPC の関連付け](#)」を参照してください。

オンラインで利用可能な多数のツールのいずれかを使用して、サブネット定義をすばやく生成し、使用可能な IP アドレスの範囲を確認できます。

Amazon EMR Serverless アーキテクチャオプション

Amazon EMR Serverless アプリケーションの命令セットアーキテクチャは、アプリケーションがジョブの実行に使用するプロセッサのタイプを決定します。Amazon EMR には、x86_64 と arm64 の 2 つのアーキテクチャオプションがあります。EMR Serverless は、最新世代のインスタンスが利用可能になると自動的に更新されるため、追加の労力を必要とせずにアプリケーションで新しいインスタンスを使用できます。

トピック

- [x86_64 アーキテクチャの使用](#)

- [arm64 アーキテクチャの使用 \(Graviton\)](#)
- [Graviton サポートによる新しいアプリケーションの起動](#)
- [Graviton を使用するように既存のアプリケーションを設定する](#)
- [Graviton を使用するときの考慮事項](#)

x86_64 アーキテクチャの使用

x86_64 アーキテクチャは、x86 64 ビットまたは x64 とも呼ばれます。x86_64 は EMR Serverless アプリケーションのデフォルトオプションです。このアーキテクチャは x86 ベースのプロセッサを使用し、ほとんどのサードパーティーのツールやライブラリと互換性があります。

ほとんどのアプリケーションは x86 ハードウェアプラットフォームと互換性があり、デフォルトの x86_64 アーキテクチャで正常に稼働できます。ただし、アプリケーションが 64 ビット ARM と互換性がある場合は、arm64 に切り替えて Graviton プロセッサを使用しパフォーマンス、コンピューティングパワー、メモリが向上します。x86 アーキテクチャで同じサイズのインスタンスを実行する場合よりも、arm64 アーキテクチャでインスタンスを実行する方がコストが低減されます。

arm64 アーキテクチャの使用 (Graviton)

AWS Graviton プロセッサは、64 ビット ARM Neoverse コア AWS を使用して によってカスタム設計され、arm64 アーキテクチャ (Arch64 または 64 ビット ARM と呼ばれます) を活用します。EMR Serverless AWS で使用できるプロセッサの Graviton ラインには、Graviton3 および Graviton2 プロセッサが含まれます。これらのプロセッサは、x86_64 アーキテクチャで実行される同等のワークロードと比較して、Spark および Hive ワークロードに対して優れた価格パフォーマンスを実現します。EMR Serverless は、最新世代のプロセッサにアップグレードするために、ユーザー側の労力を必要とせずに、最新世代のプロセッサを自動的に使用します。

Graviton サポートによる新しいアプリケーションの起動

arm64 アーキテクチャを使用するアプリケーションを起動するには、次のいずれかの方法を使用します。

AWS CLI

Graviton プロセッサを使用してアプリケーションを起動するには AWS CLI、`create-application` API の `architecture` パラメータ `ARM64` として を指定します。他のパラメータでアプリケーションに適した値を指定します。

```
aws emr-serverless create-application \  
  --name my-graviton-app \  
  --release-label emr-6.8.0 \  
  --type "SPARK" \  
  --architecture "ARM64" \  
  --region us-west-2
```

EMR Studio

EMR Studio から Graviton プロセッサを使用してアプリケーションを起動するには、アプリケーションを作成または更新するときにアーキテクチャオプションとして [arm64] を選択します。

Graviton を使用するように既存のアプリケーションを設定する

SDK、AWS CLIまたは EMR Studio で Graviton (arm64) アーキテクチャを使用するように既存の Amazon EMR Serverless アプリケーションを設定できます。

既存のアプリケーションを x86 から arm64 に変換するには

1. architecture パラメータをサポートする [AWS CLI/SDK](#) の最新バージョンを使用していることを確認します。
2. 実行中のジョブがないことを確認してから、アプリケーションを停止します。

```
aws emr-serverless stop-application \  
  --application-id application-id \  
  --region us-west-2
```

3. Graviton を使用するようにアプリケーションを更新するには、update-application API の architecture パラメータに ARM64 を指定します。

```
aws emr-serverless update-application \  
  --application-id application-id \  
  --architecture 'ARM64' \  
  --region us-west-2
```

4. アプリケーションの CPU アーキテクチャが ARM64 になったことを確認するには、get-application API を使用します。

```
aws emr-serverless get-application \  
  --application-id application-id \  
  --region us-west-2
```

```
--region us-west-2
```

5. 準備ができたら、アプリケーションを再起動します。

```
aws emr-serverless start-application \  
  --application-id application-id \  
  --region us-west-2
```

Graviton を使用するときの考慮事項

Graviton をサポートするために arm64 を使用して EMR Serverless アプリケーションを起動する前に、以下の内容を確認してください。

ライブラリの互換性

アーキテクチャオプションとして Graviton (arm64) を選択する場合は、サードパーティーのパッケージとライブラリが 64 ビット ARM アーキテクチャと互換性があることを確認してください。選択したアーキテクチャと互換性のある Python 仮想環境に Python ライブラリをパッケージ化する方法については、「[EMR Serverless での Python ライブラリの使用](#)」を参照してください。

詳細については、GitHub の「[AWS Graviton 入門リポジトリ](#)」を参照してください。このリポジトリには、ARM ベースの Graviton の使用開始に役立つ重要なリソースが含まれています。

EMR Serverless アプリケーションのジョブ同時実行とキューイング

Amazon EMR バージョン 7.0.0 以降では、アプリケーションのジョブ実行キューのタイムアウトと同時実行設定を指定します。この設定を指定すると、Amazon EMR Serverless はジョブをキューに入れ、アプリケーションの同時実行使用率に基づいて実行を開始します。例えば、ジョブ実行の同時実行が 10 の場合、アプリケーションで一度に実行されるジョブは 10 個のみです。残りのジョブは、実行中のジョブのいずれかが終了するまでキューに入れられます。キューのタイムアウトが早まると、ジョブはタイムアウトします。詳細については、「[Job run states](#)」を参照してください。

同時実行とキューイングの主な利点

ジョブの同時実行とキューイングは、多くのジョブの送信が必要な場合に次の利点をもたらします。

- これにより、同時実行ジョブを制御して、アプリケーションレベルの容量制限を効率的に使用できます。

- キューには、設定可能なタイムアウト設定で、ジョブ送信の突然のバーストを配置することができます。

同時実行とキューイングの開始方法

次の手順では、同時実行とキューイングを実装するいくつかの方法をデモンストレーションします。

の使用 AWS CLI

1. キューのタイムアウトと同時ジョブ実行を使用して Amazon EMR Serverless アプリケーションを作成します。

```
aws emr-serverless create-application \  
--release-label emr-7.0.0 \  
--type SPARK \  
--scheduler-configuration '{"maxConcurrentRuns": 1, "queueTimeoutMinutes": 30}'
```

2. アプリケーションを更新して、ジョブキューのタイムアウトと同時実行を変更します。

```
aws emr-serverless update-application \  
--application-id application-id \  
--scheduler-configuration '{"maxConcurrentRuns": 5, "queueTimeoutMinutes": 30}'
```

Note

既存のアプリケーションを更新して、ジョブの同時実行とキューイングを有効にすることができます。これを行うには、アプリケーションにリリースラベル emr-7.0.0 以降が必要です。

の使用 AWS マネジメントコンソール

次の手順では、AWS マネジメントコンソールを使用してジョブの同時実行とキューイングを開始する方法をデモンストレーションします。

1. EMR Studio に移動し、リリースラベル EMR-7.0.0 以降でアプリケーションを作成することを選択します。
2. [アプリケーションセットアップオプション] で、[カスタム設定を使用する] オプションを選択します。

3. [その他の設定] には、[ジョブ実行設定] のセクションがあります。[ジョブの同時実行を有効にする] オプションを選択して、機能を有効にします。
4. 選択後、[同時ジョブ実行] と [キュータイムアウト] を選択して、それぞれ同時ジョブ実行とキュータイムアウトの数を設定できます。これらの設定に対して値を入力しない場合は、デフォルト値が使用されます。
5. [アプリケーションを作成] を選択すると、この機能を有効にしてアプリケーションが作成されます。確認するには、ダッシュボードに移動し、アプリケーションを選択して、プロパティタブで機能が有効になっているかどうかを確認します。

設定後、この機能を有効にしてジョブを送信します。

同時実行とキューイングに関する考慮事項

同時実行とキューイングを実装する場合は、次の点を考慮してください。

- ジョブ同時実行とキューは、Amazon EMR リリース 7.0.0 以上でサポートされています。
- Amazon EMR リリース 7.3.0 以上では、ジョブの同時実行とキューがデフォルトで有効になっています。
- STARTED 状態のアプリケーションの同時実行を更新できません。
- maxConcurrentRuns の有効な範囲は 1～1000 であり、queueTimeoutMinutes の有効な範囲は 15～720 です。
- アカウントの QUEUED 状態には、最大 2,000 個のジョブを配置できます。
- 同時実行とキューイングは、バッチジョブとストリーミングジョブに適用されます。インタラクティブジョブには使用できません。詳細については、「[Run interactive workloads with EMR Serverless through EMR Studio](#)」を参照してください。

EMR Serverless を使用して S3 Express One Zone にデータを取得する

Amazon EMR リリース 7.2.0 以降では、[Amazon S3 Express One Zone](#) ストレージクラスで EMR Serverless を使用して、ジョブとワークロードを実行する際のパフォーマンスが向上します。S3 Express One Zone は、最もレイテンシーの影響を受けやすいアプリケーションに 1 桁のミリ秒単位で一貫したデータアクセスを提供する、高パフォーマンスの単一ゾーンの Amazon S3 ストレージクラスです。リリース時点で、S3 Express One Zone は、Amazon S3 の中でレイテンシーが最も低く、パフォーマンスの最も高いクラウドオブジェクトストレージを提供しています。

前提条件

- S3 Express One Zone のアクセス許可 - S3 Express One Zone が S3 オブジェクトに対して GET、LIST、PUT などのアクションを最初に行うと、ストレージクラスがユーザーに代わって CreateSession を呼び出します。S3A コネクタが CreateSession API を呼び出せるように、お使いの IAM ポリシーで s3express:CreateSession アクセス許可を付与する必要があります。このアクセス許可ポリシーの例については、「[S3 Express One Zone の使用を開始する](#)」を参照してください。
- S3A コネクタ - S3 Express One Zone ストレージクラスを使用する Amazon S3 バケットのデータにアクセスするように Spark クラスタを設定するには、Apache Hadoop コネクタ S3A を使用します。コネクタを使用するには、すべての S3 URI が s3a スキームを使用していることを確認してください。使用していない場合は、s3 スキームと s3n スキーム用にファイルシステム実装を変更します。

s3 スキームを変更するには、以下のクラスタ設定を指定します。

```
[
  {
    "Classification": "core-site",
    "Properties": {
      "fs.s3.impl": "org.apache.hadoop.fs.s3a.S3AFileSystem",
      "fs.AbstractFileSystem.s3.impl": "org.apache.hadoop.fs.s3a.S3A"
    }
  }
]
```

s3n スキームを変更するには、以下のクラスター設定を指定します。

```
[
  {
    "Classification": "core-site",
    "Properties": {
      "fs.s3n.impl": "org.apache.hadoop.fs.s3a.S3AFileSystem",
      "fs.AbstractFileSystem.s3n.impl": "org.apache.hadoop.fs.s3a.S3A"
    }
  }
]
```

S3 Express One Zone の使用を開始する

S3 Express One Zone の使用を開始するには、次の手順に従います。

1. [VPC エンドポイントを作成します](#)。エンドポイント `com.amazonaws.us-west-2.s3express` を VPC エンドポイントに追加します。
2. 「[Amazon EMR Serverless の使用を開始する](#)」に従って、Amazon EMR リリースラベル 7.2.0 以降でアプリケーションを作成します。
3. 新しく作成された VPC エンドポイント、プライベートサブネットグループ、セキュリティグループを使用するように[アプリケーションを設定します](#)。
4. `CreateSession` アクセス許可をジョブの実行ロールに追加します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Resource": [
        "*"
      ],
      "Action": [
        "s3express:CreateSession"
      ],
      "Sid": "AllowS3EXPRESSCreatesession"
    }
  ]
}
```

```
]
}
```

5. ジョブを実行します。S3 Express One Zone バケットにアクセスするには、S3A スキームを使用します。

```
aws emr-serverless start-job-run \  
--application-id <application-id> \  
--execution-role-arn <job-role-arn> \  
--name <job-run-name> \  
--job-driver '{  
  "sparkSubmit": {  
  
    "entryPoint": "s3a://<DOC-EXAMPLE-BUCKET>/scripts/wordcount.py",  
    "entryPointArguments":["s3a://<DOC-EXAMPLE-BUCKET>/emr-serverless-spark/output"],  
    "sparkSubmitParameters": "--conf spark.executor.cores=4  
--conf spark.executor.memory=8g --conf spark.driver.cores=4  
--conf spark.driver.memory=8g --conf spark.executor.instances=2  
--conf spark.hadoop.fs.s3a.change.detection.mode=none  
--conf spark.hadoop.fs.s3a.endpoint.region={<AWS_REGION>}  
--conf spark.hadoop.fs.s3a.select.enabled=false  
--conf spark.sql.sources.fastS3PartitionDiscovery.enabled=false  
}'
```


ジョブの実行

アプリケーションをプロビジョニングしたら、アプリケーションにジョブを送信します。このセクションでは、AWS CLI を使用してこれらのジョブを実行する方法について説明します。このセクションでは、EMR Serverless で使用できる各タイプのアプリケーションのデフォルト値も特定します。

トピック

- [ジョブ実行状態](#)
- [EMR Serverless 猶予期間を含むジョブ実行のキャンセル](#)
- [EMR Studio コンソールからのジョブの実行](#)
- [からジョブを実行する AWS CLI](#)
- [実行 IAM ポリシー。](#)
- [シャッフル最適化ディスクの使用](#)
- [Amazon EMR Serverless でのサーバーレスストレージの使用](#)
- [継続的にストリーミングされたデータを処理するためのストリーミングジョブ](#)
- [EMR Serverless ジョブ実行時の Spark 設定の使用](#)
- [EMR Serverless ジョブ実行時の Hive 設定の使用](#)
- [EMR Serverless ジョブの耐障害性](#)
- [EMR Serverless のメタストア設定](#)
- [EMR Serverless から別の AWS アカウントの S3 データにアクセスする](#)
- [EMR Serverless のエラーのトラブルシューティング](#)
- [ジョブレベルのコスト配分の有効化](#)

ジョブ実行状態

Amazon EMR Serverless ジョブキューにジョブ実行を送信すると、ジョブ実行は SUBMITTED 状態になります。ジョブ状態は、SUBMITTED から RUNNING を経て FAILED、SUCCESS、CANCELLING のいずれかになります。

ジョブ実行の各状態は以下のとおりです。

State	説明
Submitted (送信済み)	EMR Serverless にジョブ実行を送信するときの初期のジョブ状態。ジョブはアプリケーションのスケジュールを待機します。EMR Serverless は、ジョブ実行の優先順位付けとスケジュールを開始します。
Queued (キューに追加済み)	ジョブ実行は、アプリケーションレベルのジョブ実行の同時実行が完全に占有されていると、この状態で待機します。キューへの追加と同時実行の詳細については、「 EMR Serverless アプリケーションのジョブ同時実行とキューイング 」を参照してください。
Pending (保留中)	スケジューラは、ジョブ実行を評価して、アプリケーションの実行の優先順位付けとスケジューリングを行います。
Scheduled (スケジュール済み)	EMR Serverless は、アプリケーションのジョブ実行をスケジュールし、ジョブを実行するリソースを割り当てています。
Running (実行中)	EMR Serverless は、ジョブが最初に必要とするリソースを割り当て、ジョブはアプリケーションで実行されています。Spark アプリケーションでは、これは Spark ドライバースプロセスが running 状態にあることを意味します。
Failed (失敗)	EMR Serverless は、ジョブ実行をアプリケーションに送信できなかったか、正常に完了しませんでした。このジョブの失敗の詳細については、「StateDetails 」を参照してください。
Success (成功)	ジョブ実行は正常に完了しました。

State	説明
Cancelling (キャンセル中)	CancelJobRun API がジョブ実行のキャンセルをリクエストしたか、ジョブ実行がタイムアウトしました。EMR Serverless は、アプリケーションのジョブをキャンセルし、リソースをリリースしようとしています。
Cancelled (キャンセル)	ジョブの実行は正常にキャンセルされ、使用したリソースがリリースされました。

EMR Serverless 猶予期間を含むジョブ実行のキャンセル

データ処理システムでは、突然終了することにより、リソースの無駄、不完全なオペレーション、データの不整合が発生する可能性があります。Amazon EMR Serverless では、ジョブ実行をキャンセルする時に猶予期間を指定できます。この機能を使用することにより、ジョブが終了する前に、適切なクリーンアップと進行中の作業の完了にかかる時間を確保できます。

ジョブ実行のキャンセルする場合、パラメータ `shutdownGracePeriodInSeconds` を使用して猶予期間 (秒単位) を指定します。この間、ジョブは最終終了前にクリーンアップオペレーションを実行できます。動作とデフォルト設定は、バッチジョブとストリーミングジョブによって異なります。

バッチジョブの猶予期間

バッチジョブの場合、EMR Serverless では、猶予期間中に実行されるカスタムクリーンアップオペレーションを実装できます。これらのクリーンアップオペレーションは、JVM シャットダウンフックの一部としてアプリケーションコードに登録できます。

デフォルトの動作

シャットダウンのデフォルトの動作は、猶予期間がないことです。その構成は以下の 2 つのアクションです:

- 即時終了
- リソースはすぐにリリースされます

設定オプション

正常なシャットダウンにつながる設定を指定できます:

- シャットダウン猶予期間の有効な範囲: 15 ~ 1800 秒 (オプション)
- 即時終了 (猶予期間なし): 0 秒

正常なシャットダウンを有効にする

バッチジョブの正常なシャットダウンを実装するには、以下の手順に従います:

1. カスタムシャットダウンロジックを含むアプリケーションコードにシャットダウンフックを追加します。

Example in Scala

```
import org.apache.hadoop.util.ShutdownHookManager

// Register shutdown hook with priority (second argument)
// Higher priority hooks run first
ShutdownHookManager.get().addShutdownHook(() => {
    logger.info("Performing cleanup operations...")
}, 100)
```

[ShutdownHookManager](#) の使用

Example in PySpark

```
import atexit

def cleanup():
    # Your cleanup logic here
    print("Performing cleanup operations...")

# Register the cleanup function
atexit.register(cleanup)
```

2. ジョブをキャンセルするときに猶予期間を指定して、以前に追加されたフックの実行に時間を確保します

例

```
# Default (immediate termination)
aws emr-serverless cancel-job-run \
```

```
--application-id APPLICATION_ID \  
--job-run-id JOB_RUN_ID  
  
# With 5-minute grace period  
aws emr-serverless cancel-job-run \  
--application-id APPLICATION_ID \  
--job-run-id JOB_RUN_ID \  
--shutdown-grace-period-in-seconds 300
```

ストリーミングジョブの猶予期間

Spark 構造化ストリーミングでは、計算で外部データソースとの間で読み取りまたは書き込みが行われるため、突然シャットダウンすると望ましくない結果が生じる可能性があります。ストリーミングジョブはマイクロバッチでデータを処理し、これらのオペレーションを途中で中断することにより、その後の試行で処理が重複する可能性があります。これは、前のマイクロバッチの最新チェックポイントが書き込まれなかった場合に発生し、ストリーミングジョブの再起動時に同じデータが再度処理されます。このような重複処理はコンピューティングリソースを浪費するだけでなく、ビジネスオペレーションにも影響するため、突然のシャットダウンを避けることが重要です。

EMR Serverless は、ストリーミングクエリリスナーによる正常なシャットダウンを組み込みでサポートします。これにより、ジョブの終了前に継続的なマイクロバッチが適切に完了します。このサービスは、ストリーミングアプリケーションのマイクロバッチ間の正常なシャットダウンを自動的に管理し、現在のマイクロバッチが処理を完了し、チェックポイントが適切に書き込まれ、シャットダウンプロセス中に新しいデータを取り込まずにストリーミングコンテキストがクリーンに終了されるようにします。

デフォルトの動作

- デフォルトで有効になっている 120 秒の猶予期間。
- 組み込みのストリーミングクエリリスナーは、正常なシャットダウンを管理します。

設定オプション

- シャットダウン猶予期間の有効な範囲: 15 ~ 1800 秒 (オプション)
- 即時終了: 0 秒

正常なシャットダウンを有効にする

ストリーミングジョブに正常なシャットダウンを実装するには:

ジョブをキャンセルする時に猶予期間を指定して、進行中のマイクロバッチが完了するまでの時間を確保します。

例

```
# Default graceful shutdown (120 seconds)
aws emr-serverless cancel-job-run \
  --application-id APPLICATION_ID \
  --job-run-id JOB_RUN_ID

# Custom grace period (e.g. 300 seconds)
aws emr-serverless cancel-job-run \
  --application-id APPLICATION_ID \
  --job-run-id JOB_RUN_ID \
  --shutdown-grace-period-in-seconds 300

# Immediate Termination
aws emr-serverless cancel-job-run \
  --application-id APPLICATION_ID \
  --job-run-id JOB_RUN_ID \
  --shutdown-grace-period-in-seconds 0
```

カスタムシャットダウンフックを追加する (オプション)

EMR Serverless は、組み込みのストリーミングクエリリスナーを通じてデフォルトで正常なシャットダウンを管理しますが、オプションで個々のストリーミングクエリにカスタムシャットダウンフックを実装できます。EMR Serverless は、正常なシャットダウンリスナーを優先度 60 (ShutdownHookManager を使用) で登録します。優先度の高いフックが最初に実行されるため、カスタムクリーンアップオペレーションを 60 を超える優先度で登録して、EMR Serverless のシャットダウンプロセスが開始される前に実行されるようにすることができます。

カスタムフックを追加するには、アプリケーションコードにシャットダウンフックを追加する方法を示すこのトピックの最初の例を参照してください。ここで、優先度は 100 で、60 を超えています。したがって、このようなシャットダウンフックが最初に実行されます。

Note

カスタムシャットダウンフックはオプションであり、EMR Serverless によって自動的に処理される正常なシャットダウン機能には不要です。

猶予期間の料金とバッチ期間

猶予期間 (120 秒) のデフォルト値を使用する場合:

- バッチ期間が 120 秒未満の場合は、バッチの完了に必要な実際の時間に対してのみ課金されます。
- バッチ期間が 120 秒を超える場合、最大猶予期間 (120 秒) に対して課金されますが、クエリは強制終了されるため、正常にシャットダウンされない場合があります。

コストを最適化し、正常なシャットダウンを確保するには:

- バッチ期間 > 120 秒の場合: バッチ期間に合わせて猶予期間を増やすことを検討してください
- バッチ期間 < 120 秒の場合: 実際の処理時間に対してのみ課金されるため、猶予期間を調整する必要はありません

考慮事項

猶予期間の動作

- 猶予期間は、登録されたシャットダウンフックが完了するまでの時間を提供します。
- ジョブは、猶予期間よりかなり前であっても、シャットダウンフックが終了するとすぐに終了します。
- クリーンアップ操作が猶予期間を超えると、ジョブは強制的に終了します。

サービスの動作

- 猶予期間のシャットダウンは、RUNNING 状態のジョブでのみ使用できます。
- CANCELLING 状態の後続のキャンセルリクエストは無視されます。
- 内部サービスエラーが原因で EMR Serverless が猶予期間シャットダウンの開始に失敗した場合:

- サービスは最大 2 分間再試行します。
- 再試行が失敗することにより、ジョブは強制的に終了します。

料金

ジョブは、猶予期間中にかかった時間を含め、ジョブが完全にシャットダウンするまで使用されたコンピューティングリソースに対して請求されます。

EMR Studio コンソールからのジョブの実行

EMR Serverless アプリケーションにジョブ実行を送信し、EMR Studio コンソールからジョブアクセスできます。EMR Studio コンソールで EMR Serverless アプリケーションを作成または移動するには、「[コンソールからの開始方法](#)」の手順に従ってください。

ジョブを送信する

[ジョブの送信] ページで、次のように EMR Serverless アプリケーションにジョブを送信します。

Spark

1. [名前] フィールドにジョブ実行の名前を入力します。
2. [ランタイムロール] フィールドに、EMR Serverless アプリケーションがジョブ実行のために引き受けることができる IAM ロールの名前を入力します。ランタイムロールの詳細については、「[Amazon EMR Serverless のジョブランタイムロール](#)」を参照してください。
3. [スクリプトの場所] フィールドに、実行するスクリプトまたは JAR の Amazon S3 の場所を入力します。Spark ジョブの場合、スクリプトは Python (.py) ファイルまたは JAR (.jar) ファイルのいずれかになります。
4. スクリプトの場所が JAR ファイルの場合は、ジョブのエントリポイントであるクラス名を [メインクラス] フィールドに入力します。
5. (オプション) 残りのフィールドの値を入力します。
 - スクリプトの引数 — メイン JAR または Python スクリプトに渡す引数を入力します。コードはこれらのパラメータを読み取ります。配列の各引数は、カンマで区切ります。
 - Spark のプロパティ — Spark プロパティセクションを展開し、このフィールドに Spark 設定パラメータを入力します。

Note

Spark ドライバーとエグゼキューターのサイズを指定する場合は、メモリのオーバーヘッドを考慮します。プロパティ `spark.driver.memoryOverhead` と `spark.executor.memoryOverhead` にメモリオーバーヘッド値を指定します。メモリオーバーヘッドのデフォルト値はコンテナメモリの 10% で、最小 384 MB です。エグゼキューターメモリとメモリオーバーヘッドを合わせても、ワーカーメモリを超えることはできません。例えば、30 GB ワーカーの `spark.executor.memory` の最大値は 27 GB である必要があります。

- ジョブ設定 — このフィールドにジョブ設定を指定します。アプリケーションのデフォルトの設定を上書きするために、ジョブ設定を使用できます。
- 追加設定 — AWS Glue データカタログをメタストアとしてアクティブまたは非アクティブにして、アプリケーションログ設定を変更します。メタストア設定の詳細については、「[EMR Serverless のメタストア設定](#)」を参照してください。アプリケーションログ記録オプションの詳細については、「[ログの保存](#)」を参照してください。
- タグ — アプリケーションにカスタムタグを割り当てます。

6. [Submit job] (ジョブの送信) を選択します。

Hive

1. [名前] フィールドにジョブ実行の名前を入力します。
2. [ランタイムロール] フィールドに、EMR Serverless アプリケーションがジョブ実行のために引き受けることができる IAM ロールの名前を入力します。
3. [スクリプトの場所] フィールドに、実行するスクリプトまたは JAR の Amazon S3 の場所を入力します。Hive ジョブの場合、スクリプトは Hive (.sql) ファイルである必要があります。
4. (オプション) 残りのフィールドの値を入力します。
 - 初期化スクリプトの場所 – Hive スクリプトを実行する前にテーブルを初期化するスクリプトの場所を入力します。
 - Hive プロパティ – Hive プロパティセクションを展開し、このフィールドに Hive 設定パラメータを入力します。
 - ジョブ設定 – 任意のジョブ設定を指定します。アプリケーションのデフォルトの設定を上書きするために、ジョブ設定を使用できます。Hive ジョブの場合

合、hive.exec.scratchdir および hive.metastore.warehouse.dir は hive-site 設定に必要なプロパティです。

```
{
  "applicationConfiguration": [
    {
      "classification": "hive-site",
      "configurations": [],
      "properties": {
        "hive.exec.scratchdir": "s3://DOC-EXAMPLE_BUCKET/hive/scratch",
        "hive.metastore.warehouse.dir": "s3://DOC-EXAMPLE_BUCKET/hive/warehouse"
      }
    }
  ],
  "monitoringConfiguration": {}
}
```

- 追加設定 — AWS Glue データカタログをメタストアとしてアクティブ化または非アクティブ化し、アプリケーションログ設定を変更します。メタストア設定の詳細については、「[EMR Serverless のメタストア設定](#)」を参照してください。アプリケーションログ記録オプションの詳細については、「[ログの保存](#)」を参照してください。
- タグ — アプリケーションに任意のカスタムタグを割り当てます。

5. [Submit job] (ジョブの送信) を選択します。

ジョブ実行へのアクセス

アプリケーションの [詳細] ページの [ジョブ実行] タブからジョブ実行へアクセスして、ジョブ実行に対して次のアクションを実行できます。

ジョブのキャンセル — RUNNING 状態のジョブ実行をキャンセルするには、このオプションを選択します。ジョブ実行の移行の詳細については、「[ジョブ実行状態](#)」を参照してください。

ジョブのクローン — 以前のジョブ実行のクローンを作成して再送信するには、このオプションを選択します。

からジョブを実行する AWS CLI

AWS CLIで個々のジョブを作成、説明、削除できます。すべてのジョブを一覧表示してアクセスすることもできます。

新しいジョブを送信するには、`start-job-run` を使用します。実行するアプリケーションの ID と、ジョブ固有のプロパティを指定します。Spark の例については、「[EMR Serverless ジョブ実行時の Spark 設定の使用](#)」を参照してください。Hive の例については、「[EMR Serverless ジョブ実行時の Hive 設定の使用](#)」を参照してください。このコマンドは `application-id`、ARN、および新しい `job-id` を返します。

各ジョブの実行には、タイムアウト時間が設定されています。ジョブの実行がこの時間を超える場合、EMR Serverless は自動的にジョブをキャンセルします。デフォルトのタイムアウトは 12 時間です。ジョブの実行を開始するときに、このタイムアウト設定をジョブの要件を満たす値に設定します。`executionTimeoutMinutes` プロパティを使用してこの値を設定します。

```
aws emr-serverless start-job-run \  
  --application-id application-id \  
  --execution-role-arn job-role-arn \  
  --execution-timeout-minutes 15 \  
  --job-driver '{  
    "hive": {  
      "query": "s3://amzn-s3-demo-bucket/scripts/create_table.sql",  
      "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/  
hive/scratch --hiveconf hive.metastore.warehouse.dir=s3://amzn-s3-demo-bucket/hive/  
warehouse"  
    }  
  }' \  
  --configuration-overrides '{  
    "applicationConfiguration": [{  
      "classification": "hive-site",  
      "properties": {  
        "hive.client.cores": "2",  
        "hive.client.memory": "4GIB"  
      }  
    }  
  ]  
}'
```

ジョブを記述するには、`get-job-run` を使用します。このコマンドは、ジョブ固有の設定と新しいジョブの容量の設定を返します。

```
aws emr-serverless get-job-run \  
--job-run-id job-id \  
--application-id application-id
```

ジョブを一覧表示するには、`list-job-runs` を使用します。このコマンドは、ジョブタイプ、状態などの高レベル属性を含むプロパティの簡略版のセットを返します。すべてのジョブにアクセスしない場合は、アクセスするジョブの最大数を 50 個まで指定します。次の例では、最後の 2 つのジョブ実行をアクセスするように指定しています。

```
aws emr-serverless list-job-runs \  
--max-results 2 \  
--application-id application-id
```

ジョブをキャンセルするには、`cancel-job-run` を使用します。キャンセルするジョブの `application-id` と `job-id` を指定します。

```
aws emr-serverless cancel-job-run \  
--job-run-id job-id \  
--application-id application-id
```

からジョブを実行する方法の詳細については AWS CLI、[「EMR Serverless API リファレンス」](#) を参照してください。

実行 IAM ポリシー。

EMR Serverless でジョブ実行を送信する時に、実行ロールに加えて実行 IAM ポリシーを指定できます。ジョブ実行によって引き受けられるアクセス許可は、実行ロールのアクセス許可と指定された実行 IAM ポリシーの共通部分です。

開始方法

実行 IAM ポリシーを使用する手順:

`emr-serverless` アプリケーションを作成するか、既存のアプリケーションを使用して、以下の `aws cli` を実行して、インライン IAM ポリシーでジョブ実行を開始します:

```
aws emr-serverless start-job-run --region us-west-2 \  

```

```
--application-id application-id \  
--execution-role-arn execution-role-arn \  
--job-driver job-driver-options \  
--execution-iam-policy '{"policy": "inline-policy"}'
```

CLI コマンドの例

以下のポリシーがマシンの policy.json ファイルに格納されている場合:

JSON

```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Action": [  
        "s3:GetObject",  
        "s3:ListBucket"  
      ],  
      "Resource": [  
        "arn:aws:s3::my-test-bucket",  
        "arn:aws:s3::my-test-bucket/*"  
      ],  
      "Sid": "AllowS3GetObject"  
    }  
  ]  
}
```

その後、次の AWS CLI コマンドを使用して、このポリシーでジョブを開始できます。

```
aws emr-serverless start-job-run --region us-west-2 \  
  --application-id application-id \  
  --execution-role-arn execution-role-arn \  
  --job-driver job-driver-options \  
  --execution-iam-policy '{  
    "policy": '$(jq -c '. | @json' policy.json)'  
  }'
```

AWS とカスタマー管理ポリシーの両方を使用し、ARNsを通じて指定することもできます。

```
aws emr-serverless start-job-run --region us-west-2 \  
  --application-id application-id \  
  --execution-role-arn execution-role-arn \  
  --job-driver job-driver-options \  
  --execution-iam-policy '{  
    "policyArns": [  
      "arn:aws:iam::aws:policy/AmazonS3FullAccess",  
      "arn:aws:iam::aws:policy/CloudWatchLogsFullAccess"  
    ]  
  }'
```

同じリクエストでインライン IAM ポリシーとマネージドポリシー ARN の両方を指定することもできます:

```
aws emr-serverless start-job-run --region us-west-2 \  
  --application-id application-id \  
  --execution-role-arn execution-role-arn \  
  --job-driver job-driver-options \  
  --execution-iam-policy '{  
    "policy": '$(jq -c '. | @json' policy.json)',  
    "policyArns": [  
      "arn:aws:iam::aws:policy/AmazonS3FullAccess",  
      "arn:aws:iam::aws:policy/CloudWatchLogsFullAccess"  
    ]  
  }'
```

重要な注意事項

- `execution-role-policy` の `policy` フィールドの最大長は 2048 文字です。
- `execution-iam-policy` の `policy` フィールドで指定されたインライン IAM ポリシー文字列は、json 文字列標準に準拠する必要があります。前の例のように改行や引用符はエスケープされません。
- 最大 10 個の管理ポリシー ARNs のリストは、`execution-iam-policy` の `policyArns` フィールドの値として指定できます。
- 管理ポリシー ARNs は、有効な AWS またはカスタマー管理ポリシー ARN のリストである必要があります。カスタマー管理ポリシー ARN を指定する場合、ポリシーは EMR-S JobRun の同じ AWS アカウントに属している必要があります。
- インライン IAM ポリシーと 管理ポリシーの両方を使用する場合、インラインポリシーと 管理ポリシーの組み合わせに使用するプレーンテキストは 2,048 文字を超えることはできません。

- JobRun によって引き受けられるアクセス許可は、実行ロールのアクセス許可と指定された実行 IAM ポリシーの共通部分です。

ポリシーの交差

ジョブ実行によって引き受けられるアクセス許可は、実行ロールのアクセス許可と指定された実行 IAM ポリシーの共通部分です。つまり、JobRun を機能させるには、必要なアクセス許可を両方の場所で指定しなければなりません。ただし、更新または上書きする予定のないアクセス許可については、インラインポリシーで追加の一括許可ステートメントを指定できます。

例

以下の実行 IAM ロールポリシーを前提としています:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:*"
      ],
      "Resource": [
        "*"
      ],
      "Sid": "AllowS3"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:DescribeLogGroups"
      ],
      "Resource": [
        "arn:aws:logs:us-west-2:123456789012:log-group::log-stream"
      ],
      "Sid": "AllowLOGSDescribeloggroups"
    },
    {
      "Effect": "Allow",
```

```

    "Action": [
      "dynamodb:DescribeTable"
    ],
    "Resource": [
      "arn:aws:dynamodb:*:*:table/MyCompany1table"
    ],
    "Sid": "AllowDYNAMODBDescribetable"
  }
]
}

```

そして以下のインライン IAM ポリシー。

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::my-test-bucket/tenant1",
        "arn:aws:s3:::my-test-bucket/tenant1/*"
      ],
      "Sid": "AllowS3GetObject"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:*",
        "dynamodb:*"
      ],
      "Resource": [
        "*"
      ],
      "Sid": "AllowLOGS"
    }
  ]
}

```



```
}
```

JobRun によって引き受けられるアクセス許可は次のとおりです:

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::my-test-bucket/tenant1",
        "arn:aws:s3:::my-test-bucket/tenant1/*"
      ],
      "Sid": "AllowS3GetObject"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:DescribeLogGroups"
      ],
      "Resource": [
        "arn:aws:logs:us-west-2:123456789012:log-group:log-stream"
      ],
      "Sid": "AllowLOGSDescribeLogGroups"
    },
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:DescribeTable"
      ],
      "Resource": [
        "arn:aws:dynamodb:*:*:table/MyCompany1table"
      ],
      "Sid": "AllowDYNAMODBDescribeTable"
    }
  ]
}
```

}

シャッフル最適化ディスクの使用

Amazon EMR リリース 7.1.0 以降では、Apache Spark ジョブまたは Apache Hive ジョブを実行するときにシャッフル最適化ディスクを使用して、I/O 集約型ワークロードのパフォーマンスが向上します。標準ディスクと比較して、シャッフル最適化ディスクは IOPS (1 秒あたりの I/O オペレーション) が高く、シャッフルオペレーション中のデータ移動の高速化とレイテンシーの短縮を実現します。シャッフル最適化ディスクを使用することにより、ワーカーごとに最大 2 TB のディスクサイズをアタッチできるので、ワークロード要件に適した容量を設定します。

主な利点

シャッフル最適化ディスクには以下の利点があります。

- 高い IOPS パフォーマンス – シャッフル最適化ディスクは、標準ディスクよりも高い IOPS を実現し、Spark ジョブや Hive ジョブなどのシャッフル集約型ワークロードでデータシャッフルをより効率的かつ迅速に行うことができます。
- より大きなディスクサイズ – シャッフル最適化ディスクは、ワーカーあたり 20 GB から 2 TB のディスクサイズをサポートしているため、ワークロードに基づいて適切な容量を選択できます。

開始方法

ワークフローでシャッフル最適化ディスクを使用するには、次の手順を参照してください。

Spark

1. 次のコマンドを使用して、EMR Serverless リリース 7.1.0 アプリケーションを作成します。

```
aws emr-serverless create-application \  
  --type "SPARK" \  
  --name my-application-name \  
  --release-label emr-7.1.0 \  
  --region <AWS_REGION>
```

2. パラメータ `spark.emr-serverless.driver.disk.type` や `spark.emr-serverless.executor.disk.type` を指定して、シャッフル最適化ディスクで実行する

ように Spark ジョブを設定します。ユースケースに応じて、一方または両方のパラメータを使用できます。

```
aws emr-serverless start-job-run \  
  --application-id application-id \  
  --execution-role-arn job-role-arn \  
  --job-driver '{  
    "sparkSubmit": {  
      "entryPoint": "/usr/lib/spark/examples/jars/spark-examples.jar",  
      "entryPointArguments": ["1"],  
      "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi  
      --conf spark.executor.cores=4  
      --conf spark.executor.memory=20g  
      --conf spark.driver.cores=4  
      --conf spark.driver.memory=8g  
      --conf spark.executor.instances=1  
      --conf spark.emr-serverless.executor.disk.type=shuffle_optimized"  
    }  
  }'
```

詳細については、「[Spark ジョブのプロパティ](#)」を参照してください。

Hive

1. 次のコマンドを使用して、EMR Serverless リリース 7.1.0 アプリケーションを作成します。

```
aws emr-serverless create-application \  
  --type "HIVE" \  
  --name my-application-name \  
  --release-label emr-7.1.0 \  
  --region <AWS_REGION>
```

2. パラメータ `hive.driver.disk.type` や `hive.tez.disk.type` を指定して、シャッフル最適化ディスクで実行するように Hive ジョブを設定します。ユースケースに応じて、一方または両方のパラメータを使用できます。

```
aws emr-serverless start-job-run \  
  --application-id application-id \  
  --execution-role-arn job-role-arn \  
  --job-driver '{  
    "hive": {
```

```

        "query": "s3://<DOC-EXAMPLE-BUCKET>/emr-serverless-hive/query/hive-
query.q1",
        "parameters": "--hiveconf hive.log.explain.output=false"
    }
} \
--configuration-overrides '{
    "applicationConfiguration": [{
        "classification": "hive-site",
        "properties": {
            "hive.exec.scratchdir": "s3://<DOC-EXAMPLE-BUCKET>/emr-
serverless-hive/hive/scratch",
            "hive.metastore.warehouse.dir": "s3://<DOC-EXAMPLE-BUCKET>/emr-
serverless-hive/hive/warehouse",
            "hive.driver.cores": "2",
            "hive.driver.memory": "4g",
            "hive.tez.container.size": "4096",
            "hive.tez.cpu.vcores": "1",
            "hive.driver.disk.type": "shuffle_optimized",
            "hive.tez.disk.type": "shuffle_optimized"
        }
    ]
}'

```

詳細については、「[Hive ジョブプロパティ](#)」を参照してください。

事前に初期化された容量でのアプリケーションの設定

Amazon EMR リリース 7.1.0 に基づいてアプリケーションを作成するには、次の例を参照してください。これらのアプリケーションには、次のプロパティがあります。

- 5 台の事前に初期化された Spark ドライバー。それぞれ 2 つの vCPU、4 GB のメモリ、50 GB のシャッフル最適化ディスクを備えています。
- 50 個の事前に初期化されたエグゼキューター。それぞれ 4 つの vCPU、8 GB のメモリ、500 GB のシャッフル最適化ディスクを備えています。

このアプリケーションが Spark ジョブを実行すると、最初に初期化済みのワーカーを消費し、その後オンデマンドワーカーを最大 400 個の vCPU と 1024 GB のメモリ容量までスケールリングします。必要に応じて、DRIVER または EXECUTOR のいずれかの容量を省略できます。

Spark

```
aws emr-serverless create-application \  
--type "SPARK" \  
--name <my-application-name> \  
--release-label emr-7.1.0 \  
--initial-capacity '{  
  "DRIVER": {  
    "workerCount": 5,  
    "workerConfiguration": {  
      "cpu": "2vCPU",  
      "memory": "4GB",  
      "disk": "50GB",  
      "diskType": "SHUFFLE_OPTIMIZED"  
    }  
  },  
  "EXECUTOR": {  
    "workerCount": 50,  
    "workerConfiguration": {  
      "cpu": "4vCPU",  
      "memory": "8GB",  
      "disk": "500GB",  
      "diskType": "SHUFFLE_OPTIMIZED"  
    }  
  }  
}' \  
--maximum-capacity '{  
  "cpu": "400vCPU",  
  "memory": "1024GB"  
}'
```

Hive

```
aws emr-serverless create-application \  
--type "HIVE" \  
--name <my-application-name> \  
--release-label emr-7.1.0 \  
--initial-capacity '{  
  "DRIVER": {  
    "workerCount": 5,  
    "workerConfiguration": {  
      "cpu": "2vCPU",  
      "memory": "4GB",
```

```

        "disk": "50GB",
        "diskType": "SHUFFLE_OPTIMIZED"
    },
    "EXECUTOR": {
        "workerCount": 50,
        "workerConfiguration": {
            "cpu": "4vCPU",
            "memory": "8GB",
            "disk": "500GB",
            "diskType": "SHUFFLE_OPTIMIZED"
        }
    }
} \
--maximum-capacity '{
    "cpu": "400vCPU",
    "memory": "1024GB"
}'

```

Amazon EMR Serverless でのサーバーレスストレージの使用

Amazon EMR リリース 7.12 以降では、Apache Spark ジョブを実行するときにサーバーレスストレージを使用して、ローカルディスクのプロビジョニングを排除し、データ処理コストを削減し、ディスク容量の制約によるジョブの失敗を防ぎます。サーバーレスストレージは、容量設定を必要とせずにジョブのシャッフル、ディスクスピル、ディスクキャッシュ操作を自動的に処理し、中間データを無償で保存します。Amazon EMR Serverless は、ワークロードの需要に基づいて自動的にスケールするフルマネージドサーバーレスストレージに中間データを保存し、Spark がアイドル状態のときにすぐにコンピューティングワーカーを解放して、コンピューティングコストを削減できるようにします。

主な利点

EMR Serverless 用のサーバーレスストレージには、次の利点があります。

- ゼロ設定ストレージ – サーバーレスストレージを使用すると、アプリケーションまたはジョブごとにローカルディスクタイプとサイズを設定する必要がなくなります。EMR Serverless は、キャパシティプランニングなしで中間データオペレーションを自動的に管理します。
- 自動スケーリングによるジョブ障害の防止 – ストレージ容量はワークロードの需要に基づいて自動的にスケーリングされ、ジョブ障害によるディスク容量不足を防ぎます。

- データ処理コストの削減 – サーバーレスストレージは、2 つのメカニズムを通じて処理コストを削減します。まず、中間データストレージは無料で提供されます。コンピューティングリソースとメモリリソースに対してのみ料金が発生します。次に、Spark の動的リソース割り当てにより、Spark はワーカーを保持してローカルディスクに中間データを保持するのではなく、アイドル時にワーカーをすぐに解放できます。これにより、Spark ステージあたりのスケールアウトとスケールインが高速化され、後のステージに必要なワーカーが初期ステージよりも少ないジョブのコンピューティングコストを削減できます。
- ジョブレベルの分離による暗号化ストレージ – すべての中間データは、転送中および保管中に厳密なジョブレベルの分離で暗号化されます。
- きめ細かなアクセスコントロールのサポート – サーバーレスストレージは、AWS Lake Formation 統合によるきめ細かなアクセスコントロールをサポートします。

開始方法

Spark ワークフローで EMR Serverless のサーバーレスストレージを使用するには、次のステップを参照してください。

1. EMR Serverless アプリケーションを作成する

spark-defaults 分類で spark プロパティ `spark.aws.serverlessStorage.enabled` を `true` に設定して、サーバーレスストレージを有効にした EMR Serverless リリース 7.12 (またはそれ以降) アプリケーションを作成します。

```
aws emr-serverless create-application \  
  --type "SPARK" \  
  --name my-application \  
  --release-label emr-7.12.0 \  
  --runtime-configuration ' [{  
    "classification": "spark-defaults",  
    "properties": {  
      "spark.aws.serverlessStorage.enabled": "true"  
    }  
  } ]' \  
  --region <AWS_REGION>
```

2. Spark ジョブを開始する

アプリケーションでジョブ実行を開始します。EMR Serverless のサーバーレスストレージは、ジョブのシャッフルなどの中間データオペレーションを自動的に処理します。

```
aws emr-serverless start-job-run \
  --application-id <application-id> \
  --execution-role-arn <job-role-arn> \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "s3://<bucket>/script.py",
      "sparkSubmitParameters": "--conf spark.executor.cores=4
        --conf spark.executor.memory=20g
        --conf spark.driver.cores=4
        --conf spark.driver.memory=8g
        --conf spark.executor.instances=10"
    }
  }'
```

アプリケーションレベルで有効になっていない場合でも、ジョブレベルで EMR Serverless のサーバーレスストレージを有効にすることもできます。これにより、サーバーレスストレージが有効になっているワーカーノードが起動され、ジョブが処理されます。同じ Spark プロパティを `spark.aws.serverlessStorage.enabled false` に設定することで、特定のジョブのサーバーレスストレージを無効にすることもできます。

```
# Turn on serverless storage for EMR serverless for a specific job
aws emr-serverless start-job-run \
  --application-id <application-id> \
  --execution-role-arn <job-role-arn> \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "/usr/lib/spark/examples/jars/spark-examples.jar",
      "entryPointArguments": ["1"],
      "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi
        --conf spark.aws.serverlessStorage.enabled": "true"
    }
  }'
```

Note

従来のローカルディスクプロビジョニングを引き続き使用するには、`spark.aws.serverlessStorage.enabled` 設定を省略するか、`false` に設定します。

考慮事項と制限事項

- リリースバージョン – サーバーレスストレージは Amazon EMR リリース 7.12 以降でサポートされています。
- データボリュームの制限 – 各ジョブは、ジョブの実行ごとに最大 200 GB の中間データを読み書きできます。この制限を超えるジョブは失敗し、サーバーレスストレージの制限に達したことを示すエラーメッセージが表示されます。
- ジョブ実行タイムアウト – サーバーレスストレージは、最大 24 時間の実行タイムアウトのジョブをサポートします。実行タイムアウトを長くするように設定されたジョブは失敗し、エラーメッセージが表示されます。
- 事前初期化されたキャパシティ – 事前初期化されたキャパシティワーカーはサーバーレスストレージをサポートしていません。事前初期化された容量を設定すると、ジョブレベルでサーバーレスストレージを明示的に無効にするジョブでのみ使用されます。サーバーレスストレージが有効になっているジョブは、アプリケーションレベルの設定に関係なく、常に新しいワーカーをオンデマンドでプロビジョニングし、事前に初期化された容量を使用しません。
- ワークロードタイプ – サーバーレスストレージは、ストリーミングジョブとインタラクティブジョブではサポートされていません。
- ワーカー設定 – サーバーレスストレージは、1 つまたは 2 つの vCPUs を持つワーカーではサポートされていません。

サポートされている AWS リージョン

EMR Serverless は、以下のリージョンでサーバーレスストレージをサポートしています。

- 米国東部 (バージニア北部)
- 米国西部 (オレゴン)
- 欧州 (アイルランド)

継続的にストリーミングされたデータを処理するためのストリーミングジョブ

EMR Serverless のストリーミングジョブは、ストリーミングデータをほぼリアルタイムで分析、処理できるジョブモードです。これらの長時間実行されるジョブは、ストリーミングデータをポーリングし、データが到着すると継続的に結果を処理します。ストリーミングジョブは、ほぼリアルタイム

の分析、不正検出、レコメンデーションエンジンなど、リアルタイムのデータ処理を必要とするタスクに最適です。EMR Serverless ストリーミングジョブは、組み込みジョブの耐障害性、リアルタイムモニタリング、拡張ログ管理、ストリーミングコネクタとの統合などの最適化を提供します。

ストリーミングジョブのユースケースを次に示します。

- ほぼリアルタイムの分析 – Amazon EMR Serverless のストリーミングジョブを使用すると、ストリーミングデータをほぼリアルタイムで処理できるため、ログデータ、センサーデータ、クリックストリームデータなどの継続的なデータストリームに対してリアルタイム分析を実行してインサイトを取得し、最新の情報に基づいてタイムリーな意思決定を行うことができます。
- 不正検出 – データストリームを分析し、疑わしいパターンや異常が発生したときに特定することにより、ストリーミングジョブを使用して、金融取引、クレジットカードオペレーション、オンラインアクティビティでほぼリアルタイムの不正検出を実行します。
- レコメンデーションエンジン – ストリーミングジョブは、ユーザーアクティビティデータを処理してレコメンデーションモデルを更新できます。そうすることで、行動や好みに基づいてパーソナライズされたリアルタイムのレコメンデーションの可能性が広がります。
- ソーシャルメディア分析 – ストリーミングジョブは、ツイート、コメント、投稿などのソーシャルメディアデータを処理できるため、組織はトレンドの監視、感情分析、ブランドの評判の管理をほぼリアルタイムで行うことができます。
- モノのインターネット (IoT) 分析 – ストリーミングジョブは、IoT デバイス、センサー、接続された機械からの高速ストリームを処理、分析できるので、異常検出、予知保全、その他の IoT 分析のユースケースを実行します。
- クリックストリーム分析 – ストリーミングジョブは、ウェブサイトやモバイルアプリケーションからのクリックストリームデータを処理して分析できます。このようなデータを使用する企業は、分析を実行してユーザーの行動を詳細に把握し、ユーザーエクスペリエンスをパーソナライズし、マーケティングキャンペーンを最適化できます。
- ログの監視と分析 – ストリーミングジョブは、サーバー、アプリケーション、ネットワークデバイスからのログデータを処理することもできます。これにより、異常検出、トラブルシューティング、システムの正常性とパフォーマンスが得られます。

主な利点

EMR Serverless でのストリーミングジョブは、ジョブの耐障害性を自動的に提供します。これは、以下の要素の組み合わせです。

- 自動再試行 – EMR Serverless は、失敗したジョブをユーザーから手動で入力することなく自動的に再試行します。

- アベイラビリティーゾーン (AZ) の耐障害性 – EMR Serverless は、元の AZ で問題が発生するとストリーミングジョブを正常な AZ に自動的に切り替えます。
- ログ管理:
 - ログローテーション – ディスクストレージ管理をより効率的にするために、EMR Serverless は長時間のストリーミングジョブのログを定期的にローテーションします。これにより、すべてのディスク領域を消費する可能性のあるログの蓄積を防ぐことができます。
 - ログの圧縮 – マネージド永続化でログファイルを効率的に管理および最適化できます。圧縮により、マネージド Spark 履歴サーバーを使用する場合のデバッグのエクスペリエンスも向上します。

サポートされているデータソースとデータシンク

EMR Serverless は、多数の入力データソースと出力データシンクで動作します。

- サポートされている入力データソース – Amazon Kinesis Data Streams、Amazon Managed Streaming for Apache Kafka、セルフマネージド Apache Kafka クラスター。デフォルトでは、Amazon EMR リリース 7.1.0 以降には [Amazon Kinesis Data Streams コネクタ](#)が含まれているため、追加のパッケージを構築またはダウンロードする必要はありません。
- サポートされている出力データシンク – AWS Glue データカタログテーブル、Amazon S3、Amazon Redshift、MySQL、PostgreSQL Oracle、Oracle、Microsoft SQL、Apache Iceberg、Delta Lake、Apache Hudi。

考慮事項と制限事項

ストリーミングジョブを使用するときは、以下の考慮事項と制限事項に留意してください。

- ストリーミングジョブは、[Amazon EMR リリース 7.1.0 以降](#)でサポートされています。
- EMR Serverless ではストリーミングジョブが長時間実行されることが予想されるため、ジョブの実行時間を制限する実行タイムアウトを設定できません。
- ストリーミングジョブは、[構造化ストリーミングフレームワーク](#)に構築された Spark エンジンとのみ互換性があります。
- EMR Serverless はストリーミングジョブを無期限に再試行します。最大試行回数はカスタマイズできません。失敗した試行回数が 1 時間ごとの時間枠で設定されたしきい値を超えた場合、スラッシュ防止が自動的に組み込まれてジョブの再試行が停止します。デフォルトのしきい値は、1 時間あたり 5 回の試行の失敗です。このしきい値は、1~10 回試行するように設定できます。詳細については、「[Job resiliency](#)」を参照してください。

- ストリーミングジョブにはランタイムの状態と進行状況を保存するチェックポイントがあるため、EMR Serverless は最新のチェックポイントからストリーミングジョブを再開できます。詳細については、Apache Spark ドキュメントの「[Recovering from failures with Checkpointing](#)」を参照してください。

ストリーミングジョブの使用開始

ストリーミングジョブを使用開始する方法については、以下の手順を参照してください。

1. 「[Amazon EMR Serverless の使用を開始する](#)」に従ってアプリケーションを作成します。アプリケーションは [Amazon EMR リリース 7.1.0](#) 以降を実行する必要があります。
2. アプリケーションの準備ができたら、modeパラメータを に設定STREAMINGして、次の AWS CLI 例のようにストリーミングジョブを送信します。

```
aws emr-serverless start-job-run \  
--application-id <APPLICATION_ID> \  
--execution-role-arn <JOB_EXECUTION_ROLE> \  
--mode 'STREAMING' \  
--job-driver '{  
    "sparkSubmit": {  
        "entryPoint": "s3://<streaming script>",  
        "entryPointArguments": ["s3://<DOC-EXAMPLE-BUCKET-OUTPUT>/output"],  
        "sparkSubmitParameters": "--conf spark.executor.cores=4  
            --conf spark.executor.memory=16g  
            --conf spark.driver.cores=4  
            --conf spark.driver.memory=16g  
            --conf spark.executor.instances=3"  
    }  
}'
```

サポートされているストリーミングコネクタ

ストリーミングコネクタは、ストリーミングソースからのデータの読み取りを容易にし、ストリーミングシンクにデータを書き込むこともできます。

サポートされているストリーミングコネクタは次のとおりです。

Amazon Kinesis Data Streams コネクタ

Apache Spark 用の [Amazon Kinesis Data Streams コネクタ](#)を使用すると、Amazon Kinesis Data Streams との間でデータを消費し、データを書き込むストリーミングアプリケーションとパイプラインを構築できます。コネクタは、シャードあたり最大 2 MB/秒の専用の読み取りスループットレートで、拡張ファンアウト消費をサポートします。デフォルトでは、Amazon EMR Serverless 7.1.0 以降にはコネクタが組み込まれているため、追加のパッケージを構築またはダウンロードする必要はありません。コネクタの詳細については、[GitHub の spark-sql-kinesis-connector ページ](#)を参照してください。

以下は、Kinesis Data Streams コネクタの依存関係を使用してジョブ実行を開始する方法の例です。

```
aws emr-serverless start-job-run \  
--application-id <APPLICATION_ID> \  
--execution-role-arn <JOB_EXECUTION_ROLE> \  
--mode 'STREAMING' \  
--job-driver '{  
  "sparkSubmit": {  
    "entryPoint": "s3://<Kinesis-streaming-script>",  
    "entryPointArguments": ["s3://<DOC-EXAMPLE-BUCKET-OUTPUT>/output"],  
    "sparkSubmitParameters": "--conf spark.executor.cores=4  
      --conf spark.executor.memory=16g  
      --conf spark.driver.cores=4  
      --conf spark.driver.memory=16g  
      --conf spark.executor.instances=3  
      --jars /usr/share/aws/kinesis/spark-sql-kinesis/lib/spark-streaming-  
sql-kinesis-connector.jar"  
  }  
}'
```

Kinesis Data Streams に接続するには、VPC アクセスを使用して EMR Serverless アプリケーションを設定し、VPC エンドポイントを使用してプライベートアクセスを許可するか、NAT Gateway を使用してパブリックアクセスを取得します。詳細については、「[Configuring VPC access](#)」を参照してください。また、ジョブのランタイムロールに、必要とするデータストリームにアクセスするために必要な読み取り権限と書き込み権限があることを確認する必要があります。ジョブランタイムロールの設定方法の詳細については、「[Job runtime roles for Amazon EMR Serverless](#)」を参照してください。必要なすべてのアクセス許可の完全なリストについては、[GitHub の spark-sql-kinesis-connector ページ](#)を参照してください。

Apache Kafka コネクタ

Spark 構造化ストリーミング用の Apache Kafka コネクタは Spark コミュニティのオープンソースコネクタであり、Maven リポジトリで利用できます。このコネクタにより、Spark 構造化スト

リーミングアプリケーションや、セルフマネージド Apache Kafka と Amazon Managed Streaming for Apache Kafka との間でデータを読み書きするのが容易になります。コネクタの詳細については、Apache Spark ドキュメントの「[Structured Streaming + Kafka Integration Guide](#)」を参照してください。

次の例は、ジョブ実行リクエストに Kafka コネクタを含める方法を示しています。

```
aws emr-serverless start-job-run \  
--application-id <APPLICATION_ID> \  
--execution-role-arn <JOB_EXECUTION_ROLE> \  
--mode 'STREAMING' \  
--job-driver '{  
  "sparkSubmit": {  
    "entryPoint": "s3://<Kafka-streaming-script>",  
    "entryPointArguments": ["s3://<DOC-EXAMPLE-BUCKET-OUTPUT>/output"],  
    "sparkSubmitParameters": "--conf spark.executor.cores=4  
      --conf spark.executor.memory=16g  
      --conf spark.driver.cores=4  
      --conf spark.driver.memory=16g  
      --conf spark.executor.instances=3  
      --packages org.apache.spark:spark-sql-  
kafka-0-10_2.12:<KAFKA_CONNECTOR_VERSION>"  
  }  
'
```

Apache Kafka コネクタのバージョンは、EMR Serverless リリースバージョンと対応する Spark バージョンによって異なります。正しい Kafka バージョンを確認するには、「[Structured Streaming + Kafka Integration Guide](#)」を参照してください。

IAM 認証で Amazon Managed Streaming for Apache Kafka を使用するには、Kafka コネクタが IAM で Amazon MSK に接続できるように別の依存関係を含めます。詳細については、[GitHub の aws-msk-iam-auth リポジトリ](#)を参照してください。また、ジョブのランタイムロールに必要な IAM アクセス許可があることを確認する必要があります。次の例は、IAM 認証でコネクタを使用する方法を示しています。

```
aws emr-serverless start-job-run \  
--application-id <APPLICATION_ID> \  
--execution-role-arn <JOB_EXECUTION_ROLE> \  
--mode 'STREAMING' \  
--job-driver '{  
  "sparkSubmit": {
```

```

    "entryPoint": "s3://<Kafka-streaming-script>",
    "entryPointArguments": ["s3://<DOC-EXAMPLE-BUCKET-OUTPUT>/output"],
    "sparkSubmitParameters": "--conf spark.executor.cores=4
        --conf spark.executor.memory=16g
        --conf spark.driver.cores=4
        --conf spark.driver.memory=16g
        --conf spark.executor.instances=3
        --packages org.apache.spark:spark-sql-
kafka-0-10_2.12:<KAFKA_CONNECTOR_VERSION>,software.amazon.msk:aws-msk-iam-
auth:<MSK_IAM_LIB_VERSION>"
    }
}'

```

Amazon MSK から Kafka コネクタと IAM 認証ライブラリを使用するには、VPC アクセスを使用して EMR Serverless アプリケーションを設定します。サブネットにはインターネットアクセスが必要で、Maven の依存関係にアクセスするには NAT Gateway を使用する必要があります。詳細については、「[Configuring VPC access](#)」を参照してください。Kafka クラスターにアクセスするには、サブネットにネットワーク接続が必要です。これは、Kafka クラスターがセルフマネージド型かどうか、Amazon Managed Streaming for Apache Kafka を使用しているかどうか、に関係なく当てはまります。

ストリーミングジョブのログ管理

ストリーミングジョブは、Spark アプリケーションログとイベントログのログローテーション、および Spark イベントログのログ圧縮をサポートしています。これにより、リソースを効果的に管理できます。

