使用 TypeScript AWS CDK 中的 建立 IaC 專案的最佳實務

AWS 方案指引

AWS 方案指引: 使用 TypeScript AWS CDK 中的 建立 IaC 專案的最佳實務

Table of Contents

使用 TypeScript AWS CDK 中的 建立 IaC 專案的最佳實務

Sandeep Gawande、Mason Cahill、Sandip Gangapadhyay、Siamak Heshmati 和 Rajneesh Tyagi，Amazon Web Services (AWS)

2025 年 10 月 (文件歷史記錄)

本指南提供了使用 TypeScript 中的 AWS Cloud Development Kit (AWS CDK) 建置和部署大規模基礎設施即程式碼 (IaC) 專案的建議和最佳實務。 AWS CDK 是一個架構，用於在程式碼中定義雲端基礎設施，並透過 佈建該基礎設施 AWS CloudFormation。如果您沒有定義明確的專案結構，為大規模專案建置和管理 AWS CDK 程式碼庫可能具有挑戰性。為了應對這些挑戰，一些組織對大型專案使用反面模式，但這些模式可能會減慢您的專案速度並產生其他問題，對您的組織產生負面影響。例如，反面模式可能會使開發人員登入、錯誤修正和新功能的採用變得複雜並減慢速度。

本指南提供了使用反面模式的替代方案，並顯示如何組織程式碼以實現可擴展性、測試以及與安全最佳實務的一致性。您可以使用本指南來提高 IaC 專案的程式碼品質並最大限度地提高業務敏捷性。本指南適用於架構師、技術主管、基礎設施工程師，以及尋求為大規模 AWS CDK 專案建置架構良好的專案的任何其他角色。

目標

- 降低成本 – 您可以使用 AWS CDK 來設計自己的可重複使用元件，以符合組織的安全性、合規性和控管需求。您也可以輕鬆地在組織中共用元件，以便快速啟動預設符合最佳實務的新專案。
- 加快上市時間 – 利用 中的熟悉功能 AWS CDK 來加速開發程序。這會增加部署的可重複使用性，並減少開發工作。
- 提高開發人員生產力 – 開發人員可以使用熟悉的程式設計語言來定義基礎設施。這有助於開發人員表達和維護 AWS 資源。這可能會導致提高開發人員效率和協同合作。

最佳實務

本節將概要介紹下列最佳實務：

組織大型專案的程式碼

為什麼程式碼組織很重要

大規模 AWS CDK 專案具有高品質、定義明確的結構至關重要。隨著專案變得越來越大，支援的功能和建構模組數量增加，程式碼導覽變得更加困難。此困難可能會影響生產力並減慢開發人員登入速度。

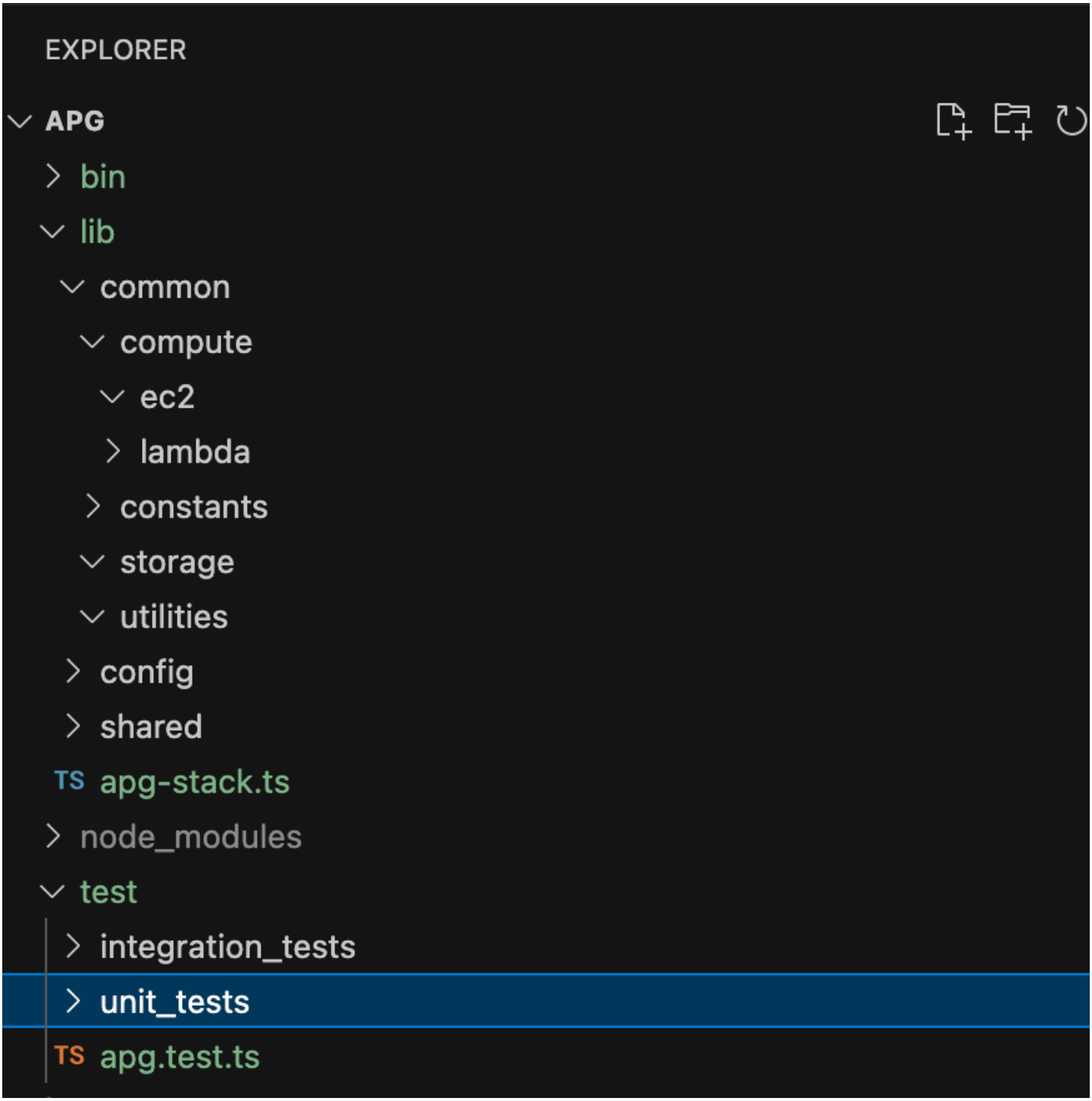
如何組織您的程式碼以進行擴展

為了實現高水準的程式碼靈活性和可讀性，我們建議您根據功能將程式碼劃分為邏輯部分。此劃分反映了這樣一個事實：您的大多數建構模組都用於不同的業務領域。例如，您的前端和後端應用程式都可能需要 AWS Lambda 函數，並使用相同的原始程式碼。工廠可以在不向用戶端暴露建立邏輯的情況下建立物件，並使用通用介面來引用新建立的物件。您可以使用工廠作為在程式碼庫中建立一致行為的有效模式。此外，工廠可以作為單一事實來源，協助您避免重複程式碼並使疑難排解變得更加容易。

為了更好地了解工廠的運作方式，請考慮汽車製造商的範例。汽車製造商不需要具備製造輪胎所需的知識和基礎設施。相反，汽車製造商將該專業知識外包給專業的輪胎製造商，然後只需根據需要從該製造商訂購輪胎即可。相同的原則也適用於程式碼。例如，您可以建立一個能夠建置高品質 Lambda 函數的 Lambda 工廠，然後在需要建立 Lambda 函數時在程式碼中呼叫該 Lambda 工廠。同樣，您可以使用此相同的外包程序來解耦應用程式並建置模組化元件。

程式碼組織範例

下列 TypeScript 範例專案 (如下列影像所示) 包含一個 common 資料夾，您可以在其中儲存所有建構模組或通用功能。



例如，compute 資料夾 (駐留在 common 資料夾中) 保存不同運算建構模組的所有邏輯。新開發人員可以輕鬆新增運算建構模組，而不會影響其他資源。所有其他建構不需要在內部建立新資源。相反，這些建構模組只需呼叫通用建構模組工廠。您可以用相同的方式組織其他建構模組，例如儲存。

組態包含環境型資料，您必須從保存邏輯的 common 資料夾解耦。我們建議您將常用 config 資料放在共用資料夾中。我們還建議您使用 utilities 資料夾來提供所有協助程式功能並清理指令碼。

開發可重複使用的模式

軟體設計模式是軟體開發中常見問題的可重複使用解決方案。其作為指南或範例，可協助軟體工程師建立遵循最佳實務的產品。本節提供兩種可重複使用模式的概觀，您可以在 AWS CDK 程式碼庫中使用：抽象工廠模式和責任鏈模式。您可以使用每個模式作為藍圖，並針對程式碼中的特定設計問題進行自訂。如需有關設計模式的詳細資訊，請參閱 Refactoring.Guru 文件中的設計模式。

抽象工廠

「抽象工廠」模式提供了用於建立相關或相依物件的系列的介面，而無需指定其具體類別。此模式適用於下列使用案例：

- 當用戶端獨立於您在系統中建立和撰寫物件的方式時
- 當系統由多個物件系列組成，且這些系列設計為一起使用時
- 當您必須具有執行期值來建構特定相依性時

如需有關「抽象工廠」模式的詳細資訊，請參閱 Refactoring.Guru 文件中的 TypeScript 中的抽象工廠。

下列程式碼範例顯示如何使用「抽象工廠」模式建構 Amazon Elastic Block Store (Amazon EBS) 儲存工廠。

```
abstract class EBSStorage {
    abstract initialize(): void;
}

class ProductEbs extends EBSStorage{
    constructor(value: String) {
        super();
        console.log(value);
    }
    initialize(): void {}
```

```
}

abstract class AbstractFactory {
    abstract createEbs(): EBSStorage
}

class EbsFactory extends AbstractFactory {
    createEbs(): ProductEbs{
        return new ProductEbs('EBS Created.')
    }
}

const ebs = new EbsFactory();
ebs.createEbs();
```

責任鏈

責任鏈是一種行為設計模式，可讓您沿著潛在處理常式鏈傳遞請求，直到其中一個處理常式處理該請求。「責任鏈」模式適用於下列使用案例：

- 當在執行期確定的多個物件是處理請求的候選者時
- 當您不想在程式碼中明確指定處理常式時
- 當您想要向多個物件之一發出請求而不明確指定接收者時

如需有關「責任鏈」模式的詳細資訊，請參閱 Refactoring.Guru 文件中的 TypeScript 中的責任鏈。

下列程式碼顯示如何使用「責任鏈」模式建置完成任務所需的一系列動作的範例。

```
interface Handler {
    setNext(handler: Handler): Handler;
    handle(request: string): string;
}
abstract class AbstractHandler implements Handler
{
    private nextHandler: Handler;
    public setNext(handler: Handler): Handler {
        this.nextHandler = handler;
        return handler;
    }

    public handle(request: string): string {
```

```
        if (this.nextHandler) {
            return this.nextHandler.handle(request);
        }
        return '';
    }
}

class KMSHandler extends AbstractHandler {
    public handle(request: string): string {
        return super.handle(request);
    }
}
```

建立或延伸建構模組

什麼是建構模組

建構是 AWS CDK 應用程式的基本建置區塊。建構可以代表單一 AWS 資源，例如 Amazon Simple Storage Service (Amazon S3) 儲存貯體，也可以是由多個 AWS 相關資源組成的更高層級抽象。建構模組的元件可以包括具有關聯運算容量的工作者佇列，或具有監控資源和儀表板的排定作業。AWS CDK 包含稱為 AWS 建構程式庫的建構集合。程式庫包含每個 的建構 AWS 服務。您可以使用 Construct Hub 來探索來自 AWS第三方和開放原始碼 AWS CDK 社群的其他建構。

不同類型的建構模組是什麼

有三種不同類型的建構 AWS CDK：

- L1 建構模組 – 第 1 層即 L1 建構模組正是 CloudFormation 定義的資源 – 不多也不少。您必須自行提供組態所需的資源。這些 L1 建構非常基本，必須手動設定。 L1 建構具有`Cfn`字首並直接對應至 CloudFormation 規格。 AWS CDK 只要 CloudFormation 支援這些服務， AWS 服務 就會支援新的 。CfnBucket 是 L1 建構的良好範例。此類別代表 S3 儲存貯體，您必須在其中明確設定所有屬性。如果您找不到 L2 或 L3 建構，建議您只使用 L1 建構。
- L2 建構模組 – 第 2 層即 L2 建構模組具有通用樣板程式碼和 glue 邏輯。這些建構模組隨附方便的預設值，並減少您需要知道的知識量。L2 建構會使用意圖型 APIs 來建構您的資源，並通常會封裝其對應的 L1 模組。L2 建構模組的一個良好範例是儲存貯體。此類別建立具有預設屬性和方法的 S3 儲存貯體，例如 bucket.addLifeCycleRule()，它會為儲存貯體新增生命週期規則。
- L3 建構模組 – 第 3 層即 L3 建構模組稱為模式。L3 建構模組旨在協助您完成 中的常見任務 AWS，通常涉及多種資源。這些比 L2 建構模組更加具體、更固定，並且為特定使用案例提供服務。例如， aws-ecs-patterns.ApplicationLoadBalancedFargateService 建構函數代表包含 AWS Fargate

使用 Application Load Balancer 之容器叢集的架構。另一個範例是 aws-apigateway.LambdaRestApi 建構模組。此建構模組代表 Lambda 函數支援的 Amazon API Gateway API。

隨著建構模組層級變得更高，對如何使用這些建構模組進行了更多假設。這可讓您為高度特定的使用案例提供具有更有效預設值的介面。

如何建立自己的建構模組

若要定義自己的建構模組，您必須遵循特定的方法。這是因為所有建構模組都延伸 `Construct` 類別。`Construct` 類別是建構模組樹狀目錄的建構區塊。建構模組在延伸 `Construct` 基本類別的類別中實作。所有建構模組在初始化時都採用以下三個參數：

- 範圍 – 建構的父系或擁有者，可以是堆疊或其他建構，決定其在建構樹中的位置。通常必須為範圍傳遞 `this` (或 Python 中的 `self`)，它表示目前物件。
- id – 在此範圍內必須是唯一的識別符。識別符可做為目前建構中定義之所有項目的命名空間，並用於配置唯一身分，例如資源名稱和 CloudFormation 邏輯 IDs。
- Props – 定義建構結構初始組態的一組屬性。

