Best practices for using the AWS CDK in TypeScript to create IaC projects

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Best practices for using the AWS CDK in TypeScript to create IaC projects

Table of Contents

Introduction **1**
Objectives **2**
Best practices **3**
Organize code for large-scale projects 3
Why code organization is important 3
How to organize your code for scale 3
Sample code organization 4
Develop reusable patterns 6
Abstract Factory 6
Chain of Responsibility 7
Create or extend constructs 8
What a construct is 8
What the different types of constructs are 8
How to create your own construct 9
Create or extend an L2 construct 10
Create an L3 construct 11
Escape hatch 12
Custom resources 13
Follow TypeScript best practices 16
Describe your data 16
Use enums 16
Use interfaces 17
Extend interfaces 18
Avoid empty interfaces 18
Use factories 19
Use destructuring on properties 19
Define standard naming conventions 19
Don't use the var keyword 20
Consider using ESLint and Prettier 20
Use access modifiers 21
Use utility types 21
Scan for security vulnerabilities and formatting errors 22
Security approaches and tools 22
Common development tools 23

Develop and refine documentation ... 23
Why code documentation is required for AWS CDK constructs ... 24
Using TypeDoc with the AWS Construct Library ... 24
Adopt a test-driven development approach ... 25
Unit test ... 26
Integration test ... 30
Use release and version control for constructs ... 30
Version control for the AWS CDK ... 30
Repository and packaging for AWS CDK constructs ... 31
Construct releasing for the AWS CDK ... 32
Enforce library version management ... 33
FAQ ... 34
What problems can TypeScript solve? ... 34
Why should I use TypeScript? ... 34
Should I use the AWS CDK or CloudFormation? ... 34
What if the AWS CDK doesn't support a newly launched AWS service? ... 34
What are the different programming languages supported by the AWS CDK? ... 35
How much does the AWS CDK cost? ... 35
Next steps ... 36
Resources ... 37
Document history ... 38
Glossary ... 39
... 39
A ... 40
B ... 43
C ... 45
D ... 48
E ... 52
F ... 54
G ... 56
H ... 57
I ... 58
L ... 61
M ... 62
O ... 66
P ... 69

Q 71
R 72
S 75
T 79
U 80
V 81
W 81
Z 82

Best practices for using the AWS CDK in TypeScript to create IaC projects

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

February 2024 (document history)

This guide provides recommendations and best practices for using the AWS Cloud Development Kit (AWS CDK) in TypeScript to build and deploy large-scale infrastructure as code (IaC) projects. The AWS CDK is a framework for defining cloud infrastructure in code and provisioning that infrastructure through AWS CloudFormation. If you don't have a well-defined project structure, building and managing an AWS CDK codebase for large-scale projects can be challenging. To deal with these challenges, some organizations use anti-patterns for large-scale projects, but these patterns can slow down your project and create other issues that negatively impact your organization. For example, anti-patterns can complicate and slow down developer onboarding, bug fixes, and the adoption of new features.

This guide provides an alternative to using anti-patterns and shows you how to organize your code for scalability, testing, and alignment with security best practices. You can use this guide to improve code quality for your IaC projects and maximize your business agility. This guide is intended for architects, technical leads, infrastructure engineers, and any other role seeking to build a well-architected AWS CDK project for large-scale projects.

Objectives

This guide can help you achieve the following targeted business outcomes:

- **Reduced costs** – You can use the AWS CDK to design your own reusable components that meet your organization's security, compliance, and governance requirements. You can also easily share components around your organization, so that you can rapidly bootstrap new projects that align with best practices by default.
- **Faster time to market** – Take advantage of familiar features in the AWS CDK to accelerate your development process. This increases reusability for deployment and reduces development efforts.
- **Increased developer productivity** – Developers can use familiar programming languages to define infrastructure. This helps developers express and maintain AWS resources. This can lead to increased developer efficiency and collaboration.