ログローテーション

ストリーミングジョブは、Spark アプリケーションログとイベントログのログローテーションをサポートします。ログローテーションにより、長時間のストリーミングジョブで大きなログファイルが生成して使用可能なディスク領域をすべて使ってしまうのを防ぎます。ログローテーションはディスクストレージを節約し、ディスク容量が少ないためにジョブが失敗するのを防ぐのに役立ちます。詳細については、「[Rotating logs](#)」を参照してください。

ログ圧縮

ストリーミングジョブは、マネージドログ記録が利用可能な場合は必ず Spark イベントログのログ圧縮もサポートします。マネージドログ記録の詳細については、「[Logging with managed storage](#)」を参照してください。ストリーミングジョブは長時間実行できるため、イベントデータが時間の経過とともに蓄積されて、ログファイルのサイズが著しく増加する可能性があります。Spark History

Server は、これらのイベントを読み取り、Spark アプリケーション UI のメモリにロードします。このプロセスは、特に Amazon S3 に保存されているイベントログが非常に大きい場合、レイテンシーが大きくなりコストが高くなる可能性があります。

ログ圧縮によりイベントログのサイズが小さくなるため、Spark History Server が常に 1 GB を超えるイベントログをロードする必要はなくなります。詳細については、Apache Spark ドキュメントの「[Monitoring and Instrumentation](#)」を参照してください。

EMR Serverless ジョブ実行時の Spark 設定の使用

type パラメータを SPARK に設定して、アプリケーションで Spark ジョブを実行できます。ジョブは、Amazon EMR リリースバージョンと互換性のある Spark バージョンと互換性がある必要があります。例えば、Amazon EMR リリース 6.6.0 でジョブを実行する場合、ジョブは Apache Spark 3.2.0 と互換性がある必要があります。各リリースのアプリケーションバージョンの詳細については、「[Amazon EMR Serverless のリリースバージョン](#)」を参照してください。

Spark ジョブのパラメータ

[StartJobRun API](#) を使用して Spark ジョブを実行する場合は、次のパラメータを指定します。

必須パラメータ

- [Spark ジョブのランタイムロール](#)
- [Spark ジョブのドライバーパラメータ](#)
- [Spark 設定オーバーライドパラメータ](#)
- [Spark 動的リソース割り当ての最適化](#)

Spark ジョブのランタイムロール

executionRoleArn を使用して、アプリケーションが Spark ジョブの実行に使用する IAM ロールの ARN を指定します。このロールには、以下のアクセス許可が含まれている必要があります。

- データが存在する S3 バケットまたはその他のデータソースからの読み取り
- PySpark スクリプトまたは JAR ファイルが存在する S3 バケットまたはプレフィックスからの読み取り
- 最終出力を書き込む予定の S3 バケットへの書き込み
- S3MonitoringConfiguration が指定した S3 バケットまたはプレフィックスへのログの書き込み

- KMS キーを使用して S3 バケット内のデータを暗号化する場合の KMS キーへのアクセス
- SparkSQL AWS を使用している場合の Glue データカタログへのアクセス

Spark ジョブが他のデータソースとの間でデータを読み書きする場合は、この IAM ロールに適切なアクセス許可を指定します。これらのアクセス許可を IAM ロールに指定しなければ、ジョブが失敗する可能性があります。詳細については、[Amazon EMR Serverless のジョブランタイムロール](#) および [ログの保存](#) を参照してください。

Spark ジョブのドライバーパラメータ

jobDriver を使用してジョブに入力します。ジョブのドライバーパラメータは、実行するジョブタイプに対して 1 つの値のみを受け入れます。Spark ジョブの場合、パラメータの値は `sparkSubmit` です。このジョブタイプを使用して、Scala、PySpark およびその他のサポートされているジョブを Spark 送信を通じて実行できます。Spark ジョブには、以下のパラメータがあります。

- **sparkSubmitParameters** - これらは、ジョブに送信する追加の Spark パラメータです。このパラメータを使用して、ドライバーメモリや、引数 `--conf` や `--class` で定義されたものなどのエグゼキューターの数など、デフォルトの Spark プロパティを上書きします。
- **entryPointArguments** - これはメイン JAR または Python ファイルに渡す引数の配列です。これらのパラメータの読み取りは、エントリーポイントコードを使用して処理する必要があります。配列の各引数は、カンマで区切ります。
- **entryPoint** - これは、実行するメイン JAR または Python ファイルへの Amazon S3 での参照です。Scala または Java JAR を実行している場合は、`--class` 引数を使用して `SparkSubmitParameters` でメインエントリクラスを指定します。

詳細については、[Launching Applications with spark-submit](#) を参照してください。

Spark 設定オーバーライドパラメータ

configurationOverrides を使用して、モニタリングレベルおよびアプリケーションレベルの設定プロパティを上書きします。このパラメータは、次の 2 つのフィールドを持つ JSON オブジェクトを受け付けます。

- **monitoringConfiguration** - このフィールドを使用して、EMR Serverless ジョブに Spark ジョブのログを保存する Amazon S3 URL (`s3MonitoringConfiguration`) を指定します。このバケットが、アプリケーションをホスト AWS アカウント すると同じ と、ジョブが実行されている AWS リージョン と同じ で作成されていることを確認します。

- **applicationConfiguration** – アプリケーションのデフォルトの設定を上書きするために、このフィールドの設定オブジェクトを指定できます。短縮構文を使用して、設定を指定したり、JSON ファイルの設定オブジェクトを参照したりできます。設定オブジェクトは、分類、プロパティ、オプションの入れ子になっている設定で構成されます。プロパティは、そのファイル内で上書きする設定で構成されます。単一の JSON オブジェクトで、複数のアプリケーションに複数の分類を指定できます。

 Note

特定の EMR Serverless リリースによって使用可能な設定分類は異なります。例えば、カスタム Log4j spark-driver-log4j2 と spark-executor-log4j2 の分類は、リリース 6.8.0 以降でのみ使用できます。

アプリケーションの上書きと Spark 送信パラメータで同じ設定を使用すると、Spark 送信パラメータが優先されます。設定は、優先順位の高いものから低いものへとランク付けしたものです。

- EMR Serverless が SparkSession を作成するときに指定する設定。
- --conf 引数で sparkSubmitParameters の一部として指定する設定。
- アプリケーションの一部として提供する設定は、ジョブの開始時に上書きされます。
- アプリケーションの作成時に runtimeConfiguration の一部として提供する設定。
- リリース用に Amazon EMR によって使用された最適化された設定。
- アプリケーションのデフォルトのオープンソース構成。

アプリケーションレベルでの設定の宣言と、ジョブ実行時の設定の上書きの詳細については、「[EMR Serverless のデフォルトのアプリケーション設定](#)」を参照してください。

Spark 動的リソース割り当ての最適化

EMR Serverless のリソース使用量を最適化するには、dynamicAllocationOptimization を使用します。このプロパティを Spark 設定分類で true に設定すると、EMR Serverless がエグゼキューターリソースの割り当てを最適化して、Spark がエグゼキューターをリクエストおよびキャンセルするレートと、EMR Serverless がワーカーを作成およびリリースするレートをより適切に調整できることを示します。これにより、EMR Serverless はステージ間でワーカーをより最適に再利用できるため、同じパフォーマンスを維持しながら複数のステージでジョブを実行する場合のコストが削減されます。

このプロパティは、すべての Amazon EMR リリースバージョンで使用できます。

以下は、dynamicAllocationOptimization を使用した設定分類の例です。

```
[
  {
    "Classification": "spark",
    "Properties": {
      "dynamicAllocationOptimization": "true"
    }
  }
]
```

動的割り当て最適化を使用している場合は、次の点を考慮してください。

- この最適化は、動的リソース割り当てを有効にした Spark ジョブで使用できます。
- 最高のコスト効率を実現するには、ワークロードに基づいてジョブレベル設定 `spark.dynamicAllocation.maxExecutors` または [アプリケーションレベルの最大容量](#) 設定を使用して、ワーカーの上限スケーリングを設定することを提案します。
- 単純なジョブではコスト改善が見られない場合があります。例えば、ジョブが小さなデータセットで実行する場合や、1つのステージで実行が終了する場合は、Spark は多数のエグゼキューターや複数のスケーリングイベントを必要としないことがあります。
- 大きなステージ、小さなステージ、そして再び大きなステージのシーケンスを持つジョブでは、ジョブの実行時にリグレッションが発生する可能性があります。EMR Serverless がリソースをより効率的に使用すると、より大きなステージで利用可能なワーカーが少なくなり、ランタイムが長くなる可能性があります。

Spark ジョブのプロパティ

次の表に、オプションの Spark プロパティと、Spark ジョブを送信するときに上書きできるデフォルト値を示します。

オプションの Spark プロパティとデフォルト値

キー	説明	デフォルトの値
<code>spark.archives</code>	Spark が各エグゼキューターの作業ディレクトリに抽出するアーカイブのコン	NULL

キー	説明	デフォルトの値
	マ区切りリスト。サポートされるタイプには .jar、.tar.gz、.tgz、.zip などがあります。抽出するディレクトリ名を指定するには、抽出するファイル名の後に # を追加します。例えば、file.zip#directory。	
spark.authenticate	Spark の内部接続の認証を有効にするオプション。	TRUE
spark.driver.cores	ドライバーが使用するコアの数。	4
spark.driver.extraJavaOptions	Spark ドライバーの追加 Java オプション。	NULL
spark.driver.memory	ドライバーが使用するメモリの量。	14G
spark.dynamicAllocation.enabled	動的リソース割り当てを有効にするオプション。このオプションは、ワークロードに基づいて、アプリケーションに登録されたエグゼキューターの数スケールアップまたはスケールダウンします。	TRUE
spark.dynamicAllocation.executorIdleTimeout	Spark がエグゼキューターを削除するまで、エグゼキューターがアイドル状態を維持できる時間の長さ。これは、動的割り当てを有効にした場合にのみ適用されます。	60 秒

キー	説明	デフォルトの値
<code>spark.dynamicAllocation.initialExecutors</code>	動的割り当てを有効にした場合に実行するエグゼキューターの初期数。	3
<code>spark.dynamicAllocation.maxExecutors</code>	動的割り当てを有効にした場合のエグゼキューター数の上限。	6.10.0 以降では infinity 6.9.0 以前では 100
<code>spark.dynamicAllocation.minExecutors</code>	動的割り当てを有効にした場合のエグゼキューター数の下限。	0
<code>spark.emr-serverless.allocation.batch.size</code>	エグゼキューター割り当ての各サイクルでリクエストするコンテナの数。各割り当てサイクル間には 1 秒のギャップがあります。	20
<code>spark.emr-serverless.driver.disk</code>	Spark ドライバーディスク。	20G
<code>spark.emr-serverless.driverEnv</code> [KEY]	Spark ドライバーに環境変数を追加するオプション。	NULL
<code>spark.emr-serverless.executor.disk</code>	Spark エグゼキューターディスク。	20G
<code>spark.emr-serverless.memoryOverheadFactor</code>	ドライバーとエグゼキューターのコンテナメモリに追加するメモリオーバーヘッドを設定します。	0.1
<code>spark.emr-serverless.driver.disk.type</code>	Spark ドライバーにアタッチされたディスクタイプ。	標準

キー	説明	デフォルトの値
<code>spark.emr-serverless.executor.disk.type</code>	Spark エグゼキューターにアタッチされたディスクタイプ。	標準
<code>spark.executor.cores</code>	各エグゼキューターが使用するコアの数。	4
<code>spark.executor.extraJavaOptions</code>	Spark エグゼキューターの追加 Java オプション。	NULL
<code>spark.executor.instances</code>	割り当てる Spark エグゼキューターコンテナの数。	3
<code>spark.executor.memory</code>	各エグゼキューターが使用するメモリの量。	14G
<code>spark.executorEnv. [KEY]</code>	Spark エグゼキューターに環境変数を追加するオプション。	NULL
<code>spark.files</code>	各エグゼキューターの作業ディレクトリに配置されるファイルのカンマ区切りリスト。これらのファイルのファイルパスには、 <code>SparkFiles.get(<i>fileName</i>)</code> を使用してエグゼキューターでアクセスできます。	NULL
<code>spark.hadoop.hive.metastore.client.factory.class</code>	Hive メタストア実装クラス。	NULL
<code>spark.jars</code>	ドライバーとエグゼキューターのランタイムクラスパスに追加する追加のジャー。	NULL

キー	説明	デフォルトの値
<code>spark.network.crypto.enabled</code>	AES ベースの RPC 暗号化を有効にするオプション。これには、Spark 2.2.0 で追加された認証プロトコルが含まれます。	FALSE
<code>spark.sql.warehouse.dir</code>	マネージドデータベースとテーブルのデフォルトの場所。	\$PWD/spark-warehouse の値
<code>spark.submit.pyFiles</code>	Python アプリの PYTHONPATH に配置する .zip、.egg、または .py ファイルのカンマ区切りリスト。	NULL

次の表に、デフォルトの Spark 送信パラメータを示します。

デフォルトの Spark 送信パラメータ

キー	説明	デフォルトの値
<code>archives</code>	Spark が各エグゼキューターの作業ディレクトリに抽出するアーカイブのカンマ区切りリスト。	NULL
<code>class</code>	アプリケーションのメインクラス (Java アプリおよび Scala アプリの場合)。	NULL
<code>conf</code>	任意の Spark 設定プロパティ。	NULL
<code>driver-cores</code>	ドライバーが使用するコアの数。	4

キー	説明	デフォルトの値
driver-memory	ドライバーが使用するメモリの量。	14G
executor-cores	各エグゼキューターが使用するコアの数。	4
executor-memory	エグゼキューターが使用するメモリの量。	14G
files	各エグゼキューターの作業ディレクトリに配置されるファイルのカンマ区切りリスト。これらのファイルのファイルパスには、SparkFile s.get(<i>fileName</i>) を使用してエグゼキューターでアクセスできます。	NULL
jars	ドライバーとエグゼキューターのクラスパスに含める jar のカンマ区切りリスト。	NULL
num-executors	起動するエグゼキューターの数。	3
py-files	Python アプリの PYTHONPATH に配置する .zip、.egg、または .py ファイルのカンマ区切りリスト。	NULL
verbose	追加のデバッグ出力を有効にするオプション。	NULL

リソース設定のベストプラクティス

StartJobRun API を使用したドライバーとエグゼキューターのリソースの設定

Note

Spark ドライバーとエグゼキューターのコアとメモリプロパティを指定する場合は、StartJobRun API リクエストで直接指定しなければなりません。

この方法でリソースを設定することにより、EMR Serverless はジョブを実行する前に正しいリソースを割り当てることができます。これは、.py ファイルや .jar ファイルなど、ユーザースクリプトで提供される設定とは対照的です。これは、ドライバーワーカーとエグゼキューターワーカーがスクリプトの実行を開始する前に事前プロビジョニングされる可能性があるため、評価が遅すぎます。ジョブの送信中にこれらのリソースを設定する方法は 2 つサポートされています:

オプション 1: sparkSubmitParameters を使用する

```
"jobDriver": {
  "sparkSubmit": {
    "entryPoint": "s3://your-script-path.py",
    "sparkSubmitParameters": "-conf spark.driver.memory=4g \
-conf spark.driver.cores=2 \
-conf spark.executor.memory=8g \
-conf spark.executor.cores=4"
  }
}
```

オプション 2: spark-defaults 分類に configurationOverrides を使用する

```
"configurationOverrides": {
  "applicationConfiguration": [
    {
      "classification": "spark-defaults",
      "properties": {
        "spark.driver.memory": "4g",
        "spark.driver.cores": "2",
        "spark.executor.memory": "8g",
        "spark.executor.cores": "4"
      }
    }
  ]
}
```

```
}  
]  
}
```

Spark の例

次の例では、StartJobRun API を使用して Python スクリプトを実行する方法を示しています。この例を使用するエンドツーエンドのチュートリアルについては、「[Amazon EMR Serverless の使用を開始する](#)」を参照してください。PySpark ジョブを実行し、Python 依存関係を追加する方法のその他の例は、[EMR Serverless Samples](#) GitHub リポジトリにあります。

```
aws emr-serverless start-job-run \  
  --application-id application-id \  
  --execution-role-arn job-role-arn \  
  --job-driver '{  
    "sparkSubmit": {  
      "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/  
wordcount/scripts/wordcount.py",  
      "entryPointArguments": ["s3://amzn-s3-demo-destination-bucket/  
wordcount_output"],  
      "sparkSubmitParameters": "--conf spark.executor.cores=1 --conf  
spark.executor.memory=4g --conf spark.driver.cores=1 --conf spark.driver.memory=4g --  
conf spark.executor.instances=1"  
    }  
  }'
```

次の例は、StartJobRun API を使用して Spark JAR を実行する方法を示しています。

```
aws emr-serverless start-job-run \  
  --application-id application-id \  
  --execution-role-arn job-role-arn \  
  --job-driver '{  
    "sparkSubmit": {  
      "entryPoint": "/usr/lib/spark/examples/jars/spark-examples.jar",  
      "entryPointArguments": ["1"],  
      "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi --conf  
spark.executor.cores=4 --conf spark.executor.memory=20g --conf spark.driver.cores=4 --  
conf spark.driver.memory=8g --conf spark.executor.instances=1"  
    }  
  }'
```

EMR Serverless ジョブ実行時の Hive 設定の使用

type パラメータを HIVE に設定して、アプリケーションで Hive ジョブを実行できます。ジョブは、Amazon EMR のリリースバージョンと互換性のある Hive バージョンと互換性がある必要があります。例えば、Amazon EMR リリース 6.6.0 のアプリケーションでジョブを実行する場合、ジョブは Apache Hive 3.1.2 と互換性がある必要があります。各リリースのアプリケーションバージョンの詳細については、「[Amazon EMR Serverless のリリースバージョン](#)」を参照してください。

Hive ジョブのパラメータ

[StartJobRun API](#) を使用して Hive ジョブを実行する場合は、次のパラメータを指定します。

必須パラメータ

- [Hive ジョブのランタイムロール](#)
- [Hive ジョブのドライバーパラメータ](#)
- [Hive 設定オーバーライドパラメータ](#)

Hive ジョブのランタイムロール

executionRoleArn を使用して、アプリケーションが Hive ジョブの実行に使用する IAM ロールの ARN を指定します。このロールには、以下のアクセス許可が含まれている必要があります。

- データが存在する S3 バケットまたはその他のデータソースからの読み取り
- Hive クエリファイルと init クエリファイルが存在する S3 バケットまたはプレフィックスからの読み取り
- Hive Scratch ディレクトリと Hive メタストアウェアハウスディレクトリが存在する S3 バケットへの読み取りと書き込み
- 最終出力を書き込む予定の S3 バケットへの書き込み
- S3MonitoringConfiguration が指定した S3 バケットまたはプレフィックスへのログの書き込み
- KMS キーを使用して S3 バケット内のデータを暗号化する場合の KMS キーへのアクセス
- AWS Glue データカタログへのアクセス

Hive ジョブが他のデータソースとの間でデータを読み書きする場合は、この IAM ロールに適切なアクセス許可を指定します。これらのアクセス許可を IAM ロールに指定しなければ、ジョブが失敗す

る可能性があります。詳細については、[Amazon EMR Serverless のジョブランタイムロール](#) を参照してください。

Hive ジョブのドライバーパラメータ

jobDriver を使用してジョブに入力します。ジョブのドライバーパラメータは、実行するジョブタイプに対して 1 つの値のみを受け入れます。hive をジョブタイプとして指定すると、EMR Serverless は Hive クエリを jobDriver パラメータに渡します。Hive ジョブには、以下のパラメータがあります。

- **query** – これは、実行する Hive クエリファイルへの Amazon S3 での参照です。
- **parameters** – 上書きする追加の Hive 設定プロパティです。プロパティを上書きするには、プロパティを `--hiveconf property=value` としてこのパラメータに渡します。変数を上書きするには、変数を `--hivevar key=value` としてこのパラメータに渡します。
- **initQueryFile** – これは init Hive クエリファイルです。Hive はクエリの前にこのファイルを実行し、このファイルでテーブルを初期化できます。

Hive 設定オーバーライドパラメータ

configurationOverrides を使用して、モニタリングレベルおよびアプリケーションレベルの設定プロパティを上書きします。このパラメータは、次の 2 つのフィールドを持つ JSON オブジェクトを受け入れます。

- **monitoringConfiguration** – このフィールドを使用して、EMR Serverless ジョブに Hive ジョブのログを保存する Amazon S3 URL (s3MonitoringConfiguration) を指定します。このバケットは、アプリケーションをホスト AWS アカウント すると同じと、ジョブが実行されている AWS リージョン と同じ で作成してください。
- **applicationConfiguration** – このフィールドに設定オブジェクトを指定すると、アプリケーションのデフォルト設定を上書きできます。短縮構文を使用して、設定を指定したり、JSON ファイルの設定オブジェクトを参照したりできます。設定オブジェクトは、分類、プロパティ、オプションの入れ子になっている設定で構成されます。プロパティは、そのファイル内で上書きする設定で構成されます。単一の JSON オブジェクトで、複数のアプリケーションに複数の分類を指定できます。

Note

特定の EMR Serverless リリースによって使用可能な設定分類は異なります。例えば、カスタム Log4j spark-driver-log4j2 と spark-executor-log4j2 の分類は、リリース 6.8.0 以降でのみ使用できます。

アプリケーションの上書きと Hive パラメータで同じ設定を渡すと、Hive パラメータが優先されます。次のリストは、設定を優先順位の高いものから低いものへとランク付けしたものです。

- `--hiveconf property=value` で Hive パラメータの一部として提供する設定。
- アプリケーションの一部として提供する設定は、ジョブの開始時に上書きされます。
- アプリケーションの作成時に `runtimeConfiguration` の一部として提供する設定。
- リリース用に Amazon EMR が割り当てた最適化された設定。
- アプリケーションのデフォルトのオープンソース設定。

アプリケーションレベルでの設定の宣言と、ジョブ実行時の設定の上書きの詳細については、「[EMR Serverless のデフォルトのアプリケーション設定](#)」を参照してください。

Hive ジョブプロパティ

次の表に、Hive ジョブの送信時に設定する必須プロパティを示します。

必須の Hive ジョブプロパティ

設定	説明
<code>hive.exec.scratchdir</code>	Hive ジョブの実行中に EMR Serverless が一時ファイルを作成する Amazon S3 の場所。
<code>hive.metastore.warehouse.dir</code>	Hive のマネージドテーブル用データベースの Amazon S3 の場所。

次の表に、オプションの Hive プロパティと、Hive ジョブを送信するときに上書きできるデフォルト値を示します。

オプションの Hive プロパティとデフォルト値

設定	説明	デフォルトの値
<code>fs.s3.customAWSCredentialsProvider</code>	使用する AWS 認証情報プロバイダー。	<code>com.amazonaws.auth.DefaultAWSCredentialsProviderChain</code>
<code>fs.s3a.aws.credentials.provider</code>	S3A ファイルシステムで使用する AWS 認証情報プロバイダー。	<code>com.amazonaws.auth.DefaultAWSCredentialsProviderChain</code>
<code>hive.auto.convert.join</code>	入力ファイルサイズに基づいた共通結合のマップ結合への自動変換を有効にするオプション。	TRUE
<code>hive.auto.convert.join.noconditionaltask</code>	Hive が入力ファイルサイズに基づいて共通結合をマップ結合に変換するときに最適化を有効にするオプション。	TRUE
<code>hive.auto.convert.join.noconditionaltask.size</code>	このサイズ以下では、結合はマップ結合に直接変換されます。	最適な値は Tez タスクメモリに基づいて計算されます
<code>hive.cbo.enable</code>	Calcite フレームワークでコストベースの最適化を有効にするオプション。	TRUE
<code>hive.cli.tez.session.async</code>	Hive クエリのコンパイル中にバックグラウンド Tez セッションを開始するオプション。false に設定すると、Hive クエリのコンパイル後に Tez AM が起動されます。	TRUE
<code>hive.compute.query.using.stats</code>	Hive をアクティブ化して、メタストアに保存された統計を	TRUE

設定	説明	デフォルトの値
	使用して特定のクエリに回答するオプション。基本的な統計については、 <code>hive.stats.autogather</code> を TRUE に設定します。クエリのより高度なコレクションについては、 <code>analyze table queries</code> を実行します。	
<code>hive.default.fileformat</code>	CREATE TABLE ステートメントのデフォルトのファイル形式。CREATE TABLE コマンドで STORED AS [FORMAT] を指定した場合、これを明示的に上書きできます。	TEXTFILE
<code>hive.driver.cores</code>	Hive ドライバークラスに使用するコアの数。	2
<code>hive.driver.disk</code>	Hive ドライバークラスのディスクサイズ。	20G
<code>hive.driver.disk.type</code>	Hive ドライバークラスのディスクタイプ。	標準
<code>hive.tez.disk.type</code>	tez ワーカーのディスクサイズ。	標準
<code>hive.driver.memory</code>	Hive ドライバークラスごとに使用するメモリ量。Hive CLI と Tez Application Master は、このメモリを 20% のヘッドルームを取って均等に共有します。	6G

設定	説明	デフォルトの値
hive.emr-serverless.launch.env.[KEY]	Hive ドライバー、Tez AM、Tez タスクなど、すべての Hive 固有のプロセスで KEY 環境変数を設定するオプション。	
hive.exec.dynamic.partition	DML/DDL で動的パーティションを有効にするオプション。	TRUE
hive.exec.dynamic.partition.mode	厳格モードを使用するか、非厳格モードを使用するかを指定するオプション。厳格モードでは、すべてのパーティションを誤って上書きする場合に備えて、少なくとも1つの静的パーティションを指定します。非厳格モードでは、すべてのパーティションを動的にすることができます。	strict
hive.exec.max.dynamic.partitions	Hive が全体で作成する動的パーティションの最大数。	1,000
hive.exec.max.dynamic.partitions.per node	Hive がマッパードとリデューサーノードのそれぞれに作成する動的パーティションの最大数。	100

設定	説明	デフォルトの値
hive.exec.orc.split.strategy	次のいずれかの値が予想されます。BI、ETL、HYBRID。これはユーザーレベルの設定ではありません。BI は、クエリの実行よりも分割生成に費やす時間を減らすように指定します。ETL は分割生成により多くの時間を費やすように指定します。HYBRID は、ヒューリスティックに基づいて上記の戦略の選択を指定します。	HYBRID
hive.exec.reducers.bytes.per.reducer	リデューサーあたりのサイズ。デフォルトは 256 MB です。入力サイズが 1G の場合、ジョブは 4 つのリデューサーを使用します。	256000000
hive.exec.reducers.max	リデューサーの最大数。	256
hive.exec.stagingdir	Hive がテーブルの場所内および hive.exec.scratchdir プロパティで指定されたスクラッチディレクトリの場所内に作成する一時ファイルを保存するディレクトリの名前。	.hive-staging

設定	説明	デフォルトの値
<code>hive.fetch.task.conversion</code>	次のいずれかの値が予想されます。NONE、MINIMAL、MORE。トは、選択クエリを 1 つの FETCH タスクに変換できます。これにより、レイテンシーが最小限に抑えられます。	MORE
<code>hive.groupby.position.alias</code>	Hive が GROUP BY ステートメントで列位置エイリアスを使用するようにするオプション。	FALSE
<code>hive.input.format</code>	デフォルトの入力形式。CombineHiveInputFormat で問題が発生した場合は、HiveInputFormat に設定します。	<code>org.apache.hadoop.hive ql.io.CombineHiveInputFormat</code>
<code>hive.log.explain.output</code>	Hive ログ内のクエリの拡張出力の説明を有効にするオプション。	FALSE
<code>hive.log.level</code>	Hive ロギングレベル。	INFO
<code>hive.mapred.reduce.tasks.speculative.execution</code>	リデューサーの投機的起動を有効にするオプション。Amazon EMR 6.10.x 以前でのみサポートされています。	TRUE

設定	説明	デフォルトの値
<code>hive.max-task-containers</code>	同時コンテナの最大数。設定されたマッパーメモリにこの値を乗じて、分割計算とタスクプリエンプションが使用する利用可能なメモリを決定します。	1,000
<code>hive.merge.mapfiles</code>	マップのみのジョブの最後に小さなファイルがマージされるオプション。	TRUE
<code>hive.merge.size.per.task</code>	ジョブの最後にマージされたファイルのサイズ。	256000000
<code>hive.merge.tezfiles</code>	Tez DAG の最後に小さなファイルのマージを有効にするオプション。	FALSE
<code>hive.metastore.client.factory.class</code>	IMetaStoreClient インターフェイスを実装するオブジェクトを生成するファクトリークラスの名前。	<code>com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory</code>
<code>hive.metastore.glue.catalogid</code>	AWS Glue データカタログがメタストアとして機能し、ジョブとは異なる AWS アカウント で実行される場合、ジョブが実行されている AWS アカウント の ID。	NULL
<code>hive.metastore.uris</code>	メタストアクライアントがリモートメタストアへの接続に使用する thrift URI。	NULL
<code>hive.optimize.ppd</code>	述語プッシュダウンを有効にするオプション。	TRUE

設定	説明	デフォルトの値
hive.optimize.ppd.storage	ストレージハンドラーへの述語プッシュダウンを有効にするオプション。	TRUE
hive.orderby.position.alias	Hive が ORDER BY ステートメントで列位置エイリアスを使用するようにするオプション。	TRUE
hive.prewarm.enabled	Tez のコンテナプレウォームをオンにするオプション。	FALSE
hive.prewarm.numcontainers	Tez でプレウォームするコンテナの数。	10
hive.stats.autogather	INSERT OVERWRITE コマンド中に Hive が基本統計を自動的に収集できるようにするオプション。	TRUE
hive.stats.fetch.column.stats	メタストアからの列統計の取得を無効にするオプション。列の数が多い場合、列統計の取得は高価になる可能性があります。	FALSE
hive.stats.gather.num.threads	partialscan および noscan ANALYZE コマンドがパーティションテーブルに使用するスレッドの数。これは、StatsProvidingRecordReader を実装するファイル形式 (ORC など) にのみ適用されます。	10

設定	説明	デフォルトの値
hive.strict.checks.cartesian.product	厳密なデカルト結合チェックを有効にするオプション。これらのチェックでは、直積集合 (クロス結合) は許可されません。	FALSE
hive.strict.checks.type.safety	厳密なタイプの安全チェックを有効にし、string と double の両方と bigint の比較を無効にするオプション。	TRUE
hive.support.quoted.identifiers	NONE または COLUMN の値が必要です。NONE は、英数字とアンダースコア文字のみが識別子として有効であることを示します。COLUMN は、列名に任意の文字を含めることができることを示します。	COLUMN
hive.tez.auto.reducer.parallelism	Tez 自動リデューサー並列処理機能を有効にするオプション。Hive は引き続きデータサイズを推定し、並列処理の推定値を設定します。Tez はソース頂点の出力サイズをサンプリングし、必要に応じて実行時に推定値を調整します。	TRUE
hive.tez.container.size	Tez タスクプロセスごとに使用するメモリ量。	6144
hive.tez.cpu.vcores	Tez タスクごとに使用するコアの数。	2

設定	説明	デフォルトの値
<code>hive.tez.disk.size</code>	各タスクコンテナのディスクサイズ。	20G
<code>hive.tez.input.format</code>	Tez AM での分割生成の入力形式。	<code>org.apache.hadoop.hive ql.io.HiveInputFormat</code>
<code>hive.tez.min.partition.factor</code>	自動リデューサー並列処理を有効にするときに Tez が指定するリデューサーの下限。	0.25
<code>hive.vectorized.execution.enabled</code>	ベクトル化されたクエリ実行モードを有効にするオプション。	TRUE
<code>hive.vectorized.execution.reduce.enabled</code>	クエリ実行のリデュース側のベクトル化モードを有効にするオプション。	TRUE
<code>javax.jdo.option.ConnectionDriverName</code>	JDBC メタストアのドライバークラス名。	<code>org.apache.derby.jdbc.EmbeddedDriver</code>
<code>javax.jdo.option.ConnectionPassword</code>	メタストアデータベースに関連付けられたパスワード。	NULL
<code>javax.jdo.option.ConnectionURL</code>	JDBC メタストアの JDBC 接続文字列。	<code>jdbc:derby:::databaseName=metastore_db;create=true</code>
<code>javax.jdo.option.ConnectionUserName</code>	メタストアデータベースに関連付けられたユーザー名。	NULL

設定	説明	デフォルトの値
<code>mapreduce.input.fileinputformat.split.maxsize</code>	入力形式が <code>org.apache.hadoop.hive ql.io.CombineHiveInputFormat</code> の場合、分割計算時の分割の最大サイズ。値 0 (ゼロ) は、制限がないことを示します。	0
<code>tez.am.dag.cleanup.on.completion</code>	DAG の完了時にシャッフルデータのクリーンアップを有効にするオプション。	TRUE
<code>tez.am.emr-serverless.launch.env.[KEY]</code>	Tez AM プロセスで KEY 環境変数を設定するオプション。Tez AM の場合、この値は <code>hive.emr-serverless.launch.env.[KEY]</code> 値を上書きします。	
<code>tez.am.log.level</code>	EMR Serverless が Tez App プライマリーに渡すルートロギングレベル。	INFO
<code>tez.am.sleep.time.before.exit.millis</code>	EMR Serverless は、AM シャットダウンリクエストからこの期間の後に ATS イベントをプッシュする必要があります。	0

設定	説明	デフォルトの値
<code>tez.am.speculation.enabled</code>	遅いタスクを投機的に起動するオプション。これにより、マシンの不良や遅延のために一部のタスクの実行が遅くなっている場合、ジョブのレイテンシーを減らすことができます。Amazon EMR 6.10.x 以前でのみサポートされています。	FALSE
<code>tez.am.task.max.failed.attempts</code>	特定のタスクが失敗するまでの最大試行回数。この数は、手動で終了した試行数をカウントしません。	3
<code>tez.am.vertex.cleanup.height</code>	すべての依存頂点が完了すると、Tez AM が頂点シャッフルデータを削除する距離。この機能は、値が 0 の場合に無効になります。Amazon EMR バージョン 6.8.0 以降でこの機能がサポートされています。	0
<code>tez.client.asynchronous-stop</code>	EMR Serverless が Hive ドライバーを終了する前に ATS イベントをプッシュするオプション。	FALSE
<code>tez.grouping.max-size</code>	グループ化された分割のサイズの上限 (バイト単位)。この制限により、過度に大きな分割が防止されます。	1073741824

設定	説明	デフォルトの値
<code>tez.grouping.min-size</code>	グループ化された分割のサイズの下限 (バイト単位)。この制限により、小さい分割が多発するのが防止されます。	16777216
<code>tez.runtime.io.sort.mb</code>	Tez が出力をソートするときのソフトバッファのサイズ。	最適な値は Tez タスクメモリに基づいて計算されます
<code>tez.runtime.unordered.output.buffer.size-mb</code>	Tez がディスクに直接書き込まない場合に使用するバッファのサイズ。	最適な値は Tez タスクメモリに基づいて計算されます
<code>tez.shuffle-vertex-manager.max-src-fraction</code>	EMR Serverless が現在の頂点のすべてのタスクをスケジューリングする前に完了する必要があるソースタスクの割合 (ScatterGather 接続の場合)。現在の頂点でスケジューリングできるタスクの数は、 <code>min-fraction</code> と <code>max-fraction</code> の間で直線的にスケールされます。これはデフォルト値または <code>tez.shuffle-vertex-manager.min-src-fraction</code> のいずれか大きい方がデフォルトになります。	0.75
<code>tez.shuffle-vertex-manager.min-src-fraction</code>	EMR Serverless が現在の頂点のタスクをスケジューリングする前に完了する必要があるソースタスクの割合 (ScatterGather 接続の場合)。	0.25

設定	説明	デフォルトの値
<code>tez.task.emr-serverless.launch.env.[<i>KEY</i>]</code>	Tez タスクプロセスで <i>KEY</i> 環境変数を設定するオプション。Tez タスクの場合、この値は <code>hive.emr-serverless.launch.env.[<i>KEY</i>]</code> 値を上書きします。	
<code>tez.task.log.level</code>	EMR Serverless が Tez タスクに渡すルートログインレベル。	INFO
<code>tez.yarn.ats.event .flush.timeout.millis</code>	シャットダウンする前にイベントがフラッシュされるまでの AM の最大待機時間。	300000

Hive ジョブの例

次のコード例は、StartJobRun API を使用して Hive クエリを実行する方法を示しています。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "hive": {
      "query": "s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-
query.q1",
      "parameters": "--hiveconf hive.log.explain.output=false"
    }
  }' \
  --configuration-overrides '{
    "applicationConfiguration": [{
      "classification": "hive-site",
      "properties": {
        "hive.exec.scratchdir": "s3://amzn-s3-demo-bucket/emr-serverless-hive/
hive/scratch",
        "hive.metastore.warehouse.dir": "s3://amzn-s3-demo-bucket/emr-
serverless-hive/hive/warehouse",
        "hive.driver.cores": "2",
```

```

        "hive.driver.memory": "4g",
        "hive.tez.container.size": "4096",
        "hive.tez.cpu.vcores": "1"
    }
}
}'

```

Hive ジョブを実行する方法のその他の例は、[EMR Serverless Samples](#) GitHub リポジトリにあります。

EMR Serverless ジョブの耐障害性

EMR Serverless リリース 7.1.0 以降には、ジョブの耐障害性のサポートが含まれているため、失敗したジョブはユーザーによる手動入力を必要とせず自動的に再試行されます。ジョブの障害耐性のもう 1 つの利点は、AZ で問題が発生した場合に EMR Serverless がジョブ実行を異なるアベイラビリティゾーン (AZ) に移動することです。

ジョブの障害耐性を有効にするには、ジョブの再試行ポリシーを設定します。再試行ポリシーは、ジョブがいずれかの時点で失敗した場合に EMR Serverless が自動的に再起動するようにします。再試行ポリシーはバッチジョブとストリーミングジョブでサポートされるため、ユースケースに応じてジョブの耐障害性をカスタマイズします。次の表は、バッチジョブとストリーミングジョブ全体におけるジョブの耐障害性の動作と相違点を比較したものです。

	バッチジョブ	ストリーミングジョブ
デフォルトの 動作	ジョブを再実行しません。	アプリケーションがジョブの実行中にチェックポイントを作成するため、常にジョブの実行を再試行します。
再試行ポイント	バッチジョブにはチェックポイントがないため、EMR Serverless は常にジョブを最初から再実行します。	ストリーミングジョブはチェックポイントをサポートしているため、ランタイム状態を保存し、Amazon S3 のチェックポイントの場所に進むようにストリーミングクエリを設定します。EMR Serverless は、チェックポイントからのジョブ実行

	バッチジョブ	ストリーミングジョブ
		を再開します。詳細については、Apache Spark ドキュメントの「 Recovering from failures with Checkpointing 」を参照してください。
再試行の最大回数	最大 10 回の再試行を許可します。	ストリーミングジョブにはスラッシュ防止コントロールが組み込まれているため、1 時間にわたってジョブが失敗し続けると、アプリケーションは再試行を停止します。1 時間以内のデフォルトの再試行回数は 5 回です。この再試行回数は 1～10 に設定できます。最大試行回数をカスタマイズすることはできません。値 1 は、再試行が行われないことを示します。

EMR Serverless はジョブの再実行を試みる際に、ジョブに試行番号のインデックスも付加するため、試行全体でジョブのライフサイクルを追跡します。

EMR Serverless API オペレーションまたは を使用して AWS CLI、ジョブの耐障害性を変更したり、ジョブの耐障害性に関連する情報にアクセスしたりできます。詳細については、「[EMR Serverless API guide](#)」を参照してください。

デフォルトでは、EMR Serverless はバッチジョブを再実行しません。バッチジョブの再試行を有効にするには、バッチジョブの実行を開始するときに `maxAttempts` パラメータを設定します。`maxAttempts` パラメータはバッチジョブにのみ適用されます。デフォルトは 1 です。つまり、ジョブを再実行しません。使用できる値は 1～10 (両端の値を含む) です。

次の例は、ジョブ実行の開始時に最大試行回数として 10 回を指定する方法を示しています。

```
aws emr-serverless start-job-run
  --application-id <APPLICATION_ID> \
```

```
--execution-role-arn <JOB_EXECUTION_ROLE> \
--mode 'BATCH' \
--retry-policy '{
  "maxAttempts": 10
}' \
--job-driver '{
  "sparkSubmit": {
    "entryPoint": "/usr/lib/spark/examples/jars/spark-examples-does-not-
exist.jar",
    "entryPointArguments": ["1"],
    "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi"
  }
}'
```

EMR Serverless は、失敗したストリーミングジョブを無期限に再試行します。回復不可能な障害が繰り返し発生することによるスラッシングを回避するには、`maxFailedAttemptsPerHour` を使用してストリーミングジョブの再試行のスラッシュ防止コントロールを設定します。このパラメータを使用すると、EMR Serverless が再試行を停止するまでの 1 時間に許容される失敗の最大試行回数を指定できます。デフォルトは 5 回です。使用できる値は 1~10 (両端の値を含む) です。

```
aws emr-serverless start-job-run
--application-id <APPLICATION_ID> \
--execution-role-arn <JOB_EXECUTION_ROLE> \
--mode 'STREAMING' \
--retry-policy '{
  "maxFailedAttemptsPerHour": 7
}' \
--job-driver '{
  "sparkSubmit": {
    "entryPoint": "/usr/lib/spark/examples/jars/spark-examples-does-not-
exist.jar",
    "entryPointArguments": ["1"],
    "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi"
  }
}'
```

他のジョブ実行 API オペレーションを使用して、ジョブに関する情報を取得することもできます。例えば、`GetJobRun` オペレーションで `attempt` パラメータを使用して、特定のジョブの試行に関する詳細情報を取得します。`attempt` パラメータを設定しない場合は、オペレーションによって最新の試行に関する情報が返されます。

```
aws emr-serverless get-job-run \  
  --job-run-id job-run-id \  
  --application-id application-id \  
  --attempt 1
```

ListJobRunAttempts オペレーションは、ジョブ実行に関連するすべての試行に関する情報を返します。

```
aws emr-serverless list-job-run-attempts \  
  --application-id application-id \  
  --job-run-id job-run-id
```

GetDashboardForJobRun オペレーションは、ジョブ実行のアプリケーション UI にアクセスするために使用する URL を作成して返します。attempt パラメータを使用すると、特定の試行の URL を取得できます。attempt パラメータを設定しない場合は、オペレーションによって最新の試行に関する情報が返されます。

```
aws emr-serverless get-dashboard-for-job-run \  
  --application-id application-id \  
  --job-run-id job-run-id \  
  --attempt 1
```

再試行ポリシーを使用してジョブをモニタリングする

ジョブの耐障害性サポートでは、新しいイベント EMR Serverless ジョブ実行の再試行も追加されます。EMR Serverless は、ジョブの再試行ごとにこのイベントを発行します。この通知を使用して、ジョブの再試行を追跡できます。イベントの詳細については、「[Amazon EventBridge events](#)」を参照してください。

再試行ポリシーを使用したログ記録

EMR Serverless がジョブを再試行するたびに、試行によって独自のログセットが生成されます。EMR Serverless がこれらのログを上書きせずに Amazon S3 と Amazon CloudWatch に正常に配信できるようにするため、EMR Serverless は S3 ログパスと CloudWatch ログストリーム名の形式にプレフィックスを追加して、ジョブの試行数を設定します。

以下に示しているのは、形式の具体的な例です。

```
 '/applications/<applicationId>/jobs/<jobId>/attempts/<attemptNumber>/'.
```

この形式により、EMR Serverless はジョブの試行ごとにすべてのログを Amazon S3 および CloudWatch の独自の指定場所に発行します。詳細については、「[Storing logs](#)」を参照してください。

Note

EMR Serverless は、すべてのストリーミングジョブと再試行が有効になっているバッチジョブでのみ、このプレフィックス形式を使用します。

EMR Serverless のメタストア設定

Hive メタストアは、スキーマ、パーティション名、データ型など、テーブルに関する構造情報を一元的に保存する場所です。EMR Serverless を使用することにより、このテーブルメタデータをジョブにアクセスできるメタストアに保持します。

Hive メタストアとして 2 つのオプションがあります。

- AWS Glue データカタログ
- 外部の Apache Hive メタストア

AWS Glue データカタログをメタストアとして使用する

Glue データカタログをメタストアとして使用するよう Spark AWS ジョブと Hive ジョブを設定できます。永続的なメタストア、またはさまざまなアプリケーション、サービス、あるいは AWS アカウントで共有されるメタストアが必要である場合は、この設定をお勧めします。データカタログの詳細については、「[AWS Glue データカタログの入力](#)」を参照してください。AWS Glue の料金については、「[AWS Glue の料金](#)」を参照してください。

EMR Serverless ジョブは、AWS アカウント アプリケーションと同じ または別の で AWS Glue データカタログを使用するように設定できます AWS アカウント。

AWS Glue データカタログを設定する

データカタログを設定するには、使用する EMR Serverless アプリケーションのタイプを選択します。

Spark

EMR Studio を使用して EMR Serverless Spark アプリケーションでジョブを実行する場合、AWS Glue データカタログがデフォルトのメタストアです。

SDKs または を使用する場合は AWS CLI、ジョブ実行の `sparkSubmit` パラメータ `com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory` で `spark.hadoop.hive.metastore.client.factory.class` 設定を に設定します。次の例は、AWS CLIを使用してデータカタログを設定する方法を示しています。

```
aws emr-serverless start-job-run \  
  --application-id application-id \  
  --execution-role-arn job-role-arn \  
  --job-driver '{  
    "sparkSubmit": {  
      "entryPoint": "s3://amzn-s3-demo-bucket/code/pyspark/  
extreme_weather.py",  
      "sparkSubmitParameters": "--conf  
spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory  
--conf spark.driver.cores=1 --conf spark.driver.memory=3g --conf  
spark.executor.cores=4 --conf spark.executor.memory=3g"  
    }  
  }'
```

または、Spark コードで新しい `SparkSession` を作成するときにこの設定をすることができま

```
from pyspark.sql import SparkSession  
  
spark = (  
    SparkSession.builder.appName("SparkSQL")  
    .config(  
        "spark.hadoop.hive.metastore.client.factory.class",  
        "com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory",  
    )  
    .enableHiveSupport()  
    .getOrCreate()  
)  
  
# we can query tables with SparkSQL  
spark.sql("SHOW TABLES").show()
```



```
# we can also them with native Spark
print(spark.catalog.listTables())
```

Hive

EMR Serverless Hive アプリケーションの場合、データカタログはデフォルトのメタストアです。つまり、EMR Serverless Hive アプリケーションでジョブを実行すると、Hive は AWS アカウント アプリケーションと同じ 内のデータカタログにメタストア情報を記録します。データカタログをメタストアとして使用するのに、仮想プライベートクラウド (VPC) は必要ありません。

Hive メタストアテーブルにアクセスするには、「Glue AWS の [IAM アクセス許可の設定](#)」で説明されている必要な [AWS Glue](#) ポリシーを追加します。

EMR Serverless および AWS Glue データカタログのクロスアカウントアクセスを設定する

EMR Serverless のクロスアカウントアクセスを設定するには、まず以下にサインインします AWS アカウント。

- AccountA – EMR Serverless アプリケーションを作成 AWS アカウント している。
 - AccountB – EMR Serverless AWS ジョブを実行する Glue データカタログ AWS アカウント を含む。
1. AccountB の管理者またはその他の承認された ID が、AccountB のデータカタログにリソースポリシーをアタッチしていることを確認します。このポリシーは、AccountB カタログ内のリソースに対してオペレーションを実行するための特定のクロスアカウントアクセス許可を AccountA に付与します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "glue:GetDatabase",
        "glue:CreateDatabase",
        "glue:GetDataBases",
```

```

        "glue:CreateTable",
        "glue:GetTable",
        "glue:UpdateTable",
        "glue>DeleteTable",
        "glue:GetTables",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:CreatePartition",
        "glue:BatchCreatePartition",
        "glue:GetUserDefinedFunctions"
    ],
    "Resource": [
        "arn:aws:glue:*:123456789012:catalog"
    ],
    "Sid": "AllowGLUEGetdatabase"
}
]
}

```

2. IAM ポリシーを AccountA の EMR Serverless ジョブランタイムロールに追加して、ロールが AccountB のデータカタログリソースにアクセスできるようにします。

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "glue:GetDatabase",
        "glue:CreateDatabase",
        "glue:GetDataBases",
        "glue:CreateTable",
        "glue:GetTable",
        "glue:UpdateTable",
        "glue>DeleteTable",
        "glue:GetTables",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:CreatePartition",
        "glue:BatchCreatePartition",
        "glue:GetUserDefinedFunctions"
      ]
    }
  ]
}

```

```

    ],
    "Resource": [
        "arn:aws:glue:*:123456789012:catalog"
    ],
    "Sid": "AllowGLUEGetdatabase"
  }
]
}

```

3. ジョブの実行を開始します。このステップは、AccountA の EMR Serverless アプリケーションタイプによって少し異なります。

Spark

以下の例に示されているように、`spark.hadoop.hive.metastore.glue.catalogid` プロパティを `sparkSubmitParameters` に渡します。*AccountB-catalog-id* は、AccountB のデータカタログの ID に置き換えます。

```

aws emr-serverless start-job-run \
--application-id "application-id" \
--execution-role-arn "job-role-arn" \
--job-driver '{
  "sparkSubmit": {
    "entryPoint": "s3://amzn-s3-demo-bucket/scripts/test.py",
    "sparkSubmitParameters": "--conf
spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.A
--conf spark.hadoop.hive.metastore.glue.catalogid=AccountB-catalog-
id --conf spark.executor.cores=1 --conf spark.executor.memory=1g
--conf spark.driver.cores=1 --conf spark.driver.memory=1g --conf
spark.executor.instances=1"
  }
}' \
--configuration-overrides '{
  "monitoringConfiguration": {
    "s3MonitoringConfiguration": {
      "logUri": "s3://amzn-s3-demo-bucket/logs/"
    }
  }
}'

```

Hive

以下の例に示されているように、hive.metastore.glue.catalogid プロパティを hive-site 分類に設定します。*AccountB-catalog-id* は、AccountB のデータカタログの ID に置き換えます。

```
aws emr-serverless start-job-run \  
--application-id "application-id" \  
--execution-role-arn "job-role-arn" \  
--job-driver '{  
    "hive": {  
        "query": "s3://amzn-s3-demo-bucket/hive/scripts/create_table.sql",  
        "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/  
hive/scratch --hiveconf hive.metastore.warehouse.dir=s3://amzn-s3-demo-bucket/  
hive/warehouse"  
    }  
' \  
--configuration-overrides '{  
    "applicationConfiguration": [{  
        "classification": "hive-site",  
        "properties": {  
            "hive.metastore.glue.catalogid": "AccountB-catalog-id"  
        }  
    }]  
'
```

AWS Glue データカタログを使用する際の考慮事項

Hive スクリプトで ADD JAR に補助 JAR を追加できます。その他の考慮事項については、「[Glue Data Catalog AWS を使用する際の考慮事項](#)」を参照してください。

外部の Hive メタストアの使用

EMR Serverless Spark ジョブと Hive ジョブを設定して、Amazon Aurora や Amazon RDS for MySQL などの外部 Hive メタストアに接続できます。このセクションでは、Amazon RDS Hive メタストアの設定方法、VPC の設定方法、外部メタストアを使用するように EMR Serverless ジョブを設定する方法について説明します。

外部の Hive メタストアを作成する

1. [VPC を作成する](#) の手順に従って、プライベートサブネットを使用して Amazon Virtual Private Cloud (Amazon VPC) を作成します。
2. 新しい Amazon VPC とプライベートサブネットを使用して EMR Serverless アプリケーションを作成します。EMR Serverless アプリケーションを VPC で設定すると、最初に、指定したサブネットごとに Elastic Network Interface がプロビジョニングされます。次に、指定されたセキュリティグループをそのネットワークインターフェイスにアタッチします。これにより、アプリケーションへのアクセスコントロールが可能になります。VPC のセットアップ方法の詳細については、「[データに接続するための EMR Serverless アプリケーションの VPC アクセスの設定](#)」を参照してください。
3. Amazon VPC のプライベートサブネットに MySQL または Aurora PostgreSQL データベースを作成します。Amazon RDS データベースを作成する方法については、「[Amazon RDS DB インスタンスの作成](#)」を参照してください。
4. [Amazon RDS DB インスタンスを変更する](#) の手順に従って、MySQL または Aurora データベースのセキュリティグループを変更して、EMR Serverless セキュリティグループからの JDBC 接続を許可します。EMR Serverless セキュリティグループの 1 つから RDS セキュリティグループへのインバウンドトラフィックのルールを追加します。

タイプ	プロトコル	ポート範囲	ソース
すべての TCP	TCP	3306	emr-serverless-security-group

Spark オプションの設定

JDBC の使用

Amazon RDS for MySQL または Amazon Aurora MySQL インスタンスに基づいて Hive メタストアに接続するように EMR Serverless Spark アプリケーションを設定するには、JDBC 接続を使用します。ジョブ実行の `spark-submit` パラメータで `--jars` の付いた `mariadb-connector-java.jar` を渡します。

```
aws emr-serverless start-job-run \
  --application-id "application-id" \
  --execution-role-arn "job-role-arn" \
```

```
--job-driver '{
    "sparkSubmit": {
        "entryPoint": "s3://amzn-s3-demo-bucket/scripts/spark-jdbc.py",
        "sparkSubmitParameters": "--jars s3://amzn-s3-demo-bucket/mariadb-
connector-java.jar
        --conf
spark.hadoop.javax.jdo.option.ConnectionDriverName=org.mariadb.jdbc.Driver
        --conf spark.hadoop.javax.jdo.option.ConnectionUserName=<connection-user-
name>
        --conf spark.hadoop.javax.jdo.option.ConnectionPassword=<connection-
password>
        --conf spark.hadoop.javax.jdo.option.ConnectionURL=<JDBC-Connection-
string>
        --conf spark.driver.cores=2
        --conf spark.executor.memory=10G
        --conf spark.driver.memory=6G
        --conf spark.executor.cores=4"
    }
}' \
--configuration-overrides '{
    "monitoringConfiguration": {
        "s3MonitoringConfiguration": {
            "logUri": "s3://amzn-s3-demo-bucket/spark/logs/"
        }
    }
}'
```

次のコード例は、Amazon RDS の Hive メタストアとやり取りする Spark エントリポイントスクリプトです。

```
from os.path import expanduser, join, abspath
from pyspark.sql import SparkSession
from pyspark.sql import Row
# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')
spark = SparkSession \
    .builder \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .enableHiveSupport() \
    .getOrCreate()
spark.sql("SHOW DATABASES").show()
spark.sql("CREATE EXTERNAL TABLE `sampledb`.`sparknyctaxi`(`dispatching_base_num`
string, `pickup_datetime` string, `dropoff_datetime` string, `pulocationid` bigint,
```

```
`dolocationid` bigint, `sr_flag` bigint) STORED AS PARQUET LOCATION 's3://<s3 prefix>/
nyctaxi_parquet/')
spark.sql("SELECT count(*) FROM sampled_b.sparknyctaxi").show()
spark.stop()
```

thrift サービスの使用

Amazon RDS for MySQL または Amazon Aurora MySQL インスタンスに基づいて Hive メタストアに接続するように EMR Serverless Hive アプリケーションを設定できます。これを行うには、既存の Amazon EMR クラスターのプライマリノードで thrift サーバーを実行します。このオプションは、EMR Serverless ジョブ設定を簡素化するために使用する thrift サーバーを備えた Amazon EMR クラスターが既にある場合に最適です。

```
aws emr-serverless start-job-run \
  --application-id "application-id" \
  --execution-role-arn "job-role-arn" \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "s3://amzn-s3-demo-bucket/thriftscript.py",
      "sparkSubmitParameters": "--jars s3://amzn-s3-demo-bucket/mariadb-
connector-java.jar
      --conf spark.driver.cores=2
      --conf spark.executor.memory=10G
      --conf spark.driver.memory=6G
      --conf spark.executor.cores=4"
    }
  }' \
  --configuration-overrides '{
    "monitoringConfiguration": {
      "s3MonitoringConfiguration": {
        "logUri": "s3://amzn-s3-demo-bucket/spark/logs/"
      }
    }
  }'
```

次のコード例は、thrift プロトコルを使用して Hive メタストアに接続するエントリポイントスクリプト (thriftscript.py) です。hive.metastore.uris プロパティは、外部 Hive メタストアから読み取るように設定する必要があります。

```
from os.path import expanduser, join, abspath
from pyspark.sql import SparkSession
from pyspark.sql import Row
```

```
# warehouse_location points to the default location for managed databases and tables
warehouse_location = abspath('spark-warehouse')
spark = SparkSession \
    .builder \
    .config("spark.sql.warehouse.dir", warehouse_location) \
    .config("hive.metastore.uris", "thrift://thrift-server-host:thrift-server-port") \
    .enableHiveSupport() \
    .getOrCreate()
spark.sql("SHOW DATABASES").show()
spark.sql("CREATE EXTERNAL TABLE sampled.`sparknyctaxi`(`dispatching_base_num`
    string, `pickup_datetime` string, `dropoff_datetime` string, `pulocationid` bigint,
    `dolocationid` bigint, `sr_flag` bigint) STORED AS PARQUET LOCATION 's3://<s3 prefix>/
    nyctaxi_parquet/'")
spark.sql("SELECT * FROM sampled.sparknyctaxi").show()
spark.stop()
```

Hive オプションの設定

JDBC の使用

Amazon RDS MySQL または Amazon Aurora インスタンスのいずれかで外部 Hive データベースの場合を指定する場合は、デフォルトのメタストア設定を上書きできます。

Note

Hive では、メタストアテーブルへの複数の書き込みを同時に実行できます。2 つのジョブ間でメタストア情報を共有する場合は、同じメタストアテーブルに同時に書き込まないようにします (同じメタストアテーブルの別のパーティションに書き込む場合は除きます)。

hive-site 分類で次の設定を行って、外部 Hive メタストアをアクティブ化します。

```
{
    "classification": "hive-site",
    "properties": {
        "hive.metastore.client.factory.class":
"org.apache.hadoop.hive.ql.metadata.SessionHiveMetaStoreClientFactory",
        "javax.jdo.option.ConnectionDriverName": "org.mariadb.jdbc.Driver",
        "javax.jdo.option.ConnectionURL": "jdbc:mysql://db-host:db-port/db-name",
        "javax.jdo.option.ConnectionUserName": "username",
        "javax.jdo.option.ConnectionPassword": "password"
    }
}
```



```
}
```

thrift サーバーの使用

Amazon RDS for MySQL または Amazon Aurora MySQL インスタンスに基づいて Hive メタストアに接続するように EMR Serverless Hive アプリケーションを設定できます。これを行うには、既存の Amazon EMR クラスターのメインノードで thrift サーバーを実行します。このオプションは、thrift サーバーを実行する Amazon EMR クラスターが既にあり、EMR Serverless ジョブ設定を使用する場合に最適です。

EMR Serverless がリモート thrift メタストアにアクセスできるように、hive-site 分類で次の設定を行います。外部 Hive メタストアから読み取るように hive.metastore.uris プロパティを設定する必要があります。

```
{
  "classification": "hive-site",
  "properties": {
    "hive.metastore.client.factory.class":
"org.apache.hadoop.hive.q1.metadata.SessionHiveMetaStoreClientFactory",
    "hive.metastore.uris": "thrift://thrift-server-host:thrift-server-port"
  }
}
```

EMR Serverless AWS での Glue マルチカタログ階層の使用

Glue マルチカタログ階層で動作するように EMR Serverless AWS アプリケーションを設定できます。次の例は、EMR-S Spark を AWS Glue マルチカタログ階層で使用方法を示しています。

マルチカタログ階層の詳細については、[「Working with a multi-catalog hierarchy in AWS Glue Data Catalog with Spark on Amazon EMR」](#)を参照してください。

Iceberg および Glue データカタログでの Redshift マネージドストレージ (RMS) AWS の使用

以下は、Iceberg と Glue データカタログを統合するように Spark AWS を設定する方法を示しています。

```
aws emr-serverless start-job-run \
  --application-id application-id \
```

```
--execution-role-arn job-role-arn \
--job-driver '{
  "sparkSubmit": {
    "entryPoint": "s3://amzn-s3-demo-bucket/myscript.py",
    "sparkSubmitParameters": "--conf spark.sql.catalog.nfgac_rms =
org.apache.iceberg.spark.SparkCatalog
    --conf spark.sql.catalog.rms.type=glue
    --conf spark.sql.catalog.rms.glue.id=Glue RMS catalog ID
    --conf spark.sql.defaultCatalog=rms
    --conf
spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
  }
}'
```

統合後のカタログ内のテーブルからのサンプルクエリ:

```
SELECT * FROM my_rms_schema.my_table
```

Iceberg REST API と AWS Glue データカタログでの Redshift マネージドストレージ (RMS) の使用

以下は、Iceberg REST カタログで動作するように Spark を設定する方法を示しています:

```
aws emr-serverless start-job-run \
--application-id application-id \
--execution-role-arn job-role-arn \
--job-driver '{
  "sparkSubmit": {
    "entryPoint": "s3://amzn-s3-demo-bucket/myscript.py",
    "sparkSubmitParameters": "
    --conf spark.sql.catalog.rms=org.apache.iceberg.spark.SparkCatalog
    --conf spark.sql.catalog.rms.type=rest
    --conf spark.sql.catalog.rms.warehouse=Glue RMS catalog ID
    --conf spark.sql.catalog.rms.uri=Glue endpoint URI/iceberg
    --conf spark.sql.catalog.rms.rest.sigv4-enabled=true
    --conf spark.sql.catalog.rms.rest.signing-name=glue
    --conf
spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
  }
}'
```

カタログ内のテーブルからのサンプルクエリ:

```
SELECT * FROM my_rms_schema.my_table
```

外部メタストアを使用する場合の考慮事項

- MariaDB JDBC と互換性のあるデータベースをメタストアとして設定できます。これらのデータベースの例としては、RDS for MariaDB、MySQL、Amazon Aurora などがあります。
- メタストアは自動初期化されません。メタストアが Hive バージョンのスキーマで初期化されていない場合は、[Hive スキーマツール](#)を使用します。
- EMR Serverless は Kerberos 認証をサポートしていません。EMR Serverless の Spark ジョブまたは Hive ジョブでは、Kerberos 認証で thrift メタストアサーバーを使用することはできません。
- マルチカタログ階層を使用するように VPC アクセスを設定しなければなりません。

EMR Serverless から別の AWS アカウントの S3 データにアクセスする

ある AWS アカウントから Amazon EMR Serverless ジョブを実行し、別の AWS アカウントに属する Amazon S3 バケットのデータにアクセスするようにジョブを設定できます。このページでは、EMR Serverless から S3 へのクロスアカウントアクセスを設定する方法について説明します。

EMR Serverless で実行されるジョブは、S3 バケットポリシーまたは引き受けたロールを使用して、別の AWS アカウントから Amazon S3 のデータにアクセスできます。

前提条件

Amazon EMR Serverless のクロスアカウントアクセスを設定するには、2 つの AWS アカウントにサインインしながらタスクを完了します。

- **AccountA** – これは、Amazon EMR Serverless アプリケーションを作成した AWS アカウントです。クロスアカウントアクセスを設定する前に、このアカウントで以下の準備をします。
 - ジョブを実行する Amazon EMR Serverless アプリケーション。
 - アプリケーションでジョブを実行するために必要な許可を持つジョブ実行ロール。詳細については、[Amazon EMR Serverless のジョブランタイムロール](#)を参照してください。
- **AccountB** – これは、Amazon EMR Serverless ジョブがアクセスする S3 バケットを含む AWS アカウントです。

S3 バケットポリシーを使用してクロスアカウント S3 データにアクセスする

account A から account B の S3 バケットにアクセスするには、次のポリシーを account B の S3 バケットにアタッチします。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ExamplePermissions1",
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::my-bucket-name"
      ]
    },
    {
      "Sid": "ExamplePermissions2",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:DeleteObject"
      ],
      "Resource": [
        "arn:aws:s3:::my-bucket-name/*"
      ]
    }
  ]
}
```

S3 バケットポリシーを使用した S3 クロスアカウントアクセスの詳細については、「Amazon Simple Storage Service ユーザーガイド」の「[例 2: バケット所有者がクロスアカウントのバケットのアクセス許可を付与する](#)」を参照してください。

引き受けたロールを使用してクロスアカウント S3 データにアクセスする

Amazon EMR Serverless のクロスアカウントアクセスを設定するもう 1 つの方法は、AWS Security Token Service (AWS STS) の AssumeRole アクションを使用することです。AWS STS は、ユーザーの権限が制限された一時的な認証情報をリクエストできるグローバルウェブサービスです。AssumeRole で作成した一時的なセキュリティ認証情報を使用して、EMR Serverless と Amazon S3 に API コールを行うことができます。

次の手順は、引き受けたロールを使用して EMR Serverless からクロスアカウント S3 データにアクセスする方法を示しています。

1. Amazon S3 バケット (#####) を AccountB に作成します。詳細については、「Amazon Simple Storage Service ユーザーガイド」の「[バケットの作成](#)」を参照してください。DynamoDB へのクロスアカウントアクセスが必要な場合は、AccountB で DynamoDB テーブルを作成します。詳細については、「Amazon DynamoDB デベロッパーガイド」の「[DynamoDB にテーブルを作成する](#)」を参照してください。
2. ##### にアクセスできる Cross-Account-Role-B IAM ロールを AccountB に作成します。
 - a. にサインイン AWS マネジメントコンソールし、<https://console.aws.amazon.com/iam/> で IAM コンソールを開きます。
 - b. [ロール] を選択し、新しいロール Cross-Account-Role-B を作成します。IAM ロールの作成方法の詳細については、「IAM ユーザーガイド」の「[IAM ロールの作成](#)」を参照してください。
 - c. 以下のポリシーステートメントに示すように、##### の S3 バケットにアクセスするための Cross-Account-Role-B の許可を指定する IAM ポリシーを作成します。IAM ポリシーを Cross-Account-Role-B にアタッチします。詳細については、「IAM ユーザーガイド」の「[IAM ポリシーを作成する](#)」を参照してください。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:*"
      ],
      "Resource": [
```

```

        "arn:aws:s3:::cross-account-bucket",
        "arn:aws:s3:::cross-account-bucket/*"
    ],
    "Sid": "AllowS3"
  }
]
}

```

DynamoDB アクセスが必要な場合は、クロスアカウントの DynamoDB テーブルにアクセスするための許可を指定する IAM ポリシーを作成します。IAM ポリシーを Cross-Account-Role-B にアタッチします。詳細については、「IAM ユーザーガイド」の「[Amazon DynamoDB: 特定のテーブルへのアクセスの許可](#)」を参照してください。

DynamoDB テーブル CrossAccountTable へのアクセスを許可するためのポリシーを以下に示します。

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:*"
      ],
      "Resource": [
        "arn:aws:dynamodb:*:123456789012:table/CrossAccountTable"
      ],
      "Sid": "AllowDYNAMODB"
    }
  ]
}

```

3. Cross-Account-Role-B ロールの信頼関係を編集します。

- a. ロールの信頼関係を設定するには、IAM コンソールでステップ 2 で作成したロール Cross-Account-Role-B の [信頼関係] タブを選択します。
- b. [信頼関係の編集] を選択します。
- c. 次のポリシードキュメントを追加します。これにより、AccountA の Job-Execution-Role-A に Cross-Account-Role-B ロールを引き受けることが許可されます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sts:AssumeRole"
      ],
      "Resource": "arn:aws:iam::123456789012:role/Job-Execution-Role-A",
      "Sid": "AllowSTSAssumerole"
    }
  ]
}
```

4. を引き受けるアクセスAssumeRole許可を Job-Execution-Role-Aに付与AccountAします
AWS STS Cross-Account-Role-B。
 - a. AWS アカウント の IAM コンソールでAccountA、 を選択しますJob-Execution-Role-A。
 - b. 次のポリシーステートメントを Job-Execution-Role-A に追加して、Cross-Account-Role-B ロールに対して AssumeRole アクションを許可します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sts:AssumeRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/Cross-Account-Role-B"
      ],
      "Sid": "AllowSTSAssumerole"
    }
  ]
}
```

引き受けたロールの例

1 つの引き受けたロールを使用して、アカウント内のすべての S3 リソースにアクセスします。また、Amazon EMR 6.11 以降では、別々のクロスアカウント S3 バケットにアクセスするときに引き受けるための複数の IAM ロールを設定します。

トピック

- [1 つの引き受けたロールを使用して S3 リソースにアクセスする](#)
- [複数の引き受けたロールを使用して S3 リソースにアクセスする](#)

1 つの引き受けたロールを使用して S3 リソースにアクセスする

Note

1 つの引き受けたロールを使用するようにジョブを設定すると、ジョブ全体のすべての S3 リソースが `entryPoint` スクリプトを含むそのロールを使用します。

1 つの引き受けたロールを使用してアカウント B 内のすべての S3 リソースにアクセスする場合は、次の設定を指定します。

1. EMRFS 設定 `fs.s3.customAWSCredentialsProvider` を `com.amazonaws.emr.AssumeRoleAWSCredentialsProvider` に指定します。
2. Spark の場合は、`spark.emr-serverless.driverEnv.ASSUME_ROLE_CREDENTIALS_ROLE_ARN` と `spark.executorEnv.ASSUME_ROLE_CREDENTIALS_ROLE_ARN` を使用してドライバーとエグゼキュターの環境変数を指定します。
3. Hive の場合は、`hive.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN`、`tez.am.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN`、および `tez.task.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN` を使用して、Hive ドライバー、Tez アプリケーションプライマリー、および Tez タスクコンテナの環境変数を指定します。

次の例は、引き受けたロールを使用してクロスアカウントアクセスで EMR Serverless ジョブの実行を開始する方法をデモンストレーションしています。

Spark

次の例は、引き受けたロールを使用して、S3 へのクロスアカウントアクセスで EMR Serverless Spark ジョブの実行を開始する方法を示しています。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "entrypoint_location",
      "entryPointArguments": [":argument_1", ":argument_2"],
      "sparkSubmitParameters": "--conf spark.executor.cores=4 --conf
spark.executor.memory=20g --conf spark.driver.cores=4 --conf spark.driver.memory=8g
--conf spark.executor.instances=1"
    }
  }' \
  --configuration-overrides '{
    "applicationConfiguration": [{
      "classification": "spark-defaults",
      "properties": {
        "spark.hadoop.fs.s3.customAWSCredentialsProvider":
"com.amazonaws.emr.AssumeRoleAWSCredentialsProvider",
        "spark.emr-serverless.driverEnv.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
"arn:aws:iam::AccountB:role/Cross-Account-Role-B",
        "spark.executorEnv.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
"arn:aws:iam::AccountB:role/Cross-Account-Role-B"
      }
    }]
  }'
```

Hive

次の例は、引き受けたロールを使用して、S3 へのクロスアカウントアクセスで EMR Serverless Hive ジョブの実行を開始する方法を示しています。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "hive": {
      "query": "query_location",
      "parameters": "hive_parameters"
    }
  }'
```

```

    }
  }' \
  --configuration-overrides '{
    "applicationConfiguration": [{
      "classification": "hive-site",
      "properties": {
        "fs.s3.customAWSCredentialsProvider":
"com.amazonaws.emr.serverless.credentialsprovider.AssumeRoleAWSCredentialsProvider",
        "hive.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
"arn:aws:iam::AccountB:role/Cross-Account-Role-B",
        "tez.am.emr-serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
"arn:aws:iam::AccountB:role/Cross-Account-Role-B",
        "tez.task.emr-
serverless.launch.env.ASSUME_ROLE_CREDENTIALS_ROLE_ARN":
"arn:aws:iam::AccountB:role/Cross-Account-Role-B"
      }
    }]
  }'

```

複数の引き受けたロールを使用して S3 リソースにアクセスする

EMR Serverless リリース 6.11.0 以降では、別々のクロスアカウントバケットにアクセスするときに引き受ける複数の IAM ロールを設定します。アカウント B でさまざまな引き受けたロールを使用して別々の S3 リソースにアクセスする場合は、ジョブの実行を開始するときに次の設定を使用します。

1. EMRFS 設定 `fs.s3.customAWSCredentialsProvider` を `com.amazonaws.emr.serverless.credentialsprovider.BucketLevelAssumeRoleCredentialsProvider` に指定します。
2. EMRFS 設定 `fs.s3.bucketLevelAssumeRoleMapping` を指定して、S3 バケット名から引き受けるアカウント B の IAM ロールへのマッピングを定義します。値は、`bucket1->role1;bucket2->role2` の形式である必要があります。

例えば、`arn:aws:iam::AccountB:role/Cross-Account-Role-B-1` を使用してバケット `bucket1` にアクセスし、`arn:aws:iam::AccountB:role/Cross-Account-Role-B-2` を使用してバケット `bucket2` にアクセスします。次の例は、複数の引き受けたロールを使用してクロスアカウントアクセスで EMR Serverless ジョブの実行を開始する方法をデモンストレーションしています。

Spark

次の例は、複数の引き受けたロールを使用して EMR Serverless Spark ジョブの実行を作成する方法を示しています。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "entrypoint_location",
      "entryPointArguments": [":argument_1", ":argument_2"],
      "sparkSubmitParameters": "--conf spark.executor.cores=4 --conf
spark.executor.memory=20g --conf spark.driver.cores=4 --conf spark.driver.memory=8g
--conf spark.executor.instances=1"
    }
  }' \
  --configuration-overrides '{
    "applicationConfiguration": [{
      "classification": "spark-defaults",
      "properties": {
        "spark.hadoop.fs.s3.customAWSCredentialsProvider":
"com.amazonaws.emr.serverless.credentialsprovider.BucketLevelAssumeRoleCredentialsProvider"
        "spark.hadoop.fs.s3.bucketLevelAssumeRoleMapping":
"bucket1->arn:aws:iam::AccountB:role/Cross-Account-Role-B-1;bucket2->arn:aws:iam::AccountB:role/Cross-Account-Role-B-2"
      }
    }]
  }'
```

Hive

次の例は、複数の引き受けたロールを使用して EMR Serverless Hive ジョブの実行を作成する方法をデモンストレーションしています。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "hive": {
      "query": "query_location",
      "parameters": "hive_parameters"
    }
  }'
```

```
}' \
--configuration-overrides '{
  "applicationConfiguration": [{
    "classification": "hive-site",
    "properties": {
      "fs.s3.customAWSCredentialsProvider":
"com.amazonaws.emr.serverless.credentialsprovider.AssumeRoleAWSCredentialsProvider",
      "fs.s3.bucketLevelAssumeRoleMapping": "bucket1-
>arn:aws:iam::AccountB:role/Cross-Account-Role-B-1;bucket2-
>arn:aws:iam::AccountB:role/Cross-Account-Role-B-2"
    }
  }]
}'
```

EMR Serverless のエラーのトラブルシューティング

Amazon EMR Serverless の使用時に発生する一般的な問題の診断や修正には、次の情報が役立ちます。

トピック

- [エラー: アカウントが同時に使用できる最大 vCPU のサービス制限に達したため、ジョブは失敗しました。](#)
- [エラー: アプリケーションが maximumCapacity 設定を超えたため、ジョブが失敗しました。](#)
- [エラー: アプリケーションが maximumCapacity を超え、ワーカーを割り当てることができなかつたため、ジョブが失敗しました。](#)
- [エラー: S3 アクセスが拒否されました。必要な S3 リソースのジョブランタイムロールの S3 アクセス許可を確認してください。](#)
- [エラー: ModuleNotFoundError: <module> という名前のモジュールがありません。EMR Serverless で Python ライブラリを使用する方法については、ユーザーガイドを参照してください。](#)
- [エラー: 実行ロール <role name> が存在しないか、必要な信頼関係で設定されていないため、引き受けることができませんでした。](#)

エラー: アカウントが同時に使用できる最大 vCPU のサービス制限に達したため、ジョブは失敗しました。

このエラーは、アカウントが設定された最大容量を超えたため、EMR Serverless がジョブを送信できなかったことを示します。アカウントの最大容量を増やします。[EMR Serverless サービスクォータでサービス制限を確認](#)します。

エラー: アプリケーションが maximumCapacity 設定を超えたため、ジョブが失敗しました。

このエラーは、アプリケーションが設定された最大容量を超えたため、EMR Serverless がジョブを送信できなかったことを示します。アプリケーションの最大容量を引き上げます。

エラー: アプリケーションが maximumCapacity を超え、ワーカーを割り当てるができなかったため、ジョブが失敗しました。

このエラーは、ジョブを完了できなかったことを示します。アプリケーションが maximumCapacity 設定を超えているため、ワーカーを割り当てることができませんでした。

エラー: S3 アクセスが拒否されました。必要な S3 リソースのジョブランタイムロールの S3 アクセス許可を確認してください。

このエラーは、ジョブが S3 リソースにアクセスできないことを示します。ジョブランタイムロールに、ジョブが使用する必要がある S3 リソースにアクセスする許可があることを確認します。ランタイムロールの詳細については、「[Amazon EMR Serverless のジョブランタイムロール](#)」を参照してください。

エラー: ModuleNotFoundError: <module> という名前のモジュールがありません。EMR Serverless で Python ライブラリを使用する方法については、[ユーザーガイドを参照](#)してください。

このエラーは、Spark ジョブで Python モジュールが使用できないことを示します。依存する Python ライブラリがジョブで使えることを確認します。Python ライブラリをパッケージ化する方法については、「[EMR Serverless での Python ライブラリの使用](#)」を参照してください。

エラー: 実行ロール <role name> が存在しないか、必要な信頼関係で設定されていないため、引き受けることができませんでした。

このエラーは、ジョブに指定したジョブランタイムロールが存在しないか、ロールに EMR Serverless アクセス許可の信頼関係がないことを示します。IAM ロールが存在していることを検証し、ロールの信頼ポリシーが正しく設定されていることを確認するには、「[Amazon EMR Serverless のジョブランタイムロール](#)」の手順を参照してください。

ジョブレベルのコスト配分の有効化

ジョブレベルのコスト配分により、アプリケーションレベルですべてのコストを集計するのではなく、個々のジョブ実行レベルで EMR Serverless の詳細な請求属性が可能になります。有効にすると、AWS Cost Explorer とコストと使用状況レポートのコストを、ジョブ実行に関連付けられた特定のジョブ実行 IDs とタグでフィルタリングして追跡できるため、送信されたジョブ実行の料金をより詳細に把握できます。

デフォルトの 動作

ジョブレベルのコスト配分はデフォルトでは有効になっていません。

機能を有効または無効にする方法

アプリケーションの作成時にジョブレベルのコスト配分を設定したり、既存のアプリケーション用に更新したりできます。

新しいアプリケーションを作成するときに `jobLevelCostAllocation` パラメータを指定します。

```
# Enable job-level cost allocation:
aws emr-serverless create-application \
  --name "my-application" \
  --release-label "emr-7.12.0" \
  --type "SPARK" \
  --job-level-cost-allocation-configuration '{
    "enabled": true
  }'

# Disable job-level cost allocation:
aws emr-serverless create-application \
  --name "my-application" \
  --release-label "emr-7.12.0" \
```

```
--type "SPARK" \  
--job-level-cost-allocation-configuration '{  
    "enabled": false  
'
```

既存のアプリケーションの `jobLevelCostAllocationConfiguration` パラメータを更新します。

```
# Enable job-level cost allocation:  
aws emr-serverless update-application \  
    --application-id <application-id> \  
    --job-level-cost-allocation-configuration '{  
        "enabled": true  
    }'  
  
# Disable job-level cost allocation:  
aws emr-serverless update-application \  
    --application-id <application-id> \  
    --job-level-cost-allocation-configuration '{  
        "enabled": false  
    }'
```

考慮事項と制約事項

- ジョブレベルのコスト配分を有効にしても、この機能が有効になる前に完了したジョブ実行のコストは遡及的には帰属しません。この機能を有効にした後に開始されたジョブの実行には、きめ細かなコスト属性があります。
- ジョブレベルのコスト配分パラメータは、アプリケーションが `CREATED` または `STOPPED` 状態にある場合にのみ更新できます。
- ジョブレベルのコスト配分が有効になっている場合、コストはアプリケーションではなく個々のジョブ実行に帰属します。アプリケーションレベルで総コストを表示するには、そのアプリケーション内のすべてのジョブ実行に一貫したタグ (`application-name` や `application-id` など) を適用し、Cost Explorer または Cost and Usage Reports でそれらのタグでフィルタリングする必要があります。

EMR Studio を使用して EMR Serverless でインタラクティブワークロードを実行する

EMR Serverless インタラクティブアプリケーションを使用することにより、EMR Studio でホストされているノートブックを使用して、EMR Serverless で Spark のインタラクティブワークロードを実行します。

概要:

インタラクティブアプリケーションは、インタラクティブ機能が有効になっている EMR Serverless アプリケーションです。Amazon EMR Serverless インタラクティブアプリケーションを使用すると、Amazon EMR Studio で管理されている Jupyter ノートブックを使用してインタラクティブワークロードを実行できます。これにより、データエンジニア、データサイエンティスト、データアナリストは EMR Studio を使用して、Amazon S3 や Amazon DynamoDB などのデータストア内のデータセットでインタラクティブな分析を実行できます。

EMR Serverless のインタラクティブアプリケーションのユースケースには、次のものがあります。

- データエンジニアは、EMR Studio の IDE エクスペリエンスを使用して ETL スクリプトを作成します。スクリプトはオンプレミスからデータを取り込み、分析のためにデータを変換し、Amazon S3 にデータを保存します。
- データサイエンティストはノートブックを使用してデータセットを調べ、データセットの異常を検出するように機械学習 (ML) モデルをトレーニングします。
- データアナリストはデータセットを調査し、ビジネスダッシュボードなどのアプリケーションを更新する日次レポートを生成するスクリプトを作成します。

前提条件

EMR Serverless でインタラクティブワークロードを使用するには、次の要件を満たします。

- EMR Serverless インタラクティブアプリケーションは、Amazon EMR 6.14.0 以降でサポートされています。
- インタラクティブアプリケーションにアクセスし、送信するワークロードを実行し、EMR Studio からインタラクティブノートブックを実行するには、特定のアクセス許可とロールが必要です。詳細については、[インタラクティブワークロードに必要なアクセス許可](#) を参照してください。

インタラクティブワークロードに必要なアクセス許可

[EMR Serverless へのアクセスに必要な基本的なアクセス許可](#)に加えて、IAM ID またはロールに追加のアクセス許可を設定します。

インタラクティブアプリケーションにアクセスするには

EMR Studio のユーザーアクセス許可と Workspace アクセス許可を設定します。詳細については、「Amazon EMR 管理ガイド」の「[Configure EMR Studio user permissions](#)」を参照してください。

EMR Serverless で送信するワークロードを実行するには

ジョブのランタイムロールを設定する 詳細については、[ジョブランタイムロールを作成する](#) を参照してください。

EMR Studio からインタラクティブノートブックを実行するには

Studio ユーザーの IAM ポリシーに次の権限を追加します。

- **emr-serverless:AccessInteractiveEndpoints** - Resource として指定したインタラクティブアプリケーションにアクセスして接続するためのアクセス許可を付与します。このアクセス許可は、EMR Studio Workspace から EMR Serverless アプリケーションにアタッチするために必要です。
- **iam:PassRole** - アプリケーションにアタッチするときに使用する予定の IAM 実行ロールにアクセスするためのアクセス許可を付与します。EMR Studio Workspace から EMR Serverless アプリケーションにアタッチするには、適切な PassRole アクセス許可が必要です。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EMRServerlessInteractiveAccess",
      "Effect": "Allow",
      "Action": [
        "emr-serverless:AccessInteractiveEndpoints"
      ],
      "Resource": [
        "arn:aws:emr-serverless:*:123456789012:/applications/*"
      ]
    }
  ],
}
```

```
{
  "Sid": "EMRServerlessRuntimeRoleAccess",
  "Effect": "Allow",
  "Action": [
    "iam:PassRole"
  ],
  "Resource": [
    "arn:aws:iam::123456789012:role/EMRServerlessInteractiveRole"
  ],
  "Condition": {
    "StringLike": {
      "iam:PassedToService": "emr-serverless.amazonaws.com"
    }
  }
}
```

インタラクティブアプリケーションの設定

以下の概略的な手順を使用して、AWS マネジメントコンソールの Amazon EMR Studio のインタラクティブ機能を備えた EMR Serverless アプリケーションを作成します。

1. [Amazon EMR Serverless の使用を開始する](#) の手順に従ってアプリケーションを作成します。
2. 次に、EMR Studio からワークスペースを起動し、コンピューティングオプションとして EMR Serverless アプリケーションにアタッチします。詳細については、[EMR Serverless Getting Started](#) ドキュメントのステップ 2 のインタラクティブワークロードタブを参照してください。

Studio Workspace にアプリケーションをアタッチすると、まだ実行されていない場合、アプリケーションは自動的にトリガーを開始します。また、アプリケーションを事前に起動し、ワークスペースにアタッチする前に準備しておくこともできます。

インタラクティブアプリケーションの考慮事項

- EMR Serverless インタラクティブアプリケーションは、Amazon EMR 6.14.0 以降でサポートされています。
- EMR Studio は、EMR Serverless インタラクティブアプリケーションと統合されている唯一のクライアントです。EMR Serverless インタラクティブアプリケーションでは、Workspace コラボ

レーション、SQL Explorer、ノートブックのプログラムによる実行などの EMR Studio 機能はサポートされていません。

- インタラクティブアプリケーションは Spark エンジンでのみサポートされています。
- インタラクティブアプリケーションは、Python 3、PySpark、Spark Scala カーネルをサポートしています。
- 1 つのインタラクティブアプリケーションで最大 25 個のノートブックを同時に実行できます。
- インタラクティブアプリケーションでセルフホスト Jupyter ノートブックをサポートするエンドポイントや API インターフェイスはありません。
- 起動エクスペリエンスを最適化するには、ドライバーとエグゼキューターの事前初期化された容量を設定し、アプリケーションを事前に起動することを提案します。アプリケーションを事前に起動する場合は、Workspace にアタッチする際に準備が整っていることを確認します。

```
aws emr-serverless start-application \  
--application-id your-application-id
```

- デフォルトでは、autoStopConfig はアプリケーションに対して有効になっています。これにより、アイドル時間が 30 分経過するとアプリケーションがシャットダウンされます。この設定は、create-application または update-application リクエストの一部として変更できます。
- インタラクティブアプリケーションを使用する場合は、ノートブックを実行するカーネル、ドライバー、エグゼキューターの事前に初期化された容量を設定することを提案します。Spark インタラクティブセッションごとに 1 つのカーネルと 1 つのドライバーが必要なため、EMR Serverless は、事前に初期化されたドライバーごとに、事前に初期化されたカーネルワーカーを維持します。デフォルトでは、EMR Serverless は、ドライバーに事前初期化された容量を指定しない場合でも、アプリケーション全体で 1 つのカーネルワーカーの事前初期化された容量を維持します。各カーネルワーカーは 4 つの vCPU と 16 GB のメモリを使用します。現在の料金については、[Amazon EMR の料金](#)ページを参照してください。
- インタラクティブワークロードを実行する AWS アカウント には、に十分な vCPU サービスクォータが必要です。Lake Formation 対応のワークロードを実行しない場合は、少なくとも 24 個の vCPU を提案します。その場合、少なくとも 28 個の vCPU を提案します。
- EMR Serverless は、60 分を超える時間アイドル状態になっている場合に、ノートブックからカーネルを自動的に終了します。EMR Serverless は、ノートブックセッション中に完了した最後のアクティビティからカーネルアイドル時間を計算します。現在、カーネルアイドルタイムアウト設定を変更することはできません。

- インタラクティブワークロードで Lake Formation を有効にするには、[EMR Serverless アプリケーションを作成する](#)ときに、runtime-configuration オブジェクトの spark-defaults 分類で設定 spark.emr-serverless.lakeformation.enabled を true に設定します。詳細については、「[Enabling Lake Formation in Amazon EMR](#)」を参照してください。

Apache Livy エンドポイントを介して EMR Serverless でインタラクティブワークロードを実行する

Amazon EMR リリース 6.14.0 以降では、EMR Serverless アプリケーションの作成時に Apache Livy エンドポイントを作成および有効化し、セルフホストノートブックまたはカスタムクライアントを使用してインタラクティブワークロードを実行します。Apache Livy エンドポイントには以下の利点があります。

- Jupyter ノートブックを介して Apache Livy エンドポイントに安全に接続し、Apache Livy の REST インターフェイスを使用して Apache Spark ワークロードを管理できます。
- Apache Livy REST API オペレーションは、Apache Spark ワークロードのデータを使用するインタラクティブなウェブアプリケーションに使用します。

前提条件

EMR Serverless で Apache Livy エンドポイントを使用するには、次の要件を満たします。

- [Amazon EMR Serverless の使用を開始する](#)の手順を完了します。
- Apache Livy エンドポイントを介してインタラクティブワークロードを実行するには、特定のアクセス許可とロールが必要です。詳細については、「[Required permissions for interactive workloads](#)」を参照してください。

必要なアクセス許可

EMR Serverless にアクセスするために必要なアクセス許可に加えて、Apache Livy エンドポイントにアクセスしてアプリケーションを実行するために、IAM ロールに次のアクセス許可を追加します。

- `emr-serverless:AccessLivyEndpoints` – Resource として指定した Livy 対応アプリケーションにアクセスして接続するためのアクセス許可を付与します。Apache Livy エンドポイントから利用可能な REST API オペレーションを実行するには、このアクセス許可が必要です。

- `iam:PassRole` – Apache Livy セッション作成時に IAM 実行ロールにアクセスするためのアクセス許可を付与します。EMR Serverless は、このロールを使用してワークロードを実行します。
- `emr-serverless:GetDashboardForJobRun` – Spark Live UI とドライバーのログリンクを生成するためのアクセス許可を付与し、Apache Livy セッションの結果の一部としてログへのアクセスを許可します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EMRServerlessInteractiveAccess",
      "Effect": "Allow",
      "Action": [
        "emr-serverless:AccessLivyEndpoints"
      ],
      "Resource": [
        "arn:aws:emr-serverless*:123456789012:/applications/*"
      ]
    },
    {
      "Sid": "EMRServerlessRuntimeRoleAccess",
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/EMRServerlessExecutionRole"
      ],
      "Condition": {
        "StringLike": {
          "iam:PassedToService": "emr-serverless.amazonaws.com"
        }
      }
    },
    {
      "Sid": "EMRServerlessDashboardAccess",
      "Effect": "Allow",
      "Action": [
        "emr-serverless:GetDashboardForJobRun"
      ]
    }
  ]
}
```

```
    ],  
    "Resource": [  
        "arn:aws:emr-serverless:*:123456789012:/applications/*"  
    ]  
  }  
]  
}
```

開始方法

Apache Livy 対応アプリケーションを作成して実行するには、次の手順に従います。

1. Apache Livy 対応アプリケーションを作成するには、次のコマンドを実行します。

```
aws emr-serverless create-application \  
--name my-application-name \  
--type 'application-type' \  
--release-label <Amazon EMR-release-version>  
--interactive-configuration '{"livyEndpointEnabled": true}'
```

2. EMR Serverless がアプリケーションを作成したら、アプリケーションを起動して Apache Livy エンドポイントを使用可能にします。

```
aws emr-serverless start-application \  
--application-id application-id
```

アプリケーションの状態を確認するには、次のコマンドを使用します。ステータスが **STARTED** になったら、Apache Livy エンドポイントにアクセスします。

```
aws emr-serverless get-application \  
--region <AWS_REGION> --application-id >application_id<
```

3. 以下の URL を使用してエンドポイントにアクセスします。

```
https://_<application-id>_.livy.emr-serverless-  
services._<AWS_REGION>_.amazonaws.com
```

エンドポイントの準備ができたなら、ユースケースに基づいてワークロードを送信します。[SIGv4 プロトコル](#)を使用してエンドポイントへのすべてのリクエストに署名し、認証ヘッダーを渡す必要があります。次の方法を使用してワークロードを実行できます。

- HTTP クライアント – カスタム HTTP クライアントを使用して Apache Livy エンドポイント API オペレーションを送信します。
- Sparkmagic カーネル – Sparkmagic カーネルをローカルで実行し、Jupyter ノートブックを使用してインタラクティブクエリを送信します。

HTTP クライアント

Apache Livy セッションを作成するには、リクエスト本文の `conf` パラメータで `emr-serverless.session.executionRoleArn` を送信します。次の例は、サンプルの POST /sessions リクエストです。

```
{
  "kind": "pyspark",
  "heartbeatTimeoutInSeconds": 60,
  "conf": {
    "emr-serverless.session.executionRoleArn": "<executionRoleArn>"
  }
}
```

次の表は、使用可能なすべての Apache Livy API オペレーションを示しています。

API オペレーション:	説明
GET /sessions	アクティブなすべてのインタラクティブセッションのリストを返します。
POST /sessions	spark または pyspark を使用して新しいインタラクティブセッションを作成します。
GET /sessions/< <i>sessionId</i> >	セッション情報を返します。
GET /sessions/< <i>sessionId</i> >/state	セッションの状態を返します。
DELETE /sessions/< <i>sessionId</i> >	セッションを停止して削除します。

API オペレーション:	説明
GET /sessions/<sessionId>/statements	セッション内のすべてのステートメントを返します。
POST /sessions/<sessionId>/statements	セッションでステートメントを実行します。
GET /sessions/<sessionId>/statements/<statementId>	セッションで指定されたステートメントの詳細を返します。
POST /sessions/<sessionId>/statements/<statementId>/cancel	このセッションで指定されたステートメントをキャンセルします。

Apache Livy エンドポイントへのリクエストの送信

HTTP クライアントから Apache Livy エンドポイントに直接リクエストを送信することもできます。これにより、ノートブックの外部でユースケースのコードをリモート実行できます。

エンドポイントへのリクエストの送信を開始するには、まず次のライブラリがインストールされていることを確認してください。

```
pip3 install boto3 awscli requests
```

以下に示すのは、HTTP リクエストをエンドポイントに直接送信するための Python スクリプトのサンプルです。

```
from boto3 import client
import requests
from boto3.awsrequest import AWSRequest
from boto3.credentials import Credentials
import boto3.session
import json, pprint, textwrap

endpoint = 'https://<application_id>.livy.emr-serverless-
services.<AWS_REGION>.amazonaws.com'
headers = {'Content-Type': 'application/json'}

session = boto3.session.Session()
signer = crt.auth.CrtS3SigV4Auth(session.get_credentials(), 'emr-serverless',
    '<AWS_REGION>')
```


Create session request

```
data = {'kind': 'pyspark', 'heartbeatTimeoutInSeconds': 60, 'conf': { 'emr-  
serverless.session.executionRoleArn': 'arn:aws:iam::123456789012:role/role1'}}  
  
request = AWSRequest(method='POST', url=endpoint + "/sessions", data=json.dumps(data),  
    headers=headers)  
  
request.context["payload_signing_enabled"] = False  
  
signer.add_auth(request)  
  
prepped = request.prepare()  
  
r = requests.post(prepped.url, headers=prepped.headers, data=json.dumps(data))  
  
pprint.pprint(r.json())
```

List Sessions Request

```
request = AWSRequest(method='GET', url=endpoint + "/sessions", headers=headers)  
  
request.context["payload_signing_enabled"] = False  
  
signer.add_auth(request)  
  
prepped = request.prepare()  
  
r2 = requests.get(prepped.url, headers=prepped.headers)  
pprint.pprint(r2.json())
```

Get session state

```
session_url = endpoint + r.headers['location']  
  
request = AWSRequest(method='GET', url=session_url, headers=headers)  
  
request.context["payload_signing_enabled"] = False  
  
signer.add_auth(request)
```

```
prepped = request.prepare()

r3 = requests.get(prepped.url, headers=prepped.headers)

pprint.pprint(r3.json())

### Submit Statement

data = {
    'code': "1 + 1"
}

statements_url = endpoint + r.headers['location'] + "/statements"

request = AWSRequest(method='POST', url=statements_url, data=json.dumps(data),
    headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()

r4 = requests.post(prepped.url, headers=prepped.headers, data=json.dumps(data))

pprint.pprint(r4.json())

### Check statements results

specific_statement_url = endpoint + r4.headers['location']

request = AWSRequest(method='GET', url=specific_statement_url, headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()

r5 = requests.get(prepped.url, headers=prepped.headers)

pprint.pprint(r5.json())
```

```
### Delete session

session_url = endpoint + r.headers['location']

request = AWSRequest(method='DELETE', url=session_url, headers=headers)

request.context["payload_signing_enabled"] = False

signer.add_auth(request)

prepped = request.prepare()

r6 = requests.delete(prepped.url, headers=prepped.headers)

pprint.pprint(r6.json())
```

Sparkmagic カーネル

sparkmagic をインストールする前に、sparkmagic をインストールするインスタンスで AWS 認証情報が設定されていることを確認してください。

1. [インストール手順](#)に従って sparkmagic をインストールします。最初の 4 つのステップのみを実行することに注意してください。
2. sparkmagic カーネルはカスタム認証をサポートしているため、認証ツールを sparkmagic カーネルと統合して、すべてのリクエストが SIGv4 署名されるようにできます。
3. EMR Serverless カスタム認証ツールをインストールします。

```
pip install emr-serverless-customauth
```

4. 次に、カスタム認証ツールへのパスと、sparkmagic の設定 json ファイル内の Apache Livy エンドポイント URL を指定します。以下のコマンドを使用して設定ファイルを開きます。

```
vim ~/.sparkmagic/config.json
```

次に、サンプル config.json ファイルを示します。

```
{
  "kernel_python_credentials" : {
```

```

    "username": "",
    "password": "",
    "url": "https://<application-id>.livy.emr-serverless-
services.<AWS_REGION>.amazonaws.com",
    "auth": "Custom_Auth"
  },

  "kernel_scala_credentials" : {
    "username": "",
    "password": "",
    "url": "https://<application-id>.livy.emr-serverless-
services.<AWS_REGION>.amazonaws.com",
    "auth": "Custom_Auth"
  },
  "authenticators": {
    "None": "sparkmagic.auth.customauth.Authenticator",
    "Basic_Access": "sparkmagic.auth.basic.Basic",
    "Custom_Auth":
"emr_serverless_customauth.customauthenticator.EMRServerlessCustomSigV4Signer"
  },
  "livy_session_startup_timeout_seconds": 600,
  "ignore_ssl_errors": false
}

```

5. Jupyter ラボを起動します。最後のステップで設定したカスタム認証を使用する必要があります。
6. その後、次のノートブックコマンドとコードを実行して開始できます。

```
%info //Returns the information about the current sessions.
```

```

%configure -f //Configure information specific to a session. We supply
executionRoleArn in this example. Change it for your use case.
{
  "driverMemory": "4g",
  "conf": {
    "emr-serverless.session.executionRoleArn":
"arn:aws:iam::123456789012:role/JobExecutionRole"
  }
}

```

```
<your code> //Run your code to start the session
```

内部的には、各命令は、設定された Apache Livy エンドポイント URL を介して各 Apache Livy API オペレーションを呼び出します。その後、ユースケースに従って指示を記述できます。

考慮事項

Apache Livy エンドポイントを介してインタラクティブワークロードを実行する場合は、次の考慮事項を検討してください。

- EMR Serverless は、発信者プリンシパルを使用してセッションレベルの分離を維持します。セッションを作成する発信者プリンシパルは、セッションにアクセスできる唯一のプリンシパルです。より詳細な分離のために、認証情報を引き受けるときにソース ID を設定します。この場合、EMR Serverless は発信者プリンシパルとソース ID に基づいてセッションレベルの分離を適用します。ソース ID の詳細については、「[引き受けたロールで実行されるアクションのモニタリングと制御](#)」を参照してください。
- Apache Livy エンドポイントは、EMR Serverless リリース 6.14.0 以降でサポートされています。
- Apache Livy エンドポイントは、Apache Spark エンジンでのみサポートされています。
- Apache Livy エンドポイントは Scala Spark と PySpark をサポートしています。
- デフォルトでは、autoStopConfig はアプリケーションで有効になっています。これは、アイドル状態が 15 分間継続するとアプリケーションがシャットダウンすることを表します。この設定は、create-application または update-application リクエストの一部として変更できます。
- 単一の Apache Livy エンドポイント対応アプリケーションで最大 25 件の同時セッションを実行できます。
- 最適な起動エクスペリエンスを実現するには、ドライバーとエグゼキューターの事前初期化された容量を設定することを提案します。
- Apache Livy エンドポイントに接続する前に、アプリケーションを手動で起動する必要があります。
- Apache Livy エンドポイントでインタラクティブワークロードを実行する AWS アカウント には、十分な vCPU サービスクォータが必要です。少なくとも 24 個の vCPU を提案します。
- Apache Livy セッションのデフォルトのタイムアウトは 1 時間です。ステートメントを 1 時間実行しない場合、Apache Livy はセッションを削除し、ドライバーとエグゼキューターを解放します。リリース emr-7.8.0 から、この値は ttl パラメータを Livy /sessions POST リクエストの一部として指定することで設定できます。例えば、2h(時間)、120m(分)、7200s(秒)、7200000ms(ミリ秒) などです。

Note

この設定を emr-7.8.0 より前に変更できません。以下の例は、サンプルの POST /sessions リクエスト本文です。

```
{
  "kind": "pyspark",
  "heartbeatTimeoutInSeconds": 60,
  "conf": {
    "emr-serverless.session.executionRoleArn": "executionRoleArn"
  },
  "ttl": "2h"
}
```

- LakeFormation によるきめ細かなアクセスコントロールが有効になっているアプリケーション用の Amazon EMR リリース emr-7.8.0 以降、セッションごとに設定を無効にできます。EMR Serverless アプリケーションのきめ細かなアクセスコントロールを有効にする方法の詳細については、「[Methods for fine-grained access control](#)」を参照してください。

Note

Lake Formation は、アプリケーションに対して有効になっていないセッションに対して有効にできません。以下の例は、サンプルの POST /sessions リクエスト本文です。

```
{
  "kind": "pyspark",
  "heartbeatTimeoutInSeconds": 60,
  "conf": {
    "emr-serverless.session.executionRoleArn": "executionRoleArn"
  },
  "spark.emr-serverless.lakeformation.enabled" : "false"
}
```

- アクティブなセッションのみが Apache Livy エンドポイントとやり取りできます。セッションが終了、キャンセル、または停止すると、Apache Livy エンドポイントからセッションにアクセスできなくなります。

ログ記録とモニタリング

モニタリングは、EMR Serverless アプリケーションとジョブの信頼性、可用性、パフォーマンスを維持する際の重要な部分です。マルチポイント障害が発生した場合は、障害をより簡単にデバッグできるように、EMR Serverless ソリューションのすべての部分からモニタリングデータを収集する必要があります。

トピック

- [ログの保存](#)
- [ログのローテーション](#)
- [ログの暗号化](#)
- [Amazon EMR Serverless の Apache Log4j2 プロパティを設定する](#)
- [EMR Serverless のモニタリング](#)
- [を使用した EMR Serverless の自動化 Amazon EventBridge](#)

ログの保存

EMR Serverless でのジョブの進行状況をモニタリングし、ジョブの失敗をトラブルシューティングするには、EMR Serverless がアプリケーションログを保存して処理する方法を選択します。ジョブ実行を送信する際に、ログ記録オプションとしてマネージドストレージ、Amazon S3、Amazon CloudWatch を指定します。

CloudWatch では、使用するログタイプとログの場所を指定するか、デフォルトのタイプと場所を受け入れます。CloudWatch のログの詳細については、「[the section called “Amazon CloudWatch”](#)」を参照してください。マネージドストレージと S3 のログ記録について、次の表に[マネージドストレージ](#)、[Amazon S3 バケット](#)、または両方を選択した場合に想定されるログの場所と UI の可用性を一覧表示します。

オプション	イベントログ	コンテナログ	アプリケーション UI
マネージドストレージ	マネージドストレージに保存	マネージドストレージに保存	サポート

オプション	イベントログ	コンテナログ	アプリケーション UI
マネージドストレージと S3 バケットの両方	両方の場所に保存	S3 バケットに保存	サポート
Amazon S3 バケット	S3 バケットに保存	S3 バケットに保存	サポート外 ¹

¹ マネージドストレージオプションを選択したままにすることを提案します。それ以外の場合は、組み込みのアプリケーション UI を使用できません。

マネージドストレージを使用した EMR Serverless のログ記録

デフォルトでは、EMR Serverless はアプリケーションログを Amazon EMR マネージドストレージに最大 30 日間安全に保存します。

Note

デフォルトのオプションをオフにすると、Amazon EMR はユーザーに代わってジョブのトラブルシューティングを行うことができません。例: EMR Serverless コンソールから Spark-UI にアクセスできません。

EMR Studio からこのオプションをオフにするには、ジョブの送信ページの「追加設定」セクションの「30 日間のログ AWS 保持を許可する」チェックボックスをオフにします。

からこのオプションをオフにするには AWS CLI、ジョブ実行を送信するときに `managedPersistenceMonitoringConfiguration` 設定を使用します。

```
{
  "monitoringConfiguration": {
    "managedPersistenceMonitoringConfiguration": {
      "enabled": false
    }
  }
}
```

EMR Serverless アプリケーションが Amazon S3 の VPC エンドポイントを持つプライベートサブネットにあり、アクセスを制御するエンドポイントポリシーをアタッチする場合は、EMR

Serverless がアプリケーションログを保存して処理するための次のアクセス許可を追加します。Resource を、[Amazon S3 にアクセスするプライベートサブネットのサンプルポリシー](#)の使用可能なリージョンテーブルの AppInfo バケットに置き換えます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EMRServerlessManagedLogging",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:PutObjectAcl"
      ],
      "Resource": [
        "arn:aws:s3:::prod.us-east-1.appinfo.src",
        "arn:aws:s3:::prod.us-east-1.appinfo.src/*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:PrincipalServiceName": "emr-serverless.amazonaws.com",
          "aws:SourceVpc": "vpc-12345678"
        }
      }
    }
  ]
}
```

さらに、aws:SourceVpc 条件キーを使用して、リクエストが VPC エンドポイントがアタッチされている VPC を通過することを確認します。

Amazon S3 バケットを使用した EMR Serverless のログ記録

ジョブが Amazon S3 にログデータを送信できるようにするには、ジョブのランタイムロールのアクセス許可ポリシーに次のアクセス許可を設定します。*amzn-s3-demo-logging-bucket* をログ記録バケットの名前に置き換えます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::amzn-s3-demo-bucket/*"
      ],
      "Sid": "AllowS3Putobject"
    }
  ]
}
```

からのログを保存するように Amazon S3 バケットを設定するには AWS CLI、ジョブ実行を開始するときに `s3MonitoringConfiguration` 設定を使用します。これを行うには、設定で以下の `--configuration-overrides` を指定します。

```
{
  "monitoringConfiguration": {
    "s3MonitoringConfiguration": {
      "logUri": "s3://amzn-s3-demo-logging-bucket/logs/"
    }
  }
}
```

再試行が有効になっていないバッチジョブの場合、EMR Serverless はログを次のパスに送信します。

```
'/applications/<applicationId>/jobs/<jobId>'
```

Spark ドライバーログは、EMR Serverless によって以下のパスに保存されます

```
'/applications/<applicationId>/jobs/<jobId>/SPARK_DRIVER/'
```

Spark エグゼキューターログは、EMR Serverless によって以下のパスに保存されます

```
'/applications/<applicationId>/jobs/<jobId>/SPARK_EXECUTOR/<EXECUTOR-ID>'
```

<EXECUTOR-ID> は整数です。

EMR Serverless リリース 7.1.0 以降では、ストリーミングジョブとバッチジョブの再試行がサポートされています。再試行を有効にしてジョブを実行すると、EMR Serverless はログパスプレフィックスに試行番号を自動的に追加するため、ログをより適切に区別して追跡できます。

```
'/applications/<applicationId>/jobs/<jobId>/attempts/<attemptNumber>/'
```

Amazon CloudWatch を使用した EMR Serverless のログ記録

EMR Serverless アプリケーションにジョブを送信する場合は、アプリケーションログを保存するためのオプションとして Amazon CloudWatch を選択します。これにより、CloudWatch Logs Insights や Live Tail などの CloudWatch ログ分析機能を使用できます。CloudWatch から OpenSearch などの他のシステムにログをストリーミングして、詳細に分析することもできます。

EMR Serverless は、ドライバーログのリアルタイムログ記録を可能にします。CloudWatch ライブテール機能、または CloudWatch CLI テールコマンドを使用して、ログにリアルタイムでアクセスできます。

デフォルトでは、EMR Serverless の CloudWatch ログ記録は無効になっています。有効にするには、[AWS CLI](#) の設定を使用してください。

Note

Amazon CloudWatch はログをリアルタイムで発行するため、ワーカーのより多くのリソースが消費されます。ワーカーの容量に対して小さな値を選択すると、ジョブの実行時間への影響が増加する可能性があります。CloudWatch のログ記録を有効にする場合は、より大きなワーカー容量の値を選択することを提案します。また、PutLogEvents の 1 秒あたりのトランザクション (TPS) レートが過剰に低い場合は、ログ発行がスロットリングする可能性があります。CloudWatch のスロットリング設定は、EMR Serverless を含むすべてのサービスに対してグローバルです。詳細については、AWS re:post の「[How do I determine throttling in my CloudWatch logs?](#)」を参照してください。

CloudWatch でのログ記録に必要なアクセス許可

ジョブが Amazon CloudWatch にログデータを送信できるようにするには、ジョブランタイムロールのアクセス許可ポリシーに次のアクセス許可を設定します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:DescribeLogGroups"
      ],
      "Resource": [
        "arn:aws:logs:*:123456789012:*"
      ],
      "Sid": "AllowLOGSDescribeLogGroups"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:DescribeLogStreams"
      ],
      "Resource": [
        "arn:aws:logs:*:123456789012:log-group:my-log-group-name:*"
      ],
      "Sid": "AllowLOGSPutLogEvents"
    }
  ]
}
```

AWS CLI

から EMR Serverless のログを保存するように Amazon CloudWatch を設定するには AWS CLI、ジョブ実行を開始するときに `cloudWatchLoggingConfiguration` 設定を使用します。これを行

うには、次の設定のオーバーライドを指定します。必要に応じて、ロググループ名、ログストリームプレフィックス名、ログタイプ、および暗号化キー ARN を指定します。

オプションの値を指定しない場合、CloudWatch はデフォルトのログストリーム / applications/*applicationId*/jobs/*jobId*/*worker-type* を使用して、デフォルトのロググループ /aws/emr-serverless にログを発行します。

EMR Serverless リリース 7.1.0 以降では、ストリーミングジョブとバッチジョブの再試行がサポートされています。ジョブの再試行を有効にした場合、EMR Serverless はログパスプレフィックスに試行番号を自動的に追加するため、ログをより適切に区別して追跡できます。

```
'/applications/<applicationId>/jobs/<jobId>/attempts/<attemptNumber>/worker-type'
```

以下では、EMR Serverless のデフォルト設定で Amazon CloudWatch ログ記録を有効にするために必要な最小設定をデモンストレーションします。

```
{
  "monitoringConfiguration": {
    "cloudWatchLoggingConfiguration": {
      "enabled": true
    }
  }
}
```

次の例は、EMR Serverless の Amazon CloudWatch ログ記録を有効にする際に指定する必須およびオプションのすべての設定を示しています。サポートされている logTypes 値もこの例の下に示されています。

```
{
  "monitoringConfiguration": {
    "cloudWatchLoggingConfiguration": {
      "enabled": true, // Required
      "logGroupName": "Example_logGroup", // Optional
      "logStreamNamePrefix": "Example_logStream", // Optional
      "encryptionKeyArn": "key-arn", // Optional
      "logTypes": {
        "SPARK_DRIVER": ["stdout", "stderr"] //List of values
      }
    }
  }
}
```

```
}
```

デフォルトでは、EMR Serverless はドライバーの stdout と stderr ログのみを CloudWatch に発行します。他のログが必要な場合は、logTypes フィールドでコンテナロールと対応するログタイプを指定します。

次のリストは、logTypes 設定で指定するサポートされているワーカータイプを示しています:

Spark

- SPARK_DRIVER : ["STDERR", "STDOUT"]
- SPARK_EXECUTOR : ["STDERR", "STDOUT"]

[Hive]

- HIVE_DRIVER : ["STDERR", "STDOUT", "HIVE_LOG", "TEZ_AM"]
- TEZ_TASK : ["STDERR", "STDOUT", "SYSTEM_LOGS"]

ログのローテーション

Amazon EMR Serverless は、Spark アプリケーションログとイベントログをローテーションできます。ログローテーションによって、長時間実行されるジョブが大きなログファイルを生成し、すべてのディスク領域を占有する可能性がある問題を解消できます。ログをローテーションすると、ディスクストレージを節約し、ディスクに空き領域がなくなるため、ジョブの失敗を低減することができます。

ログローテーションはデフォルトで有効になっており、Spark ジョブでのみ利用できます。

Spark イベントログ

Note

Spark イベントログローテーションは、すべての Amazon EMR リリースラベルで利用できます。

EMR Serverless は、単一のイベントログファイルを生成する代わりに、イベントログを定期的にローテーションし、古いイベントログファイルを削除します。ログをローテーションしても、S3 バケットにアップロードされたログには影響しません。

Spark アプリケーションログ

Note

Spark アプリケーションのログローテーションは、すべての Amazon EMR リリースラベルで利用できます。

EMR Serverless は、stdout ファイルや stderr ファイルなどのドライバーやエグゼキュターの Spark アプリケーションログもローテーションします。Spark 履歴サーバーと Live UI リンクを使用して Studio のログリンクを選択すると、最新のログファイルにアクセスできます。ログファイルは、最新のログの切り捨てられたバージョンです。古いローテーションされたログを参照するには、ログを保存するときに Amazon S3 の場所を指定します。詳細については、「[Logging for EMR Serverless with Amazon S3 buckets](#)」を参照してください。

最新のログファイルは、次の場所にあります。EMR Serverless は 15 秒ごとにファイルを更新します。これらのファイルの範囲は 0 MB から 128 MB です。

```
<example-S3-logUri>/applications/<application-id>/jobs/<job-id>/SPARK_DRIVER/stderr.gz
```

次の場所には、古いローテーションされたファイルが格納されています。各ファイルは 128 MB です。

```
<example-S3-logUri>/applications/<application-id>/jobs/<job-id>/SPARK_DRIVER/archived/  
stderr_<index>.gz
```

Spark エグゼキュターにも同じ動作が適用されます。この変更は S3 ログ記録にのみ適用されます。ログローテーションでは、Amazon CloudWatch にアップロードされたログストリームに変更は加えられません。

EMR Serverless リリース 7.1.0 以降では、ストリーミングジョブとバッチジョブの再試行がサポートされています。ジョブで再試行を有効にした場合、EMR Serverless はそれらのジョブのログパスにプレフィックスを追加して、ログをより適切に追跡し、互いに区別できるようにします。このパスには、ローテーションされたすべてのログが含まれます。

```
'/applications/<applicationId>/jobs/<jobId>/attempts/<attemptNumber>/'.
```

ログの暗号化

マネージドストレージによる EMR Serverless ログの暗号化

独自の KMS キーを使用してマネージドストレージのログを暗号化するには、ジョブ実行を送信するときに `managedPersistenceMonitoringConfiguration` 設定を使用します。

```
{
  "monitoringConfiguration": {
    "managedPersistenceMonitoringConfiguration": {
      "encryptionKeyArn": "key-arn"
    }
  }
}
```

Amazon S3 バケットを使用した EMR Serverless ログの暗号化

独自の KMS キーを使用して Amazon S3 バケットのログを暗号化するには、ジョブ実行を送信するときに `s3MonitoringConfiguration` 設定を使用します。

```
{
  "monitoringConfiguration": {
    "s3MonitoringConfiguration": {
      "logUri": "s3://amzn-s3-demo-logging-bucket/logs/",
      "encryptionKeyArn": "key-arn"
    }
  }
}
```

Amazon CloudWatch による EMR Serverless ログの暗号化

独自の KMS キーを使用して Amazon CloudWatch のログを暗号化するには、ジョブ実行を送信するときに `cloudWatchLoggingConfiguration` 設定を使用します。

```
{
  "monitoringConfiguration": {
    "cloudWatchLoggingConfiguration": {
      "enabled": true,
      "encryptionKeyArn": "key-arn"
    }
  }
}
```



```
}  
}  
}
```

ログ暗号化に必要なアクセス許可

このセクションの内容

- [必要なユーザーアクセス許可](#)
- [Amazon S3 とマネージドストレージの暗号化キーのアクセス許可](#)
- [Amazon CloudWatch の暗号化キーアクセス許可](#)

必要なユーザーアクセス許可

ジョブを送信する、またはログやアプリケーション UI を表示するユーザーは、キーを使用するためのアクセス許可を付与されている必要があります。ユーザー、グループ、またはロールの KMS キーポリシーまたは IAM ポリシーでアクセス許可を指定できます。ジョブを送信するユーザーに KMS キーのアクセス許可が付与されていない場合、EMR Serverless はジョブ実行の送信を拒否します。

キーポリシーの例

次のキーポリシーによって、`kms:GenerateDataKey` と `kms:Decrypt` にアクセス許可が付与されます。

```
{  
  "Effect": "Allow",  
  "Principal": {  
    "AWS": "arn:aws:iam::111122223333:user/user-name"  
  },  
  "Action": [  
    "kms:GenerateDataKey",  
    "kms:Decrypt"  
  ],  
  "Resource": "*"   
}
```

IAM ポリシーの例

次の IAM ポリシーによって、`kms:GenerateDataKey` と `kms:Decrypt` にアクセス許可が付与されます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "kms:GenerateDataKey",
        "kms:Decrypt"
      ],
      "Resource": [
        "arn:aws:kms:*:123456789012:key/12345678-1234-1234-1234-123456789012"
      ],
      "Sid": "AllowKMSGeneratedakey"
    }
  ]
}
```

Spark UI または Tez UI を起動するには、次のように `emr-serverless:GetDashboardForJobRun` API にアクセスするためのアクセス許可をユーザー、グループ、またはロールに付与します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:GetDashboardForJobRun"
      ],
      "Resource": [
        "*"
      ],
      "Sid": "AllowEMRSERVERLESSGetdashboardforjobrun"
    }
  ]
}
```

Amazon S3 とマネージドストレージの暗号化キーのアクセス許可

マネージドストレージまたは S3 バケットで独自の暗号化キーを使用してログを暗号化する場合は、次のように KMS キーのアクセス許可を設定します。

emr-serverless.amazonaws.com プリンシパルには、KMS キーのポリシーで次のアクセス許可が設定されている必要があります。

```
{
  "Effect": "Allow",
  "Principal": {
    "Service": "emr-serverless.amazonaws.com"
  },
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey"
  ],
  "Resource": "*"
  "Condition": {
    "StringLike": {
      "aws:SourceArn": "arn:aws:emr-serverless:region:aws-account-id:/
applications/application-id"
    }
  }
}
```

セキュリティのベストプラクティスとして、KMS キーポリシーに `aws:SourceArn` 条件キーを追加することを提案します。IAM グローバル条件キー `aws:SourceArn` によって、EMR Serverless がアプリケーション ARN にのみ KMS キーを使用するように設定できます。

ジョブのランタイムロールには、IAM ポリシーで次のアクセス許可が付与されている必要があります。

JSON

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

```

        "kms:GenerateDataKey",
        "kms:Decrypt"
    ],
    "Resource": [
        "arn:aws:kms:*:123456789012:key/12345678-1234-1234-1234-123456789012"
    ],
    "Sid": "AllowKMSGeneratedatakey"
}
]
}
```

Amazon CloudWatch の暗号化キーアクセス許可

KMS キー ARN をロググループに関連付けるには、ジョブランタイムロールに次の IAM ポリシーを使用します。

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:AssociateKmsKey"
      ],
      "Resource": [
        "arn:aws:logs:*:123456789012:log-group:my-log-group-name:*"
      ],
      "Sid": "AllowLOGSAssociatekmskey"
    }
  ]
}
```

Amazon CloudWatch に KMS アクセス許可を付与するように KMS キーポリシーを設定します。

JSON

```

{
```

```
"Version": "2012-10-17",
"Id": "key-default-1",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "kms:Decrypt",
      "kms:GenerateDataKey"
    ],
    "Resource": [
      "*"
    ],
    "Condition": {
      "ArnLike": {
        "kms:EncryptionContext:aws:logs:arn": "arn:aws:logs:*:123456789012:*"
      }
    },
    "Sid": "AllowKMSDecrypt"
  }
]
```

Amazon EMR Serverless の Apache Log4j2 プロパティを設定する

このページでは、StartJobRun で EMR Serverless ジョブのカスタム [Apache Log4j 2.x](#) プロパティを設定する方法について説明します。アプリケーションレベルで Log4j 分類を設定する場合は、「[EMR Serverless のデフォルトのアプリケーション設定](#)」を参照してください。

Amazon EMR Serverless の Spark Log4j2 プロパティを設定する

Amazon EMR リリース 6.8.0 以降では、[Apache Log4j 2.x](#) プロパティをカスタマイズして、詳細なログ設定を指定できます。これにより、EMR Serverless での Spark ジョブのトラブルシューティングが容易になります。これらのプロパティを設定するには、spark-driver-log4j2 および spark-executor-log4j2 の分類を使用します。

トピック

- [Spark の Log4j2 分類](#)
- [Spark の Log4j2 設定例](#)
- [サンプル Spark ジョブの Log4j2](#)

- [Spark の Log4j2 に関する考慮事項](#)

Spark の Log4j2 分類

Spark ログ設定をカスタマイズするには、[applicationConfiguration](#) で次の分類を使用します。Log4j 2.x プロパティを設定するには、次の [properties](#) を使用します。

spark-driver-log4j2

この分類は、ドライバーの log4j2.properties ファイル内の値を設定します。

spark-executor-log4j2

この分類は、エグゼキューターの log4j2.properties ファイル内の値を設定します。

Spark の Log4j2 設定例

次の例は、applicationConfiguration を使用して Spark ジョブを送信し、Spark ドライバーとエグゼキューターの Log4j2 設定をカスタマイズする方法を示しています。

ジョブを送信するときではなく、アプリケーションレベルで Log4j 分類を設定するには、「[EMR Serverless のデフォルトのアプリケーション設定](#)」を参照してください。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "/usr/lib/spark/examples/jars/spark-examples.jar",
      "entryPointArguments": ["1"],
      "sparkSubmitParameters": "--class org.apache.spark.examples.SparkPi --conf
spark.executor.cores=4 --conf spark.executor.memory=20g --conf spark.driver.cores=4 --
conf spark.driver.memory=8g --conf spark.executor.instances=1"
    }
  }'
  --configuration-overrides '{
    "applicationConfiguration": [
      {
        "classification": "spark-driver-log4j2",
        "properties": {
          "rootLogger.level": "error", // will only display Spark error logs
          "logger.IdentifierForClass.name": "classpath for setting logger",
```

```

        "logger.IdentifierForClass.level": "info"
    }
},
{
    "classification": "spark-executor-log4j2",
    "properties": {
        "rootLogger.level": "error", // will only display Spark error logs
        "logger.IdentifierForClass.name": "classpath for setting logger",
        "logger.IdentifierForClass.level": "info"
    }
}
]
}'

```

サンプル Spark ジョブの Log4j2

次のコードサンプルは、アプリケーションのカスタム Log4j2 設定を初期化する際に Spark アプリケーションを作成する方法を示しています。

Python

Example- Python で Spark ジョブに Log4j2 を使用する

```

import os
import sys

from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession

app_name = "PySparkApp"
if __name__ == "__main__":
    spark = SparkSession\
        .builder\
        .appName(app_name)\
        .getOrCreate()

    spark.sparkContext._conf.getAll()
    sc = spark.sparkContext
    log4jLogger = sc._jvm.org.apache.log4j
    LOGGER = log4jLogger.LogManager.getLogger(app_name)

    LOGGER.info("pyspark script logger info")

```

```
LOGGER.warn("pyspark script logger warn")
LOGGER.error("pyspark script logger error")

// your code here

spark.stop()
```

Spark ジョブの実行時にドライバーの Log4j2 をカスタマイズするには、以下の設定を使用します:

```
{
  "classification": "spark-driver-log4j2",
  "properties": {
    "rootLogger.level": "error", // only display Spark error logs
    "logger.PySparkApp.level": "info",
    "logger.PySparkApp.name": "PySparkApp"
  }
}
```

Scala

Example- Scala で Spark ジョブに Log4j2 を使用する

```
import org.apache.log4j.Logger
import org.apache.spark.sql.SparkSession

object ExampleClass {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession
      .builder
      .appName(this.getClass.getName)
      .getOrCreate()

    val logger = Logger.getLogger(this.getClass);
    logger.info("script logging info logs")
    logger.warn("script logging warn logs")
    logger.error("script logging error logs")