下列範例顯示如何設定建構模組。

```
import { Construct } from 'constructs';

export interface CustomProps {
  // List all the properties
  Name: string;
}
export class MyConstruct extends Construct {
  constructor(scope: Construct, id: string, props: CustomProps) {
    super(scope, id);

    // TODO
  }
}
```

建立或延伸 L2 建構模組

L2 建構模組代表「雲端元件」，並封裝了 CloudFormation 建立此元件必須具有的所有內容。L2 建構可以包含一或多個 AWS 資源，您可以自行自訂建構。建立或延伸 L2 建構模組的優點是您可以重複使用 CloudFormation 堆疊中的元件，而無需重新定義程式碼。您可以簡單地將建構模組匯入為類別。

當與現有建構有「是」關係時，您可以擴展現有建構，以新增額外的預設功能。最佳實務是重複使用現有 L2 建構的屬性。您可以直接在建構函數中修改屬性來覆寫屬性。

下列範例顯示如何與最佳實務保持一致並延伸稱為 s3.Bucket 的現有 L2 建構模組。此延伸建立預設屬性 (例如 versioned、publicReadAccess、blockPublicAccess)，以確保從此新的建構模組建立的所有物件 (在本範例中為 S3 儲存貯體) 永遠設定這些預設值。

```
import * as s3 from 'aws-cdk-lib/aws-s3';
import { Construct } from 'constructs';
export class MySecureBucket extends s3.Bucket {
  constructor(scope: Construct, id: string, props?: s3.BucketProps) {

    super(scope, id, {
      ...props,
      versioned: true,
      publicReadAccess: false,
      blockPublicAccess: s3.BlockPublicAccess.BLOCK_ALL
    });
  }
}
```

建立 L3 建構模組

當與現有建構合成有「有」關係時，合成是更好的選擇。組合意味著您在其他現有建構模組之上建置自己的建構模組。您可以建立自己的模式，將所有資源及其預設值封裝在單一可共用的更高層級的 L3 建構模組內。建立自己的 L3 建構模組 (模式) 的好處是您可以重複使用堆疊中的元件，而無需重新定義程式碼。您可以簡單地將建構模組匯入為類別。這些模式旨在協助取用者以簡潔的方式利用有限的知識，並基於通用模式提供多種資源。

下列程式碼範例會建立名為 的 AWS CDK 建構ExampleConstruct。您可以使用此建構模組作為範本定義雲端元件。

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';

export interface ExampleConstructProps {
  //insert properties you wish to expose
}

export class ExampleConstruct extends Construct {
  constructor(scope: Construct, id: string, props: ExampleConstructProps) {
```

```
    super(scope, id);
    //Insert the AWS components you wish to integrate
  }
}
```

下列範例示範如何在 AWS CDK 應用程式或堆疊中匯入新建立的建構。

```
import { ExampleConstruct } from './lib/construct-name';
```

下列範例顯示如何執行個體化從基本類別延伸的建構模組的執行個體。

```
import { ExampleConstruct } from './lib/construct-name';

new ExampleConstruct(this, 'newConstruct', {
  //insert props which you exposed in the interface `ExampleConstructProps`
});
```

如需詳細資訊，請參閱AWS CDK 研討會文件中的 AWS CDK 研討會。

逃生艙

您可以在 中使用逃生艙 AWS CDK 來提升抽象層級，以便存取較低層級的建構。逃生艙用於延伸 建構，以使用目前版本的 未公開 AWS 但可在 CloudFormation 中使用的功能。

我們建議您在下列案例中使用逃生艙：

- AWS 服務 此功能可透過 CloudFormation `Construct` 使用，但沒有其建構。
- AWS 服務 此功能可透過 CloudFormation `Construct` 使用，並且有服務的建構，但這些建構尚未公開此功能。由於 `Construct` 建構模組是「手動」開發的，因此其有時可能落後於 CloudFormation 資源建構模組。

下列範例程式碼顯示使用逃生艙的常見使用案例。在此範例中，較高層級建構模組中尚未實作的功能是新增 `httpPutResponseHopLimit` 以自動擴展 `LaunchConfiguration`。

```
const launchConfig = autoscaling.onDemandASG.node.findChild("LaunchConfig") as
 CfnLaunchConfiguration;
            launchConfig.metadataOptions = {
                   httpPutResponseHopLimit: autoscalingConfig.httpPutResponseHopLimit||
 2
            }
```

上述程式碼範例顯示下列工作流程：

1. 您可以使用 L2 建構模組來定義 AutoScalingGroup。L2 建構不支援更新 httpPutResponseHopLimit，因此您必須使用逃生艙。
2. 您可以使用 node.findChild()方法來尋找 L2 AutoScalingGroup 建構的特定LaunchConfig子系，並將其轉換為CfnLaunchConfiguration資源。
3. 現在您可以在 L1 CfnLaunchConfiguration 上設定 launchConfig.metadataOptions 屬性。

自訂資源

您可以使用自訂資源在範本中撰寫自訂佈建邏輯，讓 CloudFormation 在您建立、更新 (如果您變更自訂資源) 或刪除堆疊時執行。例如，如果您想要包含 中無法使用的資源，您可以使用自訂資源 AWS CDK。以此方式，您仍然可以在單一堆疊中管理所有相關資源。

建置自訂資源涉及編寫可回應資源的 CREATE、UPDATE 和 DELETE 生命週期事件的 Lambda 函數。如果您的自訂資源必須僅進行一次 API 呼叫，請考慮使用 AwsCustomResource 建構模組。這使得在 CloudFormation 部署期間執行任意 SDK 呼叫成為可能。否則，我們建議您編寫自己的 Lambda 函數來執行您必須完成的工作。

如需有關自訂資源的詳細資訊，請參閱 CloudFormation 文件中的自訂資源。如需如何使用自訂資源的範例，請參閱 GitHub 上的自訂資源儲存庫。

下列範例顯示如何建立自訂資源類別來啟動 Lambda 函數並向 CloudFormation 傳送成功或失敗訊號。

```
import cdk = require('aws-cdk-lib');
import customResources = require('aws-cdk-lib/custom-resources');
import lambda = require('aws-cdk-lib/aws-lambda');
import { Construct } from 'constructs';

import fs = require('fs');

export interface MyCustomResourceProps {
  /**
   * Message to echo
   */
  message: string;
}

export class MyCustomResource extends Construct {
```

```
  public readonly response: string;

  constructor(scope: Construct, id: string, props: MyCustomResourceProps) {
    super(scope, id);

    const fn = new lambda.SingletonFunction(this, 'Singleton', {
      uuid: 'f7d4f730-4ee1-11e8-9c2d-fa7ae01bbebc',
      code: new lambda.InlineCode(fs.readFileSync('custom-resource-handler.py',
 { encoding: 'utf-8' })),
      handler: 'index.main',
      timeout: cdk.Duration.seconds(300),
      runtime: lambda.Runtime.PYTHON_3_6,
    });

    const provider = new customResources.Provider(this, 'Provider', {
      onEventHandler: fn,
    });

    const resource = new cdk.CustomResource(this, 'Resource', {
      serviceToken: provider.serviceToken,
      properties: props,
    });

    this.response = resource.getAtt('Response').toString();
    }
}
```

下列範例顯示自訂資源的主要邏輯。

```
def main(event, context):
    import logging as log
    import cfnresponse
    log.getLogger().setLevel(log.INFO)

    # This needs to change if there are to be multiple resources in the same stack
    physical_id = 'TheOnlyCustomResource'

    try:
        log.info('Input event: %s', event)

        # Check if this is a Create and we're failing Creates
        if event['RequestType'] == 'Create' and
 event['ResourceProperties'].get('FailCreate', False):
```

```
            raise RuntimeError('Create failure requested')

        # Do the thing
        message = event['ResourceProperties']['Message']
        attributes = {
            'Response': 'You said "%s"' % message
        }

        cfnresponse.send(event, context, cfnresponse.SUCCESS, attributes, physical_id)
    except Exception as e:
        log.exception(e)
        # cfnresponse's error message is always "see CloudWatch"
        cfnresponse.send(event, context, cfnresponse.FAILED, {}, physical_id)
```

下列範例顯示 AWS CDK 堆疊如何呼叫自訂資源。

```
import cdk = require('aws-cdk-lib');
import { MyCustomResource } from './my-custom-resource';

/**
 * A stack that sets up MyCustomResource and shows how to get an attribute from it
 */
class MyStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const resource = new MyCustomResource(this, 'DemoResource', {
      message: 'CustomResource says hello',
    });

    // Publish the custom resource output
    new cdk.CfnOutput(this, 'ResponseMessage', {
      description: 'The message that came back from the Custom Resource',
      value: resource.response
    });
  }
}

const app = new cdk.App();
new MyStack(app, 'CustomResourceDemoStack');
app.synth();
```

遵循 TypeScript 最佳實務

TypeScript 是一種可延伸 JavaScript 功能的語言。這是一種強類型且以物件為導向的語言。您可以使用 TypeScript 指定程式碼中傳遞的資料類型，並且能夠在類型不符時報告錯誤。本節將概要介紹 TypeScript 最佳實務。

描述您的資料

您可以使用 TypeScript 來描述您的程式碼中的物件和函數的形狀。使用 any 類型相當於選擇不進行變數的類型檢查。我們建議您避免在程式碼中使用 any。請見此處範例。

```
type Result = "success" | "failure"
function verifyResult(result: Result) {
    if (result === "success") {
        console.log("Passed");
    } else {
        console.log("Failed")
    }
}
```

使用列舉

您可以使用列舉來定義一組具名常數並定義可在程式碼庫中重複使用的標準。我們建議您在全域層級匯出一次列舉，然後讓其他類別匯入並使用這些列舉。假設您想要建立一組可能的動作來擷取程式碼庫中的事件。TypeScript 同時提供了數字和字串型列舉。下列範例使用列舉。

```
enum EventType {
    Create,
    Delete,
    Update
}

class InfraEvent {
    constructor(event: EventType) {
        if (event === EventType.Create) {
            // Call for other function
            console.log(`Event Captured :${event}`);
        }
    }
}
```

```
let eventSource: EventType = EventType.Create;
const eventExample = new InfraEvent(eventSource)
```

使用介面

介面是類別的合約。如果您建立合約，則您的使用者必須遵守合約。在下列範例中，使用介面來標準化 `props` 並確保呼叫者在使用此類別時提供預期參數。

```
import { Stack, App } from "aws-cdk-lib";
import { Construct } from "constructs";

interface BucketProps {
    name: string;
    region: string;
    encryption: boolean;

}

class S3Bucket extends Stack {
    constructor(scope: Construct, props: BucketProps) {
        super(scope);
        console.log(props.name);

    }
}
const app = App();
const myS3Bucket = new S3Bucket(app, {
    name: "amzn-s3-demo-bucket",
    region: "us-east-1",
    encryption: false
})
```

某些屬性只能在首次建立物件時修改。您可以透過在屬性名稱之前放置 `readonly` 來指定這一點，如下列範例所示。

```
interface Position {
    readonly latitude: number;
    readonly longitute: number;
}
```

延伸介面

延伸介面可減少重複，因為您無需在介面之間複製屬性。此外，程式碼的讀取器可以輕鬆理解應用程式中的關係。

```
interface BaseInterface{
   name: string;
 }
 interface EncryptedVolume extends BaseInterface{
     keyName: string;
 }
 interface UnencryptedVolume extends BaseInterface {
     tags: string[];
 }
```

避免空介面

我們建議您避免空介面，因為其會產生潛在風險。在下列範例中，有一個名為 的空界面BucketProps。myS3Bucket1 和 myS3Bucket2 物件都是有效的，但其會遵循不同的標準，因為介面不會強制執行任何合約。下列程式碼將編譯和列印屬性，但這會在您的應用程式中導致不一致。

```
interface BucketProps {}

class S3Bucket implements BucketProps {
    constructor(props: BucketProps){
        console.log(props);
    }
}

const myS3Bucket1 = new S3Bucket({
    name: "amzn-s3-demo-bucket",
    region: "us-east-1",
    encryption: false,
});

const myS3Bucket2 = new S3Bucket({
    name: "amzn-s3-demo-bucket",
});
```

使用工廠

在「抽象工廠」模式中，介面負責建立相關物件的工廠，而無需明確指定其類別。例如，您可以建立 Lambda 工廠來建立 Lambda 函數。您不會在建構中建立新的 Lambda 函數，而是將建立程序委派給工廠。如需有關此設計模式的詳細資訊，請參閱 Refactoring.Guru 文件中的 TypeScript 中的抽象工廠。

對屬性使用解構

ECMAScript 6 (ES6) 中引入的解構是一項 JavaScript 功能，可讓您從陣列或物件中擷取多個資料片段並將其指派給自己的變數。

```
const object = {
    objname: "obj",
    scope: "this",
};

const oName = object.objname;
const oScop = object.scope;

const { objname, scope } = object;
```

定義標準命名慣例

強制執行命名慣例可以保持程式碼庫的一致性，並減少在考慮如何命名變數時的額外負荷。我們建議下列作法：

- 對變數和函數名稱使用 camelCase。
- 將 UPPER_CASE 用於全域常數，以清楚指出不可變的編譯時間值。
- 對類別名稱和介面名稱使用 PascalCase。
- 對介面成員使用 camelCase。
- 對類型名稱和列舉名稱使用 PascalCase。
- 使用 camelCase 命名檔案 (例如，`ebsVolumes.tsx` 或 `storage.ts`)

以下顯示這些建議命名慣例的範例：

```
// Variables and functions
const userName = 'john';
```

```
function getUserData() { }

// Global constants
const MAX_RETRY_ATTEMPTS = 3;
const API_BASE_URL = 'https://api.example.com';

// Classes and interfaces
class DatabaseConnection { }
interface UserProfile { }

// Types and enums
type ResponseStatus = 'success' | 'error';
enum HttpStatusCode { }
```

請勿使用 var 關鍵字

let 陳述式用於在 TypeScript 中宣告區域變數。它類似於 var關鍵字，但與 var關鍵字相比，它在範圍方面有一些限制。在具有 let 的區塊中宣告的變數只能在該區塊內使用。var 關鍵字不能是區塊範圍，這表示它可以在特定區塊之外存取（由 表示{})，但不能在定義它的函數之外存取。您可以重新宣告和更新var變數。最佳實務是避免使用 var關鍵字。

考慮使用 ESLint 和 Prettier

ESLint 可靜態分析您的程式碼以快速發現問題。您可以使用 ESLint 建立一系列聲明 (稱為 lint 規則) 來定義程式碼的外觀或行為方式。ESLint 還提供自動修復程式建議來協助您改進程式碼。最後，您可以使用 ESLint 從共用外掛程式載入 lint 規則。

Prettier 是一個知名的程式碼格式器，支援多種不同的程式設計語言。您可以使用 Prettier 設定程式碼樣式，以便避免手動格式化程式碼。安裝後，您可以更新您的 package.json 檔案並執行 npm run format 和 npm run lint 命令。

下列範例說明如何為您的 AWS CDK 專案啟用 ESLint 和 Prettier 格式工具。