Best practices

This section provides an overview of the following best practices:

- Organize code for large-scale projects
- Develop reusable patterns
- Create or extend constructs
- Follow TypeScript best practices
- Scan for security vulnerabilities and formatting errors
- Develop and refine documentation
- Adopt a test-driven development approach
- Use release and version control for constructs
- Enforce library version management

Organize code for large-scale projects

Why code organization is important

It's critical for large-scale AWS CDK projects to have a high-quality, well-defined structure. As a project gets larger and its number of supported features and constructs grows, code navigation becomes more difficult. This difficulty can impact productivity and slow down developer onboarding.

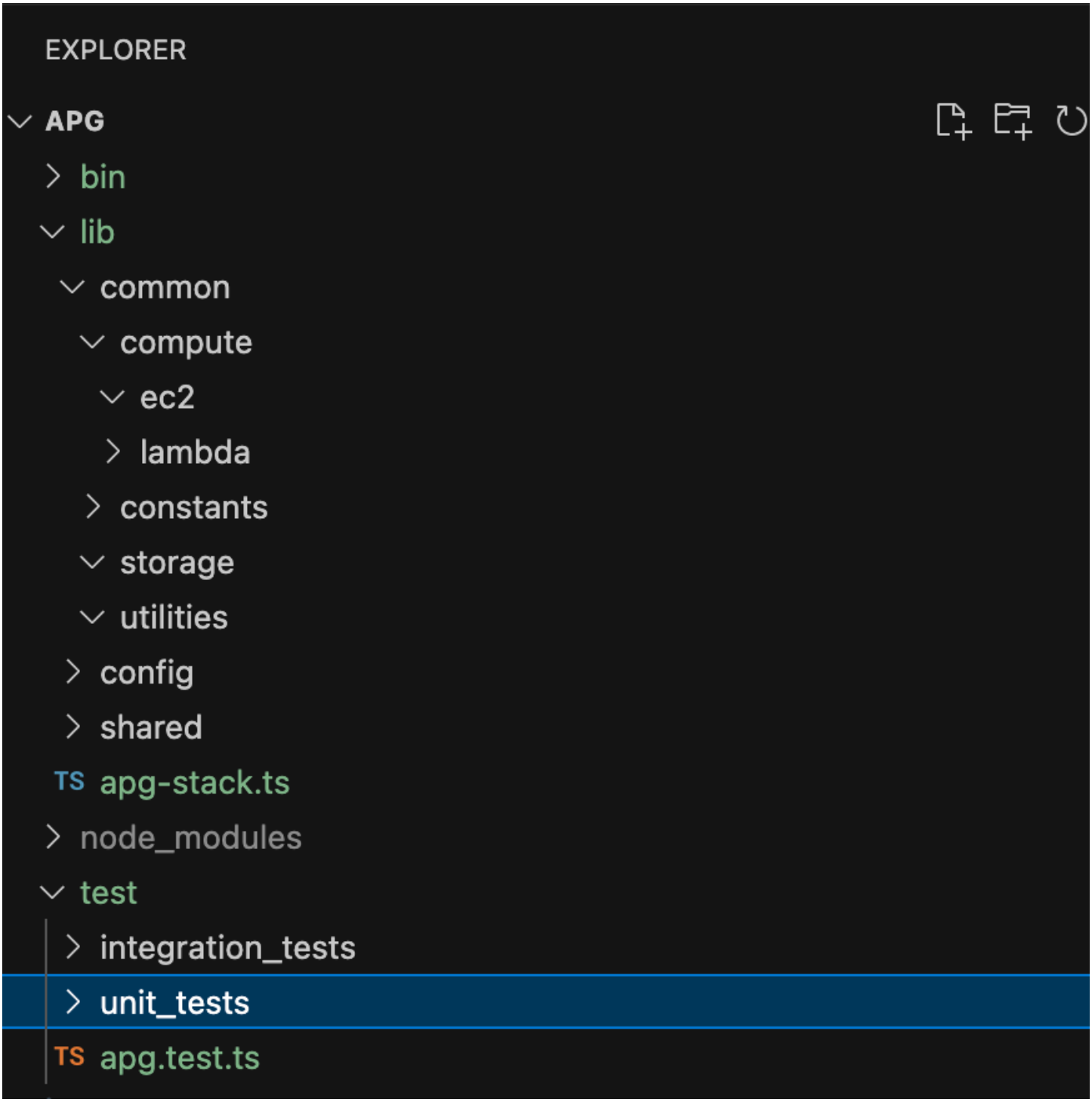
How to organize your code for scale

To achieve a high level of code flexibility and readability, we recommend that you divide your code into logical pieces based on functionality. This division reflects the fact that most of your constructs are used in different business domains. For example, both your frontend and backend applications could require an AWS Lambda function and consume the same source code. Factories can create objects without exposing the creation logic to the client and use a common interface to refer to newly created objects. You can use a factory as an effective pattern for creating a consistent behavior in your code base. Additionally, a factory can serve as a single source of truth to help you avoid repetitive code and make troubleshooting easier.

To better understand how factories work, consider the example of a car manufacturer. A car manufacturer doesn't need to have the knowledge and infrastructure required for manufacturing tires. Instead, the car manufacturer outsources that expertise to a specialized manufacturer of tires, and then simply orders the tires from that manufacturer as needed. The same principle applies to code. For example, you can create a Lambda factory that is capable of building high-quality Lambda functions, and then call the Lambda factory in your code whenever you need to create a Lambda function. Similarly, you can use this same outsourcing process to decouple your application and build modular components.

Sample code organization

The following TypeScript sample project, as shown in the following image, includes a **common** folder where you can keep all your constructs or common functionalities.