    // your code here
    spark.stop()
  }
}
```


Spark ジョブの実行時にドライバーの Log4j2 をカスタマイズするには、以下の設定を使用します:

```
{
  "classification": "spark-driver-log4j2",
  "properties": {
    "rootLogger.level": "error", // only display Spark error logs
    "logger.ExampleClass.level": "info",
    "logger.ExampleClass.name": "ExampleClass"
  }
}
```

Spark の Log4j2 に関する考慮事項

Spark プロセスでは、次の Log4j2.x プロパティは設定できません。

- `rootLogger.appenderRef.stdout.ref`
- `appender.console.type`
- `appender.console.name`
- `appender.console.target`
- `appender.console.layout.type`
- `appender.console.layout.pattern`

設定できる Log4j2.x プロパティの詳細については、GitHub の「[log4j2.properties.template ファイル](#)」を参照してください。

EMR Serverless のモニタリング

このセクションでは、Amazon EMR Serverless アプリケーションとジョブをモニタリングする方法について説明します。

トピック

- [EMR Serverless アプリケーションとジョブのモニタリング](#)
- [Amazon Managed Service for Prometheus で Spark メトリクスをモニタリングする](#)
- [EMR Serverless 使用状況メトリクス](#)

EMR Serverless アプリケーションとジョブのモニタリング

EMR Serverless の Amazon CloudWatch メトリクスを使用することにより、1 分間の CloudWatch メトリクスを受け取り、CloudWatch ダッシュボードにアクセスして、EMR Serverless アプリケーションのパフォーマンスとオペレーションにほぼリアルタイムでアクセスできます。

EMR Serverless は CloudWatch に毎分メトリクスを送信します。EMR Serverless は、これらのメトリクスをアプリケーションレベルだけでなく、ジョブ、ワーカータイプ、容量の割り当てタイプのレベルで出力します。

開始するには、[EMR Serverless GitHub リポジトリ](#)に用意されている EMR Serverless CloudWatch ダッシュボードテンプレートを使用してデプロイします。

Note

[EMR Serverless インタラクティブワークロード](#)では、アプリケーションレベルのモニタリングのみが有効化されており、新しいワーカータイプのディメンション Spark_Kernel があります。インタラクティブワークロードをモニタリングおよびデバッグするには、[EMR Studio Workspace 内](#)からログと Apache Spark UI にアクセスします。

メトリクスのモニタリング

Important

メトリクス表示を再構築して、ディメンションとして ApplicationName および JobName を追加しています。リリース 7.10 以降では、古いメトリクスは更新されなくなります。7.10 以前の EMR リリースでは、古いメトリクスは引き続き使用できます。

現在のディメンション

以下の表は、AWS/EMR Serverless 名前空間内で使用可能な EMR Serverless ディメンションを示しています。

EMR Serverless メトリクスのディメンション

ディメンション	説明	
ApplicationId	アプリケーション ID を使用して、EMR Serverless アプリケーションのすべてのメトリクスをフィルタリングします。	
ApplicationName	その名前を使用して、EMR Serverless アプリケーションのすべてのメトリクスをフィルタリングします。名前が指定されていない場合、または ASCII 以外の文字が含まれている場合、[指定なし] として発行されます。	
JobId	EMR Serverless ジョブ実行 ID のすべてのメトリクスをフィルタリングします。	
JobName	その名前を使用して、EMR Serverless ジョブ実行のすべてのメトリクスをフィルタリングします。名前が指定されていない場合、または ASCII 以外の文字が含まれている場合、[指定なし] として発行されます。	
WorkerType	特定のワーカータイプのすべてのメトリクスをフィルタリングします。例えば、Spark ジョブの SPARK_DRIVER と SPARK_EXECUTORS をフィルタリングできます。	

ディメンション	説明	
CapacityAllocationType	特定の容量割り当てタイプのすべてのメトリクスをフィルタリングします。例えば、事前に初期化された容量と、その他すべての OnDemandCapacity に対して PreInitCapacity をフィルタリングできます。	

アプリケーションレベルのモニタリング

Amazon CloudWatch メトリクスを使用して、EMR Serverless アプリケーションレベルで容量の使用状況をモニタリングできます。CloudWatch ダッシュボードでアプリケーション容量の使用状況をモニタリングするように 1 つのディスプレイを設定することもできます。

EMR Serverless アプリケーションメトリクス

メトリクス	説明	Unit	ディメンション
MaxCPUAllowed	アプリケーションに許容される CPU の最大数。	vCPU	ApplicationId , ApplicationName
MaxMemory Allowed	アプリケーションに許容されるメモリの最大 GB 数。	ギガバイト数 (GB)	ApplicationId , ApplicationName
MaxStorageAllowed	アプリケーションに許容されるストレージの最大 GB 数。	ギガバイト数 (GB)	ApplicationId , ApplicationName
CPUAllocated	割り当てられた vCPU の合計数。	vCPU	ApplicationId , ApplicationName , WorkerType , CapacityAllocationType

メトリクス	説明	Unit	ディメンション
IdleWorkerCount	アイドル状態のワーカーの合計数。	カウント	ApplicationId , ApplicationName , WorkerType ,CapacityAllocationType
MemoryAllocated	割り当てられたメモリの合計 GB 数。	ギガバイト数 (GB)	ApplicationId , ApplicationName , WorkerType ,CapacityAllocationType
PendingCreationWorkerCount	作成保留中のワーカーの合計数。	カウント	ApplicationId , ApplicationName , WorkerType ,CapacityAllocationType
RunningWorkerCount	アプリケーションで使われているワーカーの合計数。	カウント	ApplicationId , ApplicationName , WorkerType ,CapacityAllocationType
StorageAllocated	割り当てられたディスクストレージの合計 GB 数。	ギガバイト数 (GB)	ApplicationId , ApplicationName , WorkerType ,CapacityAllocationType
TotalWorkerCount	使用可能なワーカーの合計数。	カウント	ApplicationId , ApplicationName , WorkerType ,CapacityAllocationType

ジョブレベルのモニタリング

Amazon EMR Serverless は、1 分ごとに次のジョブレベルのメトリクスを Amazon CloudWatch に送信します。ジョブ実行集計のメトリクス値は、ジョブ実行状態別にアクセスできます。各メトリクスの単位はカウントです。

EMR Serverless ジョブレベルのメトリクス

メトリクス	説明	ディメンション
SubmittedJobs	送信済み状態のジョブの数。	ApplicationId , ApplicationName
PendingJobs	保留中状態のジョブの数。	ApplicationId , ApplicationName
ScheduledJobs	スケジュールされた状態のジョブの数。	ApplicationId , ApplicationName
RunningJobs	実行中状態のジョブの数。	ApplicationId , ApplicationName
SuccessJobs	成功状態のジョブの数。	ApplicationId , ApplicationName
FailedJobs	失敗状態のジョブの数。	ApplicationId , ApplicationName
CancellingJobs	キャンセル中状態のジョブの数。	ApplicationId , ApplicationName
CancelledJobs	キャンセル済み状態のジョブの数。	ApplicationId , ApplicationName

エンジン固有のアプリケーション UI を使用して、実行中の EMR Serverless ジョブと完了した EMR Serverless ジョブのエンジン固有のメトリクスをモニタリングできます。実行中のジョブの UI にアクセスすることにより、リアルタイムの更新が反映されたライブアプリケーション UI が表示されます。完了したジョブの UI にアクセスすることにより、永続アプリケーション UI が表示されます。

ジョブの実行

実行中の EMR Serverless ジョブでは、エンジン固有のメトリクスを示すリアルタイムインターフェイスにアクセスします。Apache Spark UI または Hive Tez UI のいずれかを使用して、ジョブをモニタリングおよびデバッグできます。これらの UI にアクセスするには、EMR Studio コンソールを使用するか、AWS Command Line Interface で安全な URL エンドポイントをリクエストします。

完了したジョブ

完了した EMR Serverless ジョブについては、Spark History Server または Persistent Hive Tez UI を使用して、Spark または Hive ジョブ実行のジョブの詳細、ステージ、タスク、メトリクスにアクセスします。これらの UI にアクセスするには、EMR Studio コンソールを使用するか、AWS Command Line Interface で安全な URL エンドポイントをリクエストします。

ジョブワーカーレベルのモニタリング

Amazon EMR Serverless は、AWS/EMRServerless 名前空間および Job Worker Metrics メトリクスグループで利用可能な以下のジョブワーカーレベルのメトリクスを Amazon CloudWatch に送信します。EMR Serverless は、ジョブレベル、ワーカータイプ、容量割り当てタイプのレベルでジョブの実行中に個別のワーカーからデータポイントを収集します。ApplicationId をディメンションとして使用して、同じアプリケーションに属する複数のジョブをモニタリングできます。

Note

Amazon CloudWatch コンソールでメトリクスを表示するときに EMR Serverless ジョブが使用する CPU とメモリの合計を表示するには、Statistic as Sum and Period as 1 minutes を使用します。

EMR Serverless ジョブワーカーレベルのメトリクス

メトリクス	説明	Unit	ディメンション
WorkerCpuAllocated	ジョブ実行でワーカーに割り当てられた vCPU コアの合計数。	vCPU	JobId、JobName、ApplicationId、ApplicationName、WorkerType、および CapacityAllocationType

メトリクス	説明	Unit	ディメンション	
WorkerCpuUsed	ジョブ実行でワーカーによって使用されている vCPU コアの合計数。	vCPU	JobId、JobName、ApplicationId、ApplicationName、WorkerType、およびCapacityAllocationType	
WorkerMemoryAllocated	ジョブ実行でワーカーに割り当てられたメモリの合計 GB 数。	ギガバイト数 (GB)	JobId、JobName、ApplicationId、ApplicationName、WorkerType、およびCapacityAllocationType	
WorkerMemoryUsed	ジョブ実行でワーカーによって使用されるメモリの合計 GB 数。	ギガバイト数 (GB)	JobId、JobName、ApplicationId、ApplicationName、WorkerType、およびCapacityAllocationType	
WorkerEphemeralStorageAllocated	ジョブ実行でワーカーに割り当てられたエフェメラルストレージのバイト数。	ギガバイト数 (GB)	JobId、JobName、ApplicationId、ApplicationName、WorkerType、およびCapacityAllocationType	

メトリクス	説明	Unit	ディメンション
WorkerEphemeralStorageUsed	ジョブ実行でワーカーによって使用されるエフェメラルストレージのバイト数。	ギガバイト数 (GB)	JobId、JobName、ApplicationId、ApplicationName、WorkerType、およびCapacityAllocationType
WorkerStorageReadBytes	ジョブ実行でワーカーがストレージから読み取るバイト数。	バイト	JobId、JobName、ApplicationId、ApplicationName、WorkerType、およびCapacityAllocationType
WorkerStorageWriteBytes	ジョブ実行でワーカーからストレージに書き込まれたバイト数。	バイト	JobId、JobName、ApplicationId、ApplicationName、WorkerType、およびCapacityAllocationType

以下の手順では、さまざまなタイプのメトリクスにアクセスする方法について説明します。

Console

コンソールを使用してアプリケーション UI にアクセスするには

1. [コンソールからの開始方法](#)の手順を使用して、EMR Studio の EMR Serverless アプリケーションに移動します。
2. 実行中ジョブのエンジン固有のアプリケーション UI とログにアクセスするには:
 - a. RUNNING ステータスのジョブを選択します。

- b. [アプリケーションの詳細] ページでジョブを選択するか、ジョブの [ジョブの詳細] ページに移動します。
 - c. [UI を表示] ドロップダウンメニューで、[Spark UI] または [Hive Tez UI] のいずれかを選択して、ジョブタイプのアプリケーション UI に移動します。
 - d. Spark エンジンログにアクセスするには、Spark UI の [エグゼキューター] タブに移動し、ドライバーのログリンクを選択します。Hive エンジンログにアクセスするには、Hive Tez UI で適切な DAG のログリンクを選択します。
3. 完了したジョブのエンジン固有のアプリケーション UI とログにアクセスするには:
- a. SUCCESS ステータスのジョブを選択します。
 - b. アプリケーションの [アプリケーションの詳細] ページでジョブを選択するか、ジョブの [ジョブの詳細] ページに移動します。
 - c. [UI を表示] ドロップダウンメニューで、[Spark 履歴サーバー] または [Persistent Hive Tez UI] のいずれかを選択して、ジョブタイプのアプリケーション UI に移動します。
 - d. Spark エンジンログにアクセスするには、Spark UI の [エグゼキューター] タブに移動し、ドライバーのログリンクを選択します。Hive エンジンログにアクセスするには、Hive Tez UI で適切な DAG のログリンクを選択します。

AWS CLI

を使用してアプリケーション UI にアクセスするには AWS CLI

- 実行中のジョブと完了したジョブでアプリケーション UI にアクセスするために使用できる URL を生成するには、GetDashboardForJobRun API を呼び出します。

```
aws emr-serverless get-dashboard-for-job-run /  
--application-id <application-id> /  
--job-run-id <job-id>
```

生成した URL は 1 時間有効です。

Amazon Managed Service for Prometheus で Spark メトリクスをモニタリングする

Amazon EMR リリース 7.1.0 以降では、EMR Serverless を Amazon Managed Service for Prometheus と統合して、EMR Serverless ジョブとアプリケーションの Apache Spark メトリクスを

収集できます。この統合は、AWS コンソール、EMR Serverless API、または [awscli](#) を使用してジョブを送信するか、アプリケーションを作成するときに使用できます AWS CLI。

前提条件

Spark メトリクスを Amazon Managed Service for Prometheus に配信するには、次の前提条件を満たします。

- [Amazon Managed Service for Prometheus ワークスペースを作成します。](#) このワークスペースは、取り込みエンドポイントとして機能します。エンドポイント - リモート書き込み URL に表示された URL を書き留めます。EMR Serverless アプリケーションを作成するときに URL を指定する必要があります。
- モニタリング目的で Amazon Managed Service for Prometheus にジョブへのアクセス許可を付与するには、ジョブ実行ロールに次のポリシーを追加します。

```
{
  "Sid": "AccessToPrometheus",
  "Effect": "Allow",
  "Action": ["aps:RemoteWrite"],
  "Resource": "arn:aws:aps:<AWS_REGION>:<AWS_ACCOUNT_ID>:workspace/<WORKSPACE_ID>"
}
```

セットアップ

AWS コンソールを使用して Amazon Managed Service for Prometheus と統合されたアプリケーションを作成するには

1. 「[Amazon EMR Serverless の使用を開始する](#)」を参照してアプリケーションを作成します。
2. アプリケーションの作成中に、[カスタム設定を使用する] を選択し、設定するフィールドに情報を指定してアプリケーションを設定します。
3. [アプリケーションのログとメトリクス] で、[エンジンメトリクスを Amazon Managed Service for Prometheus に配信] を選択し、リモート書き込み URL を指定します。
4. 必要な他の設定を指定し、[アプリケーションの作成と起動] を選択します。

AWS CLI または EMR Serverless API を使用する

AWS CLI または EMR Serverless API を使用して、`create-application` または `start-job-run` コマンドの実行時に EMR Serverless アプリケーションを Amazon Managed Service for Prometheus と統合することもできます。

`create-application`

```
aws emr-serverless create-application \  
--release-label emr-7.1.0 \  
--type "SPARK" \  
--monitoring-configuration '{  
    "prometheusMonitoringConfiguration": {  
        "remoteWriteUrl": "https://aps-workspaces.<AWS_REGION>.amazonaws.com/  
workspaces/<WORKSPACE_ID>/api/v1/remote_write"  
    }  
'
```

`start-job-run`

```
aws emr-serverless start-job-run \  
--application-id <APPLICATION_ID> \  
--execution-role-arn <JOB_EXECUTION_ROLE> \  
--job-driver '{  
    "sparkSubmit": {  
        "entryPoint": "local:///usr/lib/spark/examples/src/main/python/pi.py",  
        "entryPointArguments": ["10000"],  
        "sparkSubmitParameters": "--conf spark.dynamicAllocation.maxExecutors=10"  
    }  
' \  
--configuration-overrides '{  
    "monitoringConfiguration": {  
        "prometheusMonitoringConfiguration": {  
            "remoteWriteUrl": "https://aps-workspaces.<AWS_REGION>.amazonaws.com/  
workspaces/<WORKSPACE_ID>/api/v1/remote_write"  
        }  
    }  
'
```

コマンドで `prometheusMonitoringConfiguration` を指定すると、EMR Serverless は Spark メトリクスを収集し、Amazon Managed Service for Prometheus の `remoteWriteUrl` エンドポイントに書き込むエージェントを使用して Spark ジョブを実行する必要があることが示されます。そ

の後、Amazon Managed Service for Prometheus の Spark メトリクスを使用して、視覚化、アラート、分析を行うことができます。

高度な設定プロパティ

EMR Serverless は、PrometheusServlet という名前の Spark 内のコンポーネントを使用して Spark メトリクスを収集し、パフォーマンスデータを Amazon Managed Service for Prometheus と互換性のあるデータに変換します。デフォルトでは、EMR Serverless は Spark にデフォルト値を設定し、PrometheusMonitoringConfiguration を使用してジョブを送信するときにドライバーとエグゼキューターのメトリクスを解析します。

次の表は、Amazon Managed Service for Prometheus にメトリクスを送信する Spark ジョブを送信するときに設定するすべてのプロパティを示しています。

Spark プロパティ	デフォルトの値	説明
<code>spark.metrics.conf</code> <code>.*.sink.prometheusServlet.class</code>	<code>org.apache.spark.metrics.sink.PrometheusServlet</code>	Spark が Amazon Managed Service for Prometheus にメトリクスを送信するために使用するクラス。デフォルトの動作をオーバーライドするには、独自のカスタムクラスを指定します。
<code>spark.metrics.conf</code> <code>.*.source.jvm.class</code>	<code>org.apache.spark.metrics.source.JvmSource</code>	Spark が基盤となる Java 仮想マシンから重要なメトリクスを収集して送信するために使用するクラス。JVM メトリクスの収集を停止するには、このプロパティを "" などの空の文字列に設定して無効にします。デフォルトの動作をオーバーライドするには、独自のカスタムクラスを指定します。

Spark プロパティ	デフォルトの値	説明
<code>spark.metrics.conf.driver.sink.prometheusServlet.path</code>	<code>/metrics/prometheus</code>	Amazon Managed Service for Prometheus がドライバーからメトリクスを収集するために使用する個別の URL。デフォルトの動作をオーバーライドするには、独自のパスを指定します。ドライバーメトリクスの収集を停止するには、このプロパティを "" などの空の文字列に設定して無効にします。
<code>spark.metrics.conf.executor.sink.prometheusServlet.path</code>	<code>/metrics/executor/prometheus</code>	Amazon Managed Service for Prometheus がエグゼキューターからメトリクスを収集するために使用する個別の URL。デフォルトの動作をオーバーライドするには、独自のパスを指定します。エグゼキューターメトリクスの収集を停止するには、このプロパティを "" などの空の文字列に設定して無効にします。

Spark メトリクスの詳細については、「[Apache Spark メトリクス](#)」を参照してください。

考慮事項と制限事項

Amazon Managed Service for Prometheus を使用して EMR Serverless からメトリクスを収集する場合は、以下の考慮事項と制限事項を検討してください。

- Amazon Managed Service for Prometheus を EMR Serverless で使用するためのサポートは、[Amazon Managed Service for Prometheus が一般的に利用可能なAWS リージョンでのみ利用できます。](#)

- Amazon Managed Service for Prometheus で Spark メトリクスを収集するためにエージェントを実行するには、より多くのワーカーのリソースが必要です。1 つの vCPU ワーカーなど、より小さなワーカーサイズを選択すると、ジョブの実行時間が長くなる可能性があります。
- EMR Serverless で Amazon Managed Service for Prometheus を使用するためのサポートは、Amazon EMR リリース 7.1.0 以降でのみ利用できます。
- Amazon Managed Service for Prometheus は、メトリクスを収集するには EMR Serverless を実行するのと同じアカウントにデプロイする必要があります。

EMR Serverless 使用状況メトリクス

Amazon CloudWatch 使用状況メトリクスを使用して、アカウントが使用するリソースを可視化できます。これらのメトリクスを使用して、CloudWatch グラフやダッシュボードでサービスの使用状況を可視化できます。

EMR Serverless 使用状況メトリクスは、Service Quotas に対応しています。使用量がサービスクォータに近づいたときに警告するアラームを設定することもできます。詳細については、「Service Quotas ユーザーガイド」の「[Service Quotas and Amazon CloudWatch alarms](#)」を参照してください。

EMR Serverless のサービスクォータの詳細については、「[EMR Serverless のエンドポイントとクォータ](#)」を参照してください。

EMR Serverless のサービスクォータ使用状況メトリクス

EMR Serverless は、以下のサービスクォータ使用状況メトリクスを AWS/Usage 名前空間で発行します。

メトリクス	説明
ResourceCount	アカウントで実行されている指定されたリソースの合計数。リソースは、メトリクスに関連付けられた ディメンション によって定義されます。

EMR Serverless サービスクォータ使用状況メトリクスのディメンション

次のディメンションを使用して、EMR Serverless が発行する使用状況メトリクスを絞り込むことができます。

ディメンション	値	説明
Service	EMR Serverless	リソース AWS のサービス を含む の名前。
Type	[リソース]	EMR Serverless がレポートするエンティティのタイプ。
Resource	vCPU	EMR Serverless が追跡しているリソースのタイプ。
Class	なし	EMR Serverless が追跡しているリソースのクラス。

を使用した EMR Serverless の自動化 Amazon EventBridge

Amazon EventBridge を使用して を自動化 AWS のサービス し、アプリケーションの可用性の問題やリソースの変更などのシステムイベントに自動的に対応できます。EventBridge は、AWS リソースの変更を記述するシステムイベントのほぼリアルタイムのストリームを提供します。簡単なルールを記述して、注目するイベントと、イベントがルールに一致した場合に自動的に実行するアクションを指定できます。EventBridge を使用すると、自動的に以下の内容を実行できます。

- AWS Lambda 関数を呼び出す
- Amazon Kinesis Data Streams にイベントを中継する
- AWS Step Functions ステートマシンをアクティブ化する
- Amazon SNS トピックまたは Amazon SQS キューへの通知

例えば、EMR Serverless で EventBridge を使用する場合、ETL ジョブが成功したときに AWS Lambda 関数をアクティブ化する、または ETL ジョブが失敗したときに Amazon SNS トピックに通知することができます。

EMR Serverless は、次の 4 種類のイベントを出力します。

- アプリケーションの状態変更イベント – アプリケーションの状態変更をすべて出力するイベント。アプリケーションの状態の詳細については、「[アプリケーションの状態](#)」を参照してください。
- ジョブの実行状態変更イベント – ジョブ実行のすべての状態変更を出力するイベント。詳細については、「[ジョブ実行状態](#)」を参照してください。
- ジョブ実行の再試行イベント – Amazon EMR Serverless リリース 7.1.0 以降から実行されるジョブのすべての再試行を出力するイベント。
- ジョブリソース使用率の更新イベント – ジョブ実行のリソース使用率の更新を 30 分に近い間隔で実行するイベント。

EMR Serverless EventBridge のイベント例

EMR Serverless によって報告されるイベントには、次の例のように、source に `aws.emr-serverless` の値が割り当てられます。

アプリケーションの状態変更イベント

次のイベント例は、CREATING 状態のアプリケーションを示しています。

```
{
  "version": "0",
  "id": "9fd3cf79-1ff1-b633-4dd9-34508dc1e660",
  "detail-type": "EMR Serverless Application State Change",
  "source": "aws.emr-serverless",
  "account": "123456789012",
  "time": "2022-05-31T21:16:31Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "applicationId": "00f1cb5c6anuij25",
    "applicationName": "3965ad00-8fba-4932-a6c8-ded32786fd42",
    "arn": "arn:aws:emr-serverless:us-east-1:111122223333:/
applications/00f1cb5c6anuij25",
    "releaseLabel": "emr-6.6.0",
    "state": "CREATING",
    "type": "HIVE",
    "createdAt": "2022-05-31T21:16:31.547953Z",
    "updatedAt": "2022-05-31T21:16:31.547970Z",
    "autoStopConfig": {
      "enabled": true,
```

```

        "idleTimeout": 15
      },
      "autoStartConfig": {
        "enabled": true
      }
    }
  }
}

```

ジョブ実行の状態変更イベント

次のイベント例は、SCHEDULED 状態から RUNNING 状態に移行するジョブ実行を示しています。

```

{
  "version": "0",
  "id": "00df3ec6-5da1-36e6-ab71-20f0de68f8a0",
  "detail-type": "EMR Serverless Job Run State Change",
  "source": "aws.emr-serverless",
  "account": "123456789012",
  "time": "2022-05-31T21:07:42Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "jobRunId": "00f1cbn5g4bb0c01",
    "applicationId": "00f1982r1uukb925",
    "arn": "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/00f1982r1uukb925/jobruns/00f1cbn5g4bb0c01",
    "releaseLabel": "emr-6.6.0",
    "state": "RUNNING",
    "previousState": "SCHEDULED",
    "createdBy": "arn:aws:sts::123456789012:assumed-role/
TestRole-402dcef3ad14993c15d28263f64381e4cda34775/6622b6233b6d42f59c25dd2637346242",
    "updatedAt": "2022-05-31T21:07:42.299487Z",
    "createdAt": "2022-05-31T21:07:25.325900Z"
  }
}

```

ジョブ実行再試行イベント

ジョブ実行の再試行イベントの例を以下に示します。

```

{
  "version": "0",
  "id": "00df3ec6-5da1-36e6-ab71-20f0de68f8a0",

```

```

    "detail-type": "EMR Serverless Job Run Retry",
    "source": "aws.emr-serverless",
    "account": "123456789012",
    "time": "2022-05-31T21:07:42Z",
    "region": "us-east-1",
    "resources": [],
    "detail": {
      "jobRunId": "00f1cbn5g4bb0c01",
      "applicationId": "00f1982r1uukb925",
      "arn": "arn:aws:emr-serverless:us-east-1:123456789012:/
applications/00f1982r1uukb925/jobruns/00f1cbn5g4bb0c01",
      "releaseLabel": "emr-6.6.0",
      "createdBy": "arn:aws:sts::123456789012:assumed-role/
TestRole-402dcef3ad14993c15d28263f64381e4cda34775/6622b6233b6d42f59c25dd2637346242",
      "updatedAt": "2022-05-31T21:07:42.299487Z",
      "createdAt": "2022-05-31T21:07:25.325900Z",
      //Attempt Details
      "previousAttempt": 1,
      "previousAttemptState": "FAILED",
      "previousAttemptCreatedAt": "2022-05-31T21:07:25.325900Z",
      "previousAttemptEndedAt": "2022-05-31T21:07:30.325900Z",
      "newAttempt": 2,
      "newAttemptCreatedAt": "2022-05-31T21:07:30.325900Z"
    }
  }
}

```

ジョブリソース使用率の更新

次のイベント例は、実行後にターミナル状態に移行したジョブの最終的なリソース使用率の更新を示しています。

```

{
  "version": "0",
  "id": "00df3ec6-5da1-36e6-ab71-20f0de68f8a0",
  "detail-type": "EMR Serverless Job Resource Utilization Update",
  "source": "aws.emr-serverless",
  "account": "123456789012",
  "time": "2022-05-31T21:07:42Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:emr-serverless:us-east-1:123456789012:/applications/00f1982r1uukb925/
jobruns/00f1cbn5g4bb0c01"
  ],
}

```

```

    "detail": {
      "applicationId": "00f1982r1uukb925",
      "jobRunId": "00f1cbn5g4bb0c01",
      "attempt": 1,
      "mode": "BATCH",
      "createdAt": "2022-05-31T21:07:25.325900Z",
      "startedAt": "2022-05-31T21:07:26.123Z",
      "calculatedFrom": "2022-05-31T21:07:42.299487Z",
      "calculatedTo": "2022-05-31T21:07:30.325900Z",
      "resourceUtilizationFinal": true,
      "resourceUtilizationForInterval": {
        "vCPUHour": 0.023,
        "memoryGBHour": 0.114,
        "storageGBHour": 0.228
      },
      "billedResourceUtilizationForInterval": {
        "vCPUHour": 0.067,
        "memoryGBHour": 0.333,
        "storageGBHour": 0
      },
      "totalResourceUtilization": {
        "vCPUHour": 0.023,
        "memoryGBHour": 0.114,
        "storageGBHour": 0.228
      },
      "totalBilledResourceUtilization": {
        "vCPUHour": 0.067,
        "memoryGBHour": 0.333,
        "storageGBHour": 0
      }
    }
  }
}

```

`startedAt` フィールドは、ジョブが実行中の状態に移行した場合にのみイベント内に表示されます。

リソースのタグ付け

EMR Serverless リソースの管理を容易にするために、タグを使用して各リソースに独自のメタデータを割り当てます。このセクションでは、タグ機能の概要とタグの作成方法について説明します。

トピック

- [タグとは](#)
- [リソースのタグ付け](#)
- [タグ付けに関する制限事項](#)
- [AWS CLI および Amazon EMR Serverless API を使用したタグの操作](#)

タグとは

タグは、AWS リソースに割り当てるラベルです。タグはそれぞれ、1つのキーと1つの値で構成されており、どちらもお客様側が定義します。タグを使用すると、目的、所有者、環境などの属性でAWS リソースを分類できます。同じ型のリソースが多い場合に、割り当てたタグに基づいて特定のリソースをすばやく識別します。例えば、Amazon EMR Serverless に一連のタグを定義して、各アプリケーションの所有者とスタックレベルを追跡します。リソースタイプごとに一貫した一連のタグキーを考案することを提案します。

タグは自動的にリソースに割り当てられません。リソースにタグを追加したら、任意の時点でタグの値を変更する、またはリソースからタグを削除します。タグには、Amazon EMR Serverless に関連する意味論的な意味はなく、厳密に文字列として解釈されます。そのリソースの既存のタグと同じキーを持つタグを追加した場合、古い値は新しい値によって上書きされます。

IAM を使用する場合は、AWS アカウントのどのユーザーがタグを管理するアクセス許可を持っているかを制御できます。タグベースのアクセスコントロールポリシーの例については、「[タグベースのアクセスコントロールのポリシー](#)」を参照してください。

リソースのタグ付け

新規または既存のアプリケーションとジョブの実行にタグを付けることができます。Amazon EMR Serverless API、AWS CLI、または AWS SDK を使用している場合は、関連する API アクションの tags パラメータを使用して、新しいリソースにタグを適用できます。TagResource API アクションを使用して既存のリソースにタグを適用することもできます。

リソースの作成時に、リソースのタグを指定するためのいくつかのリソース作成アクションを使用できます。その場合、リソースの作成中にタグを適用できないときは、リソースの作成は失敗します。これにより、作成時にタグ付けしたリソースが、指定したタグで作成されているか、まったく作成されていないかが確認できます。作成時にリソースにタグ付けを行う場合、リソースの作成後にカスタムタグ付けスクリプトを実行する必要はありません。

次の表に、タグ付け可能な Amazon EMR Serverless リソースを示します。

タグ付け可能な リソース

[リソース]	タグをサポート	タグの伝播をサポート	作成時のタグ付けをサポート (Amazon EMR Serverless API、AWS CLIおよび AWS SDK)	作成時の API (作成時にタグを追加可能)
アプリケーション	はい	いいえ。アプリケーションに関連付けられたタグは、対象のアプリケーションに送信されたジョブ実行には反映されません。	はい	CreateApplication
ジョブ実行	はい	なし	はい	StartJobRun

タグ付けに関する制限事項

次の基本的な制限事項がタグに適用されます。

- 各リソースには、最大 50 個のユーザー作成タグを設定できます。
- タグキーはリソースごとにそれぞれ一意である必要があります。また、各タグキーに設定できる値は 1 つのみです。
- キーの最大長は UTF-8 で 128 Unicode 文字です。

- 値の最大長は UTF-8 で 256 Unicode 文字です。
- 使用できる文字は、UTF-8 対応の文字、数字、スペースと、文字 (`_ . : / = + - @`) です。
- タグキーを空の文字列にすることはできません。タグの値を空の文字列にすることはできますが、null にすることはできません。
- タグのキーと値では、大文字と小文字が区別されます。
- キーまたは値のプレフィックスとして、AWS: またはその大文字と小文字の組み合わせを変えたものを使用しないでください。これらは AWS でのみ使用するように予約されています。

AWS CLI および Amazon EMR Serverless API を使用したタグの操作

次の AWS CLI コマンドまたは Amazon EMR Serverless API オペレーションを使用して、リソースのタグを追加、更新、一覧表示、削除します。

タグの CLI コマンドと API オペレーション

[リソース]	タグをサポート	タグの伝播をサポート
1 つ以上のタグを追加、または上書きします	tag-resource	TagResource
リソースのタグの一覧表示	list-tags-for-resource	ListTagsForResource
1 つ以上のタグを削除します	untag-resource	UntagResource

以下の例では、AWS CLIを使用して、リソースに対してタグ付けまたはタグ削除する方法をデモンストレーションしています。

既存のアプリケーションにタグを付ける

次のコマンドは既存のアプリケーションにタグ付けします。

```
aws emr-serverless tag-resource --resource-arn resource_ARN --tags team=devs
```

既存のアプリケーションのタグを削除する

次のコマンドは既存のアプリケーションからタグを削除します。

```
aws emr-serverless untag-resource --resource-arn resource_ARN --tag-keys tag_key
```

リソースのタグの一覧表示

次のコマンドは、既存のリソースに関連付けられているタグのリストを取得します。

```
aws emr-serverless list-tags-for-resource --resource-arn resource_ARN
```


EMR Serverless のチュートリアル

このセクションでは、EMR Serverless アプリケーションで作業する際の一般的なユースケースについて説明します。これには、Hudi や Iceberg など、大規模なデータセットを処理し、Python や Python ライブラリを使用して Spark ジョブを送信するためのさまざまなツールが含まれます。

トピック

- [Amazon EMR Serverless での Java 17 の使用](#)
- [EMR Serverless での Apache Hudi の使用](#)
- [EMR Serverless での Apache Iceberg の使用](#)
- [EMR Serverless での Python ライブラリの使用](#)
- [EMR Serverless でのさまざまな Python バージョンの使用](#)
- [EMR Serverless での Delta Lake OSS の使用](#)
- [Airflow からの EMR Serverless ジョブの送信](#)
- [EMR Serverless での Hive ユーザー定義関数の使用](#)
- [EMR Serverless でのカスタムイメージの使用](#)
- [Amazon EMR Serverless での Amazon Redshift integration for Apache Spark の使用](#)
- [Amazon EMR Serverless を使用した DynamoDB への接続](#)

Amazon EMR Serverless での Java 17 の使用

Amazon EMR リリース 6.11.0 以降では、Java 仮想マシン (JVM) で Java 17 ランタイムを使用するように EMR Serverless Spark ジョブを設定します。次のいずれかの方法を使用して、Java 17 で Spark を設定します。

JAVA_HOME

EMR Serverless 6.11.0 以降の JVM 設定を上書きするには、`spark.emr-serverless.driverEnv` および `spark.executorEnv` の環境分類に `JAVA_HOME` 設定を指定します。

x86_64

必要なプロパティを設定して、Spark ドライバーとエグゼキューターの `JAVA_HOME` 設定として Java 17 を指定します。

```
--conf spark.emr-serverless.driverEnv.JAVA_HOME=/usr/lib/jvm/java-17-amazon-corretto.x86_64/
--conf spark.executorEnv.JAVA_HOME=/usr/lib/jvm/java-17-amazon-corretto.x86_64/
```

arm_64

必要なプロパティを設定して、Spark ドライバーとエグゼキューターの JAVA_HOME 設定として Java 17 を指定します。

```
--conf spark.emr-serverless.driverEnv.JAVA_HOME=/usr/lib/jvm/java-17-amazon-corretto.aarch64/
--conf spark.executorEnv.JAVA_HOME=/usr/lib/jvm/java-17-amazon-corretto.aarch64/
```

spark-defaults

また、spark-defaults 分類で Java 17 を指定して、EMR Serverless 6.11.0 以降の JVM 設定を上書きすることもできます。

x86_64

spark-defaults 分類で Java 17 を指定します。

```
{
  "applicationConfiguration": [
    {
      "classification": "spark-defaults",
      "properties": {
        "spark.emr-serverless.driverEnv.JAVA_HOME" : "/usr/lib/jvm/java-17-amazon-corretto.x86_64/",
        "spark.executorEnv.JAVA_HOME": "/usr/lib/jvm/java-17-amazon-corretto.x86_64/"
      }
    }
  ]
}
```

arm_64

spark-defaults 分類で Java 17 を指定します。

```
{
```

```
"applicationConfiguration": [  
  {  
    "classification": "spark-defaults",  
    "properties": {  
      "spark.emr-serverless.driverEnv.JAVA_HOME" : "/usr/lib/jvm/java-17-  
amazon-corretto.aarch64/",  
      "spark.executorEnv.JAVA_HOME": "/usr/lib/jvm/java-17-amazon-  
corretto.aarch64/"  
    }  
  }  
]  
}
```

EMR Serverless での Apache Hudi の使用

このセクションでは、EMR Serverless アプリケーションで Apache Hudi を使用する方法について説明します。Hudi は、データ処理をよりシンプルにするデータ管理フレームワークです。

EMR Serverless アプリケーションで Apache Hudi を使用するには

1. 対応する Spark ジョブ実行に必要な Spark プロパティを設定します。

```
spark.jars=/usr/lib/hudi/hudi-spark-bundle.jar,/usr/lib/hudi/hudi-utilities-  
bundle.jar,/usr/lib/hudi/hudi-aws-bundle.jar  
spark.serializer=org.apache.spark.serializer.KryoSerializer
```

2. Hudi テーブルを設定済みカタログに同期するには、AWS Glue データカタログをメタストアとして指定するか、外部メタストアを設定します。EMR Serverless は、Hudi ワークロードで Hive テーブルの同期モードとして hms をサポートします。EMR Serverless は、このプロパティをデフォルトでアクティブ化します。メタストアの設定方法の詳細については、「[EMR Serverless のメタストア設定](#)」を参照してください。

Important

EMR Serverless は、Hudi ワークロードを処理する Hive テーブルの同期モードオプションとして HIVEQL または JDBC をサポートしていません。詳細については、「[Sync modes](#)」を参照してください。

AWS Glue データカタログをメタストアとして使用する場合は、Hudi ジョブに次の設定プロパティを指定します。

```
--conf spark.jars=/usr/lib/hudi/hudi-spark-bundle.jar,  
--conf spark.serializer=org.apache.spark.serializer.KryoSerializer,  
--conf  
    spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClient
```

Amazon EMR の Apache Hudi リリースの詳細については、「[Hudi release history](#)」を参照してください。

EMR Serverless での Apache Iceberg の使用

このセクションでは、EMR Serverless アプリケーションで Apache Iceberg を使用方法について説明します。Apache Iceberg は、データレイクで大規模なデータセットを扱うのに役立つテーブル形式です。

EMR Serverless アプリケーションで Apache Iceberg を使用するには

1. 対応する Spark ジョブ実行に必要な Spark プロパティを設定します。

```
spark.jars=/usr/share/aws/iceberg/lib/iceberg-spark3-runtime.jar
```

2. Glue データカタログ AWS をメタストアとして指定するか、外部メタストアを設定します。メタストアの設定の詳細については、「[EMR Serverless のメタストア設定](#)」を参照してください。

Iceberg に使用するメタストアプロパティを設定します。たとえば、AWS Glue データカタログを使用する場合は、アプリケーション設定で次のプロパティを設定します。

```
spark.sql.catalog.dev.warehouse=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/  
spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions  
spark.sql.catalog.dev=org.apache.iceberg.spark.SparkCatalog  
spark.sql.catalog.dev.catalog-impl=org.apache.iceberg.aws.glue.GlueCatalog  
spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClient
```

AWS Glue データカタログをメタストアとして使用する場合は、Iceberg ジョブに次の設定プロパティを指定します。

```
--conf spark.jars=/usr/share/aws/iceberg/lib/iceberg-spark3-runtime.jar,  
--conf  
    spark.sql.extensions=org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions,  
--conf spark.sql.catalog.dev=org.apache.iceberg.spark.SparkCatalog,  
--conf spark.sql.catalog.dev.catalog-impl=org.apache.iceberg.aws.glue.GlueCatalog,  
--conf spark.sql.catalog.dev.warehouse=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/  
--conf  
    spark.hadoop.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGlueMetastore
```

Amazon EMR の Apache Iceberg リリースの詳細については、「[Iceberg release history](#)」を参照してください。

EMR Serverless での Python ライブラリの使用

Amazon EMR Serverless アプリケーションで PySpark ジョブを実行する際に、さまざまな Python ライブラリを依存関係としてパッケージ化します。これを行うには、ネイティブ Python 機能を使用するか、仮想環境を構築するか、Python ライブラリを使用するように PySpark ジョブを直接設定します。このページでは、それぞれの方法について説明します。

ネイティブ Python 機能の使用

以下の設定を行うと、PySpark を使用して Python ファイル (.py)、圧縮 Python パッケージ (.zip)、および Egg ファイル (.egg) を Spark エグゼキューターにアップロードします。

```
--conf spark.submit.pyFiles=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/<.py|.egg|.zip file>
```

PySpark ジョブで Python 仮想環境を使用する方法の詳細については、「[Using PySpark Native Features](#)」を参照してください。

EMR Notebook を使用する場合、次のコードを実行して、Python 依存関係を Notebook で使用できるようにできます。

```
%%configure -f  
{  
    "conf": {
```

```
"spark.submit.pyFiles": "s3:///amzn-s3-demo-bucket/EXAMPLE-PREFIX/<.py|.egg|.zip  
file>  
}  
}
```

Python 仮想環境の構築

PySpark ジョブの Python ライブラリをパッケージ化するために、独立した Python 仮想環境を構築します。

1. Python 仮想環境を構築するには、次のコマンドを実行します。次の例では、`scipy` パッケージと `matplotlib` パッケージを仮想環境パッケージにインストールし、アーカイブを Amazon S3 の場所にコピーします。

Important

EMR Serverless で使用するのと同じバージョンの Python、つまり Amazon EMR リリース 6.6.0 用の Python 3.7.10 を使用して、同様の Amazon Linux 2 環境で、次のコマンドを実行する必要があります。Dockerfile の例は、[EMR Serverless Samples](#) GitHub リポジトリにあります。

```
# initialize a python virtual environment  
python3 -m venv pyspark_venvsource  
source pyspark_venvsource/bin/activate  
  
# optionally, ensure pip is up-to-date  
pip3 install --upgrade pip  
  
# install the python packages  
pip3 install scipy  
pip3 install matplotlib  
  
# package the virtual environment into an archive  
pip3 install venv-pack  
venv-pack -f -o pyspark_venv.tar.gz  
  
# copy the archive to an S3 location  
aws s3 cp pyspark_venv.tar.gz s3:///amzn-s3-demo-bucket/EXAMPLE-PREFIX/  
  
# optionally, remove the virtual environment directory
```

```
rm -fr pyspark_venvsources
```

2. Python 仮想環境を使用するようにプロパティを設定し、Spark ジョブを送信します。

```
--conf spark.archives=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/  
pyspark_venv.tar.gz#environment  
--conf spark.emr-serverless.driverEnv.PYSPARK_DRIVER_PYTHON=./environment/bin/  
python  
--conf spark.emr-serverless.driverEnv.PYSPARK_PYTHON=./environment/bin/python  
--conf spark.executorEnv.PYSPARK_PYTHON=./environment/bin/python
```

元の Python バイナリを上書きしない場合、前述の一連の設定で 2 番目の設定が `--conf spark.executorEnv.PYSPARK_PYTHON=python` になることに注意してください。

PySpark ジョブに Python 仮想環境を使用する方法の詳細については、「[Using Virtualenv](#)」を参照してください。Spark ジョブを送信する方法の例については、「[EMR Serverless ジョブ実行時の Spark 設定の使用](#)」を参照してください。

Python ライブラリを使用するように PySpark ジョブを設定する

Amazon EMR リリース 6.12.0 以降では、新たにセットアップをしなくても、[pandas](#)、[NumPy](#)、[PyArrow](#) といった一般的なデータサイエンス向けの Python ライブラリを使用するように、直接 EMR Serverless PySpark ジョブを設定できます。

次の例は、PySpark ジョブで各 Python ライブラリをパッケージ化する方法をデモンストレーションしています。

NumPy

NumPy は、数学、並べ替え、ランダムシミュレーション、基本統計のための多次元配列とオペレーションを提供する科学計算用の Python ライブラリです。NumPy を使用するには、次のコマンドを実行します。

```
import numpy
```

pandas

pandas は、NumPy に構築された Python ライブラリです。pandas ライブラリは、[DataFrame](#) データ構造とデータ分析ツールをデータサイエンティストに提供します。pandas を使用するには、次のコマンドを実行します。

```
import pandas
```

PyArrow

PyArrow は、ジョブのパフォーマンスを向上させるためにインメモリ列指向データを管理する Python ライブラリです。PyArrow は、Apache Arrow のクロス言語開発仕様に基づいています。これは、データを列指向形式で表現して交換するための標準的な方法です。PyArrow を使用するには、次のコマンドを実行します。

```
import pyarrow
```

EMR Serverless でのさまざまな Python バージョンの使用

[EMR Serverless での Python ライブラリの使用](#) のユースケースに加えて、Python 仮想環境を使用して、Amazon EMR Serverless アプリケーションの Amazon EMR リリースでパッケージ化されたバージョンとは異なる Python バージョンを扱うこともできます。これを行うには、使用する Python バージョンで Python 仮想環境を構築します。

Python 仮想環境からジョブを送信するには

1. 次の例のコマンドを使用して仮想環境を構築します。この例では、Python 3.9.9 を仮想環境パッケージにインストールし、アーカイブを Amazon S3 の場所にコピーします。

Important

Amazon EMR リリース 7.0.0 以降を使用する場合は、EMR Serverless アプリケーションで使用するのと同様の Amazon Linux 2023 環境でコマンドを実行します。
リリース 6.15.0 以前を使用する場合は、同様の Amazon Linux 2 環境で次のコマンドを実行します。

```
# install Python 3.9.9 and activate the venv
yum install -y gcc openssl-devel bzip2-devel libffi-devel tar gzip wget make
wget https://www.python.org/ftp/python/3.9.9/Python-3.9.9.tgz && \
tar xzf Python-3.9.9.tgz && cd Python-3.9.9 && \
./configure --enable-optimizations && \
make altinstall
```



```
# create python venv with Python 3.9.9
python3.9 -m venv pyspark_venv_python_3.9.9 --copies
source pyspark_venv_python_3.9.9/bin/activate

# copy system python3 libraries to venv
cp -r /usr/local/lib/python3.9/* ./pyspark_venv_python_3.9.9/lib/python3.9/

# package venv to archive.
# **Note** that you have to supply --python-prefix option
# to make sure python starts with the path where your
# copied libraries are present.
# Copying the python binary to the "environment" directory.
pip3 install venv-pack
venv-pack -f -o pyspark_venv_python_3.9.9.tar.gz --python-prefix /home/hadoop/
environment

# stage the archive in S3
aws s3 cp pyspark_venv_python_3.9.9.tar.gz s3://<path>

# optionally, remove the virtual environment directory
rm -fr pyspark_venv_python_3.9.9
```

2. Python 仮想環境を使用するようにプロパティを設定し、Spark ジョブを送信します。

```
# note that the archive suffix "environment" is the same as the directory where you
  copied the Python binary.
--conf spark.archives=s3://amzn-s3-demo-bucket/EXAMPLE-PREFIX/
pyspark_venv_python_3.9.9.tar.gz#environment
--conf spark.emr-serverless.driverEnv.PYSPARK_DRIVER_PYTHON=./environment/bin/
python
--conf spark.emr-serverless.driverEnv.PYSPARK_PYTHON=./environment/bin/python
--conf spark.executorEnv.PYSPARK_PYTHON=./environment/bin/python
```

PySpark ジョブに Python 仮想環境を使用する方法の詳細については、「[Using Virtualenv](#)」を参照してください。Spark ジョブを送信する方法の例については、「[EMR Serverless ジョブ実行時の Spark 設定の使用](#)」を参照してください。

EMR Serverless での Delta Lake OSS の使用

Amazon EMR バージョン 6.9.0 以降

Note

Amazon EMR 7.0.0 以降では、Delta Lake 3.0.0 が使用され、`delta-core.jar` ファイル名が `delta-spark.jar` に変更されます。Amazon EMR 7.0.0 以降を使用する場合は、設定で `delta-spark.jar` を指定する必要があります。

Amazon EMR 6.9.0 以降には Delta Lake が含まれているため、Delta Lake を自分でパッケージ化したり、EMR Serverless ジョブで `--packages` フラグを指定したりする必要がなくなりました。

1. EMR Serverless ジョブを送信するときは、次の設定プロパティがあり、次のパラメータが `sparkSubmitParameters` フィールドに含まれていることを確認してください。

```
--conf spark.jars=/usr/share/aws/delta/lib/delta-core.jar,/usr/share/aws/delta/lib/delta-storage.jar
--conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension
--conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog
```

2. Delta テーブルの作成と読み取りをテストするローカル `delta_sample.py` を作成します。

```
# delta_sample.py
from pyspark.sql import SparkSession

import uuid

url = "s3://amzn-s3-demo-bucket/delta-lake/output/%s/" % str(uuid.uuid4())
spark = SparkSession.builder.appName("DeltaSample").getOrCreate()

## creates a Delta table and outputs to target S3 bucket
spark.range(5).write.format("delta").save(url)

## reads a Delta table and outputs to target S3 bucket
spark.read.format("delta").load(url).show
```

3. を使用して AWS CLI、`delta_sample.py` ファイルを Amazon S3 バケットにアップロードします。次に、`start-job-run` コマンドを使用して、既存の EMR Serverless アプリケーションにジョブを送信します。

```
aws s3 cp delta_sample.py s3://amzn-s3-demo-bucket/code/

aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --name emr-delta \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "s3://amzn-s3-demo-bucket/code/delta_sample.py",
      "sparkSubmitParameters": "--conf spark.jars=/usr/share/
aws/delta/lib/delta-core.jar,/usr/share/aws/delta/lib/delta-storage.jar --
conf spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
    }
  }'
```

Delta Lake で Python ライブラリを使用するには、`delta-core` ライブラリを[依存関係としてパッケージ化](#)するか、[カスタムイメージとして使用](#)して追加します。

また、`SparkContext.addPyFile` を使用して `delta-core` JAR ファイルから Python ライブラリを追加することもできます。

```
import glob
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()
spark.sparkContext.addPyFile(glob.glob("/usr/share/aws/delta/lib/delta-core_*.jar")[0])
```

Amazon EMR バージョン 6.8.0 以前

Amazon EMR 6.8.0 以前を使用している場合は、以下の手順に従って、EMR Serverless アプリケーションで Delta Lake OSS を使用します。

1. Amazon EMR Serverless アプリケーションの Spark のバージョンと互換性のある [Delta Lake](#) のオープンソースバージョンをビルドするには、[Delta GitHub](#) に移動して指示に従います。
2. Delta Lake ライブラリをの Amazon S3 バケットにアップロードします AWS アカウント。

3. アプリケーション設定で EMR Serverless ジョブを送信するときは、バケットに現在ある Delta Lake JAR ファイルを含めます。

```
--conf spark.jars=s3://amzn-s3-demo-bucket/jars/delta-core_2.12-1.1.0.jar
```

4. Delta テーブルとの読み書きを確実に行うには、サンプル PySpark テストを実行します。

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import HiveContext, SparkSession

import uuid

conf = SparkConf()
sc = SparkContext(conf=conf)
sqlContext = HiveContext(sc)

url = "s3://amzn-s3-demo-bucket/delta-lake/output/1.0.1/%s/" %
str(uuid.uuid4())

## creates a Delta table and outputs to target S3 bucket
session.range(5).write.format("delta").save(url)

## reads a Delta table and outputs to target S3 bucket
session.read.format("delta").load(url).show
```

Airflow からの EMR Serverless ジョブの送信

Apache Airflow の Amazon プロバイダーは、EMR Serverless 演算子を提供します。演算子の詳細については、Apache Airflow ドキュメントの「[Amazon EMR Serverless Operators](#)」を参照してください。

`EmrServerlessCreateApplicationOperator` を使用して Spark アプリケーションまたは Hive アプリケーションを作成できます。`EmrServerlessStartJobOperator` を使用して、新しいアプリケーションで 1 つ以上のジョブを開始することもできます。

Airflow 2.2.2 の Amazon Managed Workflows for Apache Airflow (MWAA) で 演算子を使用するには、次の行を `requirements.txt` ファイルに追加し、新しいファイルを使用するように MWAA 環境を更新します。

```
apache-airflow-providers-amazon==6.0.0
```

```
boto3>=1.23.9
```

Amazon プロバイダーのリリース 5.0.0 に EMR Serverless のサポートが追加されました。リリース 6.0.0 は、Airflow 2.2.2 と互換性のある最後のバージョンです。それ以降のバージョンは、MWAA の Airflow 2.4.3 で使用できます。

次の簡単な例は、アプリケーションを作成し、複数の Spark ジョブを実行してからアプリケーションを停止する方法を示しています。完全な例は、[EMR Serverless Samples](#) GitHub リポジトリにあります。sparkSubmit 設定の詳細については、「[EMR Serverless ジョブ実行時の Spark 設定の使用](#)」を参照してください。

```
from datetime import datetime

from airflow import DAG
from airflow.providers.amazon.aws.operators.emr import (
    EmrServerlessCreateApplicationOperator,
    EmrServerlessStartJobOperator,
    EmrServerlessDeleteApplicationOperator,
)

# Replace these with your correct values
JOB_ROLE_ARN = "arn:aws:iam::account-id:role/emr_serverless_default_role"
S3_LOGS_BUCKET = "amzn-s3-demo-bucket"

DEFAULT_MONITORING_CONFIG = {
    "monitoringConfiguration": {
        "s3MonitoringConfiguration": {"logUri": f"s3://{amzn-s3-demo-bucket}/logs/"}
    },
}

with DAG(
    dag_id="example_endtoend_emr_serverless_job",
    schedule_interval=None,
    start_date=datetime(2021, 1, 1),
    tags=["example"],
    catchup=False,
) as dag:
    create_app = EmrServerlessCreateApplicationOperator(
        task_id="create_spark_app",
        job_type="SPARK",
        release_label="emr-6.7.0",
        config={"name": "airflow-test"},
    )
```

```
application_id = create_app.output

job1 = EmrServerlessStartJobOperator(
    task_id="start_job_1",
    application_id=application_id,
    execution_role_arn=JOB_ROLE_ARN,
    job_driver={
        "sparkSubmit": {
            "entryPoint": "local:///usr/lib/spark/examples/src/main/python/
pi_fail.py",
        }
    },
    configuration_overrides=DEFAULT_MONITORING_CONFIG,
)

job2 = EmrServerlessStartJobOperator(
    task_id="start_job_2",
    application_id=application_id,
    execution_role_arn=JOB_ROLE_ARN,
    job_driver={
        "sparkSubmit": {
            "entryPoint": "local:///usr/lib/spark/examples/src/main/python/pi.py",
            "entryPointArguments": ["1000"]
        }
    },
    configuration_overrides=DEFAULT_MONITORING_CONFIG,
)

delete_app = EmrServerlessDeleteApplicationOperator(
    task_id="delete_app",
    application_id=application_id,
    trigger_rule="all_done",
)

(create_app >> [job1, job2] >> delete_app)
```

EMR Serverless での Hive ユーザー定義関数の使用

Hive ユーザー定義関数 (UDF) を使用すると、レコードまたはレコードのグループを処理するカスタム関数を作成できます。このチュートリアルでは、既存の Amazon EMR Serverless アプリケーショ

ンでサンプル UDF を使用して、クエリ結果を出力するジョブを実行します。アプリケーションの設定方法については、「[Amazon EMR Serverless の使用を開始する](#)」を参照してください。

EMR Serverless で UDF を使用するには

1. サンプル UDF の [GitHub](#) に移動します。リポジトリのクローンを作成し、使用する git ブランチに切り替えます。リポジトリの pom.xml ファイルの maven-compiler-plugin をソースを持つように更新します。また、ターゲットの Java のバージョン設定を 1.8 に更新します。mvn package -DskipTests を実行して、サンプル UDF を含む JAR ファイルを作成します。
2. JAR ファイルを作成したら、次のコマンドを使用して S3 バケットに JAR ファイルをアップロードします。

```
aws s3 cp brickhouse-0.8.2-JS.jar s3://amzn-s3-demo-bucket/jars/
```

3. サンプルファイルを作成して、サンプルの UDF 関数のいずれかを使用します。このクエリを udf_example.q として保存し、S3 バケットにアップロードします。

```
add jar s3://amzn-s3-demo-bucket/jars/brickhouse-0.8.2-JS.jar;
CREATE TEMPORARY FUNCTION from_json AS 'brickhouse.udf.json.FromJsonUDF';
select from_json('{"key1":[0,1,2], "key2":[3,4,5,6], "key3":[7,8,9]}', map("",
  array(cast(0 as int))));
select from_json('{"key1":[0,1,2], "key2":[3,4,5,6], "key3":[7,8,9]}', map("",
  array(cast(0 as int))))["key1"][2];
```

4. 次の Hive ジョブを送信します。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
    "hive": {
      "query": "s3://amzn-s3-demo-bucket/queries/udf_example.q",
      "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/
emr-serverless-hive/scratch --hiveconf hive.metastore.warehouse.dir=s3://'$BUCKET'/
emr-serverless-hive/warehouse"
    }
  }' --configuration-overrides '{
  "applicationConfiguration": [{
    "classification": "hive-site",
    "properties": {
      "hive.driver.cores": "2",
```

```
        "hive.driver.memory": "6G"
    }
  ]],
  "monitoringConfiguration": {
    "s3MonitoringConfiguration": {
      "logUri": "s3://amzn-s3-demo-bucket/logs/"
    }
  }
}
```

5. `get-job-run` コマンドを使用して、ジョブの状態を確認します。状態が `SUCCESS` になるまで待ちます。

```
aws emr-serverless get-job-run --application-id application-id --job-run-id job-id
```

6. 次のコマンドを使って、出力ファイルをダウンロードします。

```
aws s3 cp --recursive s3://amzn-s3-demo-bucket/logs/applications/application-id/
jobs/job-id/HIVE_DRIVER/ .
```

`stdout.gz` ファイルは次のようなものです。

```
{"key1": [0, 1, 2], "key2": [3, 4, 5, 6], "key3": [7, 8, 9]}
2
```

EMR Serverless でのカスタムイメージの使用

トピック

- [カスタム Python バージョンを使用する](#)
- [カスタム Java バージョンを使用する](#)
- [データサイエンスイメージを構築する](#)
- [Apache Sedona による地理空間データの処理](#)
- [カスタムイメージを使用するためのライセンス情報](#)

カスタム Python バージョンを使用する

別のバージョンの Python を使用するようにカスタムイメージを構築できます。例えば、Spark ジョブに Python バージョン 3.10 を使用するには、次のコマンドを実行します。

```
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest

USER root

# install python 3
RUN yum install -y gcc openssl-devel bzip2-devel libffi-devel tar gzip wget make
RUN wget https://www.python.org/ftp/python/3.10.0/Python-3.10.0.tgz && \
tar xzf Python-3.10.0.tgz && cd Python-3.10.0 && \
./configure --enable-optimizations && \
make altinstall

# EMRS runs the image as hadoop
USER hadoop:hadoop
```

Spark ジョブを送信する前に、次のように Python 仮想環境を使用するようにプロパティを設定します。

```
--conf spark.emr-serverless.driverEnv.PYSPARK_DRIVER_PYTHON=/usr/local/bin/python3.10
--conf spark.emr-serverless.driverEnv.PYSPARK_PYTHON=/usr/local/bin/python3.10
--conf spark.executorEnv.PYSPARK_PYTHON=/usr/local/bin/python3.10
```

カスタム Java バージョンを使用する

次の例は、Spark ジョブに Java 11 を使用するようにカスタムイメージを構築する方法を示しています。

```
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest

USER root

# install JDK 11
RUN amazon-linux-extras install java-openjdk11

# EMRS runs the image as hadoop
USER hadoop:hadoop
```

Spark ジョブを送信する前に、次のように Java 11 を使用するように Spark プロパティを設定します。

```
--conf spark.executorEnv.JAVA_HOME=/usr/lib/jvm/java-11-  
openjdk-11.0.16.0.8-1.amzn2.0.1.x86_64  
--conf spark.emr-serverless.driverEnv.JAVA_HOME=/usr/lib/jvm/java-11-  
openjdk-11.0.16.0.8-
```

データサイエンスイメージを構築する

次の例は、Pandas や NumPy などの一般的なデータサイエンス Python パッケージを組み込む方法を示しています。

```
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest  
  
USER root  
  
# python packages  
RUN pip3 install boto3 pandas numpy  
RUN pip3 install -U scikit-learn==0.23.2 scipy  
RUN pip3 install sk-dist  
RUN pip3 install xgboost  
  
# EMR Serverless runs the image as hadoop  
USER hadoop:hadoop
```

Apache Sedona による地理空間データの処理

次の例は、地理空間処理のために Apache Sedona を組み込むようにイメージを構築する方法を示しています。

```
FROM public.ecr.aws/emr-serverless/spark/emr-6.9.0:latest  
  
USER root  
  
RUN yum install -y wget  
RUN wget https://repo1.maven.org/maven2/org/apache/sedona/sedona-core-3.0_2.12/1.3.0-incubating/sedona-core-3.0_2.12-1.3.0-incubating.jar -P /usr/lib/spark/jars/  
RUN pip3 install apache-sedona  
  