```
"scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk",
    "lint": "eslint --ext .js,.ts .",
    "format": "prettier --ignore-path .gitignore --write '**/*.+(js|ts|json)'"
}
```

使用 access 修飾詞

TypeScript 中的 private 修飾詞僅將可見性限制為相同類別。將 private 修飾詞新增至屬性或方法時，您可以在相同類別內存取該屬性或方法。

public 修飾詞允許從所有位置存取類別屬性和方法。如果您未指定屬性和方法的任何存取修飾詞，它們預設會採用公有修飾詞。

protected 修飾詞允許在相同類別和子類別內存取類別的屬性和方法。當您預期在 AWS CDK 應用程式中建立子類別時，請使用受保護的修飾詞。

使用公用程式類型

TypeScript 中的公用程式類型是預先定義的類型函數，可在現有類型上執行轉換和操作。這可協助您根據現有類型建立新的類型。例如，您可以變更或擷取屬性、將屬性設為選用或必要，或建立不可變的類型版本。透過使用公用程式類型，您可以定義更精確的類型，並在編譯時擷取潛在的錯誤。

部分 < 類型>

`Partial` 會將輸入類型的所有成員標記為Type選用。此公用程式會傳回代表指定類型之所有子集的類型。以下是 `Partial` 的範例。

```
interface Dog {
  name: string;
  age: number;
  breed: string;
  weight: number;
}

let partialDog: Partial<Dog> = {};
```

必要 < 類型>

`Required` 與 相反`Partial`。它會讓輸入類型的所有成員Type變成非選用 （ 也就是必要 ）。以下是 `Required` 的範例。

```
interface Dog {
  name: string;
  age: number;
  breed: string;
  weight?: number;
```

```
}

let dog: Required<Dog> = {
  name: "scruffy",
  age: 5,
  breed: "labrador",
  weight: 55 // "Required" forces weight to be defined
};
```

掃描安全漏洞和格式錯誤

基礎設施即程式碼 (IaC) 和自動化對於企業來說至關重要。由於 IaC 如此強大，您在管理安全風險方面負有重大責任。常見 IaC 安全風險可能包括下列內容：

- 超額許可 AWS Identity and Access Management (IAM) 權限
- 開放式安全群組
- 未加密的資源
- 存取日誌未開啟

安全方法和工具

我們建議您實作下列安全方法：

- 開發中的漏洞偵測 – 由於開發和分發軟體修補程式的複雜性，修復生產中的漏洞既昂貴又耗時。此外，生產中的漏洞存在被利用的風險。我們建議您對 IaC 資源使用程式碼掃描，以便在發行到生產之前偵測到並修復漏洞。
- 合規和自動修復 – AWS Config 提供 AWS 受管規則。這些規則可協助您強制執行合規，並可讓您使用AWS Systems Manager 自動化來嘗試自動修復。您也可以使用 AWS Config 規則來建立和關聯自訂自動化文件。

常見開發工具

本節介紹的工具可協助您使用自己的自訂規則來延伸內建功能。我們建議您將自訂規則與組織的標準保持一致。以下是一些要考慮的常見開發工具：

- 使用 cdk-nag 來驗證指定範圍內的建構是否符合一組定義的規則。您也可以使用 cdk-nag 進行規則隱藏和合規報告。cdk-nag 工具透過擴展 中的層面來驗證建構 AWS CDK。如需詳細資訊，請參閱

《AWS DevOps 部落格》中的管理應用程式安全和與 AWS Cloud Development Kit (AWS CDK) 和 cdk-nag 的合規。

- 使用開放原始碼工具 Checkov 對您的 IaC 環境執行靜態分析。Checkov 透過掃描 Kubernetes、Terraform 或 CloudFormation 中的基礎設施程式碼來協助識別雲端錯誤組態。您可以使用 Checkov 取得不同格式的輸出，包括 JSON、JUnit XML 或 CLI。Checkov 可以透過建置顯示動態程式碼相依性的圖表來有效地處理變數。如需詳細資訊，請參閱 Bridgecrew 中的 GitHub Checkov 儲存庫。
- 使用 TFLint 檢查錯誤和已棄用的語法，並協助您強制執行最佳實務。請注意，TFLint 可能無法驗證供應商特定的問題。如需有關 TFlint 的詳細資訊，請參閱 Terraform Linters 中的 GitHub TFLint 儲存庫。
- 使用 Amazon Q Developer 執行安全掃描。在整合開發環境 (IDE) 中使用時，Amazon Q Developer 會提供 AI 驅動的軟體開發協助。它可以討論程式碼、提供內嵌程式碼完成、產生新程式碼、掃描程式碼是否有安全漏洞，以及進行程式碼升級和改善。

開發和完善文件

文件對於專案的成功至關重要。文件不僅解釋了程式碼的工作原理，還可協助開發人員更好地了解應用程式的特性和功能。開發和完善高品質文件可以加強軟體開發程序，維護高品質軟體，並有助於開發人員之間的知識轉移。

文件分為兩個類別：程式碼內的文件和有關程式碼的支援文件。程式碼內的文件採用註解的形式。有關程式碼的支援文件可以是 README 檔案和外部文件。開發人員將文件視為額外負荷並不常見，因為程式碼本身很容易理解。對於小型專案來說可能是這樣，但對於涉及多個團隊的大型專案來說，文件至關重要。

撰寫程式碼的作者最佳實務是撰寫文件，因為他們充分了解其功能。開發人員可能會因維護單獨的支援文件而產生額外負荷。為了克服此挑戰，開發人員可以在程式碼中新增註解，並且可以自動擷取這些註解，以便每個版本的程式碼和文件都將保持同步。

有各種不同的工具可協助開發人員從程式碼中擷取註解並為其產生文件。本指南著重於 TypeDoc AWS CDK 作為建構的偏好工具。

AWS CDK 建構需要程式碼文件的原因

AWS CDK 常見建構由組織中的多個團隊建立，並在不同的團隊之間共用以供取用。良好的文件可協助建構模組庫的取用者輕鬆地整合建構模組並以最輕鬆的方式建置基礎設施。保持所有文件同步是一項艱巨的任務。我們建議您在程式碼內維護文件，文件將使用 TypeDoc 庫擷取。

搭配 AWS 建構程式庫使用 TypeDoc

TypeDoc 是 TypeScript 的文件產生器。您可以使用 TypeDoc 讀取 TypeScript 來源檔案，剖析這些文件中的註解，然後產生包含程式碼文件的靜態網站。

下列程式碼說明如何將 TypeDoc 與 AWS 建構程式庫整合，然後在 的 `package.json` 檔案中新增下列套件`devDependencies`。

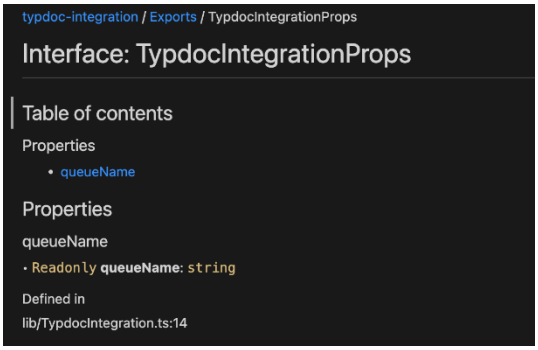
```
{

  "devDependencies": {

    "typedoc-plugin-markdown": "^3.11.7",
    "typescript": "~3.9.7"
  },

}
```

若要將 `typedoc.json` 新增在 CDK 程式庫資料夾中，請使用下列程式碼。

```
{
    "$schema": "https://typedoc.org/schema.json",
    "entryPoints": ["./lib"],
}
```

若要產生 README 檔案，請在建構程式庫專案的根目錄中執行 AWS CDK `npx typedoc`命令。

下列範例文件由 TypeDoc 產生。



如需有關 TypeDoc 整合選項的詳細資訊，請參閱 TypeDoc 文件中的文件註解。

採用測試驅動的開發方法

我們建議您使用 遵循測試驅動型開發 (TDD) 方法 AWS CDK。TDD 是一種軟體開發方法，您可以在其中開發測試案例來指定和驗證程式碼。簡言之，首先為每個功能建立測試案例，如果測試失敗，則編寫新程式碼以通過測試並使程式碼簡單且無錯誤。

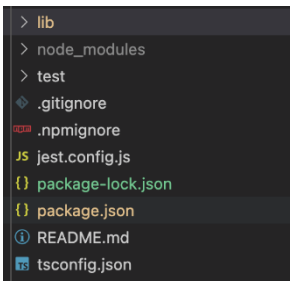
您可以先使用 TDD 編寫測試案例。這可協助您在強制執行資源安全政策和遵循專案的唯一命名慣例方面驗證具有不同設計限制的基礎設施。測試 AWS CDK 應用程式的標準方法是使用 AWS CDK assertions 模組和熱門測試架構，例如 Jest for TypeScript 和 JavaScript 或 pytest for Python。

您可以為 AWS CDK 應用程式撰寫兩種測試類別：

- 使用精細聲明來測試產生的 CloudFormation 範本的特定方面，例如「此資源具有包含此值的此屬性」。這些測試可以偵測迴歸，並且在您使用 TDD 開發新功能時也很有用 (首先編寫測試，然後透過編寫正確的實作使其通過)。精細聲明是您編寫最多的測試。
- 使用快照測試，以根據先前儲存的基準範本測試合成的 CloudFormation 範本。快照測試使自由重構成為可能，因為您可以確保重構的程式碼與原始程式碼的工作方式完全相同。如果變更是有意為之，您可以接受未來測試的新基準。不過， AWS CDK 升級也可能導致合成範本變更，因此您無法僅依賴快照來確保您的實作正確。

單位測試

本指南專門重點介紹 TypeScript 的單元測試整合。若要啟用測試，請確定您的 `package.json` 檔案具有下列程式庫： `jest`、 `@types/jest`和 `ts-jest` `devDependencies`。若要新增這些套件，請執行 `cdk init lib --language=typescript` 命令。在執行上述命令後，您會看到下列結構。



下列程式碼是使用 Jest 程式庫啟用`package.json`的檔案範例。

```
{
  ...
  "scripts": {
    "build": "npm run lint && tsc",
```

```
    "watch": "tsc -w",
    "test": "jest",
  },
  "devDependencies": {
    ...
    "@types/jest": "27.5.2",
    "jest": "27.5.1",
    "ts-jest": "27.1.5",
    ...
  }
}
```

在 Test 資料夾下，您可以編寫測試案例。下列範例顯示 AWS CodePipeline 建構的測試案例。

```
import { Stack } from 'aws-cdk-lib';
import { Template } from 'aws-cdk-lib/assertions';
import * as CodePipeline from 'aws-cdk-lib/aws-codepipeline';
import * as CodePipelineActions from 'aws-cdk-lib/aws-codepipeline-actions';
import { MyPipelineStack } from '../lib/my-pipeline-stack';
test('Pipeline Created with GitHub Source', () => {
  // ARRANGE
  const stack = new Stack();
  // ACT
  new MyPipelineStack(stack, 'MyTestStack');
  // ASSERT
  const template = Template.fromStack(stack);
  // Verify that the pipeline resource is created
  template.resourceCountIs('AWS::CodePipeline::Pipeline', 1);
  // Verify that the pipeline has the expected stages with GitHub source
  template.hasResourceProperties('AWS::CodePipeline::Pipeline', {
    Stages: [
      {
        Name: 'Source',
        Actions: [
          {
            Name: 'SourceAction',
            ActionTypeId: {
              Category: 'Source',
              Owner: 'ThirdParty',
              Provider: 'GitHub',
              Version: '1'
            },
            Configuration: {
```

```
            Owner: {
              'Fn::Join': [
                '',
                [
                  '{{resolve:secretsmanager:',
                  {
                    Ref: 'GitHubTokenSecret'
                  },
                  ':SecretString:owner}}'
                ]
              ]
            },
            Repo: {
              'Fn::Join': [
                '',
                [
                  '{{resolve:secretsmanager:',
                  {
                    Ref: 'GitHubTokenSecret'
                  },
                  ':SecretString:repo}}'
                ]
              ]
            },
            Branch: 'main',
            OAuthToken: {
              'Fn::Join': [
                '',
                [
                  '{{resolve:secretsmanager:',
                  {
                    Ref: 'GitHubTokenSecret'
                  },
                  ':SecretString:token}}'
                ]
              ]
            }
          },
          OutputArtifacts: [
            {
              Name: 'SourceOutput'
            }
          ],
          RunOrder: 1
```

```
          }
        ]
      },
      {
        Name: 'Build',
        Actions: [
          {
            Name: 'BuildAction',
            ActionTypeId: {
              Category: 'Build',
              Owner: 'AWS',
              Provider: 'CodeBuild',
              Version: '1'
            },
            InputArtifacts: [
              {
                Name: 'SourceOutput'
              }
            ],
            OutputArtifacts: [
              {
                Name: 'BuildOutput'
              }
            ],
            RunOrder: 1
          }
        ]
      }
      // Add more stage checks as needed
    ]
  });
  // Verify that a GitHub token secret is created
  template.resourceCountIs('AWS::SecretsManager::Secret', 1);
});
);
```

若要執行測試，請在專案中執行 `npm run test` 命令。此測試會傳回下列結果。

```
PASS  test/codepipeline-module.test.ts (5.972 s)
  # Code Pipeline Created (97 ms)
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
```