For example, the **compute** folder (residing in the **common** folder) holds all the logic for different compute constructs. New developers can easily add new compute constructs without impacting the other resources. All the other constructs won't need to create new resources internally. Instead, these constructs simply call the common construct factory. You can organize other constructs, such as storage, in the same way.

Configurations contain environment-based data that you must decouple from the **common** folder where you keep the logic. We recommend that you place your common **config** data in a shared folder. We also recommend that you use the **utilities** folder to serve all the helper functions and clean up scripts.

Develop reusable patterns

Software design patterns are reusable solutions to common problems in software development. They act as a guide or paradigm to help software engineers create products that follow best practices. This section provides an overview of two reusable patterns that you can use in your AWS CDK codebase: the Abstract Factory pattern and the Chain of Responsibility pattern. You can use each pattern as a blueprint and customize it for the particular design problem in your code. For more information on design patterns, see Design Patterns in the Refactoring.Guru documentation.

Abstract Factory

The Abstract Factory pattern provides interfaces for creating families of related or dependent objects without specifying their concrete classes. This pattern applies to the following use cases:

- When the client is independent of how you create and compose the objects in the system
- When the system consists of multiple families of objects, and these families are designed to be used together
- When you must have a runtime value to construct a particular dependency

For more information about the Abstract Factory pattern, see Abstract Factory in TypeScript in the Refactoring.Guru documentation.

The following code example shows how the Abstract Factory pattern can be used to build an Amazon Elastic Block Store (Amazon EBS) storage factory.

```
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();
```

Chain of Responsibility

Chain of Responsibility is a behavioral design pattern that enables you to pass a request along the chain of potential handlers until one of them handles the request. The Chain of Responsibility pattern applies to the following use cases:

- When multiple objects, determined at runtime, are candidates to handle a request
- When you don't want to specify handlers explicitly in your code
- When you want to issue a request to one of several objects without specifying the receiver explicitly

For more information about the Chain of Responsibility pattern, see Chain of Responsibility in TypeScript in the Refactoring.Guru documentation.