# EMRS runs the image as hadoop
```

```
USER hadoop:hadoop
```

カスタムイメージを使用するためのライセンス情報

EMR Serverless でカスタムイメージを構築して、特定のタスクを実行したり、特定のバージョンのソフトウェアパッケージを使用したりできます。カスタムイメージの変更と配布には、ルールとライセンス条項が適用される場合があります。ライセンステキストは、以下のサブセクションに表示されます。

カスタムイメージに適用されるライセンス

Copyright Amazon.com and its affiliates; all rights reserved。このソフトウェアは、[AWS カスタマーアグリーメント](#)に基づく AWS コンテンツであり、許可なく配布することはできません。AWS 許諾者は、[AWS 知的財産ライセンス](#)のアクセス許可に加えて、これらの追加のアクセス許可を付与します。

AWS コンテンツの派生物の作成、コピー、使用は、以下の条件が満たされている場合に許可されます。

- AWS コンテンツ自体を変更することではなく、派生物は単に新しいコンテンツを追加した結果です。
- 内部複製は、上記の著作権表示を保持しなければなりません。
- ソースまたはバイナリ形式の外部配布は、変更の有無にかかわらず、本ライセンス条項では許可されません。

カスタムイメージの使用の詳細については、「[Using custom images with EMR Serverless](#)」を参照してください。

Amazon EMR Serverless での Amazon Redshift integration for Apache Spark の使用

Amazon EMR リリース 6.9.0 以降、リリースイメージには常に [Apache Spark](#) と Amazon Redshift をつなぐコネクタが含まれます。このコネクタを使用して、Amazon EMR Serverless で Spark を使用して Amazon Redshift に保存されているデータを処理します。このインテグレーションは、[spark-redshift オープンソースコネクタ](#)をベースにしています。Amazon EMR Serverless では、[Amazon Redshift integration for Apache Spark](#) がネイティブインテグレーションとして含まれています。

トピック

- [Amazon Redshift integration for Apache Spark を使用した Spark アプリケーションの起動](#)
- [Amazon Redshift integration for Apache Spark による認証](#)
- [Amazon Redshift に対する読み書き](#)
- [Spark コネクタを使用する際の考慮事項と制限事項](#)

Amazon Redshift integration for Apache Spark を使用した Spark アプリケーションの起動

EMR Serverless 6.9.0 とのインテグレーションを使用するには、Spark ジョブに必要な Spark Redshift の依存関係を渡します。--jars を使用して、Redshift コネクタ関連のライブラリを含めます。ファイルの保存先として --jars オプションでサポートされている他の場所にアクセスするには、Apache Spark ドキュメントの「[Advanced Dependency Management](#)」セクションを参照してください。

- spark-redshift.jar
- spark-avro.jar
- RedshiftJDBC.jar
- minimal-json.jar

Amazon EMR リリース 6.10.0 以降は minimal-json.jar 依存関係を必要とせず、デフォルトで他の依存関係を各クラスターに自動的にインストールします。以下の例は、Apache Spark 用の Amazon Redshift インテグレーションを使用して Spark アプリケーションを起動する方法をデモンストレーションしています。

Amazon EMR 6.10.0 +

EMR Serverless リリース 6.10.0 以降で Amazon Redshift integration for Apache Spark を使用して、Amazon EMR Serverless の Spark ジョブを起動します。

```
spark-submit my_script.py
```

Amazon EMR 6.9.0

EMR Serverless リリース 6.9.0 で Amazon Redshift integration for Apache Spark を使用して、Amazon EMR Serverless の Spark ジョブを起動するには、次の例に示すように --jars オ

プシオンを使用します。なお、`--jars` オプションに指定されているパスは JAR ファイルのデフォルトのパスであることに注意してください。

```
--jars
  /usr/share/aws/redshift/jdbc/RedshiftJDBC.jar,
  /usr/share/aws/redshift/spark-redshift/lib/spark-redshift.jar,
  /usr/share/aws/redshift/spark-redshift/lib/spark-avro.jar,
  /usr/share/aws/redshift/spark-redshift/lib/minimal-json.jar
```

```
spark-submit \
  --jars /usr/share/aws/redshift/jdbc/RedshiftJDBC.jar,/usr/share/aws/redshift/
spark-redshift/lib/spark-redshift.jar,/usr/share/aws/redshift/spark-redshift/lib/
spark-avro.jar,/usr/share/aws/redshift/spark-redshift/lib/minimal-json.jar \
  my_script.py
```

Amazon Redshift integration for Apache Spark による認証

AWS Secrets Manager を使用して認証情報を取得し、Amazon Redshift に接続する

Secrets Manager に認証情報を保存し、Spark ジョブに `GetSecretValue` API を呼び出させて認証情報を取得させることで、Amazon Redshift を安全に認証できます。

```
from pyspark.sql import SQLContextimport boto3

sc = # existing SparkContext
sql_context = SQLContext(sc)

secretsmanager_client = boto3.client('secretsmanager',
  region_name=os.getenv('AWS_REGION'))
secret_manager_response = secretsmanager_client.get_secret_value(
  SecretId='string',
  VersionId='string',
  VersionStage='string'
)
username = # get username from secret_manager_response
password = # get password from secret_manager_response
url = "jdbc:redshift://redshifthost:5439/database?user=" + username + "&password="
+ password
```

```
# Access to Redshift cluster using Spark
```

JDBC ドライバーによる Amazon Redshift の認証

JDBC URL 内にユーザー名とパスワードを設定する

Amazon Redshift クラスターに対して Spark ジョブを認証するには、JDBC URL に Amazon Redshift データベース名とパスワードを指定します。

Note

URL にデータベース認証情報を渡すと、その URL にアクセスできるユーザーなら誰でもその認証情報にアクセスできます。この方法は、安全な方法ではないため、一般的にはお勧めしません。

ご使用のアプリケーションでセキュリティが問題にならない場合は、JDBC URL に次の形式を使用してユーザー名とパスワードを設定します。

```
jdbc:redshift://redshifthost:5439/database?user=username&password=password
```

Amazon EMR Serverless ジョブ実行ロールでの IAM ベースの認証の使用

Amazon EMR Serverless リリース 6.9.0 以降、Amazon Redshift JDBC ドライバー 2.1 以降が環境にパッケージ化されます。JDBC ドライバー 2.1 以降では、JDBC URL を指定できます。未加工のユーザー名とパスワードを含めることはできません。

代わりに、`jdbc:redshift:iam://` スキームを指定します。このコマンドは、EMR Serverless ジョブ実行ロールを使用して認証情報を自動的に取得するように JDBC ドライバーに指示しています。詳細については、「Amazon Redshift 管理ガイド」の「[Configure a JDBC or ODBC connection to use IAM credentials](#)」を参照してください。この URL の例は次のとおりです。

```
jdbc:redshift:iam://examplecluster.abc123xyz789.us-west-2.redshift.amazonaws.com:5439/  
dev
```

指定された条件を満たすためには、ジョブ実行ロールに次の権限が必要です。

アクセス許可	ジョブ実行ロールで必要になる条件
<code>redshift:GetClusterCredentials</code>	JDBC ドライバーが Amazon Redshift から認証情報を取得するために必要
<code>redshift:DescribeCluster</code>	JDBC URL に Amazon Redshift クラスターのほか、エンドポイントではなく AWS リージョン を指定する場合に必要
<code>redshift-serverless:GetCredentials</code>	JDBC ドライバーが Amazon Redshift Serverless から認証情報を取得するために必要
<code>redshift-serverless:GetWorkgroup</code>	Amazon Redshift Serverless を使用していて、URL にワークグループ名とリージョンを含めて指定している場合に必要

別の VPC 内の Amazon Redshift への接続

VPC でプロビジョニングされた Amazon Redshift クラスターまたは Amazon Redshift Serverless ワークグループを設定する場合は、リソースにアクセスするように Amazon EMR Serverless アプリケーションの VPC 接続を設定します。EMR Serverless アプリケーションで VPC 接続を設定する方法の詳細については、「[データに接続するための EMR Serverless アプリケーションの VPC アクセスの設定](#)」を参照してください。

- プロビジョニングされた Amazon Redshift クラスターまたは Amazon Redshift Serverless ワークグループにパブリックにアクセスできる場合は、EMR Serverless アプリケーションの作成時に NAT ゲートウェイがアタッチされている 1 つ以上のプライベートサブネットを指定します。
- プロビジョニングされた Amazon Redshift クラスターまたは Amazon Redshift Serverless ワークグループにパブリックにアクセスできない場合は、「[データに接続するための EMR Serverless アプリケーションの VPC アクセスの設定](#)」の説明に従って、Amazon Redshift クラスターの Amazon Redshift マネージド VPC エンドポイントを作成する必要があります。また、「Amazon Redshift 管理ガイド」の「[Amazon Redshift Serverless への接続](#)」の説明に従って、Amazon Redshift Serverless ワークグループを作成することもできます。EMR Serverless アプリケーションの作成時に指定したプライベートサブネットにクラスターまたはサブグループを関連付ける必要があります。

Note

IAM ベースの認証を使用し、EMR Serverless アプリケーションのプライベートサブネットに NAT ゲートウェイがアタッチされていない場合は、Amazon Redshift または Amazon Redshift Serverless のサブネットに VPC エンドポイントを作成する必要があります。これにより、JDBC ドライバーは認証情報を取得できます。

Amazon Redshift に対する読み書き

次のコード例では、PySpark でデータソース API と SparkSQL を使用して、Amazon Redshift データベースに対してサンプルデータを読み書きします。

Data source API

PySpark を使用して、データソース API を使用して Amazon Redshift データベースとの間でサンプルデータを読み書きします。

```
import boto3
from pyspark.sql import SQLContext

sc = # existing SparkContext
sql_context = SQLContext(sc)

url = "jdbc:redshift:iam://redshifthost:5439/database"
aws_iam_role_arn = "arn:aws:iam::account-id:role/role-name"

df = sql_context.read \
    .format("io.github.spark_redshift_community.spark.redshift") \
    .option("url", url) \
    .option("dbtable", "table-name") \
    .option("tempdir", "s3://path/for/temp/data") \
    .option("aws_iam_role", "aws-iam-role-arn") \
    .load()

df.write \
    .format("io.github.spark_redshift_community.spark.redshift") \
    .option("url", url) \
    .option("dbtable", "table-name-copy") \
    .option("tempdir", "s3://path/for/temp/data") \
    .option("aws_iam_role", "aws-iam-role-arn") \
```



```
.mode("error") \  
.save()
```

SparkSQL

PySpark を使用して、SparkSQL を使用して Amazon Redshift データベースとの間でサンプルデータを読み書きします。

```
import boto3  
import json  
import sys  
import os  
from pyspark.sql import SparkSession  
  
spark = SparkSession \  
    .builder \  
    .enableHiveSupport() \  
    .getOrCreate()  
  
url = "jdbc:redshift:iam://redshifthost:5439/database"  
aws_iam_role_arn = "arn:aws:iam::account-id:role/role-name"  
  
bucket = "s3://path/for/temp/data"  
tableName = "table-name" # Redshift table name  
  
s = f"""CREATE TABLE IF NOT EXISTS {table-name} (country string, data string)  
    USING io.github.spark_redshift_community.spark.redshift  
    OPTIONS (dbtable '{table-name}', tempdir '{bucket}', url '{url}', aws_iam_role  
    '{aws-iam-role-arn'} '); """  
  
spark.sql(s)  
  
columns = ["country" ,"data"]  
data = [("test-country", "test-data")]  
df = spark.sparkContext.parallelize(data).toDF(columns)  
  
# Insert data into table  
df.write.insertInto(table-name, overwrite=False)  
df = spark.sql(f"SELECT * FROM {table-name}")  
df.show()
```

Spark コネクタを使用する際の考慮事項と制限事項

- Amazon EMR 上の Spark から Amazon Redshift への JDBC 接続に対して SSL を有効にすることを提案します。
- ベストプラクティスとして、AWS Secrets Manager で Amazon Redshift クラスターの認証情報を管理することを提案します。例については[AWS Secrets Manager](#)、「[を使用して Amazon Redshift に接続するための認証情報を取得する](#)」を参照してください。
- Amazon Redshift 認証パラメータのパラメータ `aws_iam_role` を使用して IAM ロールを渡すことを提案します。
- 現在、パラメータ `tempformat` は Parquet 形式をサポートしていません。
- `tempdir` URI は Amazon S3 の場所を指します。この一時ディレクトリは、自動的にクリーンアップされないため、追加コストが発生する可能性があります。
- Amazon Redshift については、次の推奨事項を検討してください。
 - Amazon Redshift クラスターにパブリックにアクセスできないようにすることを提案します。
 - [Amazon Redshift 監査ログ作成](#)を有効にすることを提案します。
 - [Amazon Redshift 保管時の暗号化](#)を有効にすることを提案します。
- Amazon S3 については、次の推奨事項を検討してください。
 - [Amazon S3 バケットへのパブリックアクセスをブロックする](#)ことを提案します。
 - [Amazon S3 サーバー側の暗号化](#)を使用して、使用する Amazon S3 バケットを暗号化することを提案します。
 - [Amazon S3 ライフサイクルポリシー](#)を使用して、Amazon S3 バケットの保持ルールを定義することを提案します。
 - Amazon EMR は、常にオープンソースからイメージにインポートされるコードを検証します。セキュリティのため、Spark から Amazon S3 への次の認証方法はサポートされていません。
 - `hadoop-env` 設定分類での AWS アクセスキーの設定
 - URI `tempdir` での AWS アクセスキーのエンコード

コネクタとそのサポートされているパラメータの使用の詳細については、次のリソースを参照してください。

- 「Amazon Redshift 管理ガイド」の「[Amazon Redshift integration for Apache Spark](#)」
- Github の [spark-redshift コミュニティリポジトリ](#)

Amazon EMR Serverless を使用した DynamoDB への接続

このチュートリアルでは、[米国地名委員会](#)で提供されるデータのサブセットを Amazon S3 バケットにアップロードし、Amazon EMR Serverless の Hive または Spark を使用してクエリできる Amazon DynamoDB テーブルにデータをコピーします。

ステップ 1: データを Amazon S3 バケットにアップロードする

Amazon S3 バケットを作成するには、「Amazon Simple Storage Service コンソールユーザーガイド」の「[バケットの作成](#)」の手順に従います。[amzn-s3-demo-bucket](#) への参照を、新しく作成されたバケットの名前に置き換えます。これで、EMR Serverless アプリケーションはジョブを実行する準備が整いました。

1. 次のコマンドを使用して、サンプルデータアーカイブ `features.zip` をダウンロードします。

```
wget https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/samples/features.zip
```

2. アーカイブから `features.txt` ファイルを抽出し、ファイル内の最初の数行にアクセスします。

```
unzip features.zip  
head features.txt
```

結果は次のようになります。

```
1535908|Big Run|Stream|WV|38.6370428|-80.8595469|794  
875609|Constable Hook|Cape|NJ|40.657881|-74.0990309|7  
1217998|Gooseberry Island|Island|RI|41.4534361|-71.3253284|10  
26603|Boone Moore Spring|Spring|AZ|34.0895692|-111.410065|3681  
1506738|Missouri Flat|Flat|WA|46.7634987|-117.0346113|2605  
1181348|Minnow Run|Stream|PA|40.0820178|-79.3800349|1558  
1288759|Hunting Creek|Stream|TN|36.343969|-83.8029682|1024  
533060|Big Charles Bayou|Bay|LA|29.6046517|-91.9828654|0  
829689|Greenwood Creek|Stream|NE|41.596086|-103.0499296|3671  
541692|Button Willow Island|Island|LA|31.9579389|-93.0648847|98
```

ここにある各行のフィールドは、一意の識別子、名前、自然の特徴量のタイプ、状態、緯度(度)、経度(度)、高さ(フィート)で示しています。

3. データを Amazon S3 にアップロードする

```
aws s3 cp features.txt s3://amzn-s3-demo-bucket/features/
```

ステップ 2: Hive テーブルを作成する

Apache Spark または Hive を使用して、Amazon S3 にアップロードされたデータを含む新しい Hive テーブルを作成します。

Spark

Spark で Hive テーブルを作成するには、次のコマンドを実行します。

```
import org.apache.spark.sql.SparkSession

val sparkSession = SparkSession.builder().enableHiveSupport().getOrCreate()

sparkSession.sql("CREATE TABLE hive_features \
  (feature_id BIGINT, \
  feature_name STRING, \
  feature_class STRING, \
  state_alpha STRING, \
  prim_lat_dec DOUBLE, \
  prim_long_dec DOUBLE, \
  elev_in_ft BIGINT) \
  ROW FORMAT DELIMITED \
  FIELDS TERMINATED BY '|' \
  LINES TERMINATED BY '\n' \
  LOCATION 's3://amzn-s3-demo-bucket/features';")
```

これで、features.txt ファイルからのデータが格納された Hive テーブルが用意できます。データがテーブルにあることを確認するには、次の例に示すように Spark SQL クエリを実行します。

```
sparkSession.sql(
  "SELECT state_alpha, COUNT(*) FROM hive_features GROUP BY state_alpha;")
```

Hive

Hive で Hive テーブルを作成するには、次のコマンドを実行します。

```
CREATE TABLE hive_features
```

```
(feature_id          BIGINT,
feature_name        STRING ,
feature_class       STRING ,
state_alpha         STRING,
prim_lat_dec        DOUBLE ,
prim_long_dec       DOUBLE ,
elev_in_ft          BIGINT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
LINES TERMINATED BY '\n'
LOCATION 's3://amzn-s3-demo-bucket/features';
```

これで、features.txt ファイルからのデータを含む Hive テーブルが作成されました。データがテーブルにあることを確認するには、次の例に示すように HiveQL クエリを実行します。

```
SELECT state_alpha, COUNT(*) FROM hive_features GROUP BY state_alpha;
```

ステップ 3: データを DynamoDB にコピーする

Spark または Hive を使用してデータを新しい DynamoDB テーブルにコピーします。

Spark

前のステップで作成した Hive テーブルから DynamoDB にデータをコピーするには、[データを DynamoDB にコピーする](#)のステップ 1~3 に従います。これにより、Features という名前の新しい DynamoDB テーブルが作成されます。その後、次の例に示すように、テキストファイルから直接データを読み取って DynamoDB テーブルにコピーできます。

```
import com.amazonaws.services.dynamodbv2.model.AttributeValue
import org.apache.hadoop.dynamodb.DynamoDBItemWritable
import org.apache.hadoop.dynamodb.read.DynamoDBInputFormat
import org.apache.hadoop.io.Text
import org.apache.hadoop.mapred.JobConf
import org.apache.spark.SparkContext

import scala.collection.JavaConverters._

object EmrServerlessDynamoDbTest {

  def main(args: Array[String]): Unit = {
```

```

jobConf.set("dynamodb.input.tableName", "Features")
jobConf.set("dynamodb.output.tableName", "Features")
jobConf.set("dynamodb.region", "region")

jobConf.set("mapred.output.format.class",
"org.apache.hadoop.dynamodb.write.DynamoDBOutputFormat")
jobConf.set("mapred.input.format.class",
"org.apache.hadoop.dynamodb.read.DynamoDBInputFormat")

val rdd = sc.textFile("s3://amzn-s3-demo-bucket/ddb-connector/")
  .map(row => {
    val line = row.split("\\|")
    val item = new DynamoDBItemWritable()

    val elevInFt = if (line.length > 6) {
      new AttributeValue().withN(line(6))
    } else {
      new AttributeValue().withNULL(true)
    }

    item.setItem(Map(
      "feature_id" -> new AttributeValue().withN(line(0)),
      "feature_name" -> new AttributeValue(line(1)),
      "feature_class" -> new AttributeValue(line(2)),
      "state_alpha" -> new AttributeValue(line(3)),
      "prim_lat_dec" -> new AttributeValue().withN(line(4)),
      "prim_long_dec" -> new AttributeValue().withN(line(5)),
      "elev_in_ft" -> elevInFt)
      .asJava)
      (new Text(""), item)
    })
  rdd.saveAsHadoopDataset(jobConf)
}

```

Hive

前のステップで作成した Hive テーブルから DynamoDB にデータをコピーするには、[データを DynamoDB にコピーする](#)の手順に従います。

ステップ 4: DynamoDB からデータをクエリする

Spark または Hive を使用して DynamoDB テーブルをクエリします。

Spark

前のステップで作成した DynamoDB テーブルからデータをクエリするには、Spark SQL または Spark MapReduce API のいずれかを使用します。

Example– Spark SQL を使用して DynamoDB テーブルをクエリする

次の Spark SQL クエリは、すべての特徴量タイプのリストをアルファベット順に返します。

```
val dataframe = sparkSession.sql("SELECT DISTINCT feature_class \
FROM ddb_features \
ORDER BY feature_class;")
```

次の Spark SQL クエリは、M で始まるすべてのレイクのリストを返します。

```
val dataframe = sparkSession.sql("SELECT feature_name, state_alpha \
FROM ddb_features \
WHERE feature_class = 'Lake' \
AND feature_name LIKE 'M%' \
ORDER BY feature_name;")
```

次の Spark SQL クエリは、1 マイルを超える特徴量を 3 つ以上持つすべての状態のリストを返します。

```
val dataframe = sparkSession.dql("SELECT state_alpha, feature_class, COUNT(*) \
FROM ddb_features \
WHERE elev_in_ft > 5280 \
GROUP by state_alpha, feature_class \
HAVING COUNT(*) >= 3 \
ORDER BY state_alpha, feature_class;")
```

Example– Spark MapReduce API を使用して DynamoDB テーブルをクエリする

次の MapReduce クエリは、すべての特徴量タイプのリストをアルファベット順に返します。

```
val df = sc.hadoopRDD(jobConf, classOf[DynamoDBInputFormat], classOf[Text],
  classOf[DynamoDBItemWritable])
  .map(pair => (pair._1, pair._2.getItem))
```

```
.map(pair => pair._2.get("feature_class").getS)
.distinct()
.sortBy(value => value)
.toDF("feature_class")
```

次の MapReduce クエリは、M で始まるすべてのレイクのリストを返します。

```
val df = sc.hadoopRDD(jobConf, classOf[DynamoDBInputFormat], classOf[Text],
  classOf[DynamoDBItemWritable])
  .map(pair => (pair._1, pair._2.getItem))
  .filter(pair => "Lake".equals(pair._2.get("feature_class").getS))
  .filter(pair => pair._2.get("feature_name").getS.startsWith("M"))
  .map(pair => (pair._2.get("feature_name").getS,
    pair._2.get("state_alpha").getS))
  .sortBy(_._1)
  .toDF("feature_name", "state_alpha")
```

次の MapReduce クエリは、1 マイルを超える特徴量を 3 つ以上持つすべての状態のリストを返します。

```
val df = sc.hadoopRDD(jobConf, classOf[DynamoDBInputFormat], classOf[Text],
  classOf[DynamoDBItemWritable])
  .map(pair => pair._2.getItem)
  .filter(pair => pair.get("elev_in_ft").getN != null)
  .filter(pair => Integer.parseInt(pair.get("elev_in_ft").getN) > 5280)
  .groupBy(pair => (pair.get("state_alpha").getS, pair.get("feature_class").getS))
  .filter(pair => pair._2.size >= 3)
  .map(pair => (pair._1._1, pair._1._2, pair._2.size))
  .sortBy(pair => (pair._1, pair._2))
  .toDF("state_alpha", "feature_class", "count")
```

Hive

前のステップで作成した DynamoDB テーブルからデータをクエリするには、[DynamoDB テーブル内のデータをクエリする](#)の手順に従います。

クロスアカウントアクセスのセットアップ

EMR Serverless のクロスアカウントアクセスを設定するには、次の手順を実行します。この例では、AccountA は Amazon EMR Serverless アプリケーションを作成したアカウントであり、AccountB は Amazon DynamoDB があるアカウントです。

1. AccountB で DynamoDB テーブルを作成します。詳細については、[ステップ 1: テーブルを作成する](#)を参照してください。
2. AccountB に、DynamoDB テーブルにアクセスできる Cross-Account-Role-B IAM ロールを作成します。
 - a. にサインイン AWS マネジメントコンソール し、<https://console.aws.amazon.com/iam/> で IAM コンソールを開きます。
 - b. [ロール] を選択し、Cross-Account-Role-B という新しいロールを作成します。IAM ロールの作成方法の詳細については、ユーザーガイドの「[IAM ロールの作成](#)」を参照してください。
 - c. クロスアカウント DynamoDB テーブルへのアクセス権限を付与する IAM ポリシーを作成します。IAM ポリシーを Cross-Account-Role-B にアタッチします。

DynamoDB テーブル CrossAccountTable へのアクセスを許可するポリシーを次に示します。

- d. Cross-Account-Role-B ロールの信頼関係を編集します。

ロールの信頼関係を設定するには、IAM コンソールで、ステップ 2: Cross-Account-Role-B で作成したロールの [信頼関係] タブを選択します。

[信頼関係の編集] を選択し、次のポリシードキュメントを追加します。このドキュメントでは、AccountA の Job-Execution-Role-A がこの Cross-Account-Role-B ロールを引き受けることを許可します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sts:AssumeRole"
      ],
      "Resource": "arn:aws:iam::123456789012:role/Job-Execution-Role-A",
      "Sid": "AllowSTSAssumerole"
    }
  ]
}
```

```
}
```

- e. - STS Assume role を持つ AccountA の Job-Execution-Role-A に、Cross-Account-Role-B を引き受けるアクセス権限を付与します。

の IAM コンソールで AWS アカウント AccountA、 を選択します Job-Execution-Role-A。次のポリシーステートメントを Job-Execution-Role-A に追加して、Cross-Account-Role-B ロールに対して AssumeRole アクションを許可します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sts:AssumeRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:role/Cross-Account-Role-B"
      ],
      "Sid": "AllowSTSAssumerole"
    }
  ]
}
```

- f. core-site 分類で値を com.amazonaws.emr.AssumeRoleAWSCredentialsProvider として dynamodb.customAWSCredentialsProvider プロパティを設定します。環境変数 ASSUME_ROLE_CREDENTIALS_ROLE_ARN に ARN 値 Cross-Account-Role-B を設定します。

3. Job-Execution-Role-A を使用して Spark ジョブまたは Hive ジョブを実行します。

考慮事項

Apache Spark または Apache Hive で DynamoDB コネクタを使用する場合、これらの動作と制限に注意してください。

Apache Spark で DynamoDB コネクタを使用する場合の考慮事項

- Spark SQL は、ストレージハンドラーオプションが付いた Hive テーブルの作成をサポートしていません。詳細については、Apache Spark ドキュメントの「[Specifying storage format for Hive tables](#)」を参照してください。
- Spark SQL は、ストレージハンドラーを使用した STORED BY オペレーションをサポートしていません。外部の Hive テーブル経由で DynamoDB テーブルとやり取りする場合は、まず Hive を使用してテーブルを作成します。
- クエリを DynamoDB クエリに変換するには、DynamoDB コネクタは述語プッシュダウンを使用します。述語プッシュダウンは、DynamoDB テーブルのパーティションキーにマッピングされた列でデータをフィルタリングします。述語プッシュダウンは、MapReduce API ではなく、Spark SQL でコネクタを使用する場合にのみ動作します。

Apache Hive で DynamoDB コネクタを使用する場合の考慮事項

マッパーの最大数の調整

- SELECT クエリを使用して DynamoDB にマッピングされる外部の Hive テーブルからデータを読み取る場合、EMR Serverless のマップタスクの数は、DynamoDB テーブルに設定された合計読み取りスループットをマップタスクあたりのスループットで割って計算されます。マップタスクあたりのデフォルトのスループットは 100 です。
- Hive ジョブは、DynamoDB 用に設定された読み取りスループットに応じて、EMR Serverless アプリケーションごとに設定されたコンテナの最大数を超えるマップタスクの数を使用できます。また、長時間実行される Hive クエリは、DynamoDB テーブルのプロビジョニングされた読み込みキャパシティをすべて消費できます。これは他のユーザーに悪影響を及ぼします。
- `dynamodb.max.map.tasks` プロパティを使用して、マップタスクの上限を設定することができます。このプロパティを使用して、タスクコンテナのサイズに基づいて、各マップタスクによって読み込まれるデータの量を調整することもできます。
- `dynamodb.max.map.tasks` プロパティは、Hive クエリレベル、または `start-job-run` コマンドの `hive-site` 分類で設定できます。この値は、1 以上にする必要があります。Hive がクエリを処理する際に実行される Hive ジョブは、DynamoDB テーブルからの読み込み時、`dynamodb.max.map.tasks` の値を超えるタスクを使用することはありません。

タスクあたりの書き込みスループットの調整

- EMR Serverless のタスクあたりの書き込みスループットは、DynamoDB テーブルに設定された合計書き込みスループットを `mapreduce.job.maps` プロパティの値で割って計算されます。Hive の場合、このプロパティのデフォルト値は 2 です。したがって、Hive ジョブの最終ステージで最初の 2 つのタスクは、すべての書き込みスループットを消費できます。これにより、同じジョブや他のジョブで他のタスクの書き込みがスロットリングされます。
- 書き込みのスロットリングを回避するには、最終ステージのタスク数またはタスクごとに割り当てる書き込みスループットに基づいて `mapreduce.job.maps` プロパティの値を設定します。EMR Serverless の `start-job-run` コマンドの `mapred-site` 分類にこのプロパティを設定します。

セキュリティ

のクラウドセキュリティが最優先事項 AWS です。お客様は AWS、セキュリティを最も重視する組織の要件を満たすように構築されたデータセンターとネットワークアーキテクチャを活用できます。

セキュリティは、AWS お客様とお客様の間の責任共有です。[責任共有モデル](#)では、これをクラウドのセキュリティおよびクラウド内のセキュリティとして説明しています。

- クラウドのセキュリティ – クラウドで AWS AWS サービスを実行するインフラストラクチャを保護する AWS 責任があります。AWS また、は、お客様が安全に使用するサービスも提供します。[「AWS」コンプライアンスプログラム](#)の一環として、サードパーティーの監査が定期的にセキュリティの有効性をテストおよび検証しています。Amazon EMR Serverless に適用されるコンプライアンスプログラムについては、[「コンプライアンスプログラムによるAWS 対象範囲内のサービス」](#)を参照してください。
- クラウドのセキュリティ – お客様の責任は、使用する AWS サービスによって決まります。また、お客様は、お客様のデータの機密性、企業の要件、および適用可能な法律および規制などの他の要因についても責任を担います。

このドキュメントは、Amazon EMR Serverless を使用する際の責任共有モデルの適用について理解するのに役立ちます。トピックでは、Amazon EMR Serverless を設定し、他の AWS のサービスを使用してセキュリティとコンプライアンスの目的を達成する方法について説明します。

トピック

- [Amazon EMR Serverless のセキュリティに関するベストプラクティス](#)
- [データ保護](#)
- [Amazon EMR Serverless での Identity and Access Management \(IAM\)](#)
- [信頼できる ID の伝播](#)
- [EMR Serverless での Lake Formation の使用](#)
- [ワーカー間の暗号化](#)
- [KMS CMK によるディスク暗号化](#)
- [EMR Serverless によるデータ保護のための Secrets Manager](#)
- [EMR Serverless での Amazon S3 Access Grants の使用](#)
- [を使用した Amazon EMR Serverless API コールのログ記録 AWS CloudTrail](#)
- [Amazon EMR Serverless のコンプライアンス検証](#)

- [Amazon EMR Serverless のレジリエンス](#)
- [Amazon EMR Serverless でのインフラストラクチャセキュリティ](#)
- [Amazon EMR Serverless での設定と脆弱性の分析](#)

Amazon EMR Serverless のセキュリティに関するベストプラクティス

Amazon EMR Serverless には、独自のセキュリティポリシーを策定および実装する際に考慮すべき、さまざまなセキュリティ機能が用意されています。以下のベストプラクティスは一般的なガイドラインであり、完全なセキュリティソリューションを説明するものではありません。これらのベストプラクティスはお客様の環境に適切ではないか、十分ではない場合があるため、これらは指示ではなく、有用な考慮事項と見なしてください。

最小特権の原則を適用する

EMR Serverless は、実行ロールなどの IAM ロールを使用して、アプリケーションに対してきめ細かいアクセスポリシーを提供します。実行ロールには、アプリケーションのカバーやログ送信先へのアクセスなど、ジョブに必要な最小限の特権セットのみを付与することを提案します。また、定期的に、およびアプリケーションコードに変更があったときに、ジョブの許可を監査することをお勧めします。

信頼できないアプリケーションコードを分離する

EMR Serverless では、さまざまな EMR Serverless アプリケーションに属するジョブ間で完全なネットワーク分離を実現します。ジョブレベルの分離が必要な場合は、ジョブを別々の EMR Serverless アプリケーションに分離することを検討してください。

ロールベースのアクセスコントロール (RBAC) の許可

管理者は、EMR Serverless アプリケーションに対するロールベースのアクセスコントロール (RBAC) の許可を厳密に制御する必要があります。

データ保護

責任 AWS [共有モデル](#)は、Amazon EMR Serverless でのデータ保護に適用されます。このモデルで説明されているように、AWS はすべての AWS クラウドを実行するグローバルインフラストラク

チャを保護する責任があります。ユーザーは、このインフラストラクチャでホストされるコンテンツに対する管理を維持する責任があります。このコンテンツには、使用する AWS サービスのセキュリティ設定および管理タスクが含まれます。データプライバシーの詳細については、「[データプライバシーよくある質問](#)」を参照してください。欧州でのデータ保護の詳細については、AWS セキュリティブログの [責任 AWS 共有モデルと GDPR](#) ブログ記事を参照してください。

データ保護の目的で、AWS アカウントの認証情報を保護し、AWS Identity and Access Management (IAM) を使用して個々のアカウントを設定することをお勧めします。この方法により、それぞれのジョブを遂行するために必要なアクセス許可のみを各ユーザーに付与できます。また、次の方法でデータを保護することをお勧めします。

- 各アカウントで多要素認証 (MFA) を使用します。
- SSL/TLS を使用して AWS リソースと通信します。TLS 1.2 以降をお勧めします。
- で API とユーザーアクティビティのログ記録を設定します AWS CloudTrail。
- AWS 暗号化ソリューションと、サービス内のすべての AWS デフォルトのセキュリティコントロールを使用します。
- Amazon Macie などのアドバンスドマネージドセキュリティサービスを使用します。これは、Amazon S3 に保存されている個人データの検出と保護を支援します。
- Amazon EMR Serverless の暗号化オプションを使用して保管中および転送中のデータを暗号化します。
- コマンドラインインターフェイスまたは API AWS を介して にアクセスするときに FIPS 140-2 検証済みの暗号化モジュールが必要な場合は、FIPS エンドポイントを使用します。利用可能な FIPS エンドポイントの詳細については、「[Federal Information Processing Standard \(FIPS\) 140-2](#)」を参照してください。

顧客のアカウント番号などの機密の識別情報は、[名前] フィールドなどの自由形式のフィールドに配置しないことを強く提案します。これは、コンソール、API、または SDK を使用して Amazon EMR Serverless AWS CLI または他の AWS のサービスを使用する場合も同様です。AWS SDKs Amazon EMR Serverless や他のサービスに入力したすべてのデータは、診断ログに取り込まれる可能性があります。外部サーバーへの URL を指定するときは、そのサーバーへのリクエストを検証するための認証情報を URL に含めないでください。

保管中の暗号化

データの暗号化は、承認されていないユーザーがクラスターおよび関連するデータストレージシステムのデータを読み取れないようにするのに役立ちます。このデータには、保管中のデータと呼ばれ

る、永続的なメディアに保存されているデータや、転送中のデータと呼ばれる、ネットワークを介した転送の間に傍受される可能性のあるデータが含まれます。

データの暗号化には、キーと証明書が必要です。が管理するキー、Amazon S3 が管理するキー AWS Key Management Service、提供するカスタムプロバイダーのキーと証明書など、いくつかのオプションから選択できます。キープロバイダー AWS KMS としてを使用する場合、暗号化キーのストレージと使用には料金が適用されます。詳細については、[AWS KMS の料金](#)を参照してください。

暗号化オプションを指定する前に、使用するキーと証明書の管理システムを決定します。次に、暗号化設定の一部として指定するカスタムプロバイダーのキーと証明書を作成します。

Amazon S3 内に保管中の EMRFS データの暗号化

各 EMR Serverless アプリケーションは、EMRFS (EMR ファイルシステム) を含む特定のリリースバージョンを使用します。Amazon S3 暗号化は、Amazon S3 への読み取りおよび書き込みが行われる EMR ファイルシステム (EMRFS) オブジェクトで使用できます。保管中のデータの暗号化を有効にすると、デフォルトの暗号化モードとして、Amazon S3 サーバー側の暗号化 (SSE) またはクライアント側の暗号化 (CSE) を指定できます。オプションで、[Per バケットの暗号化オーバーライド] を使用して、バケットごとに異なる暗号化方法を指定できます。Amazon S3 の暗号化が有効かどうかにかかわらず、Transport Layer Security (TLS) は、EMR クラスターノードと Amazon S3 の間で転送される EMRFS オブジェクトを暗号化します。カスタマーマネージドキーで Amazon S3 CSE を使用する場合、EMR Serverless アプリケーションでジョブを実行するために使用される実行ロールは、キーにアクセスする必要があります。Amazon S3 の暗号化の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「[暗号化によるデータの保護](#)」を参照してください。

Note

を使用する場合 AWS KMS、暗号化キーのストレージと使用には料金が適用されます。詳細については、[AWS KMS の料金](#)を参照してください。

Amazon S3 のサーバー側の暗号化

すべての Amazon S3 バケットにはデフォルトで暗号化が設定されており、S3 バケットにアップロードされたすべての新しいオブジェクトは保管時に自動的に暗号化されます。Amazon S3 は、ディスクにデータを書き込むときにオブジェクトレベルでデータを暗号化し、アクセス時にデータを復号します。SSE の詳細については、「Amazon Simple Storage Service デベロッパーガイド」の「[サーバー側の暗号化によるデータの保護](#)」を参照してください。

Amazon EMR Serverless で SSE を指定するときに、次の 2 つの異なるキー管理システムから選択できます。

- SSE-S3 — Amazon S3 がキーを管理します。EMR Serverless では、追加のセットアップは必要ありません。
- SSE-KMS - を使用して AWS KMS key、EMR Serverless に適したポリシーをセットアップします。EMR Serverless では、追加のセットアップは必要ありません。

Amazon S3 に書き込むデータの AWS KMS 暗号化を使用するには、StartJobRun API を使用するとき、2 つのオプションがあります。Amazon S3 Amazon S3 に書き込むすべてのデータの暗号化を有効にすることも、特定のバケットに書き込むデータの暗号化を有効にすることも可能です。StartJobRun API に関する詳細については、「[EMR Serverless API Reference](#)」を参照してください。

Amazon S3 に書き込むすべてのデータの AWS KMS 暗号化を有効にするには、StartJobRun API を呼び出すときに次のコマンドを使用します。Amazon S3

```
--conf spark.hadoop.fs.s3.enableServerSideEncryption=true  
--conf spark.hadoop.fs.s3.serverSideEncryption.kms.keyId=<kms_id>
```

特定のバケットに書き込むデータの AWS KMS 暗号化を有効にするには、StartJobRun API を呼び出すときに次のコマンドを使用します。

```
--conf spark.hadoop.fs.s3.bucket.<amzn-s3-demo-bucket1>.enableServerSideEncryption=true  
--conf spark.hadoop.fs.s3.bucket.<amzn-s3-demo-bucket1>.serverSideEncryption.kms.keyId=<kms-id>
```

SSE とお客様が用意したキーとの組み合わせ (SSE-C) は、EMR Serverless では使用できません。

Amazon S3 クライアント側の暗号化

Amazon S3 のクライアント側の暗号化を使用すると、Amazon S3 の暗号化と復号は、すべての Amazon EMR リリースで利用可能な EMRFS クライアントで行われます。オブジェクトは Amazon S3 にアップロードされる前に暗号化され、ダウンロード後に復号化されます。指定するプロバイダーが、クライアントが使用する暗号化キーを提供します。クライアントは、AWS KMS によって提供されるキー (CSE-KMS) か、クライアント側のルートキーを提供するカスタム Java クラス (CSE-C) を使用できます。CSE-KMS と CSE-C では、指定するプロバイダーと、復号化または暗号化されるオブジェクトのメタデータに応じて、暗号化の仕様が少し異なります。カスタマーマネー

ジドキーで Amazon S3 CSE を使用する場合、EMR Serverless アプリケーションでジョブを実行するために使用される実行ロールは、キーにアクセスできる必要があります。追加の KMS 料金が適用される場合があります。これらの差異に関する詳細は、「Amazon Simple Storage Service デベロッパーガイド」の「[Protecting data using client-side encryption](#)」を参照してください。

ローカルディスク暗号化

エフェメラルストレージに保存されているデータは、業界標準の AES-256 暗号化アルゴリズムを使用して、サービス所有のキーで暗号化されます。

キー管理

KMS キーを自動的にローテーションするように KMS を設定できます。これにより、古いキーを無期限に保存しながら、年に一度、キーをローテーションして、データを復号することができます。詳細については、「[Rotating customer-managed keys](#)」を参照してください。

転送中の暗号化

Amazon EMR Serverless では、以下のアプリケーション固有の暗号化機能を使用できます。

- Spark
 - デフォルトでは、Spark ドライバーとエグゼキューター間の通信は認証された内部通信です。ドライバとエグゼキューター間の RPC 通信は暗号化されます。
- [Hive]
 - AWS Glue メタストアと EMR Serverless アプリケーション間の通信は TLS を介して行われます。

Amazon S3 バケット IAM ポリシーで [aws:SecureTransport 条件](#)を使用して、HTTPS (TLS) を介した暗号化接続のみを許可してください。

Amazon EMR Serverless での Identity and Access Management (IAM)

AWS Identity and Access Management (IAM) は、管理者が AWS リソースへのアクセスを安全に制御 AWS のサービスするのに役立つです。IAM 管理者は、誰を認証 (サインイン) し、誰に Amazon EMR Serverless リソースの使用を承認する (アクセス許可を付与する) かを制御します。IAM は、追加料金なしで使用できる AWS のサービスです。

トピック

- [オーディエンス](#)
- [アイデンティティを使用した認証](#)
- [ポリシーを使用したアクセスの管理](#)
- [EMR Serverless が IAM と連携する仕組み](#)
- [EMR Serverless のサービスリンクロールの使用](#)
- [Amazon EMR Serverless のジョブランタイムロール](#)
- [EMR Serverless のユーザーアクセスポリシーの例](#)
- [タグベースのアクセスコントロールのポリシー](#)
- [EMR Serverless でのアイデンティティベースのポリシー例](#)
- [AWS マネージドポリシーに対する Amazon EMR Serverless の更新](#)
- [Amazon EMR Serverless の ID とアクセスのトラブルシューティング](#)

オーディエンス

AWS Identity and Access Management (IAM) の使用方法は、ロールによって異なります。

- サービスユーザー - 機能にアクセスできない場合は、管理者にアクセス許可をリクエストします (「[Amazon EMR Serverless の ID とアクセスのトラブルシューティング](#)」を参照)。
- サービス管理者 - ユーザーアクセスを決定し、アクセス許可リクエストを送信します (「[Amazon EMR Serverless での Identity and Access Management \(IAM\)](#)」を参照)
- IAM 管理者 - アクセスを管理するためのポリシーを作成します (「[EMR Serverless でのアイデンティティベースのポリシーの例](#)」を参照)

アイデンティティを使用した認証

認証とは、ID 認証情報 AWS を使用して にサインインする方法です。、IAM ユーザー AWS アカウントのルートユーザー、または IAM ロールを引き受けることで認証される必要があります。

AWS IAM アイデンティティセンター (IAM Identity Center)、シングルサインオン認証、Google/Facebook 認証情報などの ID ソースからの認証情報を使用して、フェデレーティッド ID としてサインインできます。サインインの詳細については、「AWS サインイン ユーザーガイド」の「[AWS アカウントにサインインする方法](#)」を参照してください。

プログラムによるアクセスの場合、は SDK と CLI AWS を提供してリクエストを暗号化して署名します。詳細については、「IAM ユーザーガイド」の「[API リクエストに対するAWS 署名バージョン 4](#)」を参照してください。

AWS アカウント ルートユーザー

を作成するときは AWS アカウント、すべての AWS のサービス および リソースへの完全なアクセス権を持つ AWS アカウント root ユーザーと呼ばれる 1 つのサインインアイデンティティから始めます。日常的なタスクには、ルートユーザーを使用しないことを強くお勧めします。ルートユーザー認証情報を必要とするタスクについては、「IAM ユーザーガイド」の「[ルートユーザー認証情報が必要なタスク](#)」を参照してください。

フェデレーテッドアイデンティティ

ベストプラクティスとして、人間のユーザーが一時的な認証情報 AWS のサービス を使用して にアクセスするには、ID プロバイダーとのフェデレーションを使用する必要があります。

フェデレーテッド ID は、エンタープライズディレクトリ、ウェブ ID プロバイダー、または ID ソースからの認証情報 AWS のサービス を使用して Directory Service にアクセスするユーザーです。フェデレーテッドアイデンティティは、一時的な認証情報を提供するロールを引き受けます。

アクセスを一元管理する場合は、AWS IAM アイデンティティセンターをお勧めします。詳細については、「AWS IAM アイデンティティセンター ユーザーガイド」の「[IAM アイデンティティセンターとは](#)」を参照してください。

IAM ユーザーとグループ

[IAM ユーザー](#)は、特定の個人やアプリケーションに対する特定のアクセス許可を持つアイデンティティです。長期認証情報を持つ IAM ユーザーの代わりに一時的な認証情報を使用することをお勧めします。詳細については、IAM ユーザーガイドの「[ID プロバイダーとのフェデレーションを使用して にアクセスすることを人間のユーザーに要求する AWS](#)」を参照してください。

[IAM グループ](#)は、IAM ユーザーの集合を指定し、大量のユーザーに対するアクセス許可の管理を容易にします。詳細については、「IAM ユーザーガイド」の「[IAM ユーザーに関するユースケース](#)」を参照してください。

IAM ロール

[IAM ロール](#)は、特定のアクセス許可を持つアイデンティティであり、一時的な認証情報を提供します。ユーザーから [IAM ロール \(コンソール\)](#) に切り替えるか、または [API オペレーション](#)を呼び出すこ

とで、[ロール](#)を引き受けることができます。AWS CLI AWS 詳細については、「IAM ユーザーガイド」の「[ロールを引き受けるための各種方法](#)」を参照してください。

IAM ロールは、フェデレーションユーザーアクセス、一時的な IAM ユーザーのアクセス許可、クロスアカウントアクセス、クロスサービスアクセス、および Amazon EC2 で実行するアプリケーションに役立ちます。詳細については、IAM ユーザーガイドの [IAM でのクロスアカウントリソースアクセス](#) を参照してください。

ポリシーを使用したアクセスの管理

でアクセスを制御する AWS には、ポリシーを作成し、ID AWS またはリソースにアタッチします。ポリシーは、アイデンティティまたはリソースに関連付けられたときにアクセス許可を定義します。は、プリンシパルがリクエストを行うときにこれらのポリシー AWS を評価します。ほとんどのポリシーは JSON ドキュメント AWS として保存されます。JSON ポリシードキュメントの詳細については、「IAM ユーザーガイド」の「[JSON ポリシー概要](#)」を参照してください。

管理者は、ポリシーを使用して、どのプリンシパルがどのリソースに対して、どのような条件でアクションを実行できるかを定義することで、誰が何にアクセスできるかを指定します。

デフォルトでは、ユーザーやロールにアクセス許可はありません。IAM 管理者は IAM ポリシーを作成してロールに追加し、このロールをユーザーが引き受けられるようにします。IAM ポリシーは、オペレーションの実行方法を問わず、アクセス許可を定義します。

アイデンティティベースのポリシー

アイデンティティベースのポリシーは、アイデンティティ (ユーザー、グループ、またはロール) にアタッチできる JSON アクセス許可ポリシードキュメントです。これらのポリシーは、アイデンティティがどのリソースに対してどのような条件下でどのようなアクションを実行できるかを制御します。アイデンティティベースポリシーの作成方法については、IAM ユーザーガイドの [カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する](#) を参照してください。

アイデンティティベースのポリシーは、インラインポリシー (単一の ID に直接埋め込む) または管理ポリシー (複数の ID にアタッチされたスタンドアロンポリシー) にすることができます。管理ポリシーとインラインポリシーのいずれかを選択する方法については、「IAM ユーザーガイド」の「[管理ポリシーとインラインポリシーのいずれかを選択する](#)」を参照してください。

リソースベースのポリシー

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。例としては、IAM ロール信頼ポリシーや Amazon S3 バケットポリシーなどがあります。リソースベースのポ

リシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスを制御できます。リソースベースのポリシーでは、[プリンシパルを指定する](#)必要があります。

リソースベースのポリシーは、そのサービス内にあるインラインポリシーです。リソースベースのポリシーでは、IAM の AWS マネージドポリシーを使用できません。

その他のポリシータイプ

AWS は、より一般的なポリシータイプによって付与されるアクセス許可の最大数を設定できる追加のポリシータイプをサポートしています。

- アクセス許可の境界 – アイデンティティベースのポリシーで IAM エンティティに付与することのできるアクセス許可の数の上限を設定します。詳細については、「IAM ユーザーガイド」の「[IAM エンティティのアクセス許可境界](#)」を参照してください。
- サービスコントロールポリシー (SCP) - AWS Organizations内の組織または組織単位の最大のアクセス許可を指定します。詳細については、「AWS Organizations ユーザーガイド」の「[サービスコントロールポリシー](#)」を参照してください。
- リソースコントロールポリシー (RCP) – は、アカウント内のリソースで利用できる最大数のアクセス許可を定義します。詳細については、「AWS Organizations ユーザーガイド」の「[リソースコントロールポリシー \(RCP\)](#)」を参照してください。
- セッションポリシー – ロールまたはフェデレーションユーザーの一時セッションを作成する際にパラメータとして渡される高度なポリシーです。詳細については、「IAM ユーザーガイド」の「[セッションポリシー](#)」を参照してください。

複数のポリシータイプ

1 つのリクエストに複数のタイプのポリシーが適用されると、結果として作成されるアクセス許可を理解するのがさらに難しくなります。が複数のポリシータイプが関与する場合にリクエストを許可するかどうか AWS を決定する方法については、「IAM ユーザーガイド」の「[ポリシー評価ロジック](#)」を参照してください。

EMR Serverless が IAM と連携する仕組み

IAM を使用して Amazon EMR Serverless へのアクセスを管理する前に、Amazon EMR Serverless で使用できる IAM 機能について理解しておく必要があります。

EMR Serverless で使用する IAM の機能

IAM 機能	Amazon EMR Serverless のサポート
アイデンティティベースのポリシー	あり
リソースベースのポリシー	なし
ポリシーアクション	あり
ポリシーリソース	あり
ポリシー条件キー	いいえ
ACL	なし
ABAC (ポリシー内のタグ)	あり
一時的な認証情報	あり
プリンシパルアクセス権限	あり
サービスロール	いいえ
サービスリンクロール	はい

EMR Serverless およびその他の AWS のサービスがほとんどの IAM 機能と連携する方法の概要を把握するには、「IAM ユーザーガイド」の[AWS「IAM と連携する のサービス」](#)を参照してください。

EMR Serverless でのアイデンティティベースのポリシー

アイデンティティベースのポリシーのサポート: あり

アイデンティティベースポリシーは、IAM ユーザー、ユーザーグループ、ロールなど、アイデンティティにアタッチできる JSON 許可ポリシードキュメントです。これらのポリシーは、ユーザーとロールが実行できるアクション、リソース、および条件をコントロールします。アイデンティティベースポリシーの作成方法については、「IAM ユーザーガイド」の[「カスタマー管理ポリシーでカスタム IAM アクセス許可を定義する」](#)を参照してください。

IAM アイデンティティベースのポリシーでは、許可または拒否するアクションとリソース、およびアクションを許可または拒否する条件を指定できます。JSON ポリシーで使用できるすべての要素に

ついて学ぶには、「IAM ユーザーガイド」の「[IAM JSON ポリシーの要素のリファレンス](#)」を参照してください。

EMR Serverless でのアイデンティティベースのポリシーの例

Amazon EMR Serverless のアイデンティティベースポリシーの例にアクセスするには、「[EMR Serverless でのアイデンティティベースのポリシー例](#)」を参照してください。

EMR Serverless 内のリソースベースのポリシー

リソースベースのポリシーのサポート: なし

リソースベースのポリシーは、リソースに添付する JSON ポリシードキュメントです。リソースベースのポリシーには例として、IAM ロールの信頼ポリシーや Amazon S3 バケットポリシーがあげられます。リソースベースのポリシーをサポートするサービスでは、サービス管理者はポリシーを使用して特定のリソースへのアクセスをコントロールできます。ポリシーがアタッチされているリソースの場合、指定されたプリンシパルがそのリソースに対して実行できるアクションと条件は、ポリシーによって定義されます。リソースベースのポリシーで、[プリンシパルを指定する](#)必要があります。プリンシパルには、アカウント、ユーザー、ロール、フェデレーティッドユーザー、またはを含めることができます AWS のサービス。

クロスアカウントアクセスを有効にするには、全体のアカウント、または別のアカウントの IAM エンティティを、リソースベースのポリシーのプリンシパルとして指定します。詳細については、IAM ユーザーガイドの[IAM でのクロスアカウントリソースアクセス](#)を参照してください。

EMR Serverless のポリシーアクション

ポリシーアクションのサポート: あり

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

JSON ポリシーの Action 要素にはポリシー内のアクセスを許可または拒否するために使用できるアクションが記述されます。このアクションは関連付けられたオペレーションを実行するためのアクセス許可を付与するポリシーで使用されます。

EMR Serverless アクションのリストについては、「Service Authorization Reference」の「[Actions, resources, and condition keys for Amazon EMR Serverless](#)」を参照してください。

EMR Serverless のポリシーアクションでは、アクションの前に次のプレフィックスを使用します。

```
emr-serverless
```

単一のステートメントで複数のアクションを指定するには、アクションをカンマで区切ります。

```
"Action": [  
    "emr-serverless:action1",  
    "emr-serverless:action2"  
]
```

Amazon EMR Serverless のアイデンティティベースポリシーの例にアクセスするには、「[EMR Serverless でのアイデンティティベースのポリシー例](#)」を参照してください。

EMR Serverless のポリシーリソース

ポリシーリソースのサポート: あり

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

Resource JSON ポリシー要素はアクションが適用されるオブジェクトを指定します。ベストプラクティスとして、[Amazon リソースネーム \(ARN\)](#) を使用してリソースを指定します。リソースレベルのアクセス許可をサポートしないアクションの場合は、ステートメントがすべてのリソースに適用されることを示すために、ワイルドカード (*) を使用します。

```
"Resource": "*"
```

Amazon EMR Serverless リソースのタイプとその ARN のリストについては、「サービス認可リファレンス」の「[Resources Defined by Amazon EMR Serverless](#)」を参照してください。各リソースの ARN を指定するアクションについては、「[Actions, resources, and condition keys for Amazon EMR Serverless](#)」を参照してください。

Amazon EMR Serverless のアイデンティティベースポリシーの例にアクセスするには、「[EMR Serverless でのアイデンティティベースのポリシー例](#)」を参照してください。

EMR Serverless のポリシー条件キー

ポリシー条件キーのサポート

サービス固有のポリシー条件キーのサポート	いいえ

管理者は JSON AWS ポリシーを使用して、誰が何にアクセスできるかを指定できます。つまり、どのプリンシパルがどのリソースに対してどのような条件下でアクションを実行できるかということです。

Condition 要素は、定義された基準に基づいてステートメントが実行される時期を指定します。イコールや未満などの[条件演算子](#)を使用して条件式を作成して、ポリシーの条件とリクエスト内の値を一致させることができます。すべての AWS グローバル条件キーを確認するには、「IAM ユーザーガイド」の[AWS「グローバル条件コンテキストキー」](#)を参照してください。

Amazon EMR Serverless の条件キーのリスト、および条件キーを使用できるアクションとリソースについては、「サービス認可リファレンス」の「[Actions, resources, and condition keys for Amazon EMR Serverless](#)」を参照してください。

すべての Amazon EC2 アクションは aws:RequestedRegion および ec2:Region 条件キーをサポートします。詳細については、「[Example: Restricting access to a specific region](#)」を参照してください。

EMR Serverless のアクセスコントロールリスト (ACL)

ACL のサポート: なし

アクセスコントロールリスト (ACL) は、どのプリンシパル (アカウントメンバー、ユーザー、またはロール) がリソースにアクセスするためのアクセス許可を持つかを制御します。ACL はリソースベースのポリシーに似ていますが、JSON ポリシードキュメント形式は使用しません。

EMR Serverless での属性ベースのアクセスコントロール (ABAC)

属性ベースのアクセス制御 (ABAC) のサポート

ABAC のサポート (ポリシー内のタグ)	はい

属性ベースのアクセス制御 (ABAC) は、タグと呼ばれる属性に基づいてアクセス許可を定義する認可戦略です。IAM エンティティと AWS リソースにタグをアタッチし、プリンシパルのタグがリソースのタグと一致するときにオペレーションを許可するように ABAC ポリシーを設計できます。

タグに基づいてアクセスを管理するには、`aws:ResourceTag/key-name`、`aws:RequestTag/key-name`、または `aws:TagKeys` の条件キーを使用して、ポリシーの [条件要素](#) でタグ情報を提供します。

サービスがすべてのリソースタイプに対して 3 つの条件キーすべてをサポートする場合、そのサービスの値は `あり` です。サービスが一部のリソースタイプに対してのみ 3 つの条件キーのすべてをサポートする場合、値は「部分的」になります。

ABAC の詳細については、「IAM ユーザーガイド」の「[ABAC 認可でアクセス許可を定義する](#)」を参照してください。ABAC をセットアップする手順を説明するチュートリアルについては、「IAM ユーザーガイド」の「[属性ベースのアクセスコントロール \(ABAC\) を使用する](#)」を参照してください。

EMR Serverless での一時的な認証情報の使用

一時的な認証情報のサポート: `あり`

一時的な認証情報は、AWS リソースへの短期的なアクセスを提供し、フェデレーションまたは切り替えロールを使用する場合に自動的に作成されます。長期的なアクセスキーを使用する代わりに、一時的な認証情報を動的に生成 AWS することをお勧めします。詳細については、「IAM ユーザーガイド」の「[IAM の一時的な認証情報](#)」および「[AWS のサービス と IAM との連携](#)」を参照してください。

EMR Serverless のクロスサービスプリンシパルのアクセス許可

転送アクセスセッション (FAS) のサポート: `あり`

転送アクセスセッション (FAS) は、 を呼び出すプリンシパルのアクセス許可と AWS のサービス、ダウストリームサービス AWS のサービス へのリクエストをリクエストする を使用します。FAS リクエストを行う際のポリシーの詳細については、「[転送アクセスセッション](#)」を参照してください。

EMR Serverless のサービスロール

サービスロールのサポート	いいえ

EMR Serverless のサービスにリンクされたロール

サービスリンクロールのサポート	Yes

サービスにリンクされたロールの作成または管理の詳細については、「[IAM と提携するAWS のサービス](#)」を参照してください。表の「サービスリンクロール」列に Yes と記載されたサービスを見つめます。サービスにリンクされたロールに関するドキュメントをサービスにアクセスするには、[はい] リンクを選択します。

EMR Serverless のサービスリンクロールの使用

Amazon EMR Serverless は AWS Identity and Access Management (IAM) [サービスにリンクされたロール](#)を使用します。サービスリンクロールは、EMR Serverless に直接リンクされた一意のタイプの IAM ロールです。サービスにリンクされたロールは EMR Serverless によって事前定義されており、サービスがユーザーに代わって他の AWS サービスを呼び出すために必要なすべてのアクセス許可が含まれています。

サービスリンクロールを使用すると、必要なアクセス権限を手動で追加する必要がなくなるため、EMR Serverless の設定が簡単になります。EMR Serverless は、サービスリンクロールのアクセス権限を定義します。特に定義されている場合を除き、EMR Serverless のみがそのロールを引き受けることができます。定義される許可は信頼ポリシーと許可ポリシーに含まれており、その許可ポリシーを他の IAM エンティティにアタッチすることはできません。

サービスリンクロールを削除するには、最初に関連リソースを削除する必要があります。これにより、不注意でリソースにアクセスするアクセス許可の削除が防止され、EMR Serverless リソースは保護されます。

サービスにリンクされたロールをサポートする他のサービスについては、「[IAM と連動するAWS のサービス](#)」を参照し、[サービスにリンクされたロール] の列内で [はい] と表記されたサービスを確認してください。サービスリンクロールに関するドキュメントをサービスにアクセスするには、リンクで [はい] を選択します。

EMR Serverless のサービスリンクロールのアクセス許可

EMR Serverless は、AWSServiceRoleForAmazonEMRServerless という名前のサービスにリンクされたロールを使用して、ユーザーに代わって AWS APIsを呼び出すことを可能にします。

AWSServiceRoleForAmazonEMRServerless サービスリンクロールは、次のサービスを信頼してロールを引き受けます。

- ops.emr-serverless.amazonaws.com

AmazonEMRServerlessServiceRolePolicy という名前のロール許可ポリシーによって、EMR Serverless が次のアクションを指定されたリソースで完了できるようになります。

Note

マネージドポリシーの内容は変わるため、ここに示すポリシーは古くなっている可能性があります。AWS マネジメントコンソールで、最新のポリシー [AmazonEMRServerlessServiceRolePolicy](#) を確認してください。

- アクション: `ec2:CreateNetworkInterface`
- アクション: `ec2>DeleteNetworkInterface`
- アクション: `ec2:DescribeNetworkInterfaces`
- アクション: `ec2:DescribeSecurityGroups`
- アクション: `ec2:DescribeSubnets`
- アクション: `ec2:DescribeVpcs`
- アクション: `ec2:DescribeDhcpOptions`
- アクション: `ec2:DescribeRouteTables`
- アクション: `cloudwatch:PutMetricData`

以下は AmazonEMRServerlessServiceRolePolicy ポリシー全体です。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EC2PolicyStatement",
      "Effect": "Allow",
      "Action": [
        "ec2:CreateNetworkInterface",
        "ec2>DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:DescribeDhcpOptions",
        "ec2:DescribeRouteTables"
      ]
    }
  ],
```

```

    "Resource": [
      "*"
    ],
  },
  {
    "Sid": "CloudWatchPolicyStatement",
    "Effect": "Allow",
    "Action": [
      "cloudwatch:PutMetricData"
    ],
    "Resource": [
      "*"
    ],
    "Condition": {
      "StringEquals": {
        "cloudwatch:namespace": [
          "AWS/EMRServerless",
          "AWS/Usage"
        ]
      }
    }
  }
]
}

```

EMR Serverless プリンシパルがこのロールを引き受けることを許可するには、このロールに以下の信頼ポリシーをアタッチします。

JSON

```

{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sts:AssumeRole"
      ],
      "Resource": "arn:aws:iam::123456789012:role/aws-service-role/emr-serverless.amazonaws.com/AWSServiceRoleForEMRServerless",
      "Sid": "AllowSTSAssumerole"
    }
  ]
}

```

```
}  
]  
}
```

サービスリンクロールの作成、編集、削除を IAM エンティティ (ユーザー、グループ、ロールなど) に許可するにはアクセス許可を設定する必要があります。詳細については「IAM ユーザーガイド」の「[サービスにリンクされたロールの作成](#)」を参照してください。