```
Time:        6.142 s, estimated 9 s
```

如需測試案例的詳細資訊，請參閱《 AWS Cloud Development Kit (AWS CDK) 開發人員指南》中的測試建構。

整合測試

您也可以使用 AWS CDK `integ-tests`模組來包含建構的整合測試。整合測試應定義為 AWS CDK 應用程式。整合測試和 AWS CDK 應用程式之間應該有one-to-one的關係。如需詳細資訊，請參閱《 AWS CDK API 參考》中的 integ-tests-alpha 模組。

對建構模組使用發行版本和版本控制

的版本控制 AWS CDK

AWS CDK 常見的建構可由多個團隊建立，並跨組織共用以供取用。一般而言，開發人員會在其常見 AWS CDK 建構中發佈新功能或錯誤修正。應用程式或任何其他現有建構會使用 AWS CDK 這些 AWS CDK 建構做為相依性的一部分。因此，開發人員獨立更新和發行具有正確語意版本的建構模組至關重要。下游 AWS CDK 應用程式或其他 AWS CDK 建構模組可以更新其相依性，以使用新發行的 AWS CDK 建構模組版本。

語意版本控制 (Semver) 是一組規則或方法，用於為電腦軟體提供唯一的軟體編號。版本定義如下：

- 主要版本包含不相容的 API 變更或重大變更。
- 次要版本包含以回溯相容方式新增的功能。
- 修補版本包含回溯相容的錯誤修正。

如需有關語意版本控制的詳細資訊，請參閱語意版本控制文件中的語意版本控制規格 (SemVer)。

AWS CDK 建構的儲存庫和封裝

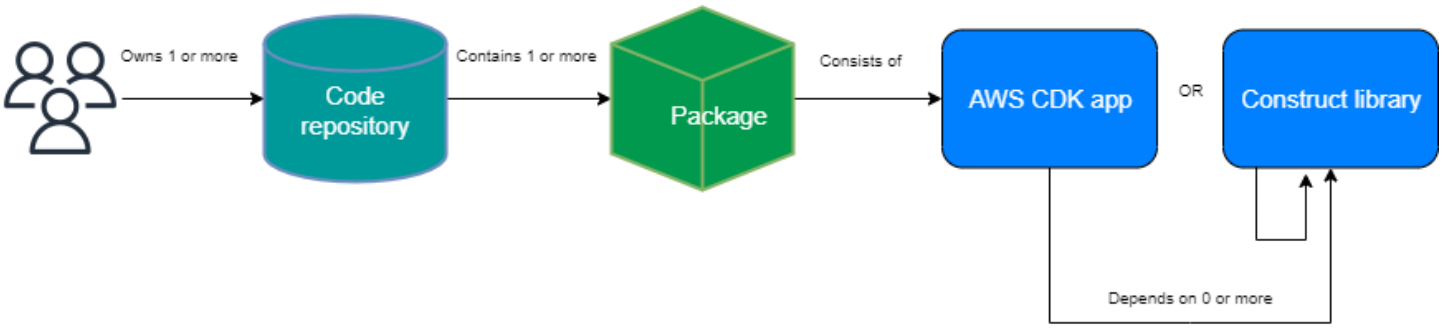
由於 AWS CDK 建構是由不同的團隊開發並由多個 AWS CDK 應用程式使用，因此您可以為每個 AWS CDK 建構使用個別的儲存庫。這也可以協助您強制執行存取控制。每個儲存庫都可以包含與相同 AWS CDK 建構相關的所有原始程式碼及其所有相依性。透過將單一應用程式 AWS CDK （即建構） 保留在單一儲存庫中，您可以減少部署期間變更的影響範圍。

AWS CDK 不僅會產生用於部署基礎設施的 CloudFormation 範本，還會綁定執行期資產，例如 Lambda 函數和 Docker 映像，並將其與您的基礎設施一起部署。不僅可以結合定義基礎設施的程式

碼，以及將執行時間邏輯實作為單一建構的程式碼，這是最佳實務。這兩種程式碼不需要位於單獨的儲存庫中，甚至不需要位於單獨的套件中。

若要跨儲存庫界限取用套件，您必須具有私有套件儲存庫 - 類似於 npm、PyPi 或 Maven Central，但位於組織內部。您還必須具有發行流程來建置、測試套件並將其發佈至私有套件儲存庫。您可以使用本機虛擬機器 (VM) 或 Amazon S3 建立私有儲存庫，例如 PyPi 伺服器。當您設計或建立私有套件登錄檔時，考慮由於高可用性和可擴展性而導致服務中斷的風險至關重要。在雲端託管以存放套件的無伺服器受管服務可以大幅降低維護開銷。例如，您可以使用 AWS CodeArtifact 來託管大多數熱門程式設計語言的套件。您也可以使用 CodeArtifact 設定外部儲存庫連線並在 CodeArtifact 內進行複寫。

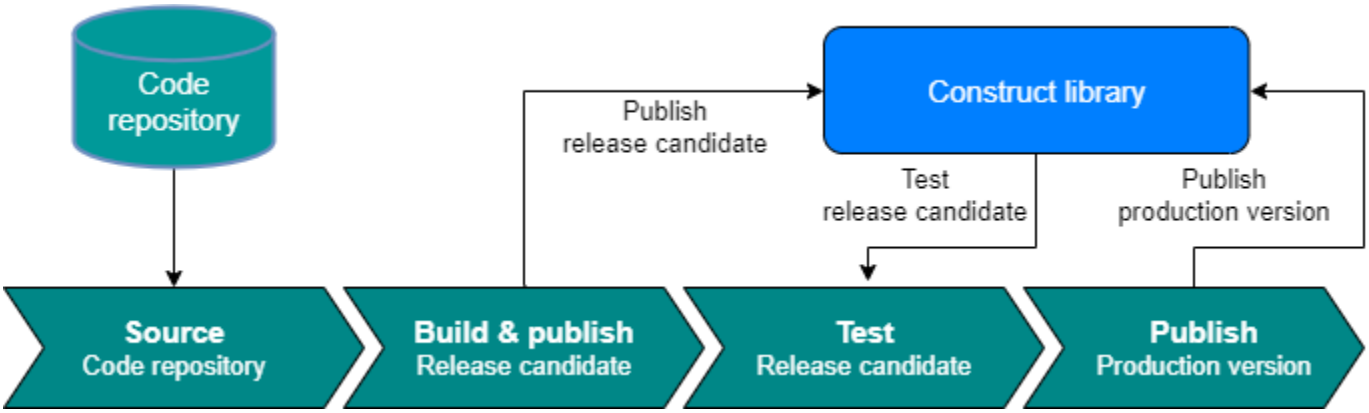
對套件儲存庫中套件的相依性由您的語言的套件管理員 (例如，用於 TypeScript 或 JavaScript 應用程式的 npm) 管理。您的套件管理員透過記錄應用程式依賴的每個套件的特定版本來確保建置是可重複的，然後讓您以受控的方式升級這些相依性，如下圖所示。



建構 的發行 AWS CDK

我們建議您建立自己的自動化管道，以建置和發行新的 AWS CDK 建構版本。如果您制定了適當的提取請求核准程序，則一旦您提交並將原始程式碼推送到儲存庫的主要分支中，管道就可以建置並建立發行候選版本。在發佈生產就緒型版本之前，可以將該版本推送至 CodeArtifact 並進行測試。或者，您可以在本機測試新的 AWS CDK 建構版本，然後再將程式碼與主要分支合併。這會導致管道發行生產就緒型版本。考慮到共用建構模組和套件必須獨立於使用應用程式進行測試，就像其向公眾發行一樣。

下圖顯示範例 AWS CDK 版本發行管道。



您可以使用下列範例命令來建置、測試和發佈 npm 套件。首先，透過執行下列命令登入成品儲存庫。