The following code shows an example of how the Chain of Responsibility pattern is used to build a series of actions that are required for completing the task.

```
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);
    }
}
```

Create or extend constructs

What a construct is

A construct is the basic building block of an AWS CDK application. A construct can represent a single AWS resource, such as an Amazon Simple Storage Service (Amazon S3) bucket, or it can be a higher-level abstraction consisting of multiple AWS related resources. The components of a construct can include a worker queue with its associated compute capacity, or a scheduled job with monitoring resources and a dashboard. The AWS CDK includes a collection of constructs called the AWS Construct Library. The library contains constructs for every AWS service. You can use the Construct Hub to discover additional constructs from AWS, third parties, and the open-source AWS CDK community.

What the different types of constructs are

There are three different types of constructs for the AWS CDK:

- **L1 constructs** – Layer 1, or L1, constructs are exactly the resources defined by CloudFormation —no more, no less. You must provide the resources required for configuration yourself. These L1 constructs are very basic and must be manually configured. L1 constructs have a `Cfn` prefix and

correspond directly to CloudFormation specifications. New AWS services are supported in AWS CDK as soon as CloudFormation has support for these services. CfnBucket is a good example of an L1 construct. This class represents an S3 bucket where you must explicitly configure all the properties. We recommend that you only use an L1 construct if you can't find the L2 or L3 construct for it.

- **L2 constructs** – Layer 2, or L2, constructs have common boilerplate code and glue logic. These constructs come with convenient defaults and reduce the amount of knowledge you need to know about them. L2 constructs use intent-based APIs to construct your resources and typically encapsulate their corresponding L1 modules. A good example of an L2 construct is Bucket. This class creates an S3 bucket with default properties and methods such as bucket.addLifeCycleRule(), which adds a lifecycle rule to the bucket.
- **L3 constructs** – A layer 3, or L3, construct is called a *pattern*. L3 constructs are designed to help you complete common tasks in AWS, often involving multiple kinds of resources. These are even more specific and opinionated than L2 constructs and serve a specific use case. For example, the aws-ecs-patterns.ApplicationLoadBalancedFargateService construct represents an architecture that includes an AWS Fargate container cluster that uses an Application Load Balancer. Another example is the aws-apigateway.LambdaRestApi construct. This construct represents an Amazon API Gateway API that's backed by a Lambda function.

As construct levels get higher, more assumptions are made as to how these constructs are going to be used. This allows you to provide interfaces with more effective defaults for highly specific use cases.

How to create your own construct

To define your own construct, you must follow a specific approach. This is because all constructs extend the `Construct` class. The `Construct` class is the building block of the construct tree. Constructs are implemented in classes that extend the `Construct` base class. All constructs take three parameters when they are initialized:

- **Scope** – A construct's parent or owner, either a stack or another construct, which determines its place in the construct tree. You usually must pass `this` (or `self` in Python), which represents the current object, for the scope.
- **id** – An identifier that must be unique within this scope. The identifier serves as a namespace for everything that's defined within the current construct and is used to allocate unique identities, such as resource names and CloudFormation logical IDs.

- **Props** – A set of properties that define the construct's initial configuration.

The following example shows how to define a construct.

```
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
  }
}
```

Create or extend an L2 construct

An L2 construct represents a "cloud component" and encapsulates everything that CloudFormation must have to create the component. An L2 construct can contain one or more AWS resources, and you're free to customize the construct yourself. The advantage of creating or extending an L2 construct is that you can reuse the components in CloudFormation stacks without redefining the code. You can simply import the construct as a class.

When there's an “is a” relationship to an existing construct you can extend an existing construct to add additional default features. It's a best practice to reuse the properties of the existing L2 construct. You can overwrite properties by modifying the properties directly in the constructor.

The following example shows how to align with best practices and extend an existing L2 construct called `s3.Bucket`. The extension establishes default properties, such as `versioned`, `publicReadAccess`, `blockPublicAccess`, to make sure that all the objects (in this example, S3 buckets) created from this new construct will always have these default values set.

```
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
    });
  }
}
```

Create an L3 construct

Composition is the better choice when there's a “has a” relationship with an existing construct composition. Composition means that you build your own construct on top of other existing constructs. You can create your own pattern to encapsulate all the resources and their default values inside a single higher-level L3 construct that can be shared. The benefit of creating your own L3 constructs (patterns) is that you can reuse the components in stacks without redefining the code. You can simply import the construct as a class. These patterns are designed to help consumers provision multiple resources based on common patterns with a limited amount of knowledge in a concise manner.