EMR Serverless のサービスリンクロールの作成

サービスリンクロールを手動で作成する必要はありません。(EMR Studio AWS マネジメントコンソール を使用して) AWS CLI、または API で新しい EMR Serverless アプリケーションを作成すると AWS、EMR Serverless によってサービスにリンクされたロールが作成されます。サービスリンクロールの作成、編集、削除を IAM エンティティ (ユーザー、グループ、ロールなど) に許可するにはアクセス許可を設定する必要があります。

IAM を使用して `AWSServiceRoleForAmazonEMRServerless` サービスリンクロールを作成するにはサービスリンクロールを作成する必要がある IAM エンティティのアクセス権限ポリシーに、次のステートメントを追加します。

```
{  
  "Effect": "Allow",  
  "Action": [  
    "iam:CreateServiceLinkedRole"  
  ],  
  "Resource": "arn:aws:iam::*:role/aws-service-role/ops.emr-serverless.amazonaws.com/  
AWSServiceRoleForAmazonEMRServerless*",  
  "Condition": {"StringLike": {"iam:AWSServiceName": "ops.emr-  
serverless.amazonaws.com"}}  
}
```

このサービスリンクロールを削除した後で再度作成する必要がある場合は同じ方法でアカウントにロールを再作成します。新しい EMR Serverless アプリケーションを作成すると、EMR Serverless はサービスリンクロールを再度作成します。

EMR Serverless のユースケースでサービスリンクロールを作成する場合も、IAM コンソールを使用できます。AWS CLI または AWS API で、サービス名を使用して `ops.emr-serverless.amazonaws.com` サービスにリンクされたロールを作成します。詳細については、

「IAM ユーザーガイド」の「[サービスにリンクされたロールの作成](#)」を参照してください。このサービスリンクロールを削除しても、同じ方法でロールを再作成できます。

EMR Serverless のサービスリンクロールの編集

EMR Serverless では、AWSServiceRoleForAmazonEMRServerless サービスリンクロールを編集できません。さまざまなエンティティがロールを参照している可能性があるためです。EMR Serverless サービスにリンクされたロールが使用する AWS 所有の IAM ポリシーは、EMR Serverless に必要なすべてのアクセス許可が含まれているため、編集できません。ただし、IAM を使用してロールの説明を編集することはできます。

AWSServiceRoleForAmazonEMRServerless サービスリンクロールの説明を IAM を使用して編集するには

サービスにリンクされたロールの説明を編集する必要がある IAM エンティティの許可ポリシーに次のステートメントを追加します。

```
{
  "Effect": "Allow",
  "Action": [
    "iam: UpdateRoleDescription"
  ],
  "Resource": "arn:aws:iam::*:role/aws-service-role/ops.emr-serverless.amazonaws.com/AWSServiceRoleForAmazonEMRServerless*",
  "Condition": {"StringLike": {"iam:AWSServiceName": "ops.emr-serverless.amazonaws.com"}}
}
```

詳細については「IAM ユーザーガイド」の「[Editing a service-linked role](#)」を参照してください。

EMR Serverless のサービスリンクロールの削除

サービスリンクロールを必要とする機能やサービスが不要になった場合は、ロールを削除することを提案します。これは、モニタリングや保守が積極的に行われていない未使用のエンティティを排除するためです。ただし、サービスリンクロールを削除する前に、すべてのリージョンのすべての EMR Serverless アプリケーションを削除します。

Note

ロールに関連するリソースを削除しようとする際に、EMR Serverless のサービスでロールが使用されていると、削除に失敗することがあります。失敗した場合は数分待ってから操作を再試行してください。

IAM を使用して `AWSServiceRoleForAmazonEMRServerless` サービスリンクロールを削除するには、サービスリンクロールを削除する必要がある IAM エンティティのアクセス権限ポリシーに、次のステートメントを追加します。

```
{
  "Effect": "Allow",
  "Action": [
    "iam:DeleteServiceLinkedRole",
    "iam:GetServiceLinkedRoleDeletionStatus"
  ],
  "Resource": "arn:aws:iam::*:role/aws-service-role/ops.emr-serverless.amazonaws.com/AWSServiceRoleForAmazonEMRServerless*",
  "Condition": {"StringLike": {"iam:AWSServiceName": "ops.emr-serverless.amazonaws.com"}}
}
```

サービスリンクロールを IAM で手動削除するには

IAM コンソール、AWS CLI、または AWS API を使用し

て、`AWSServiceRoleForAmazonEMRServerless` サービスにリンクされたロールを削除します。詳細については、「IAM ユーザーガイド」の「[Deleting a service-linked role](#)」を参照してください。

EMR Serverless のサービスリンクロールをサポートするリージョン

EMR Serverless は、このサービスが利用可できるすべてのリージョンで、サービスリンクロールの使用をサポートします。詳細については、「[AWS サービスエンドポイント](#)」を参照してください。

Amazon EMR Serverless のジョブランタイムロール

EMR Serverless ジョブの実行がユーザーに代わって他のサービスを呼び出す際に引き受けられることができる、IAM ロールのアクセス許可を指定できます。これには、データソース、ターゲット、Amazon Redshift クラスターや DynamoDB テーブルなどの他の AWS リソースに対する

Amazon S3 へのアクセスが含まれます。ロールの作成方法の詳細については、「[ジョブランタイムロールを作成する](#)」を参照してください。

ランタイムポリシーの例

ジョブランタイムロールには、次のようなランタイムポリシーをアタッチできます。次のジョブランタイムポリシーでは、以下を許可します。

- EMR サンプルを使用した Amazon S3 バケットへの読み取りアクセス。
- S3 バケットへのフルアクセス。
- AWS Glue データカタログへの作成および読み取りアクセス。

DynamoDB などの他の AWS リソースへのアクセスを追加するには、ランタイムロールを作成するときにポリシーにそれらのアクセス許可を含める必要があります。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ReadAccessForEMRSamples",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3::*.elasticmapreduce",
        "arn:aws:s3::*.elasticmapreduce/*"
      ]
    },
    {
      "Sid": "FullAccessToS3Bucket",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:DeleteObject"
      ],
    }
  ]
}
```

```

    "Resource": [
      "arn:aws:s3:::amzn-s3-demo-bucket",
      "arn:aws:s3:::amzn-s3-demo-bucket/*"
    ],
    {
      "Sid": "GlueCreateAndReadDataCatalog",
      "Effect": "Allow",
      "Action": [
        "glue:GetDatabase",
        "glue:CreateDatabase",
        "glue:GetDataBases",
        "glue:CreateTable",
        "glue:GetTable",
        "glue:UpdateTable",
        "glue>DeleteTable",
        "glue:GetTables",
        "glue:GetPartition",
        "glue:GetPartitions",
        "glue:CreatePartition",
        "glue:BatchCreatePartition",
        "glue:GetUserDefinedFunctions"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}

```

ロール権限を渡す

ユーザーのロールに IAM のアクセス許可ポリシーをアタッチして、ユーザーが承認済みのロールのみを渡すことを許可できます。これにより、管理者は特定のジョブランタイムロールを EMR Serverless ジョブに渡すことができるユーザーを制御できます。アクセス許可の設定の詳細については、[AWS 「サービスにロールを渡すアクセス許可をユーザーに付与する」](#)を参照してください。

以下は、ジョブランタイムロールを EMR Serverless サービスプリンシパルに渡すことを許可するポリシーの例です。

```

{
  "Effect": "Allow",

```

```
{
  "Action": "iam:PassRole",
  "Resource": "arn:aws:iam::1234567890:role/JobRuntimeRoleForEMRServerless",
  "Condition": {
    "StringLike": {
      "iam:PassedToService": "emr-serverless.amazonaws.com"
    }
  }
}
```

ランタイムロールに関連付けられた管理アクセス許可ポリシー

EMR Studio コンソールから EMR Serverless にジョブ実行を送信する場合、アプリケーションに関連付けるランタイムロールを選択するステップがあります。コンソールの各選択には、注意すべき重要な基盤となる管理ポリシーが関連付けられています。3 つの選択肢は以下のとおりです:

1. すべてのバケット – これを選択すると、すべてのバケットへのフルアクセスを提供する [AmazonS3FullAccess](#) AWS 管理ポリシーが指定されます。
2. 特定のバケット – 選択した各バケットの Amazon リソースネーム (ARN) 識別子を指定します。基盤となる管理ポリシーは含まれていません。
3. なし – マネージドポリシーのアクセス許可は含まれません。

特定のバケットを追加することをお勧めします。すべてのバケットを選択した場合は、すべてのバケットにフルアクセスを設定することに注意してください。

EMR Serverless のユーザーアクセスポリシーの例

EMR Serverless アプリケーションを操作する際に各ユーザーに実行させるアクションに応じて、ユーザーに対してきめ細かくポリシーを設定できます。次のポリシーは、ユーザーに適切なアクセス許可を設定するのに役立つ可能性のある例です。このセクションでは、EMR Serverless ポリシーのみに焦点を当てます。EMR Studio ユーザーポリシーのサンプルについては、「[Configure EMR Studio user permissions](#)」を参照してください。IAM ユーザー (プリンシパル) にポリシーをアタッチする方法の詳細については、「IAM ユーザーガイド」の「[IAM ポリシーを管理する](#)」を参照してください。

パワーユーザーポリシー

EMR Serverless に必要なすべてのアクションを付与するには、AmazonEMRServerlessFullAccess ポリシーを作成して必要な IAM ユーザー、ロール、またはグループにアタッチします。

以下の例は、パワーユーザーが EMR Serverless アプリケーションを作成、変更したり、ジョブの送信やデバッグなどのアクションを実行したりできるようにするポリシーです。EMR Serverless がその他のサービスに必要とするすべてのアクションを明らかにします。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EMRServerlessActions",
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CreateApplication",
        "emr-serverless:UpdateApplication",
        "emr-serverless>DeleteApplication",
        "emr-serverless:ListApplications",
        "emr-serverless:GetApplication",
        "emr-serverless:StartApplication",
        "emr-serverless:StopApplication",
        "emr-serverless:StartJobRun",
        "emr-serverless:CancelJobRun",
        "emr-serverless:ListJobRuns",
        "emr-serverless:GetJobRun"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

VPC へのネットワーク接続を有効にすると、EMR Serverless アプリケーションは VPC リソースと通信するための Amazon EC2 Elastic Network Interface (ENI) を作成します。次のポリシーは、新しい EC2 ENI が EMR Serverless アプリケーションのコンテキストでのみ作成されることを保証します。

Note

EMR Serverless アプリケーションを起動する場合を除き、ユーザーが EC2 ENI を作成できないように、このポリシーを設定することを強くお勧めします。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowEC2ENICreationWithEMRTags",
      "Effect": "Allow",
      "Action": [
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:network-interface/*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:CalledViaLast": "ops.emr-serverless.amazonaws.com"
        }
      }
    }
  ]
}
```

EMR Serverless アクセスを特定のサブネットに制限する場合は、各サブネットにタグ条件を付けることができます。この IAM ポリシーにより、EMR Serverless アプリケーションは許可されたサブネット内でのみ EC2 ENI を作成できるようになります。

```
{
  "Sid": "AllowEC2ENICreationInSubnetAndSecurityGroupWithEMRTags",
  "Effect": "Allow",
  "Action": [
    "ec2:CreateNetworkInterface"
  ],
  "Resource": [
```

```

    "arn:aws:ec2:*:*:subnet/*",
    "arn:aws:ec2:*:*:security-group/*"
  ],
  "Condition": {
    "StringEquals": {
      "aws:ResourceTag/KEY": "VALUE"
    }
  }
}

```

Important

管理者またはパワーユーザーが最初のアプリケーションを作成している場合は、EMR Serverless サービスにリンクされたロールを作成できるようにアクセス許可ポリシーを設定する必要があります。詳細については、[EMR Serverless のサービスリンクロールの使用](#) を参照してください。

次の IAM ポリシーでは、アカウントに対して EMR Serverless サービスにリンクされたロールの作成を許可します。

```

{
  "Sid": "AllowEMRServerlessServiceLinkedRoleCreation",
  "Effect": "Allow",
  "Action": "iam:CreateServiceLinkedRole",
  "Resource": "arn:aws:iam::account-id:role/aws-service-role/ops.emr-serverless.amazonaws.com/AWSServiceRoleForAmazonEMRServerless"
}

```

データエンジニアポリシー

以下の例は、EMR Serverless アプリケーションに対する読み取り専用アクセス許可と、ジョブの送信とデバッグをユーザーに許可するポリシーです。このポリシーは明示的にアクションを拒否しないため、別のポリシーステートメントが指定したアクションへのアクセス許可に使用される場合があることに注意してください。

JSON

```

{

```

```
"Version": "2012-10-17",
"Statement": [
  {
    "Sid": "EMRServerlessActions",
    "Effect": "Allow",
    "Action": [
      "emr-serverless:ListApplications",
      "emr-serverless:GetApplication",
      "emr-serverless:StartApplication",
      "emr-serverless:StartJobRun",
      "emr-serverless:CancelJobRun",
      "emr-serverless:ListJobRuns",
      "emr-serverless:GetJobRun"
    ],
    "Resource": [
      "*"
    ]
  }
]
```

アクセスコントロールにタグを使用する

きめ細かいアクセスコントロールのためにタグ条件を使用できます。例えば、1つのチームからユーザーを制限して、チーム名でタグ付けされた EMR Serverless アプリケーションにのみジョブを送信できるようにすることができます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EMRServerlessActions",
      "Effect": "Allow",
      "Action": [
        "emr-serverless:ListApplications",
        "emr-serverless:GetApplication",
        "emr-serverless:StartApplication",
        "emr-serverless:StartJobRun",
        "emr-serverless:CancelJobRun",
```



```
        "emr-serverless:ListJobRuns",
        "emr-serverless:GetJobRun"
    ],
    "Resource": [
        "*"
    ]
}
]
```

タグベースのアクセスコントロールのポリシー

アイデンティティベースのポリシーの条件を使用して、タグに基づいてアプリケーションとジョブ実行へのアクセスをコントロールできます。

次の例では、EMR Serverless 条件キーで条件演算子を使用するさまざまなシナリオと方法について説明しています。これらの IAM ポリシーステートメントは、デモンストレーションのみを目的としており、本稼働環境で使用しないでください。要件に応じて、アクセス権限を付与または拒否するようにポリシーステートメントを組み合わせる複数の方法があります。IAM ポリシーの計画およびテストの詳細については、「[IAM ユーザーガイド](#)」を参照してください。

Important

アクションをタグ付けするための権限を明示的に拒否することは重要な考慮事項です。これにより、ユーザーがリソースをタグ付けして意図せずにアクセス許可を付与することを防ぎます。リソースのタグ付けアクションが拒否されない場合、ユーザーはタグを変更して、タグベースのポリシーの意図を回避できます。タグ付けアクションを拒否するポリシーの例については、「[タグを追加および削除するためのアクセス権限を拒否する](#)」を参照してください。

以下の例では、EMR Serverless アプリケーションで許可されるアクションを制御するために使用するアイデンティティベースの許可ポリシーを説明しています。

特定のタグの値があるリソースでのみアクションを許可する

次のポリシーの例では、StringEquals 条件演算子は、dev をタグ department の値と一致させるように試みます。タグ department がアプリケーションに追加されていない場合、またはタグ department に値 dev が含まれていない場合は、ポリシーは適用されず、このアクションはポリシー

によって許可されません。アクションを許可するポリシーステートメントが他にない場合は、ユーザーはこの値を持つこのタグを持っているアプリケーションとのみ作業できます。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:GetApplication"
      ],
      "Resource": [
        "*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/department": "dev"
        }
      },
      "Sid": "AllowEMRSERVERLESSGetapplication"
    }
  ]
}
```

条件付き演算子を使用して複数のタグ値を指定できます。例えば、department タグに値 dev または test が含まれるアプリケーションでアクションを許可するには、以下のように、前術の例のような条件ブロックを置き換えます。

```
"Condition": {
  "StringEquals": {
    "emr-serverless:ResourceTag/department": ["dev", "test"]
  }
}
```

リソースの作成時にタグ付けを要求する

次の例では、アプリケーションの作成時にタグを適用する必要があります。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CreateApplication"
      ],
      "Resource": [
        "*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:RequestedRegion": "us-east-1"
        }
      },
      "Sid": "AllowEMRSERVERLESSCreateapplication"
    }
  ]
}
```

次のポリシーステートメントは、任意の値を含めることができる department タグがアプリケーションにある場合にのみ、ユーザーにアプリケーションの作成を許可します。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "emr-serverless:CreateApplication"
      ],
      "Resource": [
        "*"
      ],
      "Condition": {
```

```
    "StringEquals": {
      "aws:RequestedRegion": ["us-east-1", "us-west-2"]
    },
    "Sid": "AllowEMRSERVERLESSCreateapplication"
  }
]
}
```

タグを追加および削除するためのアクセス権限を拒否する

このポリシーは、ユーザーが、dev ではない値の department タグを使用して EMR Serverless アプリケーションでタグを追加または削除できないようにします。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "emr-serverless:TagResource",
        "emr-serverless:UntagResource"
      ],
      "Resource": [
        "*"
      ],
      "Condition": {
        "StringNotEquals": {
          "aws:PrincipalTag/department": "dev"
        }
      },
      "Sid": "AllowEMRSERVERLESSTagresource"
    }
  ]
}
```

EMR Serverless でのアイデンティティベースのポリシー例

デフォルトでは、ユーザーおよびロールには Amazon EMR Serverless リソースを作成または変更するアクセス権はありません。IAM 管理者は、リソースで必要なアクションを実行するための権限をユーザーに付与する IAM ポリシーを作成できます。

これらのサンプルの JSON ポリシードキュメントを使用して IAM アイデンティティベースのポリシーを作成する方法については、「IAM ユーザーガイド」の「[IAM ポリシーを作成する \(コンソール\)](#)」を参照してください。

Amazon EMR Serverless が定義するアクションとリソースタイプ (リソースタイプごとの ARN の形式を含む) の詳細については、「サービス認可リファレンス」の「[Amazon EMR Serverless のアクション、リソース、および条件キー](#)」を参照してください。

トピック

- [ポリシーに関するベストプラクティス](#)
- [自分の権限へのアクセスをユーザーに許可する](#)

ポリシーに関するベストプラクティス

Note

EMR Serverless は マネージドポリシーをサポートしていないため、以下に記載されている最初のプラクティスは適用されません。

アイデンティティベースのポリシーは、アカウント内で誰が Amazon EMR Serverless リソースを作成、アクセス、または削除できるを決定します。これらのアクションでは、AWS アカウントに費用が発生する場合があります。アイデンティティベースポリシーを作成したり編集したりする際には、以下のガイドラインと推奨事項に従ってください:

- AWS 管理ポリシーを開始し、最小特権のアクセス許可に移行する – ユーザーとワークロードにアクセス許可の付与を開始するには、多くの一般的なユースケースにアクセス許可を付与する AWS 管理ポリシーを使用します。これらは 使用できます AWS アカウント。ユースケースに固有の AWS カスタマー管理ポリシーを定義することで、アクセス許可をさらに減らすことをお勧めします。詳細については、IAM ユーザーガイドの [AWS マネージドポリシー](#) または [ジョブ機能のAWS マネージドポリシー](#) を参照してください。

- 最小特権を適用する – IAM ポリシーでアクセス許可を設定する場合は、タスクの実行に必要な許可のみを付与します。これを行うには、特定の条件下で特定のリソースに対して実行できるアクションを定義します。これは、最小特権アクセス許可とも呼ばれています。IAM を使用して許可を適用する方法の詳細については、IAM ユーザーガイドの [IAM でのポリシーとアクセス許可](#) を参照してください。
- IAM ポリシーで条件を使用してアクセスをさらに制限する – ポリシーに条件を追加して、アクションやリソースへのアクセスを制限できます。たとえば、ポリシー条件を記述して、すべてのリクエストを SSL を使用して送信するように指定できます。条件を使用して、サービスアクションがなどの特定の を通じて使用されている場合に AWS のサービス、サービスアクションへのアクセスを許可することもできます CloudFormation。詳細については、IAM ユーザーガイドの [IAM JSON ポリシー要素:条件](#) を参照してください。
- IAM アクセスアナライザー を使用して IAM ポリシーを検証し、安全で機能的な権限を確保する – IAM アクセスアナライザー は、新規および既存のポリシーを検証して、ポリシーが IAM ポリシー言語 (JSON) および IAM のベストプラクティスに準拠するようにします。IAM アクセスアナライザーは 100 を超えるポリシーチェックと実用的な推奨事項を提供し、安全で機能的なポリシーの作成をサポートします。詳細については、IAM ユーザーガイドの [IAM Access Analyzer でポリシーを検証する](#) を参照してください。
- 多要素認証 (MFA) を要求する – IAM ユーザーまたはルートユーザーを必要とするシナリオがある場合は AWS アカウント、MFA をオンにしてセキュリティを強化します。API オペレーションが呼び出されるときに MFA を必須にするには、ポリシーに MFA 条件を追加します。詳細については、IAM ユーザーガイドの [MFA を使用した安全な API アクセス](#) を参照してください。

IAM でのベストプラクティスの詳細については、IAM ユーザーガイドの [IAM でのセキュリティのベストプラクティス](#) を参照してください。

自分の権限へのアクセスをユーザーに許可する

この例では、ユーザーアイデンティティにアタッチされたインラインおよびマネージドポリシーの表示を IAM ユーザーに許可するポリシーの作成方法を示します。このポリシーには、コンソールで、または AWS CLI または AWS API を使用してプログラムでこのアクションを実行するアクセス許可が含まれています。

```
{
  "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 マネージドポリシーに対する Amazon EMR Serverless の更新

このサービスがこれらの変更の追跡を開始してからの Amazon EMR Serverless の AWS マネージドポリシーの更新に関する詳細にアクセスします。このページの変更に関する自動通知を入手するには、Amazon EMR Serverless [ドキュメントの履歴ページ](#) から、RSS フィードにサブスクライブしてください。

変更	説明	日付
AmazonEMRServerlessServiceRolePolicy – 既存のポリシーの更新	Amazon EMR Serverless は、新しい Sid の CloudWatchPolicyStatement と EC2PolicyStatement を AmazonEMRServerlessServiceRolePolicy ポリシー に追加しました。	2024 年 1 月 25 日
AmazonEMRServerlessServiceRolePolicy – 既存のポリシーの更新	Amazon EMR Serverless は、Amazon EMR Serverless が "AWS/Usage" 名前空間で vCPU 使用状況の集計アカウントメトリクスを発行できるようにする新しいアクセス許可を追加しました。	2023 年 4 月 20 日
Amazon EMR Serverless が変更の追跡を開始しました。	Amazon EMR Serverless は、AWS 管理ポリシーの変更の追跡を開始しました。	2023 年 4 月 20 日

Amazon EMR Serverless の ID とアクセスのトラブルシューティング

Amazon EMR Serverless と IAM の使用時に発生する可能性がある一般的な問題の診断や修正には、次の情報が役立ちます。

トピック

- [Amazon EMR Serverless でアクションを実行する権限がない](#)
- [iam:PassRole を実行する権限がない](#)
- [AWS アカウント外のユーザーに Amazon EMR Serverless リソースへのアクセスを許可したい](#)
- [EMR Studio からライブ UI/Spark 履歴サーバーを開いてジョブをデバッグできないか、get-dashboard-for-job-run を使用してログを取得しようとする API エラーが発生する](#)

Amazon EMR Serverless でアクションを実行する権限がない

がアクションを実行する権限がないと AWS マネジメントコンソール 通知した場合は、管理者に連絡してサポートを依頼してください。担当の管理者はお客様のユーザー名とパスワードを発行した人です。

以下のエラー例は、mateojackson ユーザーがコンソールを使用して架空の *my-example-widget* リソースに関する詳細情報にアクセスしようとしているが、架空の `emr-serverless:GetWidget` 許可がないという場合に発生します。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: emr-serverless:GetWidget on resource: my-example-widget
```

この場合、Mateo は、`emr-serverless:GetWidget` アクションを使用して *my-example-widget* リソースにアクセスできるように、管理者にポリシーの更新を依頼します。

iam:PassRole を実行する権限がない

`iam:PassRole` アクションを実行する権限がないというエラーが表示された場合は、ポリシーを更新して Amazon EMR Serverless にロールを渡せるようにする必要があります。

一部の AWS のサービス では、新しいサービスロールまたはサービスにリンクされたロールを作成する代わりに、既存のロールをそのサービスに渡すことができます。そのためには、サービスにロールを渡すアクセス許可が必要です。

次の例のエラーは、marymajor という IAM ユーザーがコンソールを使用して Amazon EMR Serverless でアクションを実行しようとした場合に発生します。ただし、このアクションをサービスが実行するには、サービスロールから付与されたアクセス許可が必要です。Mary には、ロールをサービスに渡すアクセス許可がありません。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform: iam:PassRole
```

この場合、Mary のポリシーを更新してメアリーに `iam:PassRole` アクションの実行を許可する必要があります。

サポートが必要な場合は、AWS 管理者にお問い合わせください。サインイン資格情報を提供した担当者が管理者です。

AWS アカウント外のユーザーに Amazon EMR Serverless リソースへのアクセスを許可したい

他のアカウントのユーザーや組織外の人々が、リソースにアクセスするために使用できるロールを作成できます。ロールの引き受けを委託するユーザーを指定できます。リソースベースのポリシーまたはアクセスコントロールリスト (ACL) をサポートするサービスの場合、それらのポリシーを使用して、リソースへのアクセスを付与できます。

詳細については、以下を参照してください:

- Amazon EMR Serverless がこれらの機能をサポートしているかどうかを確認するには、「[Amazon EMR Serverless での Identity and Access Management \(IAM\)](#)」を参照してください。
- 所有 AWS アカウント している のリソースへのアクセスを提供する方法については、IAM ユーザーガイドの「[所有 AWS アカウント している別の の IAM ユーザーへのアクセスを提供する](#)」を参照してください。
- リソースへのアクセスをサードパーティーに提供する方法については AWS アカウント、IAM ユーザーガイドの「[サードパーティー AWS アカウント が所有する へのアクセスを提供する](#)」を参照してください。
- ID フェデレーションを介してアクセスを提供する方法については、IAM ユーザーガイドの [外部で認証されたユーザー \(ID フェデレーション\) へのアクセスの許可](#) を参照してください。
- クロスアカウントアクセスにおけるロールとリソースベースのポリシーの使用方法の違いについては、IAM ユーザーガイドの [IAM でのクロスアカウントのリソースへのアクセス](#) を参照してください。

EMR Studio からライブ UI/Spark 履歴サーバーを開いてジョブをデバッグできないが、**get-dashboard-for-job-run** を使用してログを取得しようとする と API エラーが発生する

ログ記録に EMR Serverless マネージドストレージを使用し、EMR Serverless アプリケーションが Amazon S3 の VPC エンドポイントを持つプライベートサブネットにあり、アクセスを制御するエンドポイントポリシーをアタッチする場合は、「[Logging for EMR Serverless with managed storage in your VPC policy](#)」に記載されているアクセス許可を、EMR Serverless がアプリケーションログを保存して処理するための S3 ゲートウェイエンドポイントに追加します。

信頼できる ID の伝播

Amazon EMR リリース 7.8.0 以降では、Apache Livy Endpoint を介して EMR Serverless を使用して AWS、IAM Identity Center からインタラクティブなワークロードにユーザー ID を伝達できます。Apache Livy インタラクティブワークロードは、提供された ID をさらに Amazon S3、Lake Formation、Amazon Redshift などのダウンストリームサービスに伝播し、これらのダウンストリームでユーザー ID を介した安全なデータアクセスを可能にします。以下のセクションでは、Apache Livy Endpoint を介して EMR Serverless で ID を起動してインタラクティブワークロードに伝達するために必要な概念的な概要、前提条件、および手順について説明します。

概要:

[IAM Identity Center](#) は、あらゆる規模とタイプの組織に でのワークフォース認証と認可 AWS に推奨されるアプローチです。Identity Center では、 でユーザー ID を作成および管理するか AWS、Microsoft Active Directory、Okta、Ping Identity、JumpCloud、Google Workspace、Microsoft Entra ID (旧 Azure AD) などの既存の ID ソースを接続します。

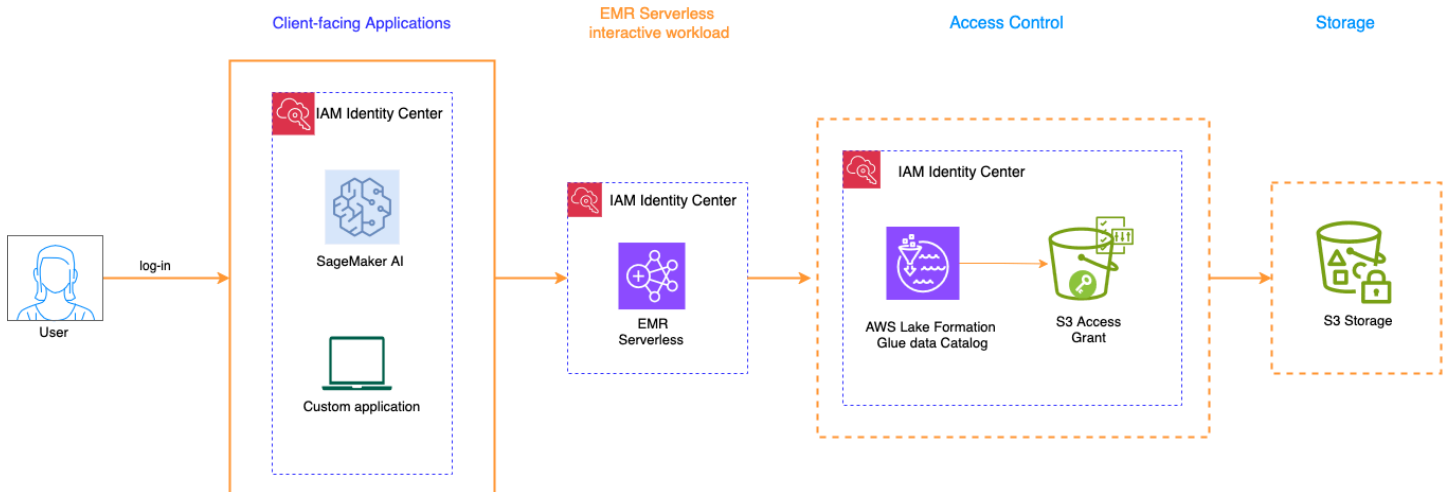
[信頼できる ID の伝播](#)は、接続された AWS サービスの管理者がサービスデータへのアクセスを許可および監査するために使用できる AWS IAM アイデンティティセンターの機能です。このデータへのアクセスは、グループの関連付けなどのユーザー属性に基づいています。信頼できる ID の伝播を設定するには、接続された AWS サービスの管理者と IAM Identity Center 管理者間のコラボレーションが必要です。詳細については、「IAM Identity Center ユーザーガイド」の「[Prerequisites and considerations](#)」を参照してください。

機能と利点

EMR Serverless Apache Livy Endpoint と IAM Identity Center の統合による[信頼された ID の伝播](#)には、次の利点があります。

- AWS Lake Formation マネージド Glue データカタログテーブルで Identity Center ID AWS を使用してテーブルレベルの認可を強制する機能。
- Amazon Redshift クラスターでアイデンティティセンター ID による認可を適用する能力。
- 監査用のユーザーアクションのエンドツーエンドの追跡の有効化。
- S3 Access Grants が管理する S3 プレフィックスの Identity Center ID を使用して Amazon S3 プレフィックスレベルの認可を執行できます。

仕組み



ユースケースの例

データの準備と特徴量エンジニアリング

複数の研究チームのデータサイエンティストが、統合されたデータプラットフォームを使用して複雑なプロジェクトで共同作業を行います。会社の認証情報を使用して SageMaker AI にログインし、複数の AWS アカウントにまたがる膨大な共有データレイクにすぐにアクセスできます。新しい機械学習モデルの特徴量エンジニアリングを開始することにより、EMR Serverless を通じて起動された Spark セッションは、伝播された ID に基づいて Lake Formation の列および行レベルのセキュリティポリシーを適用します。サイエンティストは使い慣れたツールを使用してデータやエンジニアリング機能を効率的に準備でき、コンプライアンスチームはすべてのデータインタラクションが確実に自動的に追跡および監査される体制が得られます。この安全で協調的な環境は、規制の厳しい業界で必要とされる厳格なデータ保護標準を維持しながら、研究パイプラインを加速します。

Trusted-Identity の伝播の開始方法

このセクションでは、Apache Livy Endpoint で EMR-Serverless アプリケーションを設定して AWS IAM Identity Center と統合し、[信頼できる ID の伝播](#)を有効にする方法について説明します。

前提条件

- Trusted Identity 伝達が有効な EMR Serverless Apache Livy Endpoint を作成する AWS リージョンの Identity Center インスタンス。Identity Center インスタンスは、AWS アカウントの 1 つのリージョンにのみ存在できます。[「IAM Identity Center を有効にする」](#) および [「ID のソースから IAM Identity Center にユーザーとグループをプロビジョニングする」](#) を参照してください。

- Lake Formation、S3 Access Grants、Amazon Redshift クラスターなど、インタラクティブなワークロードがデータにアクセスするためにやり取りするダウンストリームサービスに対して、信頼された ID 伝播を有効にします。

信頼できる ID の伝播が有効な EMR Serverless Application を作成するアクセス許可

[EMR Serverless へのアクセスに必要な基本的なアクセス許可](#)に加えて、信頼された ID の伝播が有効な EMR Serverless アプリケーションの作成に使用する IAM ID またはロールに追加のアクセス許可を設定する必要があります。信頼された ID の伝播のために、EMR Serverless は、ID の検証とダウンストリームへの ID の伝播にサービスが活用する 1 つのサービスマネージド Identity Center アプリケーションをアカウントに作成/ブートストラップします。

```
"sso:DescribeInstance",  
"sso:CreateApplication",  
"sso:DeleteApplication",  
"sso:PutApplicationAuthenticationMethod",  
"sso:PutApplicationAssignmentConfiguration",  
"sso:PutApplicationGrant",  
"sso:PutApplicationAccessScope"
```

- `sso:DescribeInstance` – インスタンスを説明し、`identity-center-configuration` パラメータで指定した IAM Identity Center `instanceArn` を検証するアクセス許可を付与します。
- `sso:CreateApplication` – `trusted-identity-propagation` アクションに使用される EMR Serverless マネージド IAM Identity Center アプリケーションを作成するアクセス許可を付与します。
- `sso:DeleteApplication` – EMR Serverless のマネージド IAM Identity Center アプリケーションを更新するためのアクセス許可を付与します。
- `sso:PutApplicationAuthenticationMethod` – EMR Serverless マネージド IAM Identity Center アプリケーションに `authenticationMethod` を配置するアクセス許可を付与し、`emr-serverless` サービスプリンシパルが IAM Identity Center アプリケーションとやり取りできるようにします。
- `sso:PutApplicationAssignmentConfiguration` – IAM Identity Center アプリケーションに「`User-assignment-not-required`」設定を設定するアクセス許可を付与します。
- `sso:PutApplicationGrant` – IAM Identity Center アプリケーションに `token-exchange`、`introspectToken`、`refreshToken`、`revokeToken` 許可を適用するアクセス許可を付与します。

- `sso:PutApplicationAccessScope` – 信頼されたアイデンティティの伝播が有効なダウンストリームスコープを IAM Identity Center アプリケーションに適用するアクセス許可を付与します。「`redshift:connect`」、「`lakeformation:query`」、および「`s3:read_write`」スコープを適用して、これらのサービスに `trusted-identity-propagation` を有効にします。

信頼された ID の伝播が有効な EMR Serverless アプリケーションを作成する

アプリケーションで信頼された ID の伝播を有効にする `identityCenterInstanceArn` には、`--identity-center-configuration` フィールドを指定する必要があります。次のコマンド例を使用して、信頼された ID の伝播が有効になっている EMR Serverless アプリケーションを作成します。

Note

また、Apache Livy エンドポイントでのみ信頼された ID の伝播が有効になっているため、`--interactive-configuration '{"livyEndpointEnabled":true}'` を指定する必要があります。

```
aws emr-serverless create-application \  
  --release-label emr-7.8.0 \  
  --type "SPARK" \  
  --identity-center-configuration '{"identityCenterInstanceArn" :  
"arn:aws:sso:::instance/ssoins-123456789"}' \  
  --interactive-configuration '{"livyEndpointEnabled":true}'
```

- `identity-center-configuration` – (オプション) 指定した場合、Identity Center の信頼された ID の伝播を有効にします。
- `identityCenterInstanceArn` – (必須) Identity Center インスタンスの ARN。

必要な Identity Center アクセス許可 (前述の) がない場合は、まず信頼された ID の伝播なしで EMR Serverless アプリケーションを作成し (例えば、`--identity-center-configuration` パラメータを指定しない)、後で Identity Center 管理者に `update-application` API を呼び出して信頼された ID の伝播を有効にするように依頼します。以下の例を参照してください。

```
aws emr-serverless update-application \  
  --application-id applicationId \  
  --identity-center-configuration '{"identityCenterInstanceArn" :  
"arn:aws:sso:::instance/ssoins-123456789"}' \  
  --interactive-configuration '{"livyEndpointEnabled":true}'
```

```
--identity-center-configuration '{"identityCenterInstanceArn" :  
"arn:aws:sso:::instance/ssoinstances-123456789"}'
```

EMR Serverless は、サービスが ID 検証とダウンストリームサービスへの ID 伝達に活用するサービスマネージド Identity Center アプリケーションをアカウントに作成します。EMR Serverless が作成したマネージド Identity Center アプリケーションは、アカウント内の trusted-identity-propagation が有効なすべての EMR Serverless アプリケーション間で共有されます。

Note

マネージドアイデンティセンターアプリケーションの設定を変更しないでください。変更は、アカウントで信頼された ID 伝播が有効なすべての インタラクティブセッションに影響を与える可能性があります。

ID を伝播するためのジョブ実行ロールのアクセス許可

EMR-Serverless は Identity 拡張の job-execution-role 認証情報を利用して ID をダウンストリーム AWS サービスに伝達するため、ジョブ実行ロールの信頼ポリシーには、ID を使用してジョブ実行ロール認証情報 `sts:SetContext` を強化し、S3 access-grant、Lake Formation、Amazon Redshift などのダウンストリームサービスへの trusted-identity-propagation を許可する追加のアクセス許可が必要です。ロールの作成方法の詳細については、「[Create a job runtime role](#)」を参照してください。

JSON

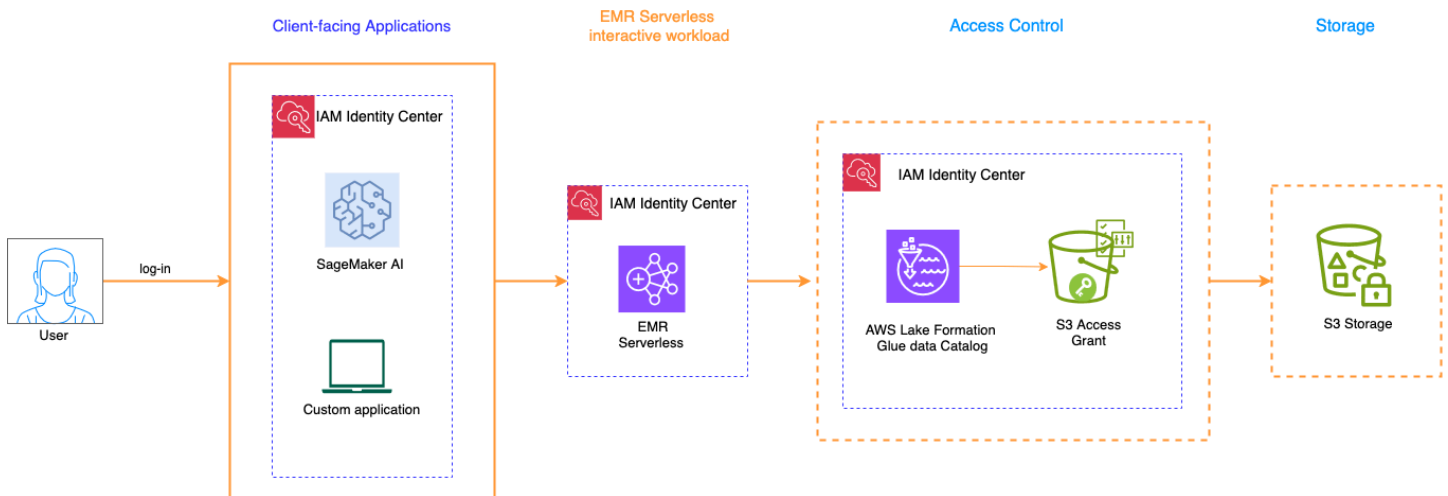
```
{  
  "Version": "2012-10-17",  
  "Statement": [  
    {  
      "Effect": "Allow",  
      "Principal": {  
        "Service": "emr-serverless.amazonaws.com"  
      },  
      "Action": [ "sts:AssumeRole", "sts:SetContext" ]  
    }  
  ]  
}
```


さらに、JobExecutionRole には、ジョブ実行がユーザー ID を使用してデータを取得するために呼び出すダウストリーム AWS サービスのアクセス許可が必要です。S3 Access Grant、Lake Formation を設定するには、以下のリンクを参照してください。

- [EMR Serverless での Lake Formation の使用](#)
- [EMR Serverless での Amazon S3 Access Grants の使用](#)

インタラクティブワークロードの信頼できる ID の伝播

Apache Livy エンドポイントを介してインタラクティブワークロードにアイデンティティを伝達するステップは、ユーザーがなどのマネージド開発環境と AWS やり取りするか Amazon SageMaker AI、クライアント向けアプリケーションとして独自のセルフホスト型ノートブック環境とやり取りするかによって異なります。



AWS マネージド開発環境

次の AWS マネージドクライアント向けアプリケーションは、EMR-Serverless Apache Livy エンドポイントによる信頼できる ID 伝達をサポートしています。

- [Amazon SageMaker AI](#)

カスタマーマネージドセルフホスト型ノートブック環境

カスタム開発アプリケーションのユーザーに対して信頼できる ID 伝達を有効にするには、AWS セキュリティブログの「[信頼できる ID 伝達を使用してプログラムで AWS サービスにアクセスします](#)」を参照してください。

ユーザーバックグラウンドセッション

ユーザーバックグラウンドセッションを使用すると、ユーザーがノートブックインターフェイスからログオフした後も、長時間実行される分析と機械学習フローを続行できます。この機能は、IAM アイデンティティセンターの信頼できる ID 伝達機能と EMR Serverless の統合を通じて実装されます。このセクションでは、ユーザーバックグラウンドセッションの設定オプションと動作について説明します。

Note

ユーザーバックグラウンドセッションは、Amazon SageMaker Unified Studio などのノートブックインターフェイスを介して開始された Spark ワークロードに適用されます。この機能の有効化または無効化は、新しい Livy セッションにのみ影響します。既存のアクティブな Livy セッションには影響しません。

ユーザーバックグラウンドセッションを設定する

適切に機能させるには、ユーザーバックグラウンドセッションを 2 つのレベルで有効にする必要があります。

1. IAM Identity Center インスタンスレベル – 通常、IdC 管理者が設定します
2. EMR Serverless アプリケーションレベル – EMR Serverless アプリケーション管理者が設定

EMR Serverless アプリケーションのユーザーバックグラウンドセッションを有効にする

EMR Serverless アプリケーションのユーザーバックグラウンドセッションを有効にするには、アプリケーションを作成または更新identityCenterConfigurationするときに、trueでuserBackgroundSessionsEnabledパラメータを に設定する必要があります。

前提条件

- EMR Serverless アプリケーションの作成/更新に使用される IAM ロールには、アクセス sso:PutApplicationSessionConfiguration 許可が必要です。このアクセス許可により、EMR Serverless は EMR Serverless マネージド IdC アプリケーションレベルでユーザーバックグラウンドセッションを有効にすることができます。
- EMR Serverless アプリケーションはリリースラベル 7.8 以降を使用し、Trusted-Identity Propagation が有効になっている必要があります。

を使用してユーザー背景セッションを有効にするには AWS CLI

```
aws emr-serverless create-application \  
  --name "my-analytics-app" \  
  --type "SPARK" \  
  --release-label "emr-7.8.0" \  
  --identity-center-configuration '{"identityCenterInstanceArn":  
"arn:aws:sso:::instance/ssoins-1234567890abcdef", "userBackgroundSessionsEnabled":  
true}'
```

既存のアプリケーションを更新するには:

```
aws emr-serverless update-application \  
  --application-id applicationId \  
  --identity-center-configuration '{"identityCenterInstanceArn":  
"arn:aws:sso:::instance/ssoins-1234567890abcdef", "userBackgroundSessionsEnabled":  
true}'
```

設定マトリックス

有効なユーザーバックグラウンドセッション設定は、EMR Serverless アプリケーション設定と IAM Identity Center インスタンスレベルの設定の両方によって異なります。

ユーザーバックグラウンドセッション設定マトリックス

IAM アイデンティティセン ター userBackgroundSession 有効	EMR Serverless userBackg roundSessionsEnabled	行動
はい	TRUE	ユーザーバックグラウンド セッションが有効
はい	FALSE	ユーザーログアウトでセッ ションの有効期限が切れる
なし	TRUE	アプリケーションの作成/更新 が例外で失敗する
いいえ	FALSE	ユーザーログアウトでセッ ションの有効期限が切れる

デフォルトのユーザーバックグラウンドセッション期間

デフォルトでは、IAM Identity Center のすべてのユーザーバックグラウンドセッションの期間制限は 7 日間です。管理者は、IAM Identity Center コンソールでこの期間を変更できます。この設定は IAM Identity Center インスタンスレベルで適用され、そのインスタンス内でサポートされているすべての IAM Identity Center アプリケーションに影響します。

- 期間は 15 分から 90 日までの任意の値に設定できます。
- この設定は、IAM Identity Center コンソールの設定 → 認証 → 設定 (非インタラクティブジョブセッション) で設定されます。

Note

EMR Serverless Livy セッションの最大期間は 24 時間です。セッションは、Livy セッションの制限またはユーザーバックグラウンドセッション期間のいずれか早い方に達したときに終了します。

ユーザーのバックグラウンドセッションを無効にした場合の影響

IAM Identity Center でユーザー背景セッションが無効になっている場合:

既存の Livy セッション

ユーザーバックグラウンドセッションを有効にして開始した場合は、中断することなく実行を続行します。これらのセッションは、自然に終了するか明示的に停止されるまで、既存のバックグラウンドセッショントークンを使用し続けます。

新しい Livy セッション

標準の信頼できる ID 伝達フローを使用し、ユーザーがログアウトするか、インタラクティブセッションの有効期限が切れると (Amazon SageMaker Unified Studio JupyterLab ノートブックを閉じる場合など) 終了します。

ユーザーバックグラウンドセッションの期間の変更

IAM Identity Center でユーザーバックグラウンドセッションの期間設定が変更された場合:

既存の Livy セッション

開始したのと同じバックグラウンドセッション期間で実行を続行します。

新しい Livy セッション

バックグラウンドセッションに新しいセッション期間を使用します。

考慮事項

セッション終了条件

ユーザーバックグラウンドセッションを使用する場合、Livy セッションは次のいずれかが発生するまで実行され続けます。

- ユーザーバックグラウンドセッションの有効期限が切れる (IdC 設定に基づく、最大 90 日間)
- ユーザーバックグラウンドセッションが管理者によって手動で取り消される
- Livy セッションがアイドルタイムアウトに達する (デフォルト: 最後に実行されたステートメントから 1 時間後)
- Livy セッションが最大期間 (24 時間) に達する
- ユーザーが、ノートブックカーネルを明示的に停止または再起動する

データの永続性

ユーザーバックグラウンドセッションを使用する場合:

- ユーザーはログアウトすると、ノートブックのインターフェースに再接続して結果を表示することはできません
- 実行が完了する前に結果を永続ストレージ (Amazon S3 など) に書き込むように Spark ステートメントを設定します

コストへの影響

- ジョブは、ユーザーが Amazon SageMaker Unified Studio JupyterLab セッションを終了した後も完了まで実行され続け、完了した実行の全期間に対して料金が発生します。
- アクティブなバックグラウンドセッションをモニタリングして、セッションを忘れたり中止したりする不要なコストを回避します。

機能を利用できるリージョン

EMR Serverless のユーザーバックグラウンドセッションは、以下に使用できます。

- Spark エンジンのみ (Hive エンジンはサポートされていません)
- Livy インタラクティブセッションのみ (バッチジョブとストリーミングジョブはサポートされていません)
- EMR Serverless リリースラベル 7.8 以降

EMR Serverless Trusted-Identity-Propagation 統合に関する考慮事項

EMR Serverless アプリケーションで IAM Identity Center Trusted-Identity-Propagation を使用する場合は、次の点を考慮してください。

- Identity Center が提供する信頼された ID 伝達は、Amazon EMR 7.8.0 以上でサポートされており、Apache Spark でのみ動作します。
- Trusted Identity Propagation は、[Apache Livy エンドポイントを介して EMR Serverless を使用するインタラクティブなワークロード](#)にのみ使用できます。EMR Studio を介したインタラクティブなワークロードは、信頼された ID の伝播をサポートしていません
- バッチジョブとストリーミングワークロードは、信頼できる ID の伝播をサポートしていません
- Trusted Identity Propagation を使用する AWS Lake Formation を使用したきめ細かなアクセスコントロールは、[Apache Livy エンドポイントを介して EMR Serverless を使用するインタラクティブなワークロード](#)で使用できます。
- Amazon EMR による信頼できる ID の伝播は、次の AWS リージョンでサポートされています。
 - af-south-1 – アフリカ (ケープタウン)
 - ap-east-1 – アジアパシフィック (香港)
 - ap-northeast-1 – アジアパシフィック (東京)
 - ap-northeast-2 – アジアパシフィック (ソウル)
 - ap-northeast-3 – アジアパシフィック (大阪)
 - ap-south-1 – アジアパシフィック (ムンバイ)
 - ap-southeast-1 – アジアパシフィック (シンガポール)
 - ap-southeast-2 – アジアパシフィック (シドニー)
 - ap-southeast-3 – アジアパシフィック (ジャカルタ)
 - ca-central-1 – カナダ (中部)

- ca-west-1 – カナダ (カルガリー)
- eu-central-1 – 欧州 (フランクフルト)
- eu-north-1 – 欧州 (ストックホルム)
- eu-south-1 – 欧州 (ミラノ)
- eu-south-2 – 欧州 (スペイン)
- eu-west-1 – 欧州 (アイルランド)
- eu-west-2 – 欧州 (ロンドン)
- eu-west-3 – 欧州 (パリ)
- me-central-1 – 中東 (UAE)
- me-south-1 – 中東 (バーレーン)
- sa-east-1 – 南米 (サンパウロ)
- us-east-1 – 米国東部 (バージニア北部)
- us-east-2 – 米国東部 (オハイオ)
- us-west-1 – 米国西部 (北カリフォルニア)
- us-west-2 – 米国西部 (オレゴン)

EMR Serverless での Lake Formation の使用

フルテーブルアクセスまたはきめ細かなアクセスコントロールで Lake Formation を使用するように EMR Serverless アプリケーションを設定できます。各アクセスモードでサポートされている機能の詳細については、次の表を参照してください。

機能を利用できるリージョン

機能	から利用可能
Hive、Iceberg テーブルの読み取りオペレーション (SELECT、DESCRIBE)	EMR 7.2 以降
マルチダイアレクトビュー	EMR 7.6 以降
Delta Lake テーブルと Hudi テーブルの読み取りオペレーション (SELECT、DESCRIBE)	EMR 7.6 以降

機能	から利用可能
Hive、Iceberg のフルテーブルアクセス	EMR 7.9 以降
Delta Lake のフルテーブルアクセス	EMR 7.11 以降
Hive、Iceberg、Delta Lake テーブルの書き込みオペレーション (DDL、DML)	EMR 7.12 以降
Hudi のフルテーブルアクセス	EMR 7.12 以降

EMR Serverless の Lake Formation フルテーブルアクセス

Amazon EMR リリース 7.8.0 以降では、ジョブランタイムロールにきめ細かなアクセスコントロールの制限なしに完全なテーブルアクセス許可がある Glue Data Catalog で AWS Lake Formation を活用できます。この機能を使用することにより、EMR Serverless Spark バッチジョブとインタラクティブジョブから Lake Formation によって保護されているテーブルを読み書きできます。Lake Formation の詳細と EMR Serverless での使用方法については、以下のセクションを参照してください。

フルテーブルアクセスでの Lake Formation の使用

AWS Lake Formation で保護された Glue データカタログテーブルには、EMR Serverless Spark ジョブまたはジョブのランタイムロールにフルテーブルアクセスがあるインタラクティブセッションからアクセスできます。EMR Serverless アプリケーションで AWS Lake Formation を有効にする必要はありません。Spark ジョブがフルテーブルアクセス (FTA) 用に設定されている場合、AWS Lake Formation 認証情報は AWS Lake Formation 登録テーブルの S3 データの読み取り/書き込みに使用され、ジョブのランタイムロール認証情報は AWS Lake Formation に登録されていないテーブルの読み取り/書き込みに使用されます。

Important

きめ細かなアクセスコントロールのために AWS Lake Formation を有効にしないでください。ジョブは、フルテーブルアクセス (FTA) ときめ細かなアクセスコントロール (FGAC) を同じ EMR クラスターまたはアプリケーションに実行することはできません。

ステップ 1: Lake Formation でフルテーブルアクセスを有効にする

フルテーブルアクセス (FTA) モードを使用するには、AWS Lake Formation の IAM セッションタグの検証なしでサードパーティーのクエリエンジンがデータにアクセスすることを許可する必要があります。有効にするには、「[フルテーブルアクセスのためのアプリケーション統合](#)」のステップに従います。

Note

クロスアカウントテーブルにアクセスする場合、プロデューサーアカウントとコンシューマーアカウントの両方でフルテーブルアクセスを有効にしなければなりません。同様に、クロスリージョンテーブルにアクセスする場合、この設定はプロデューサーリージョンとコンシューマーリージョンの両方で有効にしなければなりません。

ステップ 2: ジョブランタイムロールの IAM アクセス許可を設定する

基盤となるデータへの読み取りまたは書き込みアクセスには、Lake Formation アクセス許可に加えて、ジョブランタイムロールに `lakeformation:GetDataAccess` IAM アクセス許可が必要です。この許可があると、Lake Formation がデータにアクセスするための一時的な認証情報のリクエストを承諾します。

以下は、Amazon S3 のスクリプトにアクセスするための IAM アクセス許可、S3 AWS へのログのアップロード、Glue API アクセス許可、Lake Formation へのアクセス許可を指定する方法のポリシー例です。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ScriptAccess",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "arn:aws:s3:::*.amzn-s3-demo-bucket/scripts"
      ]
    }
  ]
}
```



```
    },
    {
      "Sid": "LoggingAccess",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::amzn-s3-demo-bucket/logs/*"
      ]
    },
    {
      "Sid": "GlueCatalogAccess",
      "Effect": "Allow",
      "Action": [
        "glue:Get*",
        "glue:Create*",
        "glue:Update*"
      ],
      "Resource": [
        "*"
      ]
    },
    {
      "Sid": "LakeFormationAccess",
      "Effect": "Allow",
      "Action": [
        "lakeformation:GetDataAccess"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```

ステップ 2.1 Lake Formation の許可を設定する

- S3 からデータを読み取る Spark ジョブには、Lake Formation SELECT アクセス許可が必要です。
- S3 でデータを書き込み/削除する Spark ジョブには、Lake Formation ALL (SUPER) アクセス許可が必要です。

- Glue Data を操作する Spark ジョブには、必要に応じて DESCRIBE、ALTER、DROP アクセス許可が必要です。

詳細については、「[Granting permissions on Data Catalog resources](#)」を参照してください。

ステップ 3: Lake Formation を使用してフルテーブルアクセスのための Spark セッションを初期化する

前提条件

AWS Glue Data Catalog は、Lake Formation テーブルにアクセスするためのメタストアとして設定する必要があります。

Glue カタログをメタストアとして設定するには、以下の設定を行います:

```
--conf spark.sql.catalogImplementation=hive
--conf
  spark.hive.metastore.client.factory.class=com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClient
```

EMR Serverless の Data Catalog の有効化の詳細については、「[Metastore configuration for EMR Serverless](#)」を参照してください。

AWS Lake Formation に登録されたテーブルにアクセスするには、Spark の初期化中に Lake Formation 認証情報を使用するように Spark を設定するには、次の設定を行う必要があります。

Hive

```
--conf
  spark.hadoop.fs.s3.credentialsResolverClass=com.amazonaws.glue.accesscontrol.AWSLakeFormationCredentialsResolver
--conf spark.hadoop.fs.s3.useDirectoryHeaderAsFolderObject=true
--conf spark.hadoop.fs.s3.folderObject.autoAction.disabled=true
--conf spark.sql.catalog.skipLocationValidationOnCreateTable.enabled=true
--conf spark.sql.catalog.createDirectoryAfterTable.enabled=true
--conf spark.sql.catalog.dropDirectoryBeforeTable.enabled=true
```

Iceberg

```
--conf spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog
--conf spark.sql.catalog.spark_catalog.warehouse=S3_DATA_LOCATION
--conf spark.sql.catalog.spark_catalog.client.region=REGION
```

```
--conf spark.sql.catalog.spark_catalog.type=glue
--conf spark.sql.catalog.spark_catalog.glue.account-id=ACCOUNT_ID
--conf spark.sql.catalog.spark_catalog.glue.lakeformation-enabled=true
--conf spark.sql.catalog.dropDirectoryBeforeTable.enabled=true
```

Delta Lake

```
--conf
  spark.hadoop.fs.s3.credentialsResolverClass=com.amazonaws.glue.accesscontrol.AWSLakeFormationCredentialsProvider
--conf spark.hadoop.fs.s3.useDirectoryHeaderAsFolderObject=true
--conf spark.hadoop.fs.s3.folderObject.autoAction.disabled=true
--conf spark.sql.catalog.skipLocationValidationOnCreateTable.enabled=true
--conf spark.sql.catalog.createDirectoryAfterTable.enabled=true
--conf spark.sql.catalog.dropDirectoryBeforeTable.enabled=true
```

Hudi

```
--conf
  spark.hadoop.fs.s3.credentialsResolverClass=com.amazonaws.glue.accesscontrol.AWSLakeFormationCredentialsProvider
--conf spark.hadoop.fs.s3.useDirectoryHeaderAsFolderObject=true
--conf spark.hadoop.fs.s3.folderObject.autoAction.disabled=true
--conf spark.sql.catalog.skipLocationValidationOnCreateTable.enabled=true
--conf spark.sql.catalog.createDirectoryAfterTable.enabled=true
--conf spark.sql.catalog.dropDirectoryBeforeTable.enabled=true
--conf spark.jars=/usr/lib/hudi/hudi-spark-bundle.jar
--conf spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension
--conf
  spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog
--conf spark.serializer=org.apache.spark.serializer.KryoSerializer
```

- `spark.hadoop.fs.s3.credentialsResolverClass=com.amazonaws.glue.accesscontrol.AWSLakeFormationCredentialsProvider`
Lake Formation 登録済みテーブルに AWS Lake Formation S3 認証情報を使用するように EMR Filesystem (EMRFS) または EMR S3A を設定します。S3 テーブルが登録されていない場合は、ジョブのランタイムロール認証情報を使用します。
- `spark.hadoop.fs.s3.useDirectoryHeaderAsFolderObject=true` および `spark.hadoop.fs.s3.folderObject.autoAction.disabled=true`: S3 フォルダの作成時に \$folder\$ サフィックスの代わりにコンテンツタイプヘッダー `application/x-directory` を使用するように EMRFS を設定します。Lake Formation 認証情報は \$folder\$ サフィックスを持つテーブルフォルダの読み取りを許可しないため、この設定は Lake Formation テーブルを読み取る際に必要です。

- `spark.sql.catalog.skipLocationValidationOnCreateTable.enabled=true`: 作成前に空のテーブルの場所を検証するステップをスキップするよう Spark を設定します。空の場所を検証するための Lake Formation 認証情報は Glue Data Catalog テーブルの作成後にしか使用できないため、この設定は Lake Formation に登録されたテーブルに対して必要です。この設定がない場合、ジョブのランタイムロール認証情報によって空のテーブルの場所が検証されます。
- `spark.sql.catalog.createDirectoryAfterTable.enabled=true`: Hive メタストアでテーブルを作成した後に Amazon S3 フォルダを作成するよう Spark を設定します。S3 フォルダを作成するための Lake Formation 認証情報は Glue Data Catalog テーブルの作成後にしか使用できないため、この設定は Lake Formation に登録されたテーブルに対して必要です。
- `spark.sql.catalog.dropDirectoryBeforeTable.enabled=true`: Hive メタストアでテーブルを削除する前に S3 フォルダを削除するよう Spark を設定します。Glue Data Catalog からテーブルを削除した後は、S3 フォルダを削除するための Lake Formation 認証情報を使用できないため、この設定は Lake Formation に登録されたテーブルに対して必要です。
- `spark.sql.catalog.<catalog>.glue.lakeformation-enabled=true`: AWS Lake Formation 登録済みテーブルに Lake Formation S3 認証情報を使用するように Iceberg カタログを設定します。テーブルが登録されていない場合は、デフォルトの環境認証情報を使用します。

SageMaker Unified Studio でフルテーブルアクセスモードを設定する

JupyterLab ノートブックのインタラクティブ Spark セッションから Lake Formation に登録されたテーブルにアクセスするには、互換性アクセス許可モードを使用します。%%configure magic コマンドを使用して Spark 設定をセットアップします。次のテーブルタイプに基づいて設定を選択します。

For Hive tables

```
%%configure -f
{
  "conf": {
    "spark.hadoop.fs.s3.credentialsResolverClass":
    "com.amazonaws.glue.accesscontrol.AWSLakeFormationCredentialResolver",
    "spark.hadoop.fs.s3.useDirectoryHeaderAsFolderObject": true,
    "spark.hadoop.fs.s3.folderObject.autoAction.disabled": true,
    "spark.sql.catalog.skipLocationValidationOnCreateTable.enabled": true,
    "spark.sql.catalog.createDirectoryAfterTable.enabled": true,
    "spark.sql.catalog.dropDirectoryBeforeTable.enabled": true
  }
}
```