```
aws codeartifact login --tool npm --domain <Domain Name> --domain-owner $(aws sts get-
caller-identity --output text --query 'Account')  \
--repository <Repository Name> --region <AWS Region Name>
```

然後，完成下列步驟：

1. 根據 `package.json` 文件安裝所需的套件：`npm install`
2. 建立候選發行版本：`npm version prerelease --preid rc`
3. 建置 npm 套件：`npm run build`
4. 測試 npm 套件：`npm run test`
5. 發佈 npm 套件：`npm publish`

強制執行程式庫版本管理

當您維護 AWS CDK 程式碼庫時，生命週期管理是一項重大挑戰。例如，假設您啟動 1.97 版的 AWS CDK 專案，然後 1.169 版在稍後推出。版本 1.169 提供了新功能和錯誤修正，但您已使用舊版部署基礎設施。現在，由於新版本中可能會引入重大變更，因此隨著差距的擴大，更新建構模組變得具有挑戰性。如果您的環境中具有許多資源，這可能是一個挑戰。本節中介紹的模式可協助您使用自動化管理 AWS CDK 程式庫版本。以下是此模式的工作流程：

1. 當您啟動新的 CodeArtifact Service Catalog 產品時，程式 AWS CDK 庫版本及其相依性會存放在 `package.json` 檔案中。
2. 您可部署一個通用管道來追蹤所有儲存庫，以便在沒有重大變更時對其套用自動升級。
3. AWS CodeBuild 階段會檢查相依性樹狀結構，並尋找重大變更。

4. 該管道建立一個功能分支，然後使用新版本執行 `cdk synth` 以確認沒有錯誤。
5. 新版本部署在測試環境中，最後執行整合測試以確保部署運作狀態正常。
6. 您可以使用兩個 Amazon Simple Queue Service (Amazon SQS) 佇列來追蹤堆疊。使用者可以在例外狀況佇列中手動檢閱堆疊並解決重大變更。通過整合測試的項目允許合併和發行。

常見問答集

TypeScript 可以解決什麼問題？

通常，您可以透過編寫自動化測試，手動驗證程式碼是否如預期般運作，最後讓另一個人驗證您的程式碼來消除程式碼中的錯誤。驗證專案的每個部分之間的連線非常耗時。為了加快驗證程序，您可以使用 TypeScript 等類型檢查語言來自動化程式碼驗證並在開發期間提供即時回饋。

為什麼我應該使用 TypeScript？

TypeScript 是一種開放原始碼語言，可簡化 JavaScript 程式碼，讓您更輕鬆地讀取和偵錯。TypeScript 也為 JavaScript IDE 和實務提供高效率的開發工具，例如靜態檢查。此外，TypeScript 還提供了 ECMAScript 6 (ES6) 的好處，可提高您的工作效率。最後，TypeScript 可以透過類型檢查程式碼來協助您避免開發人員在編寫 JavaScript 時經常遇到的令人痛苦的錯誤。

我應該使用 AWS CDK 或 CloudFormation 嗎？

如果您的組織具有利用 的開發專業知識 AWS CloudFormation，建議您使用 AWS Cloud Development Kit (AWS CDK) ，而不是 AWS CDK。這是因為 AWS CDK 比 CloudFormation 更靈活，因為您可以使用程式設計語言和 OOP 概念。請記住，您可以使用 CloudFormation 以有序且可預測的方式建立 AWS 資源。在 CloudFormation 中，資源使用 JSON 或 YAML 格式寫入文字檔案。

如果 AWS CDK 不支援新啟動的 該怎麼辦 AWS 服務？

您可以使用原始覆寫或 CloudFormation 自訂資源。

支援哪些不同的程式設計語言 AWS CDK？

AWS CDK 通常在 JavaScript、TypeScript、Python、Java、C# 和 Go （在開發人員預覽版中） 中提供。

AWS CDK 費用是多少？

無需支付額外費用 AWS CDK。您使用 時所建立 AWS 的資源 （例如 Amazon EC2 執行個體或 Elastic Load Balancing 負載平衡器） 需付費 AWS CDK ，方式與手動建立的資源相同。當您使用資源時，僅需為使用的資源付費。沒有最低費用，也不需要前期承諾。

後續步驟

我們建議您在 TypeScript AWS Cloud Development Kit (AWS CDK) 中開始使用 建置 。如需詳細資訊，請參閱AWS CDK 沉浸式日研討會。

資源

參考

- AWS 解決方案建構 (AWS 解決方案）
- AWS Cloud Development Kit (AWS CDK) (GitHub)
- AWS Construct Library API 參考 （參考文件）AWS CDK
- AWS CDK 參考文件 (AWS CDK 參考文件）
- AWS CDK 沉浸式日研討會 (AWS Workshop Studio)

工具

- cdk-nag (GitHub)
- TypeScript ESLint (TypeScript ESLint 文件)

指南和模式

- AWS 解決方案建構模式 (AWS 文件）

文件歷史紀錄

下表描述了本指南的重大變更。如果您想收到有關未來更新的通知，可以訂閱 RSS 摘要。

變更	描述	日期
更新的最佳實務	我們移除了在通用開發工具區段中使用 cfn-nag 的建議。在定義標準命名慣例區段中，我們新增了全域常數的標準命名慣例，並新增了命名範例。	2025 年 10 月 23 日
更新程式碼	我們更新了遵循 TypeScript 最佳實務區段中的程式碼範例。	2024 年 2 月 16 日
新增區段	我們新增了使用公用程式類型和整合測試區段。	2024 年 1 月 10 日
次要更新	已更新用於建立 L3 建構模組的程式碼範例。	2023 年 6 月 16 日
初次出版	—	2022 年 12 月 8 日

AWS 規範性指引詞彙表

以下是 AWS Prescriptive Guidance 提供的策略、指南和模式中常用的術語。若要建議項目，請使用詞彙表末尾的提供意見回饋連結。

數字

7 R

將應用程式移至雲端的七種常見遷移策略。這些策略以 Gartner 在 2011 年確定的 5 R 為基礎，包括以下內容：

- 重構/重新架構 – 充分利用雲端原生功能來移動應用程式並修改其架構，以提高敏捷性、效能和可擴展性。這通常涉及移植作業系統和資料庫。範例：將您的現場部署 Oracle 資料庫 遷移至 Amazon Aurora PostgreSQL 相容版本。
- 平台轉換 (隨即重塑) – 將應用程式移至雲端，並引入一定程度的優化以利用雲端功能。範例：將內部部署 Oracle 資料庫 遷移至 中的 Amazon Relational Database Service (Amazon RDS) for Oracle AWS 雲端。
- 重新購買 (捨棄再購買) – 切換至不同的產品，通常從傳統授權移至 SaaS 模型。範例：將您的客戶關係管理 (CRM) 系統 遷移至 Salesforce.com。
- 主機轉換 (隨即轉移) – 將應用程式移至雲端，而不進行任何變更以利用雲端功能。範例：將您的現場部署 Oracle 資料庫 遷移至 中 EC2 執行個體上的 Oracle AWS 雲端。
- 重新放置 (虛擬機器監視器等級隨即轉移) – 將基礎設施移至雲端，無需購買新硬體、重寫應用程式或修改現有操作。您可以將伺服器從內部部署平台遷移到相同平台的雲端服務。範例：將 Microsoft Hyper-V應用程式 遷移至 AWS。
- 保留 (重新檢視) – 將應用程式保留在來源環境中。其中可能包括需要重要重構的應用程式，且您希望將該工作延遲到以後，以及您想要保留的舊版應用程式，因為沒有業務理由來進行遷移。
- 淘汰 – 解除委任或移除來源環境中不再需要的應用程式。

A

ABAC

請參閱屬性型存取控制。

抽象服務

請參閱 受管服務。

ACID

請參閱原子性、一致性、隔離性、耐久性。

主動-主動式遷移

一種資料庫遷移方法，其中來源和目標資料庫保持同步 (透過使用雙向複寫工具或雙重寫入操作)，且兩個資料庫都在遷移期間處理來自連接應用程式的交易。此方法支援小型、受控制批次的遷移，而不需要一次性切換。它更靈活，但比主動-被動遷移需要更多的工作。

主動-被動式遷移

一種資料庫遷移方法，其中來源和目標資料庫保持同步，但只有來源資料庫會在資料複寫至目標資料庫時處理來自連線應用程式的交易。目標資料庫在遷移期間不接受任何交易。

彙總函數

在一組資料列上運作的 SQL 函數，會計算群組的單一傳回值。彙總函數的範例包括 SUM和 MAX。

AI

請參閱人工智慧。

AIOps

請參閱人工智慧操作。

匿名化

永久刪除資料集中個人資訊的程序。匿名化有助於保護個人隱私權。匿名資料不再被視為個人資料。

反模式

經常用於經常性問題的解決方案，其中解決方案具有反生產力、無效或比替代解決方案更有效。

應用程式控制

一種安全方法，僅允許使用核准的應用程式，以協助保護系統免受惡意軟體攻擊。

應用程式組合

有關組織使用的每個應用程式的詳細資訊的集合，包括建置和維護應用程式的成本及其商業價值。此資訊是產品組合探索和分析程序的關鍵，有助於識別要遷移、現代化和優化的應用程式並排定其優先順序。

人工智慧 (AI)

電腦科學領域，致力於使用運算技術來執行通常與人類相關的認知功能，例如學習、解決問題和識別模式。如需詳細資訊，請參閱什麼是人工智慧？

人工智慧操作 (AIOps)

使用機器學習技術解決操作問題、減少操作事件和人工干預以及提高服務品質的程序。如需有關如何在 AWS 遷移策略中使用 AIOps 的詳細資訊，請參閱操作整合指南。

非對稱加密

一種加密演算法，它使用一對金鑰：一個用於加密的公有金鑰和一個用於解密的私有金鑰。您可以共用公有金鑰，因為它不用於解密，但對私有金鑰存取應受到高度限制。

原子性、一致性、隔離性、耐久性 (ACID)

一組軟體屬性，即使在出現錯誤、電源故障或其他問題的情況下，也能確保資料庫的資料有效性和操作可靠性。

屬性型存取控制 (ABAC)

根據使用者屬性 (例如部門、工作職責和團隊名稱) 建立精細許可的實務。如需詳細資訊，請參閱《 AWS Identity and Access Management (IAM) 文件》中的 ABAC for AWS。

授權資料來源

存放主要版本資料的位置，被視為最可靠的資訊來源。您可以將授權資料來源中的資料複製到其他位置，以處理或修改資料，例如匿名、修訂或假名化資料。

可用區域

中的不同位置 AWS 區域 ，可隔離其他可用區域中的故障，並提供相同區域中其他可用區域的低成本、低延遲網路連線能力。

AWS 雲端採用架構 (AWS CAF)

的指導方針和最佳實務架構 AWS ，可協助組織制定高效且有效的計劃，以成功地移至雲端。 AWS CAF 將指導方針組織到六個重點領域：業務、人員、治理、平台、安全和營運。業務、人員和控管層面著重於業務技能和程序；平台、安全和操作層面著重於技術技能和程序。例如，人員層面針對處理人力資源 (HR)、人員配備功能和人員管理的利害關係人。因此， AWS CAF 為人員開發、訓練和通訊提供指引，協助組織做好成功採用雲端的準備。如需詳細資訊，請參閱 AWS CAF 網站和 AWS CAF 白皮書。

AWS 工作負載資格架構 (AWS WQF)

評估資料庫遷移工作負載、建議遷移策略並提供工作預估值的工具。 AWS WQF 隨附於 AWS Schema Conversion Tool (AWS SCT)。它會分析資料庫結構描述和程式碼物件、應用程式程式碼、相依性和效能特性，並提供評估報告。

B

錯誤的機器人

旨在中斷或傷害個人或組織的機器人。

BCP

請參閱業務持續性規劃。

行為圖

資源行為的統一互動式檢視，以及一段時間後的互動。您可以將行為圖與 Amazon Detective 搭配使用來檢查失敗的登入嘗試、可疑的 API 呼叫和類似動作。如需詳細資訊，請參閱偵測文件中的行為圖中的資料。

大端序系統

首先儲存最高有效位元組的系統。另請參閱 Endianness。

二進制分類

預測二進制結果的過程 (兩個可能的類別之一)。例如，ML 模型可能需要預測諸如「此電子郵件是否是垃圾郵件？」等問題 或「產品是書還是汽車？」

Bloom 篩選條件

一種機率性、記憶體高效的資料結構，用於測試元素是否為集的成員。

藍/綠部署

一種部署策略，您可以在其中建立兩個不同但相同的環境。您可以在一個環境 （藍色） 中執行目前的應用程式版本，並在另一個環境 （綠色） 中執行新的應用程式版本。此策略可協助您快速復原，並將影響降至最低。

機器人

透過網際網路執行自動化任務並模擬人類活動或互動的軟體應用程式。有些機器人有用或有益，例如在網際網路上編製資訊索引的 Web 爬蟲程式。某些其他機器人稱為惡意機器人，旨在中斷或傷害個人或組織。

殭屍網路

受到惡意軟體感染且受單一方控制之機器人的網路，稱為機器人繼承器或機器人運算子。殭屍網路是擴展機器人及其影響的最佳已知機制。

分支

程式碼儲存庫包含的區域。儲存庫中建立的第一個分支是主要分支。您可以從現有分支建立新分支，然後在新分支中開發功能或修正錯誤。您建立用來建立功能的分支通常稱為功能分支。當準備好發佈功能時，可以將功能分支合併回主要分支。如需詳細資訊，請參閱關於分支 (GitHub 文件)。

碎片存取

在特殊情況下，以及透過核准的程序，讓使用者快速取得他們通常無權存取 AWS 帳戶 之 的存取權。如需詳細資訊，請參閱 Well-Architected 指南中的 AWS 實作打破玻璃程序指標。

棕地策略

環境中的現有基礎設施。對系統架構採用棕地策略時，可以根據目前系統和基礎設施的限制來設計架構。如果正在擴展現有基礎設施，則可能會混合棕地和綠地策略。

緩衝快取

儲存最常存取資料的記憶體區域。

業務能力

業務如何創造價值 (例如，銷售、客戶服務或營銷)。業務能力可驅動微服務架構和開發決策。如需詳細資訊，請參閱在 AWS上執行容器化微服務白皮書的圍繞業務能力進行組織部分。

業務連續性規劃 (BCP)

一種解決破壞性事件 (如大規模遷移) 對營運的潛在影響並使業務能夠快速恢復營運的計畫。

C

CAF

請參閱AWS 雲端採用架構。

Canary 部署

版本對最終使用者的緩慢和增量版本。當您有信心時，您可以部署新版本並完全取代目前的版本。

CCoE

請參閱 Cloud Center of Excellence。

CDC

請參閱變更資料擷取。

變更資料擷取 (CDC)

追蹤對資料來源 (例如資料庫表格) 的變更並記錄有關變更的中繼資料的程序。您可以將 CDC 用於各種用途，例如稽核或複寫目標系統中的變更以保持同步。

混沌工程

故意引入故障或破壞性事件，以測試系統的彈性。您可以使用 AWS Fault Injection Service (AWS FIS) 執行實驗，為您的 AWS 工作負載帶來壓力，並評估其回應。

CI/CD

請參閱持續整合和持續交付。

分類

有助於產生預測的分類程序。用於分類問題的 ML 模型可預測離散值。離散值永遠彼此不同。例如，模型可能需要評估影像中是否有汽車。

用戶端加密

在目標 AWS 服務 接收資料之前，在本機加密資料。