The following code example creates an AWS CDK construct called `ExampleConstruct`. You can use this construct as a template to define your cloud components.

```
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
  }
}
```

The following example shows how to import the newly created construct in your AWS CDK application or stack.

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

The following example shows how you can instantiate an instance of the construct that you extended from the base class.

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

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

For more information, see AWS CDK Workshop in the AWS CDK Workshop documentation.

Escape hatch

You can use an escape hatch in the AWS CDK to go up an abstraction level so that you can access the lower level of constructs. Escape hatches are used to extend the construct for features which are not exposed with the current version of AWS but available in CloudFormation.

We recommend that you use an escape hatch in the following scenarios:

- An AWS service feature is available through CloudFormation, but there are no `Construct` constructs for it.
- An AWS service feature is available through CloudFormation and there are `Construct` constructs for the service, but these don't yet expose the feature. Because `Construct` constructs are developed "by hand," they may sometimes lag behind the CloudFormation resource constructs.

The following example code shows a common use case for using an escape hatch. In this example, the functionality that's not yet implemented in the higher-level construct is for adding `httpPutResponseHopLimit` for autoscaling `LaunchConfiguration`.

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

The preceding code example shows the following workflow:

1. You define your `AutoScalingGroup` by using an L2 construct. The L2 construct doesn't support updating the `httpPutResponseHopLimit`, so you must use an escape hatch.
2. You access the `node.defaultChild` property on the L2 `AutoScalingGroup` construct and cast it as the `CfnLaunchConfiguration` resource.
3. You can now set the `launchConfig.metadataOptions` property on the L1 `CfnLaunchConfiguration`.

Custom resources

You can use custom resources to write custom provisioning logic in templates that CloudFormation runs whenever you create, update (if you changed the custom resource), or delete stacks. For example, you can use a custom resource if you want to include resources that aren't available in the AWS CDK. That way you can still manage all your related resources in a single stack.

Building a custom resource involves writing a Lambda function that responds to a resource's CREATE, UPDATE, and DELETE lifecycle events. If your custom resource must make only a single API call, consider using the AwsCustomResource construct. This makes it possible to perform arbitrary SDK calls during a CloudFormation deployment. Otherwise, we suggest that you write your own Lambda function to perform the work that you must get done.

For more information on custom resources, see Custom resources in the CloudFormation documentation. For an example of how to use a custom resource, see the Custom Resource repository on GitHub.

The following example shows how to create a custom resource class to initiate a Lambda function and send CloudFormation a success or fail signal.

```
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();
    }
}
```

The following example shows the main logic of the custom resource.

```
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)
```

The following example shows how the AWS CDK stack calls the custom resource.

```
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();
```

Follow TypeScript best practices

TypeScript is a language that extends the capabilities of JavaScript. It's a strongly typed and object-oriented language. You can use TypeScript to specify the types of data being passed within your code and has the ability to report errors when the types don't match. This section provides an overview of TypeScript best practices.

Describe your data

You can use TypeScript to describe the shape of objects and functions in your code. Using the `any` type is equivalent to opting out of type checking for a variable. We recommend that you avoid using `any` in your code. Here is an example.

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

Use enums

You can use enums to define a set of named constants and define standards that can be reused in your code base. We recommend that you export your enums one time at the global level, and then let other classes import and use the enums. Assume that you want to create a set of possible actions to capture the events in your code base. TypeScript provides both numeric and string-based enums. The following example uses an enum.

```
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)
```

Use interfaces

An interface is a contract for the class. If you create a contract, then your users must comply with the contract. In the following example, an interface is used to standardize the `props` and ensure