For Iceberg tables

```
%%configure -f
{
  "conf": {
    "spark.sql.catalog.spark_catalog":
"org.apache.iceberg.spark.SparkSessionCatalog",
    "spark.sql.catalog.spark_catalog.warehouse": "S3_DATA_LOCATION",
    "spark.sql.catalog.spark_catalog.client.region": "REGION",
    "spark.sql.catalog.spark_catalog.type": "glue",
    "spark.sql.catalog.spark_catalog.glue.account-id": "ACCOUNT_ID",
    "spark.sql.catalog.spark_catalog.glue.lakeformation-enabled": "true",
    "spark.sql.catalog.dropDirectoryBeforeTable.enabled": "true",
  }
}
```

For Delta Lake tables

```
%%configure -f
{
  "conf": {
    "spark.hadoop.fs.s3.credentialsResolverClass":
"com.amazonaws.glue.accesscontrol.AWSLakeFormationCredentialResolver",
    "spark.hadoop.fs.s3.useDirectoryHeaderAsFolderObject": true,
    "spark.hadoop.fs.s3.folderObject.autoAction.disabled": true,
    "spark.sql.catalog.skipLocationValidationOnCreateTable.enabled": true,
    "spark.sql.catalog.createDirectoryAfterTable.enabled": true,
    "spark.sql.catalog.dropDirectoryBeforeTable.enabled": true
  }
}
```

For Hudi tables

```
%%configure -f
{
  "conf": {
    "spark.hadoop.fs.s3.credentialsResolverClass":
"com.amazonaws.glue.accesscontrol.AWSLakeFormationCredentialResolver",
    "spark.hadoop.fs.s3.useDirectoryHeaderAsFolderObject": true,
    "spark.hadoop.fs.s3.folderObject.autoAction.disabled": true,
    "spark.sql.catalog.skipLocationValidationOnCreateTable.enabled": true,
    "spark.sql.catalog.createDirectoryAfterTable.enabled": true,
    "spark.sql.catalog.dropDirectoryBeforeTable.enabled": true,
  }
}
```

```
"spark.jars": "/usr/lib/hudi/hudi-spark-bundle.jar",
"spark.sql.extensions":
"org.apache.spark.sql.hudi.HoodieSparkSessionExtension",
"spark.sql.catalog.spark_catalog":
"org.apache.spark.sql.hudi.catalog.HoodieCatalog",
"spark.serializer": "org.apache.spark.serializer.KryoSerializer"
}
}
```

プレースホルダーを置き換えます:

- S3_DATA_LOCATION: S3 バケットパス
- REGION: AWS region (us-east-1 など)
- ACCOUNT_ID: AWS アカウント ID

 Note

ノートブックで Spark 操作を実行する前に、これらの設定を行う必要があります。

サポートされているオペレーション

これらのオペレーションでは、AWS Lake Formation 認証情報を使用してテーブルデータにアクセスします。

- CREATE TABLE
- ALTER TABLE
- INSERT INTO
- INSERT OVERWRITE
- UPDATE
- MERGE INTO
- DELETE FROM
- ANALYZE TABLE
- REPAIR TABLE
- DROP TABLE

- Spark データソーススクリ
- Spark データソース書き込み

Note

上記に示されていない操作は、引き続き IAM アクセス許可を使用してテーブルデータにアクセスします。

考慮事項

- フルテーブルアクセスが有効になっていないジョブを使用して Hive テーブルが作成され、レコードが挿入されていない場合、フルテーブルアクセスを持つジョブからの後続の読み取りまたは書き込みは失敗します。これは、フルテーブルアクセスを持たない EMR Spark がテーブルフォルダ名に \$folder\$ サフィックスを追加するためです。この問題は、以下のいずれかの方法で解決できます。
 - FTA が有効になっていないジョブからテーブルに少なくとも 1 つの行を挿入します。
 - S3 のフォルダ名に \$folder\$ サフィックスが使用されないようにするため、FTA が有効になっていないジョブを設定します。そのために、Spark の設定を `spark.hadoop.fs.s3.useDirectoryHeaderAsFolderObject=true` にします。
 - S3 コンソールまたは AWS S3 CLI `s3://path/to/table/table_name` を使用して、テーブルの場所に AWS S3 フォルダを作成します。
- フルテーブルアクセスは、Amazon EMR リリース 7.8.0 以降の EMR ファイルシステム (EMRFS)、および Amazon EMR リリース 7.10.0 以降の S3A ファイルシステムでサポートされています。
- フルテーブルアクセスは、Hive、Iceberg、Delta、および Hudi テーブルでサポートされています。
- Hudi FTA 書き込みサポートに関する考慮事項:
 - Hudi FTA 書き込みでは、ジョブ実行中の認証情報の払い出しに `HoodieCredentialedHadoopStorage` を使用する必要があります。Hudi ジョブを実行するときは、次の設定を行います。
`hoodie.storage.class=org.apache.spark.sql.hudi.storage.HoodieCredentialedHadoopStorage`
 - Hudi のフルテーブルアクセス (FTA) 書き込みサポートは、Amazon EMR リリース 7.12 以降で利用できます。

- Hudi FTA 書き込みサポートは現在、デフォルトの Hudi 設定でのみ機能します。カスタムまたはデフォルト以外の Hudi 設定は完全にサポートされていない可能性があり、予期しない動作が発生する可能性があります。
- Hudi Merge-On-Read (MOR) テーブルのクラスタリングは、現時点では FTA 書き込みモードではサポートされていません。
- Lake Formation きめ細かなアクセスコントロール (FGAC) ルールまたは Glue Data Catalog ビューを持つテーブルを参照するジョブは失敗します。FGAC ルールまたは Glue Data Catalog ビューを使用してテーブルをクエリするには、FGAC モードを使用する必要があります。FGAC モードを有効にするには、AWS ドキュメントで説明されている手順に従います。[AWS Lake Formation で EMR Serverless を使用してきめ細かなアクセスコントロール](#)を行います。
- フルテーブルアクセスは Spark ストリーミングをサポートしていません。
- Lake Formation テーブルに Spark DataFrame を書き込む場合、Hive テーブルと Iceberg テーブルでは APPEND モードのみがサポートされます:
`df.write.mode("append").saveAsTable(table_name)`
- 外部テーブルを作成するには、IAM アクセス許可が必要です。
- Lake Formation は Spark ジョブ内に認証情報を一時的にキャッシュするため、現在実行中の Spark バッチジョブまたはインタラクティブセッションには、アクセス許可の変更が反映されない場合があります。
- ロールには、サービスにリンクされたロール: [Lake Formation の要件ではなく、ユーザー定義のロール](#)を使用する必要があります。

Hudi FTA 書き込みサポート - サポートされているオペレーション

次の表は、フルテーブルアクセスモードで Hudi Copy-On-Write (COW) テーブルと Merge-On-Read (MOR) テーブルでサポートされている書き込みオペレーションを示しています。

Hudi FTA でサポートされている書き込みオペレーション

テーブルタイプ	運用	SQL Write コマンド	ステータス
COW	INSERT	INSERT INTO TABLE	サポート
COW	INSERT	INSERT INTO TABLE -	サポート

テーブルタイプ	運用	SQL Write コマンド	ステータス
		PARTITION (静的、動的)	
COW	INSERT	INSERT OVERWRITE	サポート
COW	INSERT	INSERT OVERWRITE - PARTITION (静的、動的)	サポート
UPDATE	UPDATE	UPDATE TABLE	サポート
COW	UPDATE	UPDATE TABLE - パーティション の変更	サポート外
DELETE	DELETE	テーブルから削除	サポート
ALTER	ALTER	テーブルの変更 - 名前を に変更する	サポート外
COW	ALTER	テーブルの変更 - TBLPROPERTIES の設定	サポート
COW	ALTER	テーブルの変更 - TBLPROPERTIES の設定を解除する	サポート
COW	ALTER	テーブルの変更 - 列の変更	サポート

テーブルタイプ	運用	SQL Write コマンド	ステータス
COW	ALTER	テーブルの変更 - 列の追加	サポート
COW	ALTER	テーブルの変更 - パーティションの追加	サポート
COW	ALTER	テーブルの変更 - パーティションの削除	サポート
COW	ALTER	テーブルの変更 - パーティションの復元	サポート
COW	ALTER	テーブル同期 パーティションの修復	サポート
DROP	DROP	DROP TABLE	サポート
COW	DROP	DROP TABLE - 購入	サポート
CREATE	CREATE	CREATE TABLE - マネージド	サポート
COW	CREATE	テーブルの作成 - によるパーティション分割	サポート
COW	CREATE	テーブルが存在しない場合の作成	サポート

テーブルタイプ	運用	SQL Write コマンド	ステータス
COW	CREATE	CREATE TABLE LIKE	サポート
COW	CREATE	CREATE TABLE AS SELECT	サポート
CREATE	CREATE	CREATE TABLE with LOCATION - 外部テーブル	サポート外
DATAFRAME (INSERT)	DATAFRAME(INSERT)	saveAsTable.Overwrite	サポート
COW	DATAFRAME(INSERT)	saveAsTable.Append	サポート外
COW	DATAFRAME(INSERT)	saveAsTable.Ignore	サポート
COW	DATAFRAME(INSERT)	saveAsTable.ErrorIfExists	サポート
COW	DATAFRAME(INSERT)	saveAsTable - 外部テーブル (パス)	サポート外
COW	DATAFRAME(INSERT)	save(path) - DF v1	サポート外
MOR	INSERT	INSERT INTO TABLE	サポート
MOR	INSERT	INSERT INTO TABLE - PARTITION (静的、動的)	サポート

テーブルタイプ	運用	SQL Write コマンド	ステータス
MOR	INSERT	INSERT OVERWRITE	サポート
MOR	INSERT	INSERT OVERWRITE - PARTITION (静的、動的)	サポート
UPDATE	UPDATE	UPDATE TABLE	サポート
MOR	UPDATE	UPDATE TABLE - パーティションの変更	サポート外
DELETE	DELETE	テーブルから削除	サポート
ALTER	ALTER	テーブルの変更 - 名前を に変更する	サポート外
MOR	ALTER	テーブルの変更 - TBLPROPERTIES の設定	サポート
MOR	ALTER	テーブルの変更 - TBLPROPERTIES の設定を解除する	サポート
MOR	ALTER	テーブルの変更 - 列の変更	サポート
MOR	ALTER	テーブルの変更 - 列の追加	サポート

テーブルタイプ	運用	SQL Write コマンド	ステータス
MOR	ALTER	テーブルの変更 - パーティションの追加	サポート
MOR	ALTER	テーブルの変更 - パーティションの削除	サポート
MOR	ALTER	テーブルの変更 - パーティションの復元	サポート
MOR	ALTER	テーブル同期 パーティションの修復	サポート
DROP	DROP	DROP TABLE	サポート
MOR	DROP	DROP TABLE - 購入	サポート
CREATE	CREATE	CREATE TABLE - マネージド	サポート
MOR	CREATE	テーブルの作成 - によるパーティション分割	サポート
MOR	CREATE	テーブルが存在しない場合は作成	サポート
MOR	CREATE	CREATE TABLE LIKE	サポート

テーブルタイプ	運用	SQL Write コマンド	ステータス
MOR	CREATE	CREATE TABLE AS SELECT	サポート
CREATE	CREATE	CREATE TABLE with LOCATION - 外部テーブル	サポート外
DATAFRAME (UPSERT)	DATAFRAME (UPSERT)	saveAsTable.Overwrite	サポート
MOR	DATAFRAME (UPSERT)	saveAsTable.Append	サポート外
MOR	DATAFRAME (UPSERT)	saveAsTable.Ignore	サポート
MOR	DATAFRAME (UPSERT)	saveAsTable.ErrorIfExists	サポート
MOR	DATAFRAME (UPSERT)	saveAsTable - 外部テーブル (パス)	サポート外
MOR	DATAFRAME (UPSERT)	save(path) - DF v1	サポート外
DATAFRAME (削除)	DATAFRAME (削除)	saveAsTable.Append	サポート外
MOR	DATAFRAME (削除)	saveAsTable - 外部テーブル (パス)	サポート外
MOR	DATAFRAME (削除)	save(path) - DF v1	サポート外

テーブルタイプ	運用	SQL Write コマンド	ステータス
DATAFRAME (BULK_INSERT)	DATAFRAME(BULK_INSERT)	saveAsTable.Overwrite	サポート
MOR	DATAFRAME(BULK_INSERT)	saveAsTable.Append	サポート外
MOR	DATAFRAME(BULK_INSERT)	saveAsTable.Ignore	サポート
MOR	DATAFRAME(BULK_INSERT)	saveAsTable.ErrorIfExists	サポート
MOR	DATAFRAME(BULK_INSERT)	saveAsTable - 外部テーブル (パス)	サポート外
MOR	DATAFRAME(BULK_INSERT)	save(path) - DF v1	サポート外

での EMR Serverless を使用したときめ AWS Lake Formation 細かなアクセスコントロール

概要:

Amazon EMR リリース 7.2.0 以降では、AWS Lake Formation を活用して、S3 でバックアップされた Data Catalog テーブルにきめ細かなアクセスコントロールを適用します。この機能を使用すると、Amazon EMR Serverless Spark ジョブ内の read クエリにテーブル、行、列、セルレベルのアクセスコントロールを設定できます。Apache Spark バッチジョブとインタラクティブセッションできめ細かなアクセスコントロールを設定するには、EMR Studio を使用します。Lake Formation の詳細と EMR Serverless での使用方法については、以下のセクションを参照してください。

で Amazon EMR Serverless を使用すると、追加料金 AWS Lake Formation が発生します。詳細については、「[Amazon EMR の料金](#)」を参照してください。

EMR Serverless と の連携方法 AWS Lake Formation

EMR Serverless と Lake Formation を使用すると、各 Spark ジョブにアクセス許可のレイヤーを適用して、EMR Serverless がジョブを実行するときに Lake Formation アクセス許可コントロールを適用できます。EMR Serverless は、[Spark リソースプロファイル](#)を使用して 2 つのプロファイルを作成し、ジョブを効果的に実行します。ユーザープロファイルはユーザー提供のコードを実行し、システムプロファイルは Lake Formation ポリシーを適用します。詳細については、「[AWS Lake Formation とは](#)」および「[考慮事項と制限事項](#)」を参照してください。

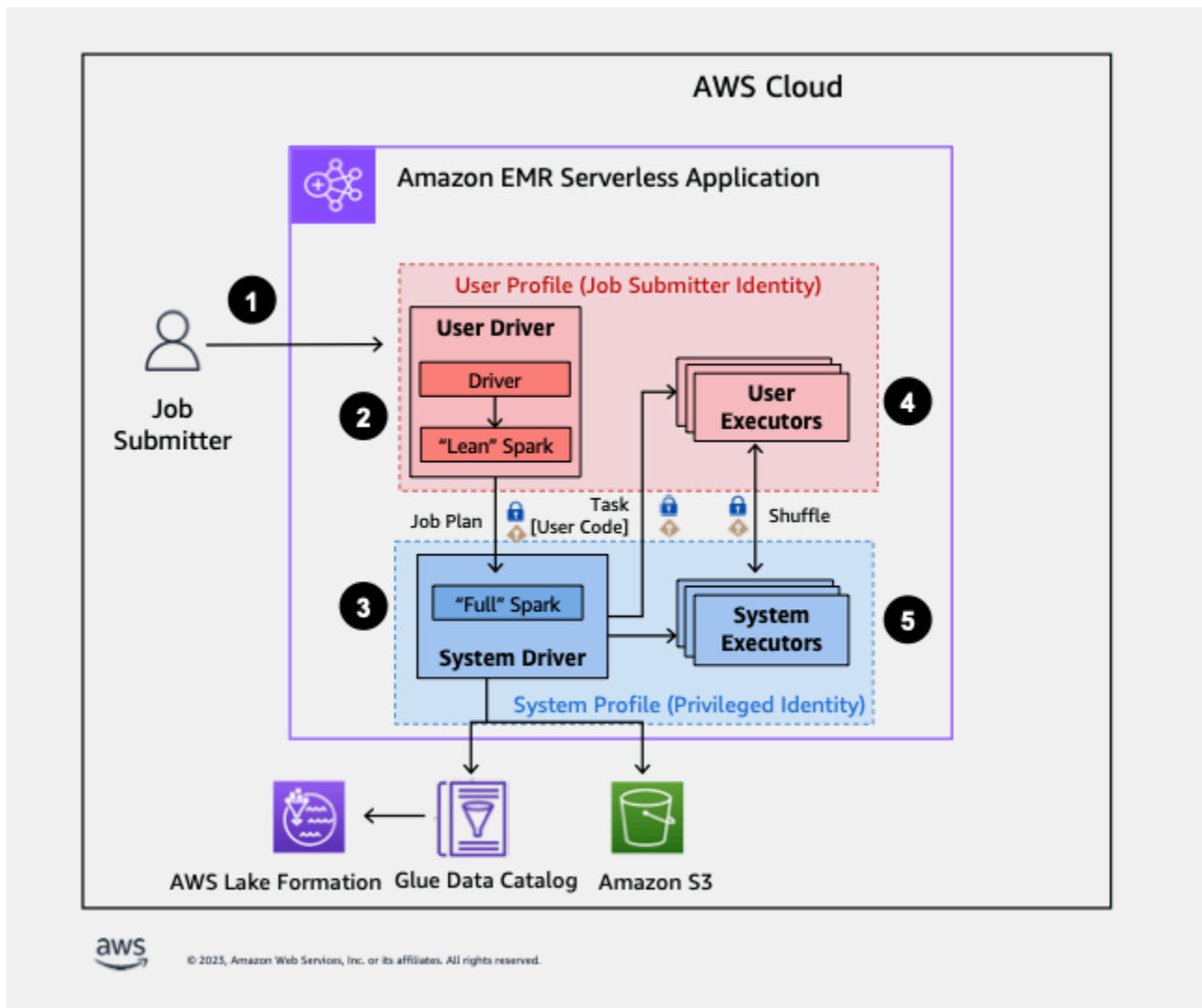
Lake Formation で事前初期化された容量を使用する場合は、少なくとも 2 つの Spark ドライバーを使用することを提案します。Lake Formation 対応ジョブごとに 2 つの Spark ドライバー (1 つはユーザープロファイル用、1 つはシステムプロファイル用) を使用します。最高のパフォーマンスを得るには、Lake Formation を使用しない場合と比較して、Lake Formation 対応ジョブのドライバーの数を 2 倍にします。

EMR Serverless で Spark ジョブを実行するときは、リソース管理とクラスターパフォーマンスに対する動的割り当ての影響も考慮します。リソースプロファイルあたりのエグゼキューターの最大数である `spark.dynamicAllocation.maxExecutors` の設定は、ユーザーエグゼキューターとシステムエグゼキューターに適用されます。その数を最大許容エグゼキューター数と等しく設定すると、利用可能なすべてのリソースを使用する 1 つのタイプのエグゼキューターが原因でジョブの実行が停止し、ジョブの実行時に他のエグゼキューターが使用できなくなる可能性があります。

したがって、リソースが不足しないように、EMR Serverless はリソースプロファイルあたりのデフォルトの最大エグゼキューター数を `spark.dynamicAllocation.maxExecutors` 値の 90% に設定します。この設定は、`spark.dynamicAllocation.maxExecutorsRatio` を 0 から 1 までの値で指定すると上書きできます。さらに、リソースの割り当てと全体的なパフォーマンスを最適化するために、次のプロパティを設定します。

- `spark.dynamicAllocation.cachedExecutorIdleTimeout`
- `spark.dynamicAllocation.shuffleTracking.timeout`
- `spark.cleaner.periodicGC.interval`

以下は、EMR Serverless が Lake Formation セキュリティポリシーで保護されたデータにアクセスする方法の概要を示します。



1. ユーザーは、Spark ジョブを AWS Lake Formation 有効な EMR Serverless アプリケーションに送信します。
2. EMR Serverless はジョブをユーザードライバーに送信し、ユーザープロファイルでジョブを実行します。ユーザードライバーは、タスクの起動、エグゼキューターのリクエスト、S3 または Glue カタログへのアクセスができない Spark のリーンバージョンを実行します。ジョブプランを構築します。
3. EMR Serverless は、システムドライバーと呼ばれる 2 番目のドライバーを設定し、システムプロファイルで (特権 ID を使用して) 実行します。EMR Serverless は、通信用の 2 つのドライバー間に暗号化された TLS チャンネルを設定します。ユーザードライバーはチャンネルを使用して、ジョブプランをシステムドライバーに送信します。システムドライバーは、ユーザーが送信したコードを実行しません。フル Spark を実行して、データアクセスのために S3 およびデータ

カタログと通信します。エグゼキュターをリクエストし、ジョブプランを一連の実行ステージにコンパイルします。

- 次に、EMR Serverless はユーザードライバーまたはシステムドライバを使用してエグゼキュターでステージを実行します。どのステージのユーザーコードも、ユーザープロファイルのエグゼキュターでのみ実行されます。
- で保護された Data Catalog テーブルからデータを読み取るステージ AWS Lake Formation、またはセキュリティフィルターを適用するステージは、システムエグゼキュターに委任されます。

Amazon EMR での Lake Formation の有効化

Lake Formation を有効にするには、[EMR Serverless アプリケーションの作成](#)時に、ランタイム設定パラメータの `spark-defaults` 分類で `spark.emr-serverless.lakeformation.enabled` を `true` に設定します。

```
aws emr-serverless create-application \  
  --release-label emr-7.12.0 \  
  --runtime-configuration '{  
    "classification": "spark-defaults",  
    "properties": {  
      "spark.emr-serverless.lakeformation.enabled": "true"  
    }  
  }' \  
  --type "SPARK"
```

EMR Studio で新しいアプリケーションを作成するときに Lake Formation を有効にすることもできます。[追加の設定] で利用可能な [Lake Formation のきめ細かなアクセスコントロールを使用する] を選択します。

EMR Serverless で Lake Formation を使用することにより、[ワーカー間の暗号化](#)がデフォルトで有効になるため、ワーカー間の暗号化を再度明示的に有効にする必要はありません。

Spark ジョブの Lake Formation の有効化

個々の Spark ジョブの Lake Formation を有効にするには、`spark-submit` の使用時に `spark.emr-serverless.lakeformation.enabled` を `true` に設定します。

```
--conf spark.emr-serverless.lakeformation.enabled=true
```

ジョブランタイムロールの IAM アクセス許可

Lake Formation のアクセス許可は、Glue Data Catalog AWS リソース、Amazon S3 ロケーション、およびそれらのロケーションの基盤となるデータへのアクセスを制御します。IAM アクセス許可は、Lake Formation および AWS Glue API とリソースへのアクセスを制御します。データカタログ内のテーブルにアクセスするための Lake Formation アクセス許可 (SELECT) を持っていない場合、glue:Get* API オペレーションに対する IAM アクセス許可がない場合、操作は失敗します。

以下は、S3 のスクリプトにアクセスするための IAM アクセス許可、S3 AWS へのログのアップロード、Glue API アクセス許可、Lake Formation へのアクセス許可を指定する方法のポリシー例です。

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ScriptAccess",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3::*.amzn-s3-demo-bucket/scripts",
        "arn:aws:s3::*.amzn-s3-demo-bucket/*"
      ]
    },
    {
      "Sid": "LoggingAccess",
      "Effect": "Allow",
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::amzn-s3-demo-bucket/logs/*"
      ]
    },
    {
      "Sid": "GlueCatalogAccess",
      "Effect": "Allow",
      "Action": [
```

```
        "glue:Get*",
        "glue:Create*",
        "glue:Update*"
    ],
    "Resource": [
        "*"
    ]
},
{
    "Sid": "LakeFormationAccess",
    "Effect": "Allow",
    "Action": [
        "lakeformation:GetDataAccess"
    ],
    "Resource": [
        "*"
    ]
}
]
```

ジョブランタイムロールの Lake Formation アクセス許可の設定

まず、Hive テーブルの場所を Lake Formation に登録します。次に、目的のテーブルにジョブランタイムロールのアクセス許可を作成します。Lake Formation の詳細については、「AWS Lake Formation デベロッパーガイド」の「[What is AWS Lake Formation?](#)」を参照してください。

Lake Formation アクセス許可を設定したら、Amazon EMR Serverless で Spark ジョブを送信します。Spark ジョブの詳細については、「[Spark example](#)」を参照してください。

ジョブ実行の送信

Lake Formation 許可の設定が完了したら、[EMR Serverless で Spark ジョブを送信](#)できます。次のセクションでは、ジョブ実行プロパティを設定して送信する方法の例を示します。

アクセス許可の要件

に登録されていないテーブル AWS Lake Formation

に登録されていないテーブルの場合 AWS Lake Formation、ジョブランタイムロールは AWS Glue データカタログと Amazon S3 の基盤となるテーブルデータの両方にアクセスします。これには、

ジョブランタイムロールに Glue オペレーションと Amazon AWS S3 オペレーションの両方に対する適切な IAM アクセス許可が必要です。Amazon S3

に登録されたテーブル AWS Lake Formation

に登録されたテーブルの場合 AWS Lake Formation、ジョブランタイムロールは AWS Glue データカタログメタデータにアクセスし、Lake Formation によって提供された一時的な認証情報は Amazon S3 の基盤となるテーブルデータにアクセスします。オペレーションの実行に必要な Lake Formation のアクセス許可は、Spark ジョブが開始する AWS Glue データカタログと Amazon S3 API コールに依存し、次のように要約できます。

- DESCRIBE アクセス許可により、ランタイムロールは Data Catalog 内のテーブルまたはデータベースメタデータを読み取ることができます。
- ALTER アクセス許可により、ランタイムロールは Data Catalog のテーブルまたはデータベースメタデータを変更できます。
- DROP アクセス許可により、ランタイムロールは Data Catalog からテーブルまたはデータベースメタデータを削除できます。
- SELECT アクセス許可により、ランタイムロールは Amazon S3 からテーブルデータを読み取ることができます。
- INSERT アクセス許可により、ランタイムロールは Amazon S3 にテーブルデータを書き込むことができます。
- DELETE アクセス許可により、ランタイムロールは Amazon S3 からテーブルデータを削除できます。

 Note

Lake Formation は、Spark ジョブが AWS Glue を呼び出してテーブルメタデータを取得し、Amazon S3 を呼び出してテーブルデータを取得するときに、アクセス許可を遅延的に評価します。アクセス許可が不十分なランタイムロールを使用するジョブは、Spark が不足しているアクセス許可を必要とする AWS Glue または Amazon S3 呼び出しを行うまで失敗しません。

 Note

次のサポートされているテーブルマトリックス:

- サポート対象とマークされたオペレーションは、Lake Formation 認証情報のみを使用して、Lake Formation に登録されたテーブルのテーブルデータにアクセスします。Lake Formation のアクセス許可が不十分な場合、オペレーションはランタイムロールの認証情報にフォールバックされません。Lake Formation に登録されていないテーブルの場合、ジョブランタイムロールの認証情報はテーブルデータにアクセスします。
- Amazon S3 ロケーションで IAM アクセス許可でサポートとマークされたオペレーションは、Amazon S3 の基盤となるテーブルデータにアクセスするために Lake Formation 認証情報を使用しません。これらのオペレーションを実行するには、テーブルが Lake Formation に登録されているかどうかにかかわらず、ジョブランタイムロールにテーブルデータにアクセスするために必要な Amazon S3 IAM アクセス許可が必要です。

Hive

運用	AWS Lake Formation アクセス許可	サポートステータス
SELECT	SELECT	サポート
CREATE TABLE	CREATE_TABLE	サポート
CREATE TABLE LIKE	CREATE_TABLE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
CREATE TABLE AS SELECT	CREATE_TABLE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
DESCRIBE TABLE	DESCRIBE	サポート
SHOW TBLPROPERTIES	DESCRIBE	サポート
SHOW COLUMNS	DESCRIBE	サポート
SHOW PARTITIONS	DESCRIBE	サポート
SHOW CREATE TABLE	DESCRIBE	サポート

運用	AWS Lake Formation アクセス許可	サポートステータス
テーブルの変更 tablename	SELECT と ALTER	サポート
テーブルtablename セットの場所の変更	-	サポートされていません
テーブルtablename 追加パーティションの変更	SELECT、INSERT、ALTER	サポート
REPAIR TABLE	SELECT と ALTER	サポート
データを読み込む		サポートされていません
INSERT	INSERT と ALTER	サポート
INSERT OVERWRITE	SELECT、INSERT、DELETE、ALTER	サポート
DROP TABLE	SELECT、DROP、DELETE、ALTER	サポート
TRUNCATE TABLE	SELECT、INSERT、DELETE、ALTER	サポート
データフレームライター V1	対応する SQL オペレーションと同じ	既存のテーブルにデータを追加するときにサポートされます。詳細については、 「考慮事項と制限事項」 を参照してください。
データフレームライター V2	対応する SQL オペレーションと同じ	既存のテーブルにデータを追加するときにサポートされます。詳細については、 「考慮事項と制限事項」 を参照してください。

Iceberg

運用	AWS Lake Formation アクセス許可	サポートステータス
SELECT	SELECT	サポート
CREATE TABLE	CREATE_TABLE	サポート
CREATE TABLE LIKE	CREATE_TABLE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
CREATE TABLE AS SELECT	CREATE_TABLE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
テーブルを SELECT に置き換える	SELECT、INSERT、ALTER	サポート
DESCRIBE TABLE	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
SHOW TBLPROPERTIES	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
SHOW CREATE TABLE	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
ALTER TABLE	SELECT、INSERT、ALTER	サポート
ALTER TABLE SET LOCATION	SELECT、INSERT、ALTER	Amazon S3 ロケーションでの IAM アクセス許可でサポート
によって順序付けられたテーブルの書き込みを変更する	SELECT、INSERT、ALTER	Amazon S3 ロケーションでの IAM アクセス許可でサポート
によって分散されたテーブルの書き込みを変更する	SELECT、INSERT、ALTER	Amazon S3 ロケーションでの IAM アクセス許可でサポート

運用	AWS Lake Formation アクセス許可	サポートステータス
テーブル名の変更テーブル	CREATE_TABLE、および DROP	サポート
INSERT INTO	SELECT、INSERT、ALTER	サポート
INSERT OVERWRITE	SELECT、INSERT、ALTER	サポート
DELETE	SELECT、INSERT、ALTER	サポート
UPDATE	SELECT、INSERT、ALTER	サポート
MERGE INTO	SELECT、INSERT、ALTER	サポート
DROP TABLE	SELECT、DELETE、DROP	サポート
DataFrame ライター V1	-	サポートされていません
DataFrame ライター V2	対応する SQL オペレーションと同じ	既存のテーブルにデータを追加するときにサポートされます。詳細については、「 考慮事項と制限事項 」を参照してください。
メタデータテーブル	SELECT	サポート対象。特定のテーブルは非表示になります。詳細については、「 考慮事項と制限事項 」を参照してください。

運用	AWS Lake Formation アクセス許可	サポートステータス
ストアドプロシージャ	-	次の条件を満たすテーブルでサポートされています。 - に登録されていないテーブル AWS Lake Formation - register_table とを使用しないテーブル migrate 詳細については、「考慮事項と制限事項」を参照してください。

Iceberg の Spark 設定: 以下の例は、Iceberg で Spark を設定する方法を示しています。Iceberg ジョブを実行するには、次の spark-submit プロパティを指定します。

```
--conf spark.sql.catalog.spark_catalog=org.apache.iceberg.spark.SparkSessionCatalog
--conf spark.sql.catalog.spark_catalog.warehouse=<S3_DATA_LOCATION>
--conf spark.sql.catalog.spark_catalog.glue.account-id=<ACCOUNT_ID>
--conf spark.sql.catalog.spark_catalog.client.region=<REGION>
--conf spark.sql.catalog.spark_catalog.glue.endpoint=https://
glue.<REGION>.amazonaws.com
```

Hudi

運用	AWS Lake Formation アクセス許可	サポートステータス
SELECT	SELECT	サポート
CREATE TABLE	CREATE_TABLE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
CREATE TABLE LIKE	CREATE_TABLE	Amazon S3 ロケーションでの IAM アクセス許可でサポート

運用	AWS Lake Formation アクセス許可	サポートステータス
CREATE TABLE AS SELECT	-	サポートされていません
DESCRIBE TABLE	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
SHOW TBLPROPERTIES	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
SHOW COLUMNS	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
SHOW CREATE TABLE	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
ALTER TABLE	SELECT	Amazon S3 ロケーションでの IAM アクセス許可でサポート
INSERT INTO	SELECT と ALTER	Amazon S3 ロケーションでの IAM アクセス許可でサポート
INSERT OVERWRITE	SELECT と ALTER	Amazon S3 ロケーションでの IAM アクセス許可でサポート
DELETE	-	サポートされていません
UPDATE	-	サポートされていません
MERGE INTO	-	サポートされていません

運用	AWS Lake Formation アクセス許可	サポートステータス
DROP TABLE	SELECT と DROP	Amazon S3 ロケーションでの IAM アクセス許可でサポート
DataFrame ライター V1	-	サポートされていません
DataFrame ライター V2	対応する SQL オペレーションと同じ	Amazon S3 ロケーションでの IAM アクセス許可でサポート
メタデータテーブル	-	サポートされていません
テーブルのメンテナンスとユーティリティの機能	-	サポートされていません

以下の例では、Hudi を使用して Spark を設定し、ファイルの場所と使用に必要なその他のプロパティを指定します。

Hudi の Spark 設定: このスニペットをノートブックで使用するにより、Hudi Spark バンドル JAR ファイルへのパスが指定され、Spark の Hudi 機能が有効になります。また、Glue データカタログをメタストアとして使用するよう AWS Spark を設定します。

```
%%configure -f
{
  "conf": {
    "spark.jars": "/usr/lib/hudi/hudi-spark-bundle.jar",
    "spark.hadoop.hive.metastore.client.factory.class":
    "com.amazonaws.glue.catalog.metastore.AWSGlueDataCatalogHiveClientFactory",
    "spark.serializer": "org.apache.spark.serializer.JavaSerializer",
    "spark.sql.catalog.spark_catalog":
    "org.apache.spark.sql.hudi.catalog.HoodieCatalog",
    "spark.sql.extensions":
    "org.apache.spark.sql.hudi.HoodieSparkSessionExtension"
  }
}
```

AWS Glue を使用した Hudi の Spark 設定: このスニペットをノートブックで使用すると、Hudi はサポートされているデータレイク形式として有効になり、Hudi ライブラリと依存関係が利用可能になります。

```
%%configure
{
  "--conf": "spark.serializer=org.apache.spark.serializer.JavaSerializer --conf
spark.sql.catalog.spark_catalog=org.apache.spark.sql.hudi.catalog.HoodieCatalog --
conf
spark.sql.extensions=org.apache.spark.sql.hudi.HoodieSparkSessionExtension",
  "--datalake-formats": "hudi",
  "--enable-glue-datacatalog": True,
  "--enable-lakeformation-fine-grained-access": "true"
}
```

Delta Lake

運用	AWS Lake Formation アクセス許可	サポートステータス
SELECT	SELECT	サポート
CREATE TABLE	CREATE_TABLE	サポート
CREATE TABLE LIKE	-	サポートされていません
CREATE TABLE AS SELECT	CREATE_TABLE	サポート
テーブルを SELECT に置き 換える	SELECT、INSERT、ALTER	サポート
DESCRIBE TABLE	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
SHOW TBLPROPERTIES	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート

運用	AWS Lake Formation アクセス許可	サポートステータス
SHOW COLUMNS	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
SHOW CREATE TABLE	DESCRIBE	Amazon S3 ロケーションでの IAM アクセス許可でサポート
ALTER TABLE	SELECT と INSERT	サポート
ALTER TABLE SET LOCATION	SELECT と INSERT	Amazon S3 ロケーションでの IAM アクセス許可でサポート
テーブルtablename クラスターの変更	SELECT と INSERT	Amazon S3 ロケーションでの IAM アクセス許可でサポート
テーブルtablename 追加制約の変更	SELECT と INSERT	Amazon S3 ロケーションでの IAM アクセス許可でサポート
テーブルtablename ドロップ制約の変更	SELECT と INSERT	Amazon S3 ロケーションでの IAM アクセス許可でサポート
INSERT INTO	SELECT と INSERT	サポート
INSERT OVERWRITE	SELECT と INSERT	サポート
DELETE	SELECT と INSERT	サポート
UPDATE	SELECT と INSERT	サポート
MERGE INTO	SELECT と INSERT	サポート
DROP TABLE	SELECT、DELETE、DROP	サポート

運用	AWS Lake Formation アクセス許可	サポートステータス
DataFrame ライター V1	-	サポートされていません
DataFrame ライター V2	対応する SQL オペレーションと同じ	サポート
テーブルのメンテナンスとユーティリティの機能	-	サポートされていません

Delta Lake での EMR Serverless: EMR Serverless で Lake Formation で Delta Lake を使用するには、以下のコマンドを実行します。

```
spark-sql \
  --conf
  spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension,com.amazonaws.emr.recordserver
  \
  --conf
  spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog \
```

ジョブのデバッグ

Note

この機能を使用することにより、フィルタリングされていない機密情報を含む可能性のあるシステムプロファイルワーカーの stdout ログと stderr ログにアクセスできます。以下のアクセス許可は、非本番データへのアクセスにのみ使用してください。本番稼働用ジョブで使用するよう作成されたアプリケーションの場合、これらのアクセス許可は、昇格されたデータアクセスを持つ管理者またはユーザーにのみ追加することを強くお勧めします。

EMR-7.3.0 以降では、EMR Serverless は Lake Formation 対応バッチジョブの自己デバッグ機能を有効にします。そのためには、[GetDashboardForJobRun](#) API で新しいパラメータ `accessSystemProfileLogs` を使用します。`accessSystemProfileLogs` が `true` に設定されている場合、システムプロファイルワーカーの stdout ログと stderr ログにアクセスできます。このログは、Lake Formation が有効な EMR Serverless バッチジョブのデバッグに使用できます。

```
aws emr-serverless get-dashboard-for-job-run \  
  --application-id application-id \  
  --job-run-id job-run-id \  
  --access-system-profile-logs
```

必要なアクセス許可

GetDashboardForJobRun を使用して Lake Formation 対応バッチジョブをデバッグするプリンシパルには、以下の追加のアクセス許可が必要です:

```
{  
  "Sid": "AccessSystemProfileLogs",  
  "Effect": "Allow",  
  "Action": [  
    "emr-serverless:GetDashboardForJobRun",  
    "emr-serverless:AccessSystemProfileLogs",  
    "glue:GetDatabases",  
    "glue:SearchTables"  
  ],  
  "Resource": [  
    "arn:aws:emr-serverless:region:account-id:/applications/applicationId/  
jobruns/jobid",  
    "arn:aws:glue:region:account-id:catalog",  
    "arn:aws:glue:region:account-id:database/*",  
    "arn:aws:glue:region:account-id:table/*/*"  
  ]  
}
```

考慮事項

デバッグ用のシステムプロファイルログは、ジョブと同じアカウント内の Lake Formation のデータベースまたはテーブルにアクセスするジョブに表示されます。以下のシナリオでは表示されません:

- Lake Formation アクセス許可を使用して管理されるデータカタログにクロスアカウントデータベースとテーブルがある場合
- Lake Formation アクセス許可を使用して管理されるデータカタログにリソースリンクがある場合

Glue データカタログビューの使用

EMR Serverless で使用する AWS Glue データカタログでビューを作成および管理できます。これらは一般的に AWS Glue データカタログビューと呼ばれます。これらのビューは複数の SQL クエリエ

ンジンをサポートしているため、EMR Serverless、Amazon Athena Amazon Redshift などのさまざまな AWS サービスで同じビューにアクセスできます。

Data Catalog でビューを作成することで、デリソース許可とタグベースのアクセスコントロールを使用して、そのビューへのアクセス AWS Lake Formation を許可します。このアクセスコントロール方法を使用することにより、ビューの作成時に参照したテーブルへの追加のアクセスを設定する必要はありません。アクセス許可を付与する方法は、definer セマンティクスと呼ばれ、これらのビューは definer ビューと呼ばれます。Lake Formation でのアクセスコントロールの詳細については、AWS「[Lake Formation デベロッパーガイド](#)」の「[データカタログリソースに対するアクセス許可の付与と取り消し](#)」を参照してください。

データカタログビューは、次のユースケースに役立ちます。

- 細やかなアクセス制御 – ユーザーが必要とするアクセス許可に基づいてデータアクセスを制限するビューを作成できます。例えば、データカタログのビューを使用して、人事 (HR) 部門に属さない従業員が個人を特定できる情報 (PII) を表示できないようにすることができます。
- 完全なビュー定義 – データカタログのビューにフィルターを適用することで、データカタログビューで利用可能なデータレコードを常に完全な状態にすることができます。
- セキュリティの強化 – ビューの作成に使用されるクエリ定義は完全である必要があります。このメリットにより、データカタログのビューは悪意のあるアクターからの SQL コマンドの影響を受けにくくなります。
- データを簡単に共有 – データを移動することなく、他の AWS アカウントとデータを共有します。詳細については、「[Cross-account data sharing in Lake Formation](#)」を参照してください。

データカタログビューの作成

Data Catalog ビューを作成するには、さまざまな方法があります。これには、AWS CLI または Spark SQL の使用が含まれます。以下にいくつかの例を示します。

Using SQL

以下は、データカタログビューを作成するための構文を示しています。MULTI DIALECT ビュータイプを書き留めます。これにより、Data Catalog ビューと他のビューが区別されます。SECURITY 述語は DEFINER として指定されます。これは、DEFINER セマンティクスを含む Data Catalog ビューを示します。

```
CREATE [ OR REPLACE ] PROTECTED MULTI DIALECT VIEW [IF NOT EXISTS] view_name
[(column_name [COMMENT column_comment], ...)]
```

```
[ COMMENT view_comment ]
[TBLPROPERTIES (property_name = property_value, ... )]
SECURITY DEFINER
AS query;
```

以下は、構文に従った CREATE ステートメントの例です:

```
CREATE PROTECTED MULTI DIALECT VIEW catalog_view
SECURITY DEFINER
AS
SELECT order_date, sum(totalprice) AS price
FROM source_table
GROUP BY order_date
```

実際にリソースを作成することなく、SQL を使用してドライランモードでビューを作成し、ビューの作成をテストすることもできます。このオプションを使用すると、入力を検証する「ドライラン」になり、検証が成功すると、ビューを表す AWS Glue テーブルオブジェクトの JSON が返されます。この場合、実際のビューは作成されません。

```
CREATE [ OR REPLACE ] PROTECTED MULTI DIALECT VIEW view_name
SECURITY DEFINER
[ SHOW VIEW JSON ]
AS view-sql
```

Using the AWS CLI

Note

CLI コマンドを使用する場合、ビューの作成に使用される SQL は解析されません。これにより、ビューは作成されますが、クエリは成功しません。ビューを作成する前に、必ず SQL 構文をテストしてください。

以下の CLI コマンドを使用して、ビューを作成します:

```
aws glue create-table --cli-input-json '{
  "DatabaseName": "database",
  "TableInput": {
    "Name": "view",
    "StorageDescriptor": {
```

```

    "Columns": [
      {
        "Name": "col1",
        "Type": "data-type"
      },
      ...
      {
        "Name": "col_n",
        "Type": "data-type"
      }
    ],
    "SerdeInfo": {}
  },
  "ViewDefinition": {
    "SubObjects": [
      "arn:aws:glue:aws-region:aws-account-id:table/database/referenced-table1",
      ...
      "arn:aws:glue:aws-region:aws-account-id:table/database/referenced-tableN",
    ],
    "IsProtected": true,
    "Representations": [
      {
        "Dialect": "SPARK",
        "DialectVersion": "1.0",
        "ViewOriginalText": "Spark-SQL",
        "ViewExpandedText": "Spark-SQL"
      }
    ]
  }
}
}'

```

サポートされているビュー操作

以下のコマンドフラグメントは、データカタログビューを操作するさまざまな方法を示します。

• CREATE VIEW

データカタログビューを作成します。以下は、既存のテーブルからビューを作成する例です。

```

CREATE PROTECTED MULTI DIALECT VIEW catalog_view
SECURITY DEFINER AS SELECT * FROM my_catalog.my_database.source_table

```

• ALTER VIEW

使用可能な構文:

- ALTER VIEW view_name [FORCE] ADD DIALECT AS query
- ALTER VIEW view_name [FORCE] UPDATE DIALECT AS query
- ALTER VIEW view_name DROP DIALECT

FORCE ADD DIALECT オプションを使用して、新しいエンジンダイアレクトに従ってスキーマとサブオブジェクトを強制的に更新できます。ただし、これを行うと、他のエンジンダイアレクトの更新にも FORCE を使用しないと、クエリエラーが発生する可能性があることに注意してください。以下に例を示します:

```
ALTER VIEW catalog_view FORCE ADD DIALECT
AS
SELECT order_date, sum(totalprice) AS price
FROM source_table
GROUP BY orderdate;
```

以下は、ダイアレクトを更新するためにビューを変更する方法をデモンストレーションします:

```
ALTER VIEW catalog_view UPDATE DIALECT AS
SELECT count(*) FROM my_catalog.my_database.source_table;
```

• DESCRIBE VIEW

ビューを記述するために使用できる構文:

- SHOW COLUMNS {FROM|IN} view_name [{FROM|IN} database_name] – ユーザーにビューを記述するために必要な AWS Glue および Lake Formation アクセス許可がある場合は、列を一覧表示できます。列を表示するためのいくつかのコマンドの例を以下に示します:

```
SHOW COLUMNS FROM my_database.source_table;
SHOW COLUMNS IN my_database.source_table;
```

- DESCRIBE view_name – ユーザーにビューを記述するために必要な AWS Glue および Lake Formation アクセス許可がある場合、ビュー内の列とそのメタデータを一覧表示できます。
- ## • DROP VIEW

使用可能な構文:

- DROP VIEW [IF EXISTS] view_name

次のサンプルは、ビューを削除する前にビューが存在するかどうかをテストする DROP ステートメントを示します。

```
DROP VIEW IF EXISTS catalog_view;
```

- 作成ビューを表示する

- SHOW CREATE VIEW view_name – 指定したビューを作成する SQL ステートメントを示します。以下は、データカタログビューの作成を示すサンプルです。

```
SHOW CREATE TABLE my_database.catalog_view;  
CREATE PROTECTED MULTI DIALECT VIEW my_catalog.my_database.catalog_view (  
    net_profit,  
    customer_id,  
    item_id,  
    sold_date)  
TBLPROPERTIES (  
    'transient_lastDdlTime' = '1736267222')  
SECURITY DEFINER AS SELECT * FROM  
my_database.store_sales_partitioned_1f WHERE customer_id IN (SELECT customer_id  
    from source_table limit 10)
```

- SHOW VIEWS

通常のビュー、マルチダイアレクトビュー (MDV)、Spark ダイアレクトのない MDV など、カタログ内のすべてのビューを一覧表示します。使用可能な構文は次のとおりです。

- SHOW VIEWS [{ FROM | IN } database_name] [LIKE regex_pattern]:

ビューを表示するコマンドの例を以下に示します:

```
SHOW VIEWS IN marketing_analytics LIKE 'catalog_view*';
```

データカタログビューの作成と設定の詳細については、AWS Lake Formation デベロッパーガイド [AWS の「Glue データカタログビューの構築」](#) を参照してください。

データカタログビューをクエリする

データカタログビューを作成したら、AWS Lake Formation きめ細かなアクセスコントロールが有効になっている Amazon EMR Serverless Spark ジョブを使用してクエリを実行できます。ジョブラ

ンタイムロールには、データカタログビューに対する Lake Formation SELECT アクセス許可が必要です。ビューで参照される基盤となるテーブルへのアクセスを許可する必要はありません。

すべてをセットアップしたら、ビューをクエリできます。例えば、EMR Studio で EMR Serverless アプリケーションを作成した後、以下のクエリを実行してビューにアクセスします。

```
SELECT * from my_database.catalog_view LIMIT 10;
```

便利な関数は `invoker_principal` です。EMRS ジョブランタイムロールの一意の識別子を返します。これは、呼び出しプリンシパルに基づいてビュー出力を制御するために使用できます。これを使用して、呼び出しロールに基づいてクエリ結果を絞り込む条件をビューに追加できます。この関数を使用するには、ジョブランタイムロールに `LakeFormation:GetDataLakePrincipal` IAM アクションに対するアクセス許可が必要です。

```
select invoker_principal();
```

例えば、この関数を WHERE 句に追加して、クエリ結果を絞り込むことができます。

考慮事項と制限事項

Data Catalog ビューを作成することにより、以下が適用されます:

- Amazon EMR 7.6 以上では、Data Catalog ビューのみを作成できます。
- Data Catalog ビュー定義者は、ビューによってアクセスされる基盤となるベーステーブルに対する SELECT アクセスを持っている必要があります。特定のベーステーブルに定義者ロールに対する Lake Formation フィルターが適用されている場合、Data Catalog ビューの作成は失敗します。
- Lake Formation のベーステーブルには `IAMAllowedPrincipals` データレイクのアクセス許可を付与してはいけません。付与されている場合は、`Multi Dialect views may only reference tables without IAMAllowedPrincipals permissions` というエラーが表示されます。
- テーブルの Amazon S3 の場所は、Lake Formation データレイクの場所として登録する必要があります。テーブルがこのように登録されていない場合、`Multi Dialect views may only reference Lake Formation managed tables` というエラーが表示されます。Lake Formation で Amazon S3 ロケーションを登録する方法については、「AWS Lake Formation デベロッパーガイド」の [Amazon S3 ロケーションの登録](#) を参照してください。
- PROTECTED データカタログビューのみを作成できます。UNPROTECTED ビューはサポートされていません。
- データカタログビュー定義の別の AWS アカウントのテーブルを参照することはできません。また、別のリージョンにあるのと同じアカウントのテーブルを参照することはできません。

- アカウントまたはリージョン間でデータを共有するには、Lake Formation リソースリンクを使用してビュー全体をクロスアカウントおよびクロスリージョンで共有する必要があります。
- ユーザー定義関数 (UDF) はサポートされていません。
- Iceberg テーブルに基づくビューを使用できます。オープンテーブル形式の Apache Hudi と Delta Lake もサポートされています。
- データカタログビューで他のビューを参照できません。
- AWS Glue データカタログビュースキーマは常に小文字を使用して保存されます。例えば、DDL ステートメントを使用して `Castle` という名前の列のある Glue データカタログビューを作成する場合、Glue Data Catalog に作成される列は小文字で `castle` になります。次に、DML クエリで列名を `Castle` または `CASTLE` として指定することにより、EMR Spark はその名前を小文字にしてクエリを実行します。ただし、列見出しは、クエリで指定した大文字と小文字を使用して表示されます。

DML クエリで指定された列名が Glue データカタログの列名と一致しない場合にクエリを失敗させるには、`spark.sql.caseSensitive=true` を設定します。

オープンテーブル形式のサポート

EMR Serverless は、Apache Hive、Apache Iceberg、Delta Lake (7.6.0 以降)、Apache Hudi (7.6.0 以降) に対する SELECT クエリをサポートしています。EMR 7.12 以降、テーブルデータを変更する DML および DDL オペレーションは、Lake Formation が提供する認証情報を使用して Apache Hive、Apache Iceberg、Delta Lake テーブルでサポートされています。

考慮事項と制限事項

General

EMR Serverless で Lake Formation を使用する場合は、次の制限事項を確認してください。

Note

EMR Serverless で Spark ジョブの Lake Formation を有効にすると、ジョブはシステムドライバーとユーザードライバーを起動します。起動時に事前初期化された容量を指定した場合、ドライバーは事前初期化された容量からプロビジョニングし、システムドライバーの数は指定したユーザードライバーの数と等しくなります。オンデマンドキャパシティを選択すると、EMR Serverless はユーザードライバーに加えてシステムドライバーを起動しま

す。Lake Formation ジョブで EMR Serverless に関連するコストを見積もるには、[AWS 料金見積りツール](#) を使用します。

- Lake Formation を使用した Amazon EMR Serverless は、サポートされているすべての [EMR Serverless リージョン](#) で使用できます。
- Lake Formation 対応アプリケーションは、[カスタマイズされた EMR Serverless イメージ](#) の使用をサポートしていません。
- Lake Formation ジョブの DynamicResourceAllocation をオフにすることはできません。
- Lake Formation は Spark ジョブでのみ使用できます。
- Lake Formation を使用した EMR Serverless は、ジョブ全体で 1 つの Spark セッションのみをサポートします。
- Lake Formation を使用した EMR Serverless は、リソースリンクを介して共有されるクロスアカウントテーブルクエリのみをサポートします。
- 次のサポートはありません。
 - 耐障害性のある分散データセット (RDD)
 - Spark ストリーミング
 - ネストされた列のアクセスコントロール
- EMR Serverless は、以下を含むシステムドライバーの完全な分離を損なう可能性のある機能をブロックします。
 - UDT、HiveUDF およびカスタムクラスを含むユーザー定義関数
 - カスタムデータソース
 - Spark 拡張機能、コネクタ、メタストア用の追加の jar の提供
 - ANALYZE TABLE コマンド
- EMR Serverless アプリケーションが Amazon S3 の VPC エンドポイントを持つプライベートサブネットにあり、アクセスを制御するエンドポイントポリシーをアタッチしている場合、ジョブが AWS Managed Amazon S3 にログデータを送信する前に、VPC ポリシーの [Managed Storage](#) に詳述されているアクセス許可を S3 ゲートウェイエンドポイントに含めます。リクエストのトラブルシューティングについては、AWS サポートにお問い合わせください。
- Amazon EMR 7.9.0 以降、Spark FGAC は s3a:// スキームで使用する S3AFileSystem をサポートします。
- Amazon EMR 7.11 は、CTAS を使用したマネージドテーブルの作成をサポートしています。

- Amazon EMR 7.12 は、CTAS を使用したマネージドテーブルと外部テーブルの作成をサポートしています。

アクセス許可

- アクセスコントロールを適用するために、EXPLAIN PLAN および DESCRIBE TABLE などの DDL オペレーションでは、制限された情報は公開されません。
- Lake Formation にテーブルの場所を登録すると、データアクセスは EMR Serverless ジョブランタイムロールの IAM アクセス許可の代わりに Lake Formation の保存された認証情報を使用します。ランタイムロールにその場所に対する S3 IAM アクセス許可がある場合でも、テーブルの場所の登録済みロールが正しく設定されていない場合、ジョブは失敗します。
- Amazon EMR 7.12 以降では、DataFrameWriter (V2) と Lake Formation 認証情報を追加モードで使用して、既存の Hive テーブルと Iceberg テーブルに書き込むことができます。上書きオペレーションの場合、または新しいテーブルを作成する場合、EMR はランタイムロールの認証情報を使用してテーブルデータを変更します。
- ビューまたはキャッシュされたテーブルをソースデータとして使用する場合は、次の制限が適用されます (これらの制限は Glue Data Catalog AWS ビューには適用されません)。
 - MERGE、DELETE、および UPDATE オペレーションの場合
 - サポートされている: ビューとキャッシュされたテーブルをソーステーブルとして使用します。
 - サポート対象外: 割り当て句と条件句でビューとキャッシュされたテーブルを使用する。
 - CREATE OR REPLACE および REPLACE TABLE AS SELECT オペレーションの場合:
 - サポート対象外: ビューとキャッシュされたテーブルをソーステーブルとして使用します。
- ソースデータの UDFsを持つ Delta Lake テーブルは、削除ベクトルが有効になっている場合にのみ MERGE、DELETE、UPDATE オペレーションをサポートします。

ログとデバッグ

- EMR Serverless は、Lake Formation 対応アプリケーションのシステムドライバー Spark ログへのアクセスを制限します。システムドライバーは昇格アクセス許可を使用して実行されるため、システムドライバーが生成するイベントとログには機密情報が含まれる可能性があります。許可されていないユーザーまたはコードがこの機密データにアクセスできないように、EMR Serverless はシステムドライバーログへのアクセスを無効にしました。

- システムプロファイルログは常にマネージドストレージに保持されます。これは無効にできない必須の設定です。これらのログは、カスタマーマネージド KMS キーまたは AWS マネージド KMS キーを使用して安全に保存され、暗号化されます。

Iceberg

Apache Iceberg を使用するときは、以下の考慮事項を確認してください。

- Apache Iceberg はセッションカタログでのみ使用でき、任意の名前のカタログでは使用できません。
- Lake Formation に登録されている Iceberg テーブルは、メタデータテーブル `history`、`metadata_log_entries`、`snapshots`、`files`、`manifests`、および `refs` のみをサポートします。Amazon EMR は `partitions`、`path`、`summaries` などの機密データを持つ可能性のある列を非表示にします。この制限は、Lake Formation に登録されていない Iceberg テーブルには適用されません。
- Lake Formation に登録されていないテーブルは、すべての Iceberg ストアドプロシージャをサポートします。`register_table` プロシージャと `migrate` プロシージャは、どのテーブルでもサポートされていません。
- V1 の代わりに Iceberg DataFrameWriterV2 を使用することを提案します。

トラブルシューティング

トラブルシューティングソリューションについては、以下のセクションを参照してください。

ログ記録

EMR Serverless は、Spark リソースプロファイルを使用してジョブ実行を分割します。EMR Serverless は、ユーザープロファイルを使用して指定したコードを実行し、システムプロファイルは Lake Formation ポリシーを適用します。ユーザープロファイルとして実行されたタスクのログにアクセスできます。

Lake Formation 対応ジョブのデバッグの詳細については、「[ジョブのデバッグ](#)」を参照してください。

Live UI と Spark 履歴サーバー

Live UI と Spark 履歴サーバーには、ユーザープロファイルから生成されたすべての Spark イベントと、システムドライバーから生成された編集済みイベントがあります。

[エグゼキューター] タブには、ユーザードライバーとシステムドライバからすべてのタスクが表示されます。ただし、ログリンクはユーザプロフィールでのみ使用できます。また、出力レコードの数など、一部の情報は Live UI から編集されます。

Lake Formation のアクセス許可が不十分なためジョブが失敗した

ジョブランタイムロールには、アクセスしているテーブルで SELECT と DESCRIBE を実行するアクセス許可があることを確認してください。

RDD の実行が失敗したジョブ

EMR Serverless は現在、Lake Formation 対応ジョブで耐障害性のある分散データセット (RDD) オペレーションをサポートしていません。

Amazon S3 のデータファイルにアクセスできない

Lake Formation にデータレイクの場所が登録されていることを確認します。

セキュリティ検証の例外

EMR Serverless がセキュリティ検証エラーを検出しました。サポートが必要な場合は、AWS サポートにお問い合わせください。

アカウント間で AWS Glue データカタログとテーブルを共有する

データベースとテーブルをアカウント間で共有し、Lake Formation を引き続き使用できます。詳細については、「[Lake Formation でのクロスアカウントデータ共有](#)」および「[を使用した AWS Glue データカタログとテーブルのクロスアカウント共有方法 AWS Lake Formation](#)」を参照してください。

Spark ネイティブのきめ細かなアクセスコントロール alllowlisted PySpark API

セキュリティとデータアクセスコントロールを維持するために、Spark のきめ細かなアクセスコントロール (FGAC) は特定の PySpark 関数を制限します。これらの制限は、以下を通じて適用されます。

- 関数の実行を妨げる明示的なブロック
- 関数を機能させないアーキテクチャの非互換性
- エラーをスローしたり、アクセス拒否メッセージを返したり、呼び出されたときに何もしない関数

Spark FGAC では、次の PySpark 機能はサポートされていません。

- RDD オペレーション (SparkRDDUnsupportedException でブロック)
- Spark Connect (サポートされていません)
- Spark ストリーミング (サポートされていません)

Native Spark FGAC 環境でリストされた関数をテストし、期待どおりに動作することを確認しましたが、テストは通常、各 API の基本的な使用のみを対象としています。複数の入力タイプまたは複雑なロジックパスを持つ関数には、テストされていないシナリオがある可能性があります。

ここに記載されておらず、上記のサポートされていないカテゴリに明確に含まれていない関数については、次のことをお勧めします。

- ガンマ環境または小規模デプロイで最初にテストする
- 本番環境で使用する前に動作を検証する

 Note

クラスメソッドがリストされているが、そのベースクラスが表示されていない場合でも、そのメソッドは機能します。つまり、ベースクラスコンストラクタを明示的に検証していないということです。

PySpark API はモジュールに整理されています。各モジュール内のメソッドの一般的なサポートの詳細については、以下の表を参照してください。

モジュール名	ステータス	注意事項
pyspark_core	サポート	このモジュールには主要な RDD クラスが含まれており、これらの関数はほとんどサポートされていません。
pyspark_sql	サポート	
pyspark_testing	サポート	
pyspark_resource	サポート	

モジュール名	ステータス	注意事項
pyspark_streaming	ブロック	ストリーミングの使用は Spark FGAC でブロックされます。
pyspark_mllib	実験的	このモジュールには RDD ベースの ML オペレーションが含まれており、これらの関数はほとんどサポートされていません。このモジュールは徹底的にテストされていません。
pyspark_ml	実験的	このモジュールには DataFrame ベースの ML オペレーションが含まれており、これらの関数は主にサポートされています。このモジュールは徹底的にテストされていません。
pyspark_pandas	サポート	
pyspark_pandas_slow	サポート	
pyspark_connect	ブロック	Spark Connect の使用が Spark FGAC でブロックされます。
pyspark_pandas_connect	ブロック	Spark Connect の使用が Spark FGAC でブロックされます。

モジュール名	ステータス	注意事項
pyspark_pandas_slow_connect	ブロック	Spark Connect の使用が Spark FGAC でブロックされます。
pyspark_errors	実験的	このモジュールは徹底的にテストされていません。カスタムエラークラスは使用できません。

API 許可リスト

ダウンロード可能で検索しやすいリストの場合、モジュールとクラスを含むファイルは、[ネイティブ FGAC で許可されている Python 関数](#)で使えます。

ワーカー間の暗号化

Amazon EMR バージョン 6.15.0 以降では、Spark ジョブの実行でワーカー間の相互 TLS 暗号化通信を有効にします。有効にすると、EMR Serverless はジョブ実行でプロビジョニングされたワーカーごとに一意の証明書を自動的に生成して配布します。これらのワーカーが通信してコントロールメッセージを交換したり、シャッフルデータを転送したりすると、相互 TLS 接続を確立し、設定された証明書を使用して相互のアイデンティティを検証します。ワーカーが別のワーカーの証明書を検証できない場合、TLS ハンドシェイクは失敗し、EMR Serverless はワーカー間の接続を中止します。

EMR Serverless で Lake Formation を使用している場合、相互 TLS 暗号化はデフォルトで有効になっています。

EMR Serverless での相互 TLS 暗号化の有効化

Spark アプリケーションで相互 TLS 暗号化を有効にするには、[EMR Serverless アプリケーションの作成](#)時に `spark.ssl.internode.enabled` を `true` に設定します。AWS コンソールを使用して EMR Serverless アプリケーションを作成している場合は、カスタム設定を使用を選択し、アプリケーション設定を展開して `runtimeConfiguration` を入力します。