雲端卓越中心 (CCoE)

一個多學科團隊，可推動整個組織的雲端採用工作，包括開發雲端最佳實務、調動資源、制定遷移時間表以及領導組織進行大規模轉型。如需詳細資訊，請參閱 AWS 雲端 企業策略部落格上的 CCoE 文章。

雲端運算

通常用於遠端資料儲存和 IoT 裝置管理的雲端技術。雲端運算通常連接到邊緣運算技術。

雲端操作模型

在 IT 組織中，用於建置、成熟和最佳化一或多個雲端環境的操作模型。如需詳細資訊，請參閱建置您的雲端操作模型。

採用雲端階段

組織在遷移至 時通常會經歷的四個階段 AWS 雲端：

- 專案 – 執行一些與雲端相關的專案以進行概念驗證和學習用途
- 基礎 – 進行基礎投資以擴展雲端採用 (例如，建立登陸區域、定義 CCoE、建立營運模型)

- 遷移 – 遷移個別應用程式
- 重塑 – 優化產品和服務，並在雲端中創新

這些階段由 Stephen Orban 在部落格文章 The Journey Toward Cloud-First 和企業策略部落格上的採用階段中定義。 AWS 雲端 如需有關它們如何與 AWS 遷移策略相關的詳細資訊，請參閱遷移整備指南。

CMDB

請參閱組態管理資料庫。

程式碼儲存庫

透過版本控制程序來儲存及更新原始程式碼和其他資產 (例如文件、範例和指令碼) 的位置。常見的雲端儲存庫包括 GitHub或 Bitbucket Cloud。程式碼的每個版本都稱為分支。在微服務結構中，每個儲存庫都專用於單個功能。單一 CI/CD 管道可以使用多個儲存庫。

冷快取

一種緩衝快取，它是空的、未填充的，或者包含過時或不相關的資料。這會影響效能，因為資料庫執行個體必須從主記憶體或磁碟讀取，這比從緩衝快取讀取更慢。

冷資料

很少存取且通常是歷史資料的資料。查詢這類資料時，通常可接受慢查詢。將此資料移至效能較低且成本較低的儲存層或類別，可以降低成本。

電腦視覺 (CV)

AI 欄位???，使用機器學習從數位影像和影片等視覺化格式分析和擷取資訊。例如，Amazon SageMaker AI 提供 CV 的影像處理演算法。

組態偏離

對於工作負載，組態會從預期狀態變更。這可能會導致工作負載不合規，而且通常是漸進和無意的。

組態管理資料庫 (CMDB)

儲存和管理有關資料庫及其 IT 環境的資訊的儲存庫，同時包括硬體和軟體元件及其組態。您通常在遷移的產品組合探索和分析階段使用 CMDB 中的資料。

一致性套件

您可以組合的 AWS Config 規則和修補動作集合，以自訂您的合規和安全檢查。您可以使用 YAML 範本，將一致性套件部署為 AWS 帳戶 和 區域中或整個組織的單一實體。如需詳細資訊，請參閱 AWS Config 文件中的一致性套件。

持續整合和持續交付 (CI/CD)

自動化軟體發行程序的來源、建置、測試、暫存和生產階段的程序。CI/CD 通常被描述為管道。CI/CD 可協助您將程序自動化、提升生產力、改善程式碼品質以及加快交付速度。如需詳細資訊，請參閱持續交付的優點。CD 也可表示持續部署。如需詳細資訊，請參閱持續交付與持續部署。

CV

請參閱電腦視覺。

D

靜態資料

網路中靜止的資料，例如儲存中的資料。

資料分類

根據重要性和敏感性來識別和分類網路資料的程序。它是所有網路安全風險管理策略的關鍵組成部分，因為它可以協助您確定適當的資料保護和保留控制。資料分類是 AWS Well-Architected Framework 中安全支柱的元件。如需詳細資訊，請參閱資料分類。

資料偏離

生產資料與用於訓練 ML 模型的資料之間有意義的變化，或輸入資料隨時間有意義的變更。資料偏離可以降低 ML 模型預測的整體品質、準確性和公平性。

傳輸中的資料

在您的網路中主動移動的資料，例如在網路資源之間移動。

資料網格

架構架構，提供分散式、分散式資料擁有權與集中式管理。

資料最小化

僅收集和處理嚴格必要資料的原則。在 中實作資料最小化 AWS 雲端 可以降低隱私權風險、成本和分析碳足跡。

資料周邊

AWS 環境中的一組預防性防護機制，可協助確保只有信任的身分才能從預期的網路存取信任的資源。如需詳細資訊，請參閱在 上建置資料周邊 AWS。

資料預先處理

將原始資料轉換成 ML 模型可輕鬆剖析的格式。預處理資料可能意味著移除某些欄或列，並解決遺失、不一致或重複的值。

資料來源

在整個生命週期中追蹤資料的原始伺服器和歷史記錄的程序，例如資料的產生、傳輸和儲存方式。

資料主體

正在收集和處理資料的個人。

資料倉儲

支援商業智慧的資料管理系統，例如 分析。資料倉儲通常包含大量歷史資料，通常用於查詢和分析。

資料庫定義語言 (DDL)

用於建立或修改資料庫中資料表和物件之結構的陳述式或命令。

資料庫處理語言 (DML)

用於修改 (插入、更新和刪除) 資料庫中資訊的陳述式或命令。

DDL

請參閱資料庫定義語言。

深度整體

結合多個深度學習模型進行預測。可以使用深度整體來獲得更準確的預測或估計預測中的不確定性。

深度學習

一個機器學習子領域，它使用多層人工神經網路來識別感興趣的輸入資料與目標變數之間的對應關係。

深度防禦

這是一種資訊安全方法，其中一系列的安全機制和控制項會在整個電腦網路中精心分層，以保護網路和其中資料的機密性、完整性和可用性。當您在 上採用此策略時 AWS，您可以在 AWS Organizations 結構的不同層新增多個控制項，以協助保護資源。例如，defense-in-depth方法可能會結合多重要素驗證、網路分割和加密。

委派的管理員

在 中 AWS Organizations，相容的服務可以註冊 AWS 成員帳戶，以管理組織的帳戶和管理該服務的許可。此帳戶稱為該服務的委派管理員。如需詳細資訊和相容服務清單，請參閱 AWS Organizations 文件中的可搭配 AWS Organizations運作的服務。

deployment

在目標環境中提供應用程式、新功能或程式碼修正的程序。部署涉及在程式碼庫中實作變更，然後在應用程式環境中建置和執行該程式碼庫。

開發環境

請參閱 環境。

偵測性控制

一種安全控制，用於在事件發生後偵測、記錄和提醒。這些控制是第二道防線，提醒您注意繞過現有預防性控制的安全事件。如需詳細資訊，請參閱在 AWS上實作安全控制中的偵測性控制。

開發值串流映射 (DVSM)

一種程序，用於識別對軟體開發生命週期中的速度和品質造成負面影響的限制並排定優先順序。DVSM 擴展了原本專為精簡製造實務設計的價值串流映射程序。它著重於透過軟體開發程序建立和移動價值所需的步驟和團隊。

數位分身

真實世界系統的虛擬呈現，例如建築物、工廠、工業設備或生產線。數位分身支援預測性維護、遠端監控和生產最佳化。

維度資料表

在星星結構描述中，較小的資料表包含有關事實資料表中量化資料的資料屬性。維度資料表屬性通常是文字欄位或離散數字，其行為類似於文字。這些屬性通常用於查詢限制、篩選和結果集標記。

災難

防止工作負載或系統在其主要部署位置中實現其業務目標的事件。這些事件可能是自然災難、技術故障或人為動作的結果，例如意外設定錯誤或惡意軟體攻擊。

災難復原 (DR)

您用來將災難造成的停機時間和資料遺失降至最低的策略和程序。如需詳細資訊，請參閱 AWS Well-Architected Framework 中的 上工作負載災難復原 AWS：雲端中的復原。

DML

請參閱資料庫處理語言。

領域驅動的設計

一種開發複雜軟體系統的方法，它會將其元件與每個元件所服務的不斷發展的領域或核心業務目標相關聯。Eric Evans 在其著作 Domain-Driven Design: Tackling Complexity in the Heart of Software (Boston: Addison-Wesley Professional, 2003) 中介紹了這一概念。如需有關如何將領域驅動的設計與 strangler fig 模式搭配使用的資訊，請參閱使用容器和 Amazon API Gateway 逐步現代化舊版 Microsoft ASP.NET (ASMX) Web 服務。

DR

請參閱災難復原。

偏離偵測

追蹤與基準組態的偏差。例如，您可以使用 AWS CloudFormation 來偵測系統資源中的偏離，也可以使用 AWS Control Tower 來偵測登陸區域中可能影響控管要求合規性的變更。 https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-stack-drift.html

DVSM

請參閱開發值串流映射。

E

EDA

請參閱探索性資料分析。

EDI

請參閱電子資料交換。

邊緣運算

提升 IoT 網路邊緣智慧型裝置運算能力的技術。與雲端運算相比，邊緣運算可以減少通訊延遲並改善回應時間。

電子資料交換 (EDI)

在組織之間自動交換商業文件。如需詳細資訊，請參閱什麼是電子資料交換。

加密

一種運算程序，可將人類可讀取的純文字資料轉換為加密文字。

加密金鑰

由加密演算法產生的隨機位元的加密字串。金鑰長度可能有所不同，每個金鑰的設計都是不可預測且唯一的。

端序

位元組在電腦記憶體中的儲存順序。大端序系統首先儲存最高有效位元組。小端序系統首先儲存最低有效位元組。

端點

請參閱 服務端點。

端點服務

您可以在虛擬私有雲端 (VPC) 中託管以與其他使用者共用的服務。您可以使用 建立端點服務，AWS PrivateLink 並將許可授予其他 AWS 帳戶 或 AWS Identity and Access Management (IAM) 委託人。這些帳戶或主體可以透過建立介面 VPC 端點私下連接至您的端點服務。如需詳細資訊，請參閱 Amazon Virtual Private Cloud (Amazon VPC) 文件中的建立端點服務。

企業資源規劃 (ERP)

一種系統，可自動化和管理企業的關鍵業務流程 （例如會計、MES 和專案管理）。

信封加密

使用另一個加密金鑰對某個加密金鑰進行加密的程序。如需詳細資訊，請參閱 AWS Key Management Service (AWS KMS) 文件中的信封加密。

環境

執行中應用程式的執行個體。以下是雲端運算中常見的環境類型：

- 開發環境 – 執行中應用程式的執行個體，只有負責維護應用程式的核心團隊才能使用。開發環境用來測試變更，然後再將開發環境提升到較高的環境。此類型的環境有時稱為測試環境。
- 較低的環境 – 應用程式的所有開發環境，例如用於初始建置和測試的開發環境。
- 生產環境 – 最終使用者可以存取的執行中應用程式的執行個體。在 CI/CD 管道中，生產環境是最後一個部署環境。
- 較高的環境 – 核心開發團隊以外的使用者可存取的所有環境。這可能包括生產環境、生產前環境以及用於使用者接受度測試的環境。

epic

在敏捷方法中，有助於組織工作並排定工作優先順序的功能類別。epic 提供要求和實作任務的高層級描述。例如， AWS CAF 安全概念包括身分和存取管理、偵測控制、基礎設施安全、資料保護和事件回應。如需有關 AWS 遷移策略中的 Epic 的詳細資訊，請參閱計畫實作指南。

ERP

請參閱企業資源規劃。

探索性資料分析 (EDA)

分析資料集以了解其主要特性的過程。您收集或彙總資料，然後執行初步調查以尋找模式、偵測異常並檢查假設。透過計算摘要統計並建立資料可視化來執行 EDA。

F

事實資料表

星狀結構描述中的中央資料表。它存放有關業務操作的量化資料。一般而言，事實資料表包含兩種類型的資料欄：包含度量的資料，以及包含維度資料表外部索引鍵的資料欄。

快速失敗

一種使用頻繁和增量測試來縮短開發生命週期的理念。這是敏捷方法的關鍵部分。

故障隔離界限

在 中 AWS 雲端，像是可用區域 AWS 區域、控制平面或資料平面等邊界會限制故障的影響，並有助於改善工作負載的彈性。如需詳細資訊，請參閱AWS 故障隔離界限。

功能分支

請參閱分支。

特徵

用來進行預測的輸入資料。例如，在製造環境中，特徵可能是定期從製造生產線擷取的影像。

功能重要性

特徵對於模型的預測有多重要。這通常表示為可以透過各種技術來計算的數值得分，例如 Shapley Additive Explanations (SHAP) 和積分梯度。如需詳細資訊，請參閱 的機器學習模型可解譯性 AWS。

特徵轉換

優化 ML 程序的資料，包括使用其他來源豐富資料、調整值、或從單一資料欄位擷取多組資訊。這可讓 ML 模型從資料中受益。例如，如果將「2021-05-27 00:15:37」日期劃分為「2021」、「五月」、「週四」和「15」，則可以協助學習演算法學習與不同資料元件相關聯的細微模式。

少量擷取提示

在要求 LLM 執行類似的任務之前，提供少量示範任務和所需輸出的範例。此技術是內容內學習的應用程式，其中模型會從內嵌在提示中的範例 (快照) 中學習。對於需要特定格式、推理或網域知識的任務，少量的提示可以有效。另請參閱零鏡頭提示。

FGAC

請參閱精細存取控制。

精細存取控制 (FGAC)

使用多個條件來允許或拒絕存取請求。

閃切遷移

一種資料庫遷移方法，透過變更資料擷取使用連續資料複寫，以盡可能在最短的時間內遷移資料，而不是使用分階段方法。目標是將停機時間降至最低。

FM

請參閱基礎模型。

基礎模型 (FM)

大型深度學習神經網路，已在廣義和未標記資料的大量資料集上進行訓練。FMs 能夠執行各種一般任務，例如了解語言、產生文字和影像，以及以自然語言交談。如需詳細資訊，請參閱什麼是基礎模型。

G

生成式 AI

已針對大量資料進行訓練的 AI 模型子集，可使用簡單的文字提示建立新的內容和成品，例如影像、影片、文字和音訊。如需詳細資訊，請參閱什麼是生成式 AI。

地理封鎖

請參閱地理限制。

地理限制 (地理封鎖)

Amazon CloudFront 中的選項，可防止特定國家/地區的使用者存取內容分發。您可以使用允許清單或封鎖清單來指定核准和禁止的國家/地區。如需詳細資訊，請參閱 CloudFront 文件中的限制內容的地理分佈。

Gitflow 工作流程

這是一種方法，其中較低和較高環境在原始碼儲存庫中使用不同分支。Gitflow 工作流程被視為舊版，而以幹線為基礎的工作流程是現代、偏好的方法。

黃金影像

系統或軟體的快照，做為部署該系統或軟體新執行個體的範本。例如，在製造中，黃金映像可用於在多個裝置上佈建軟體，並有助於提高裝置製造操作的速度、可擴展性和生產力。

綠地策略

新環境中缺乏現有基礎設施。對系統架構採用綠地策略時，可以選擇所有新技術，而不會限制與現有基礎設施的相容性，也稱為棕地。如果正在擴展現有基礎設施，則可能會混合棕地和綠地策略。

防護機制

有助於跨組織單位 (OU) 來管控資源、政策和合規的高層級規則。預防性防護機制會強制執行政策，以確保符合合規標準。透過使用服務控制政策和 IAM 許可界限來將其實作。偵測性防護機制可偵測政策違規和合規問題，並產生提醒以便修正。它們是透過使用 AWS Config AWS Security Hub CSPM、Amazon GuardDuty、、Amazon Inspector AWS Trusted Advisor和自訂 AWS Lambda 檢查來實作。

H

HA

請參閱高可用性。

異質資料庫遷移

將來源資料庫遷移至使用不同資料庫引擎的目標資料庫 (例如，Oracle 至 Amazon Aurora)。異質遷移通常是重新架構工作的一部分，而轉換結構描述可能是一項複雜任務。AWS 提供有助於結構描述轉換的 AWS SCT。

高可用性 (HA)

在遇到挑戰或災難時，工作負載能夠在不介入的情況下持續運作。HA 系統的設計目的是自動容錯移轉、持續提供高品質的效能，以及處理不同的負載和故障，並將效能影響降至最低。

歷史現代化

一種方法，用於現代化和升級操作技術 (OT) 系統，以更好地滿足製造業的需求。歷史資料是一種資料庫，用於從工廠中的各種來源收集和存放資料。

保留資料

從用於訓練機器學習模型的資料集中保留的部分歷史標記資料。您可以使用保留資料，透過比較模型預測與保留資料來評估模型效能。

異質資料庫遷移

將您的來源資料庫遷移至共用相同資料庫引擎的目標資料庫 (例如，Microsoft SQL Server 至 Amazon RDS for SQL Server)。同質遷移通常是主機轉換或平台轉換工作的一部分。您可以使用原生資料庫公用程式來遷移結構描述。

熱資料

經常存取的資料，例如即時資料或最近的轉譯資料。此資料通常需要高效能儲存層或類別，才能提供快速的查詢回應。

修補程序

緊急修正生產環境中的關鍵問題。由於其緊迫性，通常會在典型 DevOps 發行工作流程之外執行修補程式。

超級護理期間

在切換後，遷移團隊在雲端管理和監控遷移的應用程式以解決任何問題的時段。通常，此期間的長度為 1-4 天。在超級護理期間結束時，遷移團隊通常會將應用程式的責任轉移給雲端營運團隊。

I

IaC

請參閱基礎設施即程式碼。

身分型政策

連接至一或多個 IAM 主體的政策，可定義其在 AWS 雲端 環境中的許可。

閒置應用程式

90 天期間 CPU 和記憶體平均使用率在 5% 至 20% 之間的應用程式。在遷移專案中，通常會淘汰這些應用程式或將其保留在內部部署。

IIoT

請參閱工業物聯網。

不可變的基礎設施

為生產工作負載部署新基礎設施的模型，而不是更新、修補或修改現有的基礎設施。不可變基礎設施本質上比可變基礎設施更一致、可靠且可預測。如需詳細資訊，請參閱 AWS Well-Architected Framework 中的使用不可變基礎設施部署最佳實務。

傳入 (輸入) VPC

在 AWS 多帳戶架構中，接受、檢查和路由來自應用程式外部之網路連線的 VPC。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

增量遷移

一種切換策略，您可以在其中將應用程式分成小部分遷移，而不是執行單一、完整的切換。例如，您最初可能只將一些微服務或使用者移至新系統。確認所有項目都正常運作之後，您可以逐步移動其他微服務或使用者，直到可以解除委任舊式系統。此策略可降低與大型遷移關聯的風險。

工業 4.0

2016 年 Klaus Schwab 推出的術語，透過連線能力、即時資料、自動化、分析和 AI/ML 的進展，指製造程序的現代化。

基礎設施

應用程式環境中包含的所有資源和資產。

基礎設施即程式碼 (IaC)

透過一組組態檔案來佈建和管理應用程式基礎設施的程序。IaC 旨在協助您集中管理基礎設施，標準化資源並快速擴展，以便新環境可重複、可靠且一致。

工業物聯網 (IIoT)

在製造業、能源、汽車、醫療保健、生命科學和農業等產業領域使用網際網路連線的感測器和裝置。如需詳細資訊，請參閱建立工業物聯網 (IIoT) 數位轉型策略。

檢查 VPC

在 AWS 多帳戶架構中，集中式 VPC 可管理 VPCs 之間 （在相同或不同的 中 AWS 區域)、網際網路和內部部署網路之間的網路流量檢查。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

物聯網 (IoT)

具有內嵌式感測器或處理器的相連實體物體網路，其透過網際網路或本地通訊網路與其他裝置和系統進行通訊。如需詳細資訊，請參閱什麼是 IoT？

可解釋性

機器學習模型的一個特徵，描述了人類能夠理解模型的預測如何依賴於其輸入的程度。如需詳細資訊，請參閱 的機器學習模型可解譯性 AWS。

IoT

請參閱物聯網。

IT 資訊庫 (ITIL)

一組用於交付 IT 服務並使這些服務與業務需求保持一致的最佳實務。ITIL 為 ITSM 提供了基礎。

IT 服務管理 (ITSM)

與組織的設計、實作、管理和支援 IT 服務關聯的活動。如需有關將雲端操作與 ITSM 工具整合的資訊，請參閱操作整合指南。

ITIL

請參閱 IT 資訊庫。

ITSM

請參閱 IT 服務管理。

L

標籤型存取控制 (LBAC)

強制存取控制 (MAC) 的實作，其中使用者和資料本身都會獲得明確指派的安全標籤值。使用者安全標籤和資料安全標籤之間的交集會決定使用者可以看到哪些資料列和資料欄。

登陸區域

登陸區域是架構良好的多帳戶 AWS 環境，可擴展且安全。這是一個起點，您的組織可以從此起點快速啟動和部署工作負載與應用程式，並對其安全和基礎設施環境充滿信心。如需有關登陸區域的詳細資訊，請參閱設定安全且可擴展的多帳戶 AWS 環境。

大型語言模型 (LLM)

預先訓練大量資料的深度學習 AI 模型。LLM 可以執行多個任務，例如回答問題、摘要文件、將文字翻譯成其他語言，以及完成句子。如需詳細資訊，請參閱什麼是 LLMs。

大型遷移

遷移 300 部或更多伺服器。

LBAC

請參閱標籤型存取控制。

最低權限

授予執行任務所需之最低許可的安全最佳實務。如需詳細資訊，請參閱 IAM 文件中的套用最低權限許可。

隨即轉移

請參閱 7 個 R。

小端序系統

首先儲存最低有效位元組的系統。另請參閱 Endianness。

LLM

請參閱大型語言模型。

較低的環境

請參閱 環境。

M

機器學習 (ML)

一種使用演算法和技術進行模式識別和學習的人工智慧。機器學習會進行分析並從記錄的資料 (例如物聯網 (IoT) 資料) 中學習，以根據模式產生統計模型。如需詳細資訊，請參閱機器學習。

主要分支

請參閱分支。

惡意軟體

旨在危及電腦安全或隱私權的軟體。惡意軟體可能會中斷電腦系統、洩露敏感資訊，或取得未經授權的存取。惡意軟體的範例包括病毒、蠕蟲、勒索軟體、特洛伊木馬程式、間諜軟體和鍵盤記錄器。

受管服務

AWS 服務 會 AWS 操作基礎設施層、作業系統和平台，而您會存取端點來存放和擷取資料。Amazon Simple Storage Service (Amazon S3) 和 Amazon DynamoDB 是受管服務的範例。這些也稱為抽象服務。

製造執行系統 (MES)

一種軟體系統，用於追蹤、監控、記錄和控制生產程序，將原物料轉換為現場成品。

MAP

請參閱遷移加速計劃。

機制

建立工具、推動工具採用，然後檢查結果以進行調整的完整程序。機制是在操作時強化和改善自身的循環。如需詳細資訊，請參閱 AWS Well-Architected Framework 中的建置機制。

成員帳戶

除了屬於組織一部分的管理帳戶 AWS 帳戶 之外的所有 AWS Organizations。帳戶一次只能是一個組織的成員。

製造執行系統

請參閱製造執行系統。

訊息佇列遙測傳輸 (MQTT)

根據發佈/訂閱模式的輕量型machine-to-machine(M2M) 通訊協定，適用於資源受限的 IoT 裝置。

微服務

一種小型的獨立服務，它可透過定義明確的 API 進行通訊，通常由小型獨立團隊擁有。例如，保險系統可能包含對應至業務能力 (例如銷售或行銷) 或子領域 (例如購買、索賠或分析) 的微服務。微服務的優點包括靈活性、彈性擴展、輕鬆部署、可重複使用的程式碼和適應力。如需詳細資訊，請參閱使用無 AWS 伺服器服務整合微服務。

微服務架構

一種使用獨立元件來建置應用程式的方法，這些元件會以微服務形式執行每個應用程式程序。這些微服務會使用輕量型 API，透過明確定義的介面進行通訊。此架構中的每個微服務都可以進行更新、部署和擴展，以滿足應用程式特定功能的需求。如需詳細資訊，請參閱在 上實作微服務 AWS。

Migration Acceleration Program (MAP)

一種 AWS 計畫，提供諮詢支援、訓練和服務，協助組織建立強大的營運基礎，以移至雲端，並協助抵銷遷移的初始成本。MAP 包括用於有條不紊地執行舊式遷移的遷移方法以及一組用於自動化和加速常見遷移案例的工具。

大規模遷移

將大部分應用程式組合依波次移至雲端的程序，在每個波次中，都會以更快的速度移動更多應用程式。此階段使用從早期階段學到的最佳實務和經驗教訓來實作團隊、工具和流程的遷移工廠，以透過自動化和敏捷交付簡化工作負載的遷移。這是 AWS 遷移策略的第三階段。

遷移工廠

可透過自動化、敏捷的方法簡化工作負載遷移的跨職能團隊。遷移工廠團隊通常包括營運、業務分析師和擁有者、遷移工程師、開發人員以及從事 Sprint 工作的 DevOps 專業人員。20% 至 50% 之間的企業應用程式組合包含可透過工廠方法優化的重複模式。如需詳細資訊，請參閱此內容集中的遷移工廠的討論和雲端遷移工廠指南。

遷移中繼資料

有關完成遷移所需的應用程式和伺服器的資訊。每種遷移模式都需要一組不同的遷移中繼資料。遷移中繼資料的範例包括目標子網路、安全群組和 AWS 帳戶。

遷移模式

可重複的遷移任務，詳細描述遷移策略、遷移目的地以及所使用的遷移應用程式或服務。範例：使用 AWS Application Migration Service 重新託管遷移至 Amazon EC2。

遷移組合評定 (MPA)

線上工具，提供驗證商業案例以遷移至 的資訊 AWS 雲端。MPA 提供詳細的組合評定 (伺服器適當規模、定價、總體擁有成本比較、遷移成本分析) 以及遷移規劃 (應用程式資料分析和資料收集、應用程式分組、遷移優先順序，以及波次規劃)。MPA 工具 （需要登入） 可供所有 AWS 顧問和 APN 合作夥伴顧問免費使用。

遷移準備程度評定 (MRA)

使用 AWS CAF 取得組織雲端整備狀態的洞見、識別優缺點，以及建立行動計劃以消除已識別差距的程序。如需詳細資訊，請參閱遷移準備程度指南。MRA 是 AWS 遷移策略的第一階段。

遷移策略

用來將工作負載遷移至 的方法 AWS 雲端。如需詳細資訊，請參閱此詞彙表中的 7 個 Rs 項目，並請參閱調動您的組織以加速大規模遷移。

機器學習 (ML)

請參閱機器學習。

現代化

將過時的 (舊版或單一) 應用程式及其基礎架構轉換為雲端中靈活、富有彈性且高度可用的系統，以降低成本、提高效率並充分利用創新。如需詳細資訊，請參閱《》中的現代化應用程式的策略 AWS 雲端。

現代化準備程度評定

這項評估可協助判斷組織應用程式的現代化準備程度；識別優點、風險和相依性；並確定組織能夠在多大程度上支援這些應用程式的未來狀態。評定的結果就是目標架構的藍圖、詳細說明現代化程序的開發階段和里程碑的路線圖、以及解決已發現的差距之行動計畫。如需詳細資訊，請參閱《》中的評估應用程式的現代化準備 AWS 雲端程度。

單一應用程式 (單一)

透過緊密結合的程序作為單一服務執行的應用程式。單一應用程式有幾個缺點。如果一個應用程式功能遇到需求激增，則必須擴展整個架構。當程式碼庫增長時，新增或改進單一應用程式的功能也會變得更加複雜。若要解決這些問題，可以使用微服務架構。如需詳細資訊，請參閱將單一體系分解為微服務。

MPA

請參閱遷移產品組合評估。

MQTT

請參閱訊息佇列遙測傳輸。

多類別分類

一個有助於產生多類別預測的過程 (預測兩個以上的結果之一)。例如，機器學習模型可能會詢問「此產品是書籍、汽車還是電話？」 或者「這個客戶對哪種產品類別最感興趣？」

可變基礎設施

更新和修改生產工作負載現有基礎設施的模型。為了提高一致性、可靠性和可預測性， AWS Well-Architected Framework 建議使用不可變基礎設施做為最佳實務。

O

OAC

請參閱原始存取控制。

OAI

請參閱原始存取身分。

OCM

請參閱組織變更管理。

離線遷移

一種遷移方法，可在遷移過程中刪除來源工作負載。此方法涉及延長停機時間，通常用於小型非關鍵工作負載。

OI

請參閱 操作整合。

OLA

請參閱操作層級協議。

線上遷移

一種遷移方法，無需離線即可將來源工作負載複製到目標系統。連接至工作負載的應用程式可在遷移期間繼續運作。此方法涉及零至最短停機時間，通常用於關鍵的生產工作負載。

OPC-UA

請參閱開放程序通訊 - 統一架構。

開放程序通訊 - 統一架構 (OPC-UA)

用於工業自動化的machine-to-machine(M2M) 通訊協定。OPC-UA 提供資料加密、身分驗證和授權機制的互通性標準。

操作水準協議 (OLA)

一份協議，闡明 IT 職能群組承諾向彼此提供的內容，以支援服務水準協議 (SLA)。

操作整備審查 (ORR)

問題和相關最佳實務的檢查清單，可協助您了解、評估、預防或減少事件和可能失敗的範圍。如需詳細資訊，請參閱 AWS Well-Architected Framework 中的操作整備審查 (ORR)。

操作技術 (OT)

使用實體環境控制工業操作、設備和基礎設施的硬體和軟體系統。在製造中，整合 OT 和資訊技術 (IT) 系統是工業 4.0 轉型的關鍵重點。

操作整合 (OI)

在雲端中將操作現代化的程序，其中包括準備程度規劃、自動化和整合。如需詳細資訊，請參閱操作整合指南。

組織追蹤

由 建立的線索 AWS CloudTrail ，會記錄 AWS 帳戶 組織中所有 的所有事件 AWS Organizations。在屬於組織的每個 AWS 帳戶 中建立此追蹤，它會跟蹤每個帳戶中的活動。如需詳細資訊，請參閱 CloudTrail 文件中的建立組織追蹤。

組織變更管理 (OCM)

用於從人員、文化和領導力層面管理重大、顛覆性業務轉型的架構。OCM 透過加速變更採用、解決過渡問題，以及推動文化和組織變更，協助組織為新系統和策略做好準備，並轉移至新系統和策略。在 AWS 遷移策略中，此架構稱為人員加速，因為雲端採用專案所需的變更速度。如需詳細資訊，請參閱 OCM 指南。

原始存取控制 (OAC)

CloudFront 中的增強型選項，用於限制存取以保護 Amazon Simple Storage Service (Amazon S3) 內容。OAC 支援所有 S3 儲存貯體中的所有伺服器端加密 AWS KMS (SSE-KMS) AWS 區域，以及對 S3 儲存貯體的動態PUT和DELETE請求。

原始存取身分 (OAI)

CloudFront 中的一個選項，用於限制存取以保護 Amazon S3 內容。當您使用 OAI 時，CloudFront 會建立一個可供 Amazon S3 進行驗證的主體。經驗證的主體只能透過特定 CloudFront 分發來存取 S3 儲存貯體中的內容。另請參閱 OAC，它可提供更精細且增強的存取控制。

ORR

請參閱操作整備審核。

OT

請參閱操作技術。

傳出 (輸出) VPC

在 AWS 多帳戶架構中，處理從應用程式內啟動之網路連線的 VPC。AWS 安全參考架構建議您使用傳入、傳出和檢查 VPC 來設定網路帳戶，以保護應用程式與更廣泛的網際網路之間的雙向介面。

P

許可界限

附接至 IAM 主體的 IAM 管理政策，可設定使用者或角色擁有的最大許可。如需詳細資訊，請參閱 IAM 文件中的許可界限。

個人身分識別資訊 (PII)

當直接檢視或與其他相關資料配對時，可用來合理推斷個人身分的資訊。PII 的範例包括名稱、地址和聯絡資訊。

PII

請參閱個人身分識別資訊。

手冊

一組預先定義的步驟，可擷取與遷移關聯的工作，例如在雲端中提供核心操作功能。手冊可以採用指令碼、自動化執行手冊或操作現代化環境所需的程序或步驟摘要的形式。

PLC

請參閱可程式設計邏輯控制器。

PLM