```
aws emr-serverless create-application \
```

```
--release-label emr-6.15.0 \  
--runtime-configuration '{  
  "classification": "spark-defaults",  
  "properties": {"spark.ssl.internode.enabled": "true"}  
}' \  
--type "SPARK"
```

個々の Spark ジョブ実行の相互 TLS 暗号化を有効にする場合は、`spark-submit` を使用する際に `spark.ssl.internode.enabled` を `true` に設定します。

```
--conf spark.ssl.internode.enabled=true
```

KMS CMK によるディスク暗号化

EMR Serverless は、サービス所有の暗号化キーを使用して、デフォルトでワーカーにアタッチされたすべてのディスクを暗号化します。オプションで、独自の AWS KMS カスタマーマネージドキー (CMKs) を使用してこれらのディスクを暗号化することもできます。これにより、キーポリシーの確立と維持、キーの使用状況の監査など、暗号化キーをより詳細に制御できます。

ディスク暗号化は、アプリケーションの作成時または個々のジョブの送信時に設定できます。アプリケーションレベルで有効にすると、そのアプリケーションのすべてのジョブが暗号化設定を継承します。ジョブの送信時にディスク暗号化設定を指定することで、アプリケーションのデフォルトを上書きすることもできます。

Note

EMR Serverless ディスク暗号化は、対称 KMS キーのみをサポートします。非対称 KMS キーはサポートされていません。で作成された対称暗号化 KMS キーを使用する必要があります AWS KMS。詳細については AWS KMS、[「とは」を参照してください AWS KMS。](#)

暗号化コンテキストの使用

必要に応じて、EMR Serverless は暗号化コンテキストを使用して、暗号化オペレーションに追加の認証済みデータを提供します。暗号化コンテキストは、シークレットではない追加の認証済みデータを含むことができるキーと値のペアのセットです。暗号化コンテキストは暗号化されたデータに暗号的にバインドされるため、データの復号には同じ暗号化コンテキストが必要です。

EMR Serverless では、ディスク暗号化を設定するときにカスタム暗号化コンテキストを指定できます。この暗号化コンテキストは、KMS オペレーションの特定と理解に役立つ AWS CloudTrail ログに含まれています。

Note

機密情報は AWS CloudTrail、ログにプレーンテキストで表示されるため、暗号化コンテキストに保存しないでください。

カスタマーマネージドキーを使用したディスク暗号化の設定

CreateApplication

独自の KMS キーを使用してディスクを暗号化するには、EMR Serverless アプリケーションを作成するときに `diskEncryptionConfiguration` パラメータを含めます。

```
aws emr-serverless create-application \  
  --type TYPE \  
  --name APPLICATION_ID \  
  --release-label RELEASE_LABEL \  
  --region AWS_REGION \  
  --disk-encryption-configuration '{  
    "encryptionKeyArn": "key-arn",  
    "encryptionContext": {  
      "key": "value"  
    }  
  }'  
'
```

UpdateApplication

KMS キー ARN および/または暗号化コンテキストを更新するには、アプリケーションを更新するときに `diskEncryptionConfiguration` パラメータを新しい値で指定します。

```
aws emr-serverless update-application \  
  --name APPLICATION_ID \  
  --region AWS_REGION \  
  --disk-encryption-configuration '{  
    "encryptionKeyArn": "key-arn",  
    "encryptionContext": {  
      "key": "value"  
    }  
  }'  
'
```



```
        "key": "value"
    }
}'
```

Note

アプリケーションで設定されたディスク暗号化の設定を解除するには、更新アプリケーション `diskEncryptionConfiguration` 中に空の `diskEncryptionConfiguration` を渡します。

StartJobRun

独自の KMS キーを使用してディスクを暗号化するには、ジョブ実行を送信するときに `diskEncryptionConfiguration` 設定を使用します。

```
--configuration-overrides '{
    "diskEncryptionConfiguration": {
        "encryptionKeyArn": "key-arn",
        "encryptionContext": {
            "key": "value"
        }
    }
}'
```

パブリック Livy エンドポイント

パブリック Livy エンドポイントを介して Spark セッションを作成するときに、独自の KMS キーを使用してディスクを暗号化するには、セッションの `conf` オブジェクトで暗号化設定を指定します。

```
data = {
    "kind": "pyspark",
    "heartbeatTimeoutInSeconds": 60,
    "conf": {
        "emr-serverless.session.executionRoleArn": "role_arn",
        "spark.emr-serverless.disk.encryptionKeyArn": "key-arn",
        "spark.emr-serverless.disk.encryptionContext": "key1:value1,key2:value2" #
Optional
    }
}
```

```
# Send request to create a session with the Livy API endpoint
request = AWSRequest(method='POST', url=endpoint + "/sessions", data=json.dumps(data),
headers=headers)
```

ディスク暗号化に必要なアクセス許可

EMR Serverless の暗号化キーのアクセス許可

独自の暗号化キーを使用してディスクを暗号化する場合は、`emr-serverless.amazonaws.com` プリンシパルに次の KMS キーのアクセス許可を設定する必要があります。

- `kms:GenerateDataKey` : ディスクボリュームを暗号化するためのデータキーを生成するには
- `kms:Decrypt` : 暗号化されたディスクコンテンツにアクセスするときにデータキーを復号するには

```
{
  "Effect": "Allow",
  "Principal": {
    "Service": "emr-serverless.amazonaws.com"
  },
  "Action": [
    "kms:Decrypt",
    "kms:GenerateDataKey"
  ],
  "Resource": "*",
  "Condition": {
    "StringLike": {
      "aws:SourceArn": "arn:aws:emr-serverless:region:aws-account-id:/
applications/application-id"
    },
    "StringEquals": {
      "kms:EncryptionContext:applicationId": "application-id",
      "aws:SourceAccount": "aws-account-id"
    }
  }
}
```

セキュリティのベストプラクティスとして、KMS キーポリシーに `aws:SourceArn` 条件キーを追加することをお勧めします。IAM グローバル条件キー `aws:SourceArn` によって、EMR Serverless がアプリケーション ARN にのみ KMS キーを使用するように設定できます。さらに、

aws:SourceAccount 条件キーを含めると、KMS キーの使用を条件で指定された AWS アカウント ID から送信されるリクエストに制限することで、別のセキュリティレイヤーが提供されます。

ジョブのランタイムロールには、IAM ポリシーで次のアクセス許可が付与されている必要があります。

```
{
  "Sid": "Enable GDK and Decrypt",
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
      "kms:GenerateDataKey",
      "kms:Decrypt"
    ],
    "Resource": "key-arn"
  }
}
```

必要なユーザーアクセス許可

ジョブを送信するユーザーには、キーを使用するアクセス許可が必要です。ユーザー、グループ、またはロールの KMS キーポリシーまたは IAM ポリシーでアクセス許可を指定できます。ジョブを送信するユーザーに KMS キーのアクセス許可が付与されていない場合、EMR Serverless はジョブ実行の送信を拒否します。

キーポリシーの例

次のキーポリシーは、kms:DescribeKey、kms:GenerateDataKey および kms:Decrypt へのアクセス許可を提供します。

- kms:DescribeKey: カスタマー管理の KMS キーが有効であること、および使用する前に SYMMETRIC であることを確認します。

```
{
  "Sid": "Enable DescribeKey",
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::111122223333:user/user-name"
  },
  "Action": [
```

```

        "kms:DescribeKey"
    ],
    "Resource": "*"
},
{
    "Sid": "Enable GDK and Decrypt",
    "Effect": "Allow",
    "Principal": {
        "AWS": "arn:aws:iam::111122223333:user/user-name"
    },
    "Action": [
        "kms:GenerateDataKey",
        "kms:Decrypt"
    ],
    "Resource": "*",
    "Condition": {
        "StringEquals": {
            "kms:ViaService": "emr-serverless.region.amazonaws.com",
            "kms:EncryptionContext:key": "value"
        }
    }
}
}

```

セキュリティのベストプラクティスとして、KMS キーポリシーに `kms:viaService` 条件キーを追加することをお勧めします。これにより、KMS キーの使用が `emr-serverless` からの検証リクエストに制限されます。

IAM ポリシーの例

次の IAM ポリシーは、`kms:DescribeKey`、`kms:GenerateDataKey` および `kms:Decrypt` へのアクセス許可を提供します。

```

{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": [
            "kms:DescribeKey",
            "kms:GenerateDataKey",
            "kms:Decrypt"
        ],
        "Resource": "key-arn"
    }
}

```

```
}
```

キー使用状況のモニタリング

EMR Serverless through でカスターマネージドキーの使用をモニタリングできます AWS CloudTrail。は、EMR Serverless コンソール、EMR Serverless API、AWS CLI、または AWS SDK からの呼び出しなど、へのすべての API 呼び出しをイベント AWS KMS として AWS CloudTrail キャプチャします。

キャプチャされた情報には、指定した暗号化コンテキストが含まれており、KMS キーを使用した特定の EMR Serverless リソースの特定と監査に役立ちます。たとえば、に次のようなイベントが表示される場合があります AWS CloudTrail。の使用の詳細については AWS CloudTrail、[AWS CloudTrail 「ユーザーガイド」](#)を参照してください。

GenerateDataKey

EMR Serverless が暗号化されたディスクボリュームを作成している場合の GenerateDataKey オペレーションのサンプルイベント

```
{
  "eventVersion": "1.11",
  "userIdentity": {
    "type": "AWSService",
    "principalId": "user",
    "invokedBy": "AWS Internal"
  },
  "eventTime": "2025-07-28T21:43:51Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "GenerateDataKey",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "ipAddress",
  "userAgent": "userAgent",
  "requestParameters": {
    "encryptionContext": {
      "applicationId": "test"
    },
    "keyId": "arn:aws:kms:region:accountId:key/ffffffff-fffff-aaaaa-eeee-sample",
    "keySpec": "AES_256"
  },
  "responseElements": null,
  "additionalEventData": {
```

```

    "keyMaterialId":
      "145c963debe558dfb01848d2a4539da940f3478852f86cfe2f52d5df796a5a02"
    },
    "requestID": "cc9d1c5e-97c4-4a4f-ae7a-e576sample",
    "eventID": "0b0fef09-f28d-4da8-a5a1-17b74sample",
    "readOnly": true,
    "resources": [
      {
        "accountId": "account",
        "type": "AWS::KMS::Key",
        "ARN": "arn:aws:kms:region:accountId:key/ffffffff-fffff-aaaaa-eeee-sample"
      }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "accountId",
    "eventCategory": "Management"
  }
}

```

Decrypt

EMR Serverless が暗号化されたデータにアクセスしている場合の Decrypt オペレーションのサンプルイベント。

```

{
  "eventVersion": "1.11",
  "userIdentity": {
    "type": "AWSService",
    "principalId": "user",
    "invokedBy": "AWS Internal"
  },
  "eventTime": "2025-07-28T21:43:51Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "Decrypt",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "ipAddress",
  "userAgent": "userAgent",
  "requestParameters": {
    "encryptionContext": {
      "applicationId": "test"
    },
    "keyId": "arn:aws:kms:region:accountId:key/ffffffff-fffff-aaaaa-eeee-sample",
    "keySpec": "AES_256"
  }
}

```

```
{,
  "responseElements": null,
  "additionalEventData": {
    "keyMaterialId":
"145c963debe558dfb01848d2a4539da940f3478852f86cfe2f52d5df796a5a02"
  },
  "requestID": "cc9d1c5e-97c4-4a4f-ae7a-e576sample",
  "eventID": "0b0fef09-f28d-4da8-a5a1-17b74sample",
  "readOnly": true,
  "resources": [
    {
      "accountId": "account",
      "type": "AWS::KMS::Key",
      "ARN": "arn:aws:kms:region:accountId:key/ffffffff-fffff-aaaaa-eeee-sample"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "accountId",
  "eventCategory": "Management"
}
```

詳細はこちら

次のリソースは、保管時のデータ暗号化についての詳細を説明しています。

- AWS KMS 基本概念の詳細については、「[AWS KMS デベロッパーガイド](#)」を参照してください。
- のセキュリティのベストプラクティスの詳細については AWS KMS、「[AWS KMS デベロッパーガイド](#)」を参照してください。

EMR Serverless によるデータ保護のための Secrets Manager

AWS Secrets Manager は、データベース認証情報、API キー、およびその他のシークレット情報を保護するためのシークレットストレージサービスです。次に、コード内で、ハードコードされた認証情報を Secrets Manager への API コールに置き換えます。シークレットはそこに存在しないため、お客様のコードを調べている誰かがシークレットを漏らさないようにするのに役立ちます。この概要については、「[AWS Secrets Manager ユーザーガイド](#)」を参照してください。

Secrets Manager は、AWS Key Management Service キーを使用してシークレットを暗号化します。詳細については、「AWS Secrets Manager ユーザーガイド」の「[Secret encryption and decryption](#)」を参照してください。

指定したスケジュールに従って自動的にシークレットを更新するように Secrets Manager を設定することができます。これにより、長期のシークレットを短期のシークレットに置き換えることが可能となり、侵害されるリスクが大幅に減少します。詳細については、「AWS Secrets Manager ユーザーガイド」の「[AWS Secrets Manager シークレットのローテーション](#)」を参照してください。

Amazon EMR Serverless はと統合 AWS Secrets Manager されているため、データを Secrets Manager に保存し、設定でシークレット ID を使用できます。

EMR Serverless がシークレットを使用する方法

Secrets Manager にデータを保存して、EMR Serverless の設定でシークレット ID を使用する場合、機密設定データをプレーンテキストで EMR Serverless に渡したり、外部 API に公開したりすることはありません。キーと値のペアに Secrets Manager に保存したシークレットのシークレット ID が含まれていることを示すと、EMR Serverless は実行中のジョブの設定データをワーカーに送信するときにシークレットを取得します。

設定のキーと値のペアに Secrets Manager に保存したシークレットへの参照が含まれていることを示すには、設定値に `EMR.secret@` 注釈を追加します。シークレット ID 注釈を持つ設定プロパティの場合、EMR Serverless は Secrets Manager を呼び出し、ジョブの実行時にシークレットを解決します。

シークレットを作成する方法

シークレットを作成するには、AWS Secrets Manager 「ユーザーガイド」の「[AWS Secrets Manager シークレットを作成する](#)」の手順に従います。ステップ 3 では、[プレーンテキスト] フィールドを選択して機密値を入力します。

シークレットを設定分類で指定する

次の例は、StartJobRun の設定分類でシークレットを指定する方法をデモンストラレーションしています。アプリケーションレベルで Secrets Manager の分類を設定する場合は、「[EMR Serverless のデフォルトのアプリケーション設定](#)」を参照してください。

この例では、`SecretName` を取得するシークレットの名前に置き換えます。詳細については、[シークレットを作成する方法](#) を参照してください。

このセクションの内容

- [シークレットリファレンスを指定する - Spark](#)
- [シークレットリファレンスを指定する - Hive](#)

シークレットリファレンスを指定する - Spark

Example– Spark の外部 Hive メタストア設定でシークレットリファレンスを指定する

```
aws emr-serverless start-job-run \
  --application-id "application-id" \
  --execution-role-arn "job-role-arn" \
  --job-driver '{
    "sparkSubmit": {
      "entryPoint": "s3://amzn-s3-demo-bucket/scripts/spark-jdbc.py",
      "sparkSubmitParameters": "--jars s3://amzn-s3-demo-bucket/mariadb-
connector-java.jar
      --conf
      spark.hadoop.javax.jdo.option.ConnectionDriverName=org.mariadb.jdbc.Driver
      --conf spark.hadoop.javax.jdo.option.ConnectionUserName=connection-user-
name
      --conf
      spark.hadoop.javax.jdo.option.ConnectionPassword=EMR.secret@SecretName
      --conf spark.hadoop.javax.jdo.option.ConnectionURL=jdbc:mysql://db-host:db-
port/db-name
      --conf spark.driver.cores=2
      --conf spark.executor.memory=10G
      --conf spark.driver.memory=6G
      --conf spark.executor.cores=4"
    }
  }' \
  --configuration-overrides '{
    "monitoringConfiguration": {
      "s3MonitoringConfiguration": {
        "logUri": "s3://amzn-s3-demo-bucket/spark/logs/"
      }
    }
  }'
```

Example– spark-defaults 分類で外部 Hive メタストア設定のシークレットリファレンスを指定する

```
{
```

```

    "classification": "spark-defaults",
    "properties": {

      "spark.hadoop.javax.jdo.option.ConnectionDriverName": "org.mariadb.jdbc.Driver"
      "spark.hadoop.javax.jdo.option.ConnectionURL": "jdbc:mysql://db-host:db-
port/db-name"
      "spark.hadoop.javax.jdo.option.ConnectionUserName": "connection-user-name"
      "spark.hadoop.javax.jdo.option.ConnectionPassword":
      "EMR.secret@SecretName",
    }
  }
}

```

シークレットリファレンスを指定する - Hive

Example- Hive の外部 Hive メタストア設定でシークレットリファレンスを指定する

```

aws emr-serverless start-job-run \
  --application-id "application-id" \
  --execution-role-arn "job-role-arn" \
  --job-driver '{
    "hive": {
      "query": "s3://amzn-s3-demo-bucket/emr-serverless-hive/query/hive-query.ql",
      "parameters": "--hiveconf hive.exec.scratchdir=s3://amzn-s3-demo-bucket/emr-
serverless-hive/hive/scratch
                    --hiveconf hive.metastore.warehouse.dir=s3://amzn-s3-demo-bucket/
emr-serverless-hive/hive/warehouse
                    --hiveconf javax.jdo.option.ConnectionUserName=username
                    --hiveconf
javax.jdo.option.ConnectionPassword=EMR.secret@SecretName
                    --hiveconf
hive.metastore.client.factory.class=org.apache.hadoop.hive.ql.metadata.SessionHiveMetaStoreCli
                    --hiveconf
javax.jdo.option.ConnectionDriverName=org.mariadb.jdbc.Driver
                    --hiveconf javax.jdo.option.ConnectionURL=jdbc:mysql://db-host:db-
port/db-name"
    }
  }' \
  --configuration-overrides '{
    "monitoringConfiguration": {
      "s3MonitoringConfiguration": {
        "logUri": "s3://amzn-s3-demo-bucket"
      }
    }
  }
}

```

```
}'
```

Example– hive-site 分類で外部 Hive メタストア設定のシークレットリファレンスを指定する

```
{
  "classification": "hive-site",
  "properties": {
    "hive.metastore.client.factory.class":
"org.apache.hadoop.hive.ql.metadata.SessionHiveMetaStoreClientFactory",
    "javax.jdo.option.ConnectionDriverName": "org.mariadb.jdbc.Driver",
    "javax.jdo.option.ConnectionURL": "jdbc:mysql://db-host:db-port/db-name",
    "javax.jdo.option.ConnectionUserName": "username",
    "javax.jdo.option.ConnectionPassword": "EMR.secret@SecretName"
  }
}
```

EMR Serverless にアクセス権を付与してシークレットを取得する

EMR Serverless が Secrets Manager からシークレット値を取得できるようにするには、クラスターを作成するときにシークレットに次のポリシーステートメントを追加します。EMR Serverless がシークレット値を読み取るには、カスターマネージド KMS キーを使用してシークレットを作成する必要があります。詳細については、「AWS Secrets Manager ユーザーガイド」の「[Permissions for the KMS key](#)」を参照してください。

次のポリシーでは、*applicationId* をアプリケーションの ID に置き換えます。

シークレットのリソースポリシー

EMR Serverless がシークレット値を取得できるようにするには、AWS Secrets Manager のシークレットのリソースポリシーに次のアクセス許可を含める必要があります。特定のアプリケーションのみがこのシークレットを取得できるようにするには、オプションでポリシーの条件として EMR Serverless アプリケーション ID を指定できます。

JSON

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

```

        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret"
    ],
    "Resource": [
        "*"
    ],
    "Condition": {
        "StringEquals": {
            "aws:SourceArn": "arn:aws:emr-serverless*:123456789012:/applications/
*"
        }
    },
    "Sid": "AllowSECRETSMANAGERGetsecretvalue"
}
]
}

```

カスタマーマネージド AWS Key Management Service (AWS KMS) キーの次のポリシーを使用してシークレットを作成します。

カスタマーマネージド AWS KMS キーのポリシー

```

{
  "Sid": "Allow EMR Serverless to use the key for decrypting secrets",
  "Effect": "Allow",
  "Principal": {
    "Service": [
      "emr-serverless.amazonaws.com"
    ]
  },
  "Action": [
    "kms:Decrypt",
    "kms:DescribeKey"
  ],
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "kms:ViaService": "secretsmanager.AWS #####.amazonaws.com"
    }
  }
}

```

シークレットのローテーション

ローテーションとは、シークレットを定期的に更新することです。AWS Secrets Manager を設定して、指定したスケジュールで自動的にシークレットをローテーションするようにすることができます。そうすれば、長期のシークレットを短期のシークレットに置き換えることができます。これにより、侵害のリスクを減らすことができます。EMR Serverless は、ジョブが実行中状態に移行すると、注釈付き設定からシークレット値を取得します。お客様またはプロセスが Secrets Manager のシークレット値を更新する場合は、ジョブが更新された値を取得できるように、新しいジョブを送信する必要があります。

Note

既に実行中の状態のジョブは、更新されたシークレット値を取得できません。これにより、ジョブが失敗する可能性があります。

EMR Serverless での Amazon S3 Access Grants の使用

EMR Serverless 用 S3 Access Grants の概要

Amazon EMR リリース 6.15.0 以降では、Amazon S3 Access Grants によるスケーラブルなアクセスコントロールソリューションの提供により、EMR Serverless から Amazon S3 データへのアクセスが強化されています。S3 データのアクセス許可設定が複雑または大規模な場合は、Access Grants を使用して、ユーザー、ロール、アプリケーションの S3 データ権限をスケーリングします。

S3 Access Grants を使用すると、ランタイムロールや EMR Serverless アプリケーションへのアクセス権を持つアイデンティティにアタッチされている IAM ロールによって付与される権限を超えて、Amazon S3 データへのアクセスを強化できます。

詳細については、「Amazon EMR 管理ガイド」の「[Managing access with S3 Access Grants for Amazon EMR](#)」および「Amazon Simple Storage Service ユーザーガイド」の「[S3 Access Grants でのアクセス管理](#)」を参照してください。

このセクションでは、S3 Access Grants を使用して Amazon S3 のデータへのアクセスを提供する EMR Serverless アプリケーションを起動する方法について説明します。他の Amazon EMR デプロイで S3 Access Grants を使用する手順については、以下のドキュメントを参照してください。

- [Amazon EMR での S3 Access Grants の使用](#)

- [EKS 上の Amazon EMR での S3 Access Grants の使用](#)

データ管理に S3 Access Grants を使用した EMR Serverless アプリケーションの起動

EMR Serverless で S3 Access Grants を有効にし、Spark アプリケーションを起動できます。アプリケーションが S3 データをリクエストすると、Amazon S3 は特定のバケット、プレフィックス、またはオブジェクトを対象とする一時的な認証情報を提供します。

1. EMR Serverless アプリケーションのジョブ実行ロールを設定します。Spark ジョブの実行と S3 Access Grants の使用に必要な IAM アクセス許可 (s3:GetDataAccess および s3:GetAccessGrantsInstanceForPrefix) を含めます。

```
{
  "Effect": "Allow",
  "Action": [
    "s3:GetDataAccess",
    "s3:GetAccessGrantsInstanceForPrefix"
  ],
  "Resource": [
    //LIST ALL INSTANCE ARNS THAT THE ROLE IS ALLOWED TO QUERY
    "arn:aws_partition:s3:Region:account-id1:access-grants/default",
    "arn:aws_partition:s3:Region:account-id2:access-grants/default"
  ]
}
```

Note

ジョブ実行に、S3 に直接アクセスする追加のアクセス許可を持つ IAM ロールを指定することにより、ユーザーは S3 Access Grants からのアクセス許可がなくても、そのロールで許可されたデータにアクセスできます。

2. 次の例に示すように、Amazon EMR リリースラベルが 6.15.0 以降で spark-defaults 分類の EMR Serverless アプリケーションを起動します。*red text* の値を使用シナリオに適した値に置き換えます。

```
aws emr-serverless start-job-run \
  --application-id application-id \
  --execution-role-arn job-role-arn \
  --job-driver '{
```

```

    "sparkSubmit": {
      "entryPoint": "s3://us-east-1.elasticmapreduce/emr-containers/samples/
wordcount/scripts/wordcount.py",
      "entryPointArguments": ["s3://amzn-s3-demo-destination-bucket1/
wordcount_output"],
      "sparkSubmitParameters": "--conf spark.executor.cores=1 --conf
spark.executor.memory=4g --conf spark.driver.cores=1 --conf spark.driver.memory=4g
--conf spark.executor.instances=1"
    }
  } \
  --configuration-overrides '{
    "applicationConfiguration": [{
      "classification": "spark-defaults",
      "properties": {
        "spark.hadoop.fs.s3.s3AccessGrants.enabled": "true",
        "spark.hadoop.fs.s3.s3AccessGrants.fallbackToIAM": "false"
      }
    }
  ]
}'

```

EMR Serverless での S3 Access Grants の使用に関する考慮事項

EMR Serverless で Amazon S3 Access Grants を使用する際の重要なサポート、互換性、および動作情報については、「Amazon EMR 管理ガイド」の「[S3 Access Grants considerations with Amazon EMR](#)」を参照してください。

を使用した Amazon EMR Serverless API コールのログ記録 AWS CloudTrail

Amazon EMR Serverless は AWS CloudTrail、EMR Serverless のユーザー、ロール、またはのサービスによって実行されたアクションを記録する AWS サービスであると統合されています。CloudTrail は、EMR Serverless のすべての API 呼び出しをイベントとしてキャプチャします。キャプチャされたコールには、EMR Serverless コンソールからのコールと、EMR Serverless API オペレーションへのコードコールが含まれます。証跡を作成する場合は、EMR Serverless のイベントなど、Amazon S3 バケットに対する CloudTrail イベントの継続的な配信を有効にします。証跡を設定しない場合でも、CloudTrail コンソールの [イベント履歴] で最新のイベントにアクセスできます。CloudTrail で収集された情報を使用して、EMR Serverless に対するリクエスト、リクエスト元の IP アドレス、リクエスト者、リクエスト日時などの詳細を確認できます。

詳細については、「[AWS CloudTrail ユーザーガイド](#)」を参照してください。

CloudTrail での EMR Serverless の情報

CloudTrail は、アカウントの作成 AWS アカウント 時に 有効になります。EMR Serverless でアクティビティが発生すると、そのアクティビティはイベント履歴の他の AWS サービスイベントとともに CloudTrail イベントに記録されます。最近のイベントは、AWS アカウントアカウントでアクセス、検索、ダウンロードできます。詳細については、「[Viewing events with CloudTrail Event history](#)」を参照してください。

EMR Serverless のイベントなど AWS アカウント、 のイベントの継続的な記録については、証跡を作成します。証跡により、CloudTrail はログファイルを Amazon S3 バケットに配信できます。デフォルトでは、コンソールで証跡を作成するときに、証跡がすべての AWS リージョンに適用されます。証跡は、AWS パーティション内のすべてのリージョンからのイベントをログに記録し、指定した Amazon S3 バケットにログファイルを配信します。さらに、CloudTrail ログで収集したイベントデータをより詳細に分析し、それに基づいて対応するため、他の AWS サービスを構成します。詳細については以下を参照してください。

- [追跡を作成するための概要](#)
- [CloudTrail がサポートされているサービスと統合](#)
- 「[CloudTrail の Amazon SNS 通知の設定](#)」
- [複数のリージョンから CloudTrail ログファイルを受け取る](#) および [複数のアカウントから CloudTrail ログファイルを受け取る](#)

すべての EMR Serverless アクションは CloudTrail によってログ記録されます。これらは、「[EMR Serverless API reference](#)」に記載されています。例えば、CreateApplication、StartJobRun、CancelJobRun の各アクションを呼び出すと、CloudTrail ログファイルにエントリが生成されます。

各イベントまたはログエントリには、誰がリクエストを生成したかという情報が含まれます。アイデンティティ情報は、以下を判別するのに役立ちます。

- リクエストがルートまたは AWS Identity and Access Management (IAM) ユーザー認証情報を使用して行われたかどうか。
- リクエストがロールまたはフェデレーションユーザーのテンポラリなセキュリティ認証情報を使用して行われたかどうか。
- リクエストが別の AWS サービスによって行われたかどうか。

詳細については、「[CloudTrail userIdentity element](#)」を参照してください。

EMR Serverless のログファイルエントリの理解

「トレイル」は、指定した Amazon S3 バケットにイベントをログファイルとして配信するように設定できます。CloudTrail のログファイルは、単一か複数のログエントリを含みます。イベントは、任意の出典からの単一のリクエストを表し、リクエストされたアクション、アクションの日時、リクエストパラメータなどに関する情報が含まれます。CloudTrail ログファイルは、パブリック API コールの順序付けられたスタックトレースではないため、特定の順序では表示されません。

以下の例は、CreateApplication アクションを示す CloudTrail ログエントリです。

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE:admin",
    "arn": "arn:aws:sts::012345678910:assumed-role/Admin/admin",
    "accountId": "012345678910",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::012345678910:role/Admin",
        "accountId": "012345678910",
        "userName": "Admin"
      },
      "webIdFederationData": {},
      "attributes": {
        "creationDate": "2022-06-01T23:46:52Z",
        "mfaAuthenticated": "false"
      }
    }
  },
  "eventTime": "2022-06-01T23:49:28Z",
  "eventSource": "emr-serverless.amazonaws.com",
  "eventName": "CreateApplication",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "PostmanRuntime/7.26.10",
  "requestParameters": {
    "name": "my-serverless-application",
```

```
    "releaseLabel": "emr-6.6",
    "type": "SPARK",
    "clientToken": "0a1b234c-de56-7890-1234-567890123456"
  },
  "responseElements": {
    "name": "my-serverless-application",
    "applicationId": "1234567890abcdef0",
    "arn": "arn:aws:emr-serverless:us-west-2:555555555555:/
applications/1234567890abcdef0"
  },
  "requestID": "890b8639-e51f-11e7-b038-EXAMPLE",
  "eventID": "874f89fa-70fc-4798-bc00-EXAMPLE",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "012345678910",
  "eventCategory": "Management"
}
```

Amazon EMR Serverless のコンプライアンス検証

EMR Serverless のセキュリティとコンプライアンスは、以下を含む複数の AWS コンプライアンス プログラムの一環として、サードパーティーの監査者によって評価されます。

- System and Organization Controls (SOC)
- Payment Card Industry Data Security Standard (PCI DSS)
- Federal Risk and Authorization Management Program (FedRAMP) Moderate
- Health Insurance Portability and Accountability Act (HIPAA)

AWS は、コンプライアンスプログラムによる対象範囲内 AWS のサービスで、特定のコンプライアンスプログラムの範囲内のサービスのリストを頻繁に更新[AWS します](#)。

サードパーティーの監査レポートは、を使用してダウンロードできます AWS Artifact。詳細については、[AWS 「アーティファクトでのレポートのダウンロード」](#)を参照してください。

AWS コンプライアンスプログラムの詳細については、[AWS 「コンプライアンスプログラム」](#)を参照してください。

EMR Serverless を使用する際のお客様のコンプライアンス責任は、組織のデータの機密性や組織のコンプライアンス目的、適用可能な法律、規制によって決定されます。EMR Serverless の使用が

HIPAA、PCI、FedRAMP Moderate などのスタンダードにコンプライアンスしていることを前提としている場合、AWS は以下を支援するリソースを提供します。

- [セキュリティとコンプライアンスのクイックスタートガイド](#)では、セキュリティとコンプライアンスに重点を置いたベースライン環境をデプロイするためのアーキテクチャ上の考慮事項と手順について説明します AWS。
- [AWS カスタマーコンプライアンスガイド](#)は、コンプライアンスの観点から責任共有モデルを理解するのに役立ちます。このガイドは、AWS のサービスを保護するためのベストプラクティスを要約したものであり、複数のフレームワーク (米国標準技術研究所 (NIST)、ペイメントカード業界セキュリティ標準評議会 (PCI)、国際標準化機構 (ISO) など) にわたるセキュリティ統制へのガイダンスがまとめられています。
- [AWS Config](#) を使用すると、社内プラクティス、業界ガイドライン、およびに対するリソースの構成の準拠状態を評価できます。
- [AWS コンプライアンスリソース](#)はワークブックのコレクションであり、ガイドはお客様の業界や地域に適用される場合があります。
- [AWS Security Hub](#) は、内のセキュリティ状態を包括的に把握 AWS し、セキュリティ業界標準とベストプラクティスへの準拠を確認するのに役立ちます。
- [AWS Audit Manager](#) – これにより AWS のサービス、AWS 使用状況を継続的に監査し、リスクの管理方法と規制や業界標準への準拠を簡素化できます。

Amazon EMR Serverless のレジリエンス

AWS グローバルインフラストラクチャは、AWS リージョンとアベイラビリティゾーンを中心に構築されています。AWS リージョンは、低レイテンシー、高スループット、高度に冗長なネットワークで接続された、物理的に分離された複数のアベイラビリティゾーンを提供します。アベイラビリティゾーンでは、ゾーン間で中断することなく自動的にフェールオーバーするアプリケーションとデータベースを設計および運用します。アベイラビリティゾーンは、従来の単一または複数のデータセンターインフラストラクチャよりも可用性、フォールトトレランス、および拡張性が優れています。

AWS リージョンとアベイラビリティゾーンの詳細については、[AWS グローバルインフラストラクチャ](#)を参照してください。

AWS グローバルインフラストラクチャに加えて、Amazon EMR Serverless は EMRFS を介して Amazon S3 との統合を提供し、データの耐障害性とバックアップのニーズをサポートします。

Amazon EMR Serverless でのインフラストラクチャセキュリティ

マネージドサービスである Amazon EMR は、AWS グローバルネットワークセキュリティで保護されています。AWS セキュリティサービスと [インフラストラクチャ AWS](#) を保護する方法については、[AWS 「クラウドセキュリティ」](#) を参照してください。インフラストラクチャセキュリティのベストプラクティスを使用して環境を AWS 設計するには、「Security Pillar AWS Well-Architected Framework」の「[Infrastructure Protection](#)」を参照してください。

AWS 公開された API コールを使用して、ネットワーク経由で Amazon EMR にアクセスします。クライアントは以下をサポートする必要があります。

- Transport Layer Security (TLS)。TLS 1.2 が必須で、TLS 1.3 をお勧めします。
- DHE (楕円ディフィー・ヘルマン鍵共有) や ECDHE (楕円曲線ディフィー・ヘルマン鍵共有) などの完全前方秘匿性 (PFS) による暗号スイート。これらのモードは、Java 7 以降など、最近のほとんどのシステムでサポートされています。

Amazon EMR Serverless での設定と脆弱性の分析

AWS は、ゲストオペレーティングシステム (OS) やデータベースのパッチ適用、ファイアウォール設定、ディザスタリカバリなどの基本的なセキュリティタスクを処理します。これらの手順は適切なサードパーティーによって確認され、認証されています。詳細については、以下のリソースを参照してください:

- [Amazon EMR Serverless のコンプライアンス検証](#)
- [責任共有モデル](#)
- [Amazon Web Services: セキュリティプロセスの概要](#)

EMR Serverless のエンドポイントとクォータ

サービスエンドポイント

プログラムでに接続するには AWS のサービス、 エンドポイントを使用します。エンドポイントは、 AWS ウェブサービスのエン트리ポイントの URL です。標準 AWS エンドポイントに加えて、一部の は AWS のサービス 選択したリージョンで FIPS エンドポイントを提供します。次の表は、EMR Serverless のサービスエンドポイントを示しています。詳細については、「[AWS のサービス endpoints](#)」を参照してください。

EMR Serverless サービスエンドポイント

リージョン名	リージョン	エンドポイント	プロトコル
米国東部 (オハイオ)	us-east-2 (次のアベイラビリティゾーンに限定: use2-az1、use2-az2、use2-az3)	emr-serverless.us-east-2.amazonaws.com	HTTPS
米国東部 (バージニア北部)	us-east-1 (次のアベイラビリティゾーンに限定: use1-az1、use1-az2、use1-az4、use1-az5、use1-az6)	emr-serverless.us-east-1.amazonaws.com emr-serverless-fips.us-east-1.amazonaws.com	HTTPS
米国西部 (北カリフォルニア)	us-west-1	emr-serverless.us-west-1.amazonaws.com	HTTPS
米国西部 (オレゴン)	us-west-2	emr-serverless.us-	HTTPS

リージョン名	リージョン	エンドポイント	プロトコル
		west-2.amazonaws.com emr-serverless-fips.us-west-2.amazonaws.com	
アフリカ (ケープタウン)	af-south-1	emr-serverless.af-south-1.amazonaws.com	HTTPS
アジアパシフィック (香港)	ap-east-1	emr-serverless.ap-east-1.amazonaws.com	HTTPS
アジアパシフィック (ジャカルタ)	ap-southeast-3	emr-serverless.ap-southeast-3.amazonaws.com	HTTPS
アジアパシフィック (メルボルン)	ap-southeast-4	emr-serverless.ap-southeast-4.amazonaws.com	HTTPS
アジアパシフィック (マレーシア)	ap-southeast-5	emr-serverless.ap-southeast-5.amazonaws.com	HTTPS

リージョン名	リージョン	エンドポイント	プロトコル
アジアパシフィック (ムンバイ)	ap-south-1	emr-serverless.ap-south-1.amazonaws.com	HTTPS
アジアパシフィック (大阪)	ap-northeast-3	emr-serverless.ap-northeast-3.amazonaws.com	HTTPS
アジアパシフィック (ソウル)	ap-northeast-2	emr-serverless.ap-northeast-2.amazonaws.com	HTTPS
アジアパシフィック (シンガポール)	ap-southeast-1	emr-serverless.ap-southeast-1.amazonaws.com	HTTPS
アジアパシフィック (シドニー)	ap-southeast-2	emr-serverless.ap-southeast-2.amazonaws.com	HTTPS
アジアパシフィック (東京)	ap-northeast-1	emr-serverless.ap-northeast-1.amazonaws.com	HTTPS

リージョン名	リージョン	エンドポイント	プロトコル
カナダ (中部)	ca-central-1 (次のアベイラビリティゾーンに限定: cac1-az1、cac1-az2)	emr-serverless.ca-central-1.amazonaws.com	HTTPS
カナダ西部 (カルガリー)	ca-west-1	emr-serverless.ca-west-1.amazonaws.com	HTTPS
欧州 (フランクフルト)	eu-central-1	emr-serverless.eu-central-1.amazonaws.com	HTTPS
欧州 (チューリッヒ)	eu-central-2	emr-serverless.eu-central-2.amazonaws.com	HTTPS
欧州 (アイルランド)	eu-west-1	emr-serverless.eu-west-1.amazonaws.com	HTTPS
欧州 (ロンドン)	eu-west-2	emr-serverless.eu-west-2.amazonaws.com	HTTPS
欧州 (ミラノ)	eu-south-1	emr-serverless.eu-south-1.amazonaws.com	HTTPS

リージョン名	リージョン	エンドポイント	プロトコル
欧州 (パリ)	eu-west-3	emr-serverless.eu-west-3.amazonaws.com	HTTPS
欧州 (スペイン)	eu-south-2	emr-serverless.eu-south-2.amazonaws.com	HTTPS
欧州 (ストックホルム)	eu-north-1	emr-serverless.eu-north-1.amazonaws.com	HTTPS
イスラエル (テルアビブ)	il-central-1	emr-serverless.il-central-1.amazonaws.com	HTTPS
中東 (バーレーン)	me-south-1	emr-serverless.me-south-1.amazonaws.com	HTTPS
中東 (アラブ首長国連邦)	me-central-1	emr-serverless.me-central-1.amazonaws.com	HTTPS
南米 (サンパウロ)	sa-east-1	emr-serverless.sa-east-1.amazonaws.com	HTTPS

リージョン名	リージョン	エンドポイント	プロトコル
中国 (北京)	cn-north-1 (次のアベイラビリティーゾーンに限定: cnn1-az1、cnn1-az2)	emr-serverless.cn-north-1.amazonaws.com.cn	HTTPS
AWS GovCloud (米国 東部)	us-gov-east-1	emr-serverless.us-gov-east-1.amazonaws.com	HTTPS
AWS GovCloud (米国 西部)	us-gov-west-1	emr-serverless.us-gov-west-1.amazonaws.com	HTTPS

Service Quotas

制限とも呼ばれるサービスクォータは、が AWS アカウント 使用できるサービスリソースまたはオペレーションの最大数です。EMR Serverless は、AWS/Usage 名前空間で 1 分ごとにサービスクォータ使用状況メトリクスを収集します。

Note

新しい AWS アカウントには、時間の経過とともに増加する可能性がある初期の低いクォータがあります。Amazon EMR Serverless は、各 内のアカウントの使用状況を監視し AWS リージョン、使用状況に基づいてクォータを自動的に引き上げます。

次のテーブルに EMR Serverless のサービスクォータの一覧を示します。詳細については、「[AWS のサービスクォータ](#)」を参照してください。

名前	デフォルトの制限	引き上げ可能?	説明
アカウントあたりの最大同時実行 vCPU 数	16	はい	現在の AWS リージョンのアカウントで同時に実行できる vCPU の最大数。
アカウントあたりのキューに入れられた最大ジョブ数	2000	はい	現在の AWS リージョンのアカウントのキューに入れられたジョブの最大数。

API の制限

以下では、AWS アカウントのリージョンあたりの API 制限について説明します。

[リソース]	デフォルトのクォータ
ListApplications	1 秒あたり 10 件のトランザクション。1 秒あたり 50 件のトランザクションをバースト。
CreateApplication	1 秒あたり 1 件のトランザクション。1 秒あたり 25 件のトランザクションをバースト。
DeleteApplication	1 秒あたり 1 件のトランザクション。1 秒あたり 25 件のトランザクションをバースト。
GetApplication	1 秒あたり 10 件のトランザクション。1 秒あたり 50 件のトランザクションをバースト。
UpdateApplication	1 秒あたり 1 件のトランザクション。1 秒あたり 25 件のトランザクションをバースト。
ListJobRuns	1 秒あたり 1 件のトランザクション。1 秒あたり 25 件のトランザクションをバースト。

[リソース]	デフォルトのクォータ
StartJobRun	1 秒あたり 1 件のトランザクション。1 秒あたり 25 件のトランザクションをバースト。
GetDashboardForJobRun	1 秒あたり 1 件のトランザクション。1 秒あたり 2 件のトランザクションをバースト。
CancelJobRun	1 秒あたり 1 件のトランザクション。1 秒あたり 25 件のトランザクションをバースト。
GetJobRun	1 秒あたり 10 件のトランザクション。1 秒あたり 50 件のトランザクションをバースト。
StartApplication	1 秒あたり 1 件のトランザクション。1 秒あたり 25 件のトランザクションをバースト。
StopApplication	1 秒あたり 1 件のトランザクション。1 秒あたり 25 件のトランザクションをバースト。

その他の考慮事項

次のリストには、EMR Serverless に関するその他の考慮事項が含まれています。

• EMR Serverless は、以下でご利用いただけます AWS リージョン。

- 米国東部(オハイオ)
- 米国東部 (バージニア北部)
- 米国西部 (北カリフォルニア)
- 米国西部 (オレゴン)
- アフリカ (ケープタウン)
- アジアパシフィック (香港)
- アジアパシフィック (ジャカルタ)
- アジアパシフィック (ムンバイ)
- アジアパシフィック (大阪)
- アジアパシフィック (ソウル)
- アジアパシフィック (シンガポール)
- アジアパシフィック (シドニー)
- アジアパシフィック (東京)
- カナダ (中部)
- 欧州 (フランクフルト)
- 欧州 (アイルランド)
- 欧州 (ロンドン)
- 欧州 (ミラノ)
- 欧州 (パリ)
- 欧州 (スペイン)
- 欧州 (ストックホルム)
- 中東 (バーレーン)
- 中東 (アラブ首長国連邦)
- 南米 (サンパウロ)
- AWS GovCloud (米国東部)
- AWS GovCloud (米国西部)

これらのリージョンに関連付けられたエンドポイントのリストについては、「[サービスエンドポイント](#)」を参照してください。

- ジョブ実行のデフォルトのタイムアウトは 12 時間です。この設定は、startJobRun API または AWS SDK の executionTimeoutMinutes プロパティを使用して変更できます。ジョブの実行がタイムアウトしないようにする場合は、executionTimeoutMinutes を 0 に設定します。例えば、ストリーミングアプリケーションがある場合は、executionTimeoutMinutes を 0 に設定してストリーミングジョブを継続的に実行します。
- getJobRun API の billedResourceUtilization プロパティには、AWS がジョブ実行に対して請求した vCPU、メモリ、ストレージの集計が表示されます。課金されるリソースには、ワーカーの最小使用量である 1 分とワーカーあたり 20 GB を超える追加のストレージが含まれます。これらのリソースには、アイドル状態の事前初期化されたワーカーの使用は含まれません。
- VPC 接続がない場合、ジョブは同じの一部の AWS のサービス エンドポイントにアクセスできます AWS リージョン。これらのサービスには、Amazon S3、AWS Glue、AWS Lake Formation、Amazon CloudWatch Logs、AWS KMS AWS Security Token Service、Amazon DynamoDB、および が含まれます AWS Secrets Manager。VPC 接続を有効にして [AWS PrivateLink](#) を介して他の AWS のサービス にアクセスすることはできますが、これを行う必要はありません。外部サービスにアクセスするには、VPC を使用してアプリケーションを作成します。
- EMR Serverless は HDFS をサポートしていません。ワーカーのローカルディスクは、ジョブの実行中に EMR Serverless がデータのシャッフルと処理に使用する一時ストレージです。

Amazon EMR Serverless のリリースバージョン

Amazon EMR リリースは、ビッグデータエコシステムの一連のオープンソースアプリケーションです。各リリースには、ビッグデータアプリケーション、コンポーネント、機能が含まれ、ジョブを実行するときに Amazon EMR Serverless でデプロイして設定することを選択します。

Amazon EMR 6.6.0 以降では、EMR Serverless をデプロイします。このデプロイオプションは、これより前の Amazon EMR リリースバージョンでは使用できません。ジョブを送信する際は、次のいずれかのサポートされているリリースを指定します。

トピック

- [AWS runtime for Apache Spark \(emr-spark-8.0-preview\)](#)
- [EMR Serverless 7.12.0](#)
- [EMR Serverless 7.11.0](#)
- [EMR Serverless 7.10.0](#)
- [EMR Serverless 7.9.0](#)
- [EMR Serverless 7.8.0](#)
- [EMR Serverless 7.7.0](#)
- [EMR Serverless 7.6.0](#)
- [EMR Serverless 7.5.0](#)
- [EMR Serverless 7.4.0](#)
- [EMR Serverless 7.3.0](#)
- [EMR Serverless 7.2.0](#)
- [EMR Serverless 7.1.0](#)
- [EMR Serverless 7.0.0](#)
- [EMR Serverless 6.15.0](#)
- [EMR Serverless 6.14.0](#)
- [EMR Serverless 6.13.0](#)
- [EMR Serverless 6.12.0](#)
- [EMR Serverless 6.11.0](#)
- [EMR Serverless 6.10.0](#)
- [EMR Serverless 6.9.0](#)

- [EMR Serverless 6.8.0](#)
- [EMR Serverless 6.7.0](#)
- [EMR Serverless 6.6.0](#)

AWS runtime for Apache Spark (emr-spark-8.0-preview)

次の表に、で使用できるアプリケーションバージョンを示します AWS runtime for Apache Spark (emr-spark-8.0-preview)。

アプリケーションバージョン情報

アプリケーション	バージョン
Spark	4.0.1-amzn-0

AWS runtime for Apache Spark (emr-spark-8.0-preview) リリースノート

- プレビューリリース - これは、Apache Spark 4.0.1 AWS runtime for Apache Sparkを搭載したのプレビューリリースです。このプレビューは EMR Serverless でのみ使用できます。
- リージョンの可用性 - このプレビューリリースは、中国および AWS GovCloud (米国) AWS リージョンを除く、EMR Serverless が利用可能なすべてのリージョンで利用できます。
- アプリケーションバージョン情報 - このリリースには、次のアプリケーションバージョンが付属しています。
 - AWS SDK for Java 2.35.5, 1.12.792
 - Python3.9、3.11, 3.12
 - Scala 2.13.16
 - AmazonCloudWatchAgent 1.300034.0-amzn-0
 - デルタ 4.0.0-amzn-0-spark
 - Iceberg 1.10.0-amzn-spark-0
- このリリースには、Corretto 17 (OpenJDK) をサポートするアプリケーション用に、デフォルトで Amazon Corretto 17 (OpenJDK 上に構築) が付属しています。
- プレビューの制限 - このプレビューリリースでは、次の機能は使用できません。
 - インタラクティブおよび統合機能: SageMaker Unified Studio、EMR Studio 統合、Spark Connect、Livy、JupyterEnterpriseGateway はサポートされていません。

- テーブル形式とアクセスコントロール: Hudi、Delta ユニバーサル形式、および行レベルまたは列レベルのフィルタリングと DDL/DML 演算子を使用したきめ細かなアクセスコントロール (FGAC) はサポートされていません。
- データコネクタ: spark-sql-kinesis、emr-dynamodb、spark-redshift コネクタは使用できません。
- 履歴サーバー: 永続 Spark 履歴サーバーは、このプレビューリリースでは利用できません。ユーザーは引き続きライブ Spark UI にアクセスして、アクティブなサーバーレスジョブをリアルタイムでモニタリングおよびデバッグできます。
- 特殊な機能: マテリアライズドビューは使用できません。
- プレビュー機能 - このプレビューリリースでは、次の機能をテストできます。このプレビューリリースは、本番ワークロードにはお勧めしません。
 - SQL 機能: より厳密な型処理を使用する ANSI SQL モード、連鎖オペレーション用の SQL PIPE 構文 (`|>`)、半構造化 JSON データ用の VARIANT データ型、制御フローステートメントとセッション変数を使用した SQL スクリプト、SQL ユーザー定義関数。
 - ストリーミングの機能強化: transformWithState 演算子を使用した任意ステートフル処理 API v2、クエリ可能なストリーミング状態 (実験的) 用のステートデータソースリーダー、RocksDB 変更ログチェックポイントが改善された拡張ステートストア。
 - テーブル形式のサポート: VARIANT データ型をサポートする Apache Iceberg v3、AWS S3 Tables 統合、Iceberg、Delta Lake、Hive テーブル AWS Lake Formation 用の 1 つのフルテーブルアクセス (FTA)。
- 追加のドキュメント - 追加の Apache Spark ドキュメントについては、[「Apache Spark 4.0.1 リリースドキュメント」](#)を参照してください。

開始方法

Apache Spark 4.0.1 プレビューを開始するには、CLI を使用して EMR Serverless AWS アプリケーションを作成します。

```
aws emr-serverless create-application --type spark \
  --release-label emr-spark-8.0-preview \
  --region us-east-1 --name spark4-preview
```

EMR Serverless 7.12.0

次の表に、7.12.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.6
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.12.0 リリースノート

- 新しい特徴
 - EMR Serverless 用のサーバーレスストレージ - Amazon EMR Serverless は、EMR リリース 7.12 以降でサーバーレスストレージを導入し、Apache Spark ワークロードのローカルディスクプロビジョニングを排除します。EMR Serverless は、シャッフルなどの中間データオペレーションをストレージ料金なしで自動的に処理します。サーバーレスストレージはストレージとコンピューティングを分離するため、Spark はワーカーをアクティブのままにして一時データを保持するのではなく、アイドル状態のときにすぐにワーカーを解放できます。詳細についてはaws.amazon.com/serverless-storage-for-emr-serverlessを参照してください。
 - Iceberg マテリアライズドビュー - Amazon EMR 7.12.0 以降、Amazon EMR Spark は Iceberg マテリアライズドビュー (MV) の作成と管理をサポートしています。
 - Hudi フルテーブルアクセス - Amazon EMR 7.12.0 以降、Amazon EMR は Lake Formation で定義されたポリシーに基づいて、Apache Spark で Apache Hudi のフルテーブルアクセス (FTA) コントロールをサポートするようになりました。この機能を使用すると、ジョブロールが完全なテーブルアクセス権を持っているときに、Lake Formation 登録済みテーブルの Amazon EMR Spark ジョブからの読み取りおよび書き込みオペレーションが有効になります。
 - Iceberg バージョンアップグレード - Amazon EMR 7.12.0 が Apache Iceberg バージョン 1.10 をサポート

EMR Serverless 7.11.0

次の表に、7.11.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.6

アプリケーション	バージョン
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.11.0 リリースノート

- 最大ジョブ実行時間 - BATCH ジョブの`executionTimeoutMinutesStartJobRun`実行中の最大値は、このリリースから 7 日後です。バッチジョブ実行では、`executionTimeoutMinutes`をタイムアウト0に設定できなくなりました。

EMR Serverless 7.10.0

次の表に、7.10.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.5
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.10.0 リリースノート

- EMR Serverless のメトリクス – モニタリングメトリクスは、`ApplicationName` および `JobName` デイメンションに焦点を当てて再構築されます。今後、古いメトリクスは更新されなくなります。詳細については、「[Monitoring EMR Serverless applications and jobs](#)」を参照してください。

EMR Serverless 7.9.0

次の表に、7.9.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.5
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.8.0

次の表に、7.8.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.4
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.7.0

次の表に、7.7.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.3
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.6.0

次の表に、7.6.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.3
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.5.0

次の表に、7.5.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.2
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.4.0

次の表に、7.4.0 EMR Serverless で使用できるアプリケーションバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.2
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.3.0

次の表に、EMR Serverless 7.3.0 で使用できるアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.3.0 リリースノート

- EMR Serverless でのジョブの同時実行とキュー – Amazon EMR リリース 7.3.0 以降で新しい EMR Serverless アプリケーションを作成することにより、ジョブの同時実行とキューイングがデフォルトで有効になります。詳細については、「[the section called “ジョブの同時実行とキューイング”](#)」を参照してください。同時実行とキューの開始方法の詳細と、機能に関する考慮事項のリストも含まれています。

EMR Serverless 7.2.0

次の表に、EMR Serverless 7.2.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.2.0 リリースノート

- EMR Serverless を使用した Lake Formation – を使用して AWS Lake Formation、S3 でバックアップされた Data Catalog テーブルにきめ細かなアクセスコントロールを適用できるようになりました。この機能を使用すると、EMR Serverless Spark ジョブ内の読み取りクエリのテーブル、行、列、セルレベルのアクセスコントロールを設定できます。詳細については、[the section called “FGAC のための Lake Formation”](#) および [the section called “考慮事項と制限事項”](#) を参照してください。

EMR Serverless 7.1.0

次の表に、EMR Serverless 7.1.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.0
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 7.0.0

次の表に、EMR Serverless 7.0.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.5.0
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.15.0

次の表に、EMR Serverless 6.15.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.4.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.15.0 リリースノート

- TLS サポート – Amazon EMR Serverless リリース 6.15.0 以降では、Spark ジョブ実行のワーカー間で相互 TLS 暗号化通信を有効にします。有効にすると、EMR Serverless は、TLS ハンドシェイク中にワーカーが相互に認証し、データを安全に処理するための暗号化されたチャネルを確立するために使用するジョブ実行に基づいてプロビジョニングする、ワーカーごとに一意の証明書を自動的に生成します。相互 TLS 暗号化の詳細については、「[Inter-worker encryption](#)」を参照してください。

EMR Serverless 6.14.0

次の表に、EMR Serverless 6.14.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.4.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.13.0

次の表に、EMR Serverless 6.13.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.4.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.12.0

次の表に、EMR Serverless 6.12.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.4.0
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.11.0

次の表に、EMR Serverless 6.11.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.3.2
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.11.0 リリースノート

- [他のアカウントで S3 リソースにアクセスする](#) - リリース 6.11.0 以降では、EMR Serverless から異なる AWS アカウントで Amazon S3 バケットにアクセスするときに引き受ける複数の IAM ロールを設定できます。

EMR Serverless 6.10.0

次の表に、EMR Serverless 6.10.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.3.1
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.10.0 リリースノート

- リリース 6.10.0 以降の EMR Serverless アプリケーションの場合、`spark.dynamicAllocation.maxExecutors` プロパティのデフォルト値は `infinity` です。これより前のリリースはデフォルトで `100` です。詳細については、[Spark ジョブのプロパティ](#) を参照してください。

EMR Serverless 6.9.0

次の表に、EMR Serverless 6.9.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.3.0
Apache Hive	3.1.3
Apache Tez	0.10.2

EMR Serverless 6.9.0 リリースノート

- Apache Spark 用の Amazon Redshift の統合は、Amazon EMR リリース 6.9.0 以降に含まれています。以前はオープンソースツールであったこのネイティブインテグレーションは Spark コネクタと呼ばれるもので、これを使用して Apache Spark アプリケーションを構築することで、Amazon Redshift と Amazon Redshift Serverless 内のデータを読み書きできます。詳細については、「[Amazon EMR Serverless での Amazon Redshift integration for Apache Spark の使用](#)」を参照してください。
- EMR Serverless リリース 6.9.0 では、AWS Graviton2 (arm64) アーキテクチャのサポートが追加されました。create-application API および update-application API の `architecture` パラメータを使用して arm64 アーキテクチャを選択できます。詳細については、[Amazon EMR Serverless アーキテクチャオプション](#) を参照してください。
- EMR Serverless Spark アプリケーションおよび Hive アプリケーションから直接 Amazon DynamoDB テーブルをエクスポート、インポート、クエリ、結合できるようになりました。詳細については、[Amazon EMR Serverless を使用した DynamoDB への接続](#) を参照してください。

既知の問題

- Amazon Redshift integration for Apache Spark を使用している場合に、time、timetz、timestamp、timestampz のいずれかにマイクロ秒の精度を Parquet 形式で設定していると、コネクタがその時間値を最も近いミリ秒値に四捨五入します。回避策として、テキストアンロード形式 unload_s3_format パラメータを使用してください。

EMR Serverless 6.8.0

次の表に、EMR Serverless 6.8.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.3.0
Apache Hive	3.1.3
Apache Tez	0.9.2

EMR Serverless 6.7.0

次の表に、EMR Serverless 6.7.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.2.1
Apache Hive	3.1.3
Apache Tez	0.9.2

エンジン固有の変更、機能強化、解決された問題

次の表に、新しいエンジン固有の機能を示します。

変更	説明
機能	Tez スケジューラがコンテナのプリエンプションではなく、Tez タスクのプリエンプションをサポートするようになりました

EMR Serverless 6.6.0

次の表に、EMR Serverless 6.6.0 で使用可能なアプリケーションのバージョンを示します。

アプリケーション	バージョン
Apache Spark	3.2.0
Apache Hive	3.1.2
Apache Tez	0.9.2

EMR Serverless の初期リリースノート

- EMR Serverless は、Spark 設定分類 `spark-defaults` をサポートしています。この分類は、Spark の `spark-defaults.conf` XML ファイルの値を変更します。設定分類を使用すると、アプリケーションをカスタマイズできます。詳細については、「[Configure applications](#)」を参照してください。
- EMR Serverless は、Hive 設定分類 `hive-site`、`tez-site`、`emrfs-site`、`core-site` をサポートしています。この分類では、Hive の `hive-site.xml` ファイル、Tez の `tez-site.xml` ファイル、Amazon EMR の EMRFS 設定、または Hadoop の `core-site.xml` ファイルの値をそれぞれ変更できます。設定分類を使用すると、アプリケーションをカスタマイズできます。詳細については、「[Configure applications](#)」を参照してください。

エンジン固有の変更、機能強化、解決された問題

- 次の表に、Hive および Tez のバックポートを示します。

Hive と Tez の変更

変更	説明
バックポート	TEZ-4430 : <code>tez.task.launch.cmd-opts</code> プロパティの問題を修正しました
バックポート	HIVE-25971 : キャッシュに保存されたスレッドプールのオープンによる Tez タスクのシャットダウン遅延を修正しました

ドキュメント履歴

次の表は、EMR Serverless の前回のリリースからの重要な変更を示しています。このドキュメントの更新情報は、RSS フィードに加入して取得できます。

変更	説明	日付
新しいプレビューリリース	AWS runtime for Apache Spark (emr-spark-8.0-preview)	2025 年 11 月 21 日
新規リリース	EMR Serverless 7.2.0	2024 年 7 月 25 日
新規リリース	EMR Serverless 7.1.0	2024 年 4 月 17 日
既存のポリシーの更新。	AmazonEMRServerlessServiceRolePolicy ポリシー に新しい Sid、CloudWatchPolicyStatement、EC2PolicyStatement を追加しました。	2024 年 1 月 25 日
新規リリース	EMR Serverless 7.0.0	2023 年 12 月 29 日
新規リリース	EMR Serverless 6.15.0	2023 年 11 月 17 日
新機能	EMR Serverless (6.11 以降) とは異なるアカウントの Amazon S3 バケットにアクセスするときに引き受ける複数の IAM ロールを設定する	2023 年 10 月 18 日
新規リリース	EMR Serverless 6.14.0	2023 年 10 月 17 日
新機能	EMR Serverless のデフォルトのアプリケーション設定	2023 年 9 月 25 日

デフォルトの Hive プロパティの更新	hive.driver.disk、hive.tez.disk.size、hive.tez.auto.reducer.parallelism、tez.grouping.min-size の Hive ジョブプロパティ のデフォルト値を更新しました。	2023 年 9 月 12 日
新規リリース	EMR Serverless 6.13.0	2023 年 9 月 11 日
新規リリース	EMR Serverless 6.12.0	2023 年 7 月 21 日
新規リリース	EMR Serverless 6.11.0	2023 年 6 月 8 日
サービスにリンクされたロールポリシーの更新	AmazonEMRServerlessServiceRolePolicy _SLR ロールを更新して、 "AWS/Usage" 名前空間でアカウントレベルの使用状況を公開しました。	2023 年 4 月 20 日
EMR Serverless の一般提供 (GA)	これは EMR Serverless の最初のパブリックリリースです。	2022 年 6 月 1 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。