請參閱產品生命週期管理。

政策

可定義許可的物件（請參閱身分型政策）、指定存取條件（請參閱資源型政策），或定義組織中所有帳戶的最大許可 AWS Organizations（請參閱服務控制政策）。

混合持久性

根據資料存取模式和其他需求，獨立選擇微服務的資料儲存技術。如果您的微服務具有相同的資料儲存技術，則其可能會遇到實作挑戰或效能不佳。如果微服務使用最適合其需求的資料儲存，則可以更輕鬆地實作並達到更好的效能和可擴展性。

組合評定

探索、分析應用程式組合並排定其優先順序以規劃遷移的程序。如需詳細資訊，請參閱評估遷移準備程度。

述詞

傳回 `true`或 的查詢條件`false`，通常位於 `WHERE`子句中。

述詞下推

一種資料庫查詢最佳化技術，可在傳輸前篩選查詢中的資料。這可減少必須從關聯式資料庫擷取和處理的資料量，並改善查詢效能。

預防性控制

旨在防止事件發生的安全控制。這些控制是第一道防線，可協助防止對網路的未經授權存取或不必要變更。如需詳細資訊，請參閱在 AWS上實作安全控制中的預防性控制。

委託人

中可執行動作和存取資源 AWS 的實體。此實體通常是 AWS 帳戶、IAM 角色或使用者的根使用者。如需詳細資訊，請參閱 IAM 文件中角色術語和概念中的主體。

依設計的隱私權

透過整個開發程序將隱私權納入考量的系統工程方法。

私有託管區域

一種容器，它包含有關您希望 Amazon Route 53 如何回應一個或多個 VPC 內的域及其子域之 DNS 查詢的資訊。如需詳細資訊，請參閱 Route 53 文件中的使用私有託管區域。

主動控制

旨在防止部署不合規資源的安全控制。這些控制項會在佈建資源之前對其進行掃描。如果資源不符合控制項，則不會佈建。如需詳細資訊，請參閱 AWS Control Tower 文件中的控制項參考指南，並參閱實作安全控制項中的主動控制項。 AWS

產品生命週期管理 (PLM)

管理產品整個生命週期的資料和程序，從設計、開發和啟動，到成長和成熟，再到拒絕和移除。

生產環境

請參閱 環境。

可程式設計邏輯控制器 (PLC)

在製造中，高度可靠、可調整的電腦，可監控機器並自動化製造程序。

提示鏈結

使用一個 LLM 提示的輸出做為下一個提示的輸入，以產生更好的回應。此技術用於將複雜任務分解為子任務，或反覆精簡或展開初步回應。它有助於提高模型回應的準確性和相關性，並允許更精細、個人化的結果。

擬匿名化

以預留位置值取代資料集中個人識別符的程序。假名化有助於保護個人隱私權。假名化資料仍被視為個人資料。

發佈/訂閱 (pub/sub)

一種模式，可啟用微服務之間的非同步通訊，以提高可擴展性和回應能力。例如，在微服務型 MES 中，微服務可以將事件訊息發佈到其他微服務可訂閱的頻道。系統可以新增新的微服務，而無需變更發佈服務。

Q

查詢計劃

一系列步驟，如指示，用於存取 SQL 關聯式資料庫系統中的資料。

查詢計劃迴歸

在資料庫服務優化工具選擇的計畫比對資料庫環境進行指定的變更之前的計畫不太理想時。這可能因為對統計資料、限制條件、環境設定、查詢參數繫結的變更以及資料庫引擎的更新所導致。

R

RACI 矩陣

請參閱負責、負責、諮詢、告知 (RACI)。

RAG

請參閱擷取增強產生。

勒索軟體

一種惡意軟體，旨在阻止對計算機系統或資料的存取，直到付款為止。

RASCI 矩陣

請參閱負責、負責、諮詢、告知 (RACI)。

RCAC

請參閱資料列和資料欄存取控制。

僅供讀取複本

用於唯讀用途的資料庫複本。您可以將查詢路由至僅供讀取複本以減少主資料庫的負載。

重新架構師

請參閱 7 Rs。

復原點目標 (RPO)

自上次資料復原點以來可接受的時間上限。這會決定最後一個復原點與服務中斷之間可接受的資料遺失。

復原時間目標 (RTO)

服務中斷與服務還原之間的可接受延遲上限。

重構

請參閱 7 Rs。

區域

地理區域中的 AWS 資源集合。每個 AWS 區域 都獨立於其他 ，以提供容錯能力、穩定性和彈性。如需詳細資訊，請參閱指定 AWS 區域 您的帳戶可以使用哪些。

迴歸

預測數值的 ML 技術。例如，為了解決「這房子會賣什麼價格？」的問題 ML 模型可以使用線性迴歸模型，根據已知的房屋事實 (例如，平方英尺) 來預測房屋的銷售價格。

重新託管

請參閱 7 Rs。

版本

在部署程序中，它是將變更提升至生產環境的動作。

重新定位

請參閱 7 個 R。

Replatform

請參閱 7 個 R。

回購

請參閱 7 Rs。

彈性

應用程式抵禦中斷或從中斷中復原的能力。在 中規劃彈性時，高可用性和災難復原是常見的考量 AWS 雲端。如需詳細資訊，請參閱AWS 雲端 彈性。

資源型政策

附接至資源的政策，例如 Amazon S3 儲存貯體、端點或加密金鑰。這種類型的政策會指定允許存取哪些主體、支援的動作以及必須滿足的任何其他條件。

負責者、當責者、事先諮詢者和事後告知者 (RACI) 矩陣

定義所有涉及遷移活動和雲端操作之各方的角色和責任的矩陣。矩陣名稱衍生自矩陣中定義的責任類型：負責人 (R)、責任 (A)、諮詢 (C) 和知情 (I)。支援 (S) 類型為選用。如果您包含支援，則矩陣稱為 RASCI 矩陣，如果您排除它，則稱為 RACI 矩陣。

回應性控制

一種安全控制，旨在驅動不良事件或偏離安全基準的補救措施。如需詳細資訊，請參閱在 AWS上實作安全控制中的回應性控制。

保留

請參閱 7 個 R。

淘汰

請參閱 7 Rs。

檢索增強生成 (RAG)

一種生成式 AI 技術，其中 LLM 會在產生回應之前參考訓練資料來源以外的授權資料來源。例如，RAG 模型可能會對組織的知識庫或自訂資料執行語意搜尋。如需詳細資訊，請參閱什麼是 RAG。

輪換

定期更新秘密的程序，讓攻擊者更難存取登入資料。

資料列和資料欄存取控制 (RCAC)

使用已定義存取規則的基本、彈性 SQL 表達式。RCAC 包含資料列許可和資料欄遮罩。

RPO

請參閱復原點目標。

RTO

請參閱復原時間目標。

執行手冊

執行特定任務所需的一組手動或自動程序。這些通常是為了簡化重複性操作或錯誤率較高的程序而建置。

S

SAML 2.0

許多身分提供者 (IdP) 使用的開放標準。此功能會啟用聯合單一登入 (SSO)，讓使用者可以登入 AWS 管理主控台 或呼叫 AWS API 操作，而不必為您組織中的每個人在 IAM 中建立使用者。如需有關以 SAML 2.0 為基礎的聯合詳細資訊，請參閱 IAM 文件中的關於以 SAML 2.0 為基礎的聯合。

SCADA

請參閱監督控制和資料擷取。

SCP

請參閱服務控制政策。

秘密

您以加密形式存放的 AWS Secrets Manager機密或限制資訊，例如密碼或使用者登入資料。它由秘密值及其中繼資料組成。秘密值可以是二進位、單一字串或多個字串。如需詳細資訊，請參閱 Secrets Manager 文件中的 Secrets Manager 秘密中的什麼內容？。

依設計的安全性

透過整個開發程序將安全性納入考量的系統工程方法。

安全控制

一種技術或管理防護機制，它可預防、偵測或降低威脅行為者利用安全漏洞的能力。安全控制有四種主要類型：預防性、偵測性、回應性和主動性。

安全強化

減少受攻擊面以使其更能抵抗攻擊的過程。這可能包括一些動作，例如移除不再需要的資源、實作授予最低權限的安全最佳實務、或停用組態檔案中不必要的功能。

安全資訊與事件管理 (SIEM) 系統

結合安全資訊管理 (SIM) 和安全事件管理 (SEM) 系統的工具與服務。SIEM 系統會收集、監控和分析來自伺服器、網路、裝置和其他來源的資料，以偵測威脅和安全漏洞，並產生提醒。

安全回應自動化

預先定義和程式設計的動作，旨在自動回應或修復安全事件。這些自動化可做為偵測或回應式安全控制，協助您實作 AWS 安全最佳實務。自動化回應動作的範例包括修改 VPC 安全群組、修補 Amazon EC2 執行個體或輪換登入資料。

伺服器端加密

由接收資料的 AWS 服務 在其目的地加密資料。

服務控制政策 (SCP)

為 AWS Organizations中的組織的所有帳戶提供集中控制許可的政策。SCP 會定義防護機制或設定管理員可委派給使用者或角色的動作限制。您可以使用 SCP 作為允許清單或拒絕清單，以指定允許或禁止哪些服務或動作。如需詳細資訊，請參閱 AWS Organizations 文件中的服務控制政策。

服務端點

的進入點 URL AWS 服務。您可以使用端點，透過程式設計方式連接至目標服務。如需詳細資訊，請參閱 AWS 一般參考 中的 AWS 服務 端點。

服務水準協議 (SLA)

一份協議，闡明 IT 團隊承諾向客戶提供的服務，例如服務正常執行時間和效能。

服務層級指標 (SLI)

服務效能方面的測量，例如其錯誤率、可用性或輸送量。

服務層級目標 (SLO)

代表服務運作狀態的目標指標，由服務層級指標測量。

共同責任模式

描述您與 共同 AWS 承擔雲端安全與合規責任的模型。 AWS 負責雲端的安全，而 負責雲端的安全。如需詳細資訊，請參閱共同責任模式。

SIEM

請參閱安全資訊和事件管理系統。

單一故障點 (SPOF)

應用程式的單一關鍵元件故障，可能會中斷系統。

SLA

請參閱服務層級協議。

SLI

請參閱服務層級指標。

SLO

請參閱服務層級目標。

先拆分後播種模型

擴展和加速現代化專案的模式。定義新功能和產品版本時，核心團隊會進行拆分以建立新的產品團隊。這有助於擴展組織的能力和服務，提高開發人員生產力，並支援快速創新。如需詳細資訊，請參閱 中的階段式應用程式現代化方法 AWS 雲端。

SPOF

請參閱單一故障點。

星狀結構描述

使用一個大型事實資料表來存放交易或測量資料的資料庫組織結構，並使用一或多個較小的維度資料表來存放資料屬性。此結構旨在用於資料倉儲或商業智慧用途。

Strangler Fig 模式

一種現代化單一系統的方法，它會逐步重寫和取代系統功能，直到舊式系統停止使用為止。此模式源自無花果藤，它長成一棵馴化樹並最終戰勝且取代了其宿主。該模式由 Martin Fowler 引入，作為重寫單一系統時管理風險的方式。如需有關如何套用此模式的範例，請參閱使用容器和 Amazon API Gateway 逐步現代化舊版 Microsoft ASP.NET (ASMX) Web 服務。

子網

您 VPC 中的 IP 地址範圍。子網必須位於單一可用區域。

監控控制和資料擷取 (SCADA)

在製造中，使用硬體和軟體來監控實體資產和生產操作的系統。

對稱加密

使用相同金鑰來加密及解密資料的加密演算法。

合成測試

以模擬使用者互動的方式測試系統，以偵測潛在問題或監控效能。您可以使用 Amazon CloudWatch Synthetics 來建立這些測試。

系統提示

一種向 LLM 提供內容、指示或指導方針以指示其行為的技術。系統提示有助於設定內容，並建立與使用者互動的規則。

T

標籤

做為中繼資料以組織 AWS 資源的鍵/值對。標籤可協助您管理、識別、組織、搜尋及篩選資源。如需詳細資訊，請參閱標記您的 AWS 資源。

目標變數

您嘗試在受監督的 ML 中預測的值。這也被稱為結果變數。例如，在製造設定中，目標變數可能是產品瑕疵。

任務清單

用於透過執行手冊追蹤進度的工具。任務清單包含執行手冊的概觀以及要完成的一般任務清單。對於每個一般任務，它包括所需的預估時間量、擁有者和進度。

測試環境

請參閱 環境。

訓練

為 ML 模型提供資料以供學習。訓練資料必須包含正確答案。學習演算法會在訓練資料中尋找將輸入資料屬性映射至目標的模式 (您想要預測的答案)。它會輸出擷取這些模式的 ML 模型。可以使用 ML 模型，來預測您不知道的目標新資料。

傳輸閘道

可以用於互連 VPC 和內部部署網路的網路傳輸中樞。如需詳細資訊，請參閱 AWS Transit Gateway 文件中的什麼是傳輸閘道。

主幹型工作流程

這是一種方法，開發人員可在功能分支中本地建置和測試功能，然後將這些變更合併到主要分支中。然後，主要分支會依序建置到開發環境、生產前環境和生產環境中。

受信任的存取權

將許可授予您指定的服務，以代表您在組織中 AWS Organizations 及其帳戶中執行任務。受信任的服務會在需要該角色時，在每個帳戶中建立服務連結角色，以便為您執行管理工作。如需詳細資訊，請參閱 文件中的 AWS Organizations 搭配使用 AWS Organizations 與其他 AWS 服務。

調校

變更訓練程序的各個層面，以提高 ML 模型的準確性。例如，可以透過產生標籤集、新增標籤、然後在不同的設定下多次重複這些步驟來訓練 ML 模型，以優化模型。

雙比薩團隊

兩個比薩就能吃飽的小型 DevOps 團隊。雙披薩團隊規模可確保軟體開發中的最佳協作。

U

不確定性

這是一個概念，指的是不精確、不完整或未知的資訊，其可能會破壞預測性 ML 模型的可靠性。有兩種類型的不確定性：認知不確定性是由有限的、不完整的資料引起的，而隨機不確定性是由資料中固有的噪聲和隨機性引起的。如需詳細資訊，請參閱量化深度學習系統的不確定性指南。

未區分的任務

也稱為繁重工作，這是建立和操作應用程式的必要工作，但不為最終使用者提供直接價值或提供競爭優勢。未區分任務的範例包括採購、維護和容量規劃。

較高的環境

請參閱 環境。

V

清空

一種資料庫維護操作，涉及增量更新後的清理工作，以回收儲存並提升效能。

版本控制

追蹤變更的程序和工具，例如儲存庫中原始程式碼的變更。

VPC 對等互連

兩個 VPC 之間的連線，可讓您使用私有 IP 地址路由流量。如需詳細資訊，請參閱 Amazon VPC 文件中的什麼是 VPC 對等互連。

漏洞

危害系統安全性的軟體或硬體瑕疵。

W

暖快取

包含經常存取的目前相關資料的緩衝快取。資料庫執行個體可以從緩衝快取讀取，這比從主記憶體或磁碟讀取更快。

暖資料

不常存取的資料。查詢這類資料時，通常可接受中等速度的查詢。

視窗函數

SQL 函數，對與目前記錄在某種程度上相關的資料列群組執行計算。視窗函數適用於處理任務，例如根據目前資料列的相對位置計算移動平均值或存取資料列的值。

工作負載

提供商業價值的資源和程式碼集合，例如面向客戶的應用程式或後端流程。

工作串流

遷移專案中負責一組特定任務的功能群組。每個工作串流都是獨立的，但支援專案中的其他工作串流。例如，組合工作串流負責排定應用程式、波次規劃和收集遷移中繼資料的優先順序。組合工作串流將這些資產交付至遷移工作串流，然後再遷移伺服器和應用程式。

WORM

請參閱寫入一次，讀取許多。

WQF

請參閱AWS 工作負載資格架構。

寫入一次，讀取許多 (WORM)

儲存模型，可一次性寫入資料，並防止刪除或修改資料。授權使用者可以視需要多次讀取資料，但無法變更資料。此資料儲存基礎設施被視為不可變。

Z

零時差入侵

利用零時差漏洞的攻擊，通常是惡意軟體。

零時差漏洞

生產系統中未緩解的缺陷或漏洞。威脅行為者可以使用這種類型的漏洞來攻擊系統。開發人員經常因為攻擊而意識到漏洞。

零鏡頭提示

提供 LLM 執行任務的指示，但沒有可協助引導任務的範例 (快照)。LLM 必須使用其預先訓練的知識來處理任務。零鏡頭提示的有效性取決於任務的複雜性和提示的品質。另請參閱少量擷取提示。

殭屍應用程式

CPU 和記憶體平均使用率低於 5% 的應用程式。在遷移專案中，通常會淘汰這些應用程式。

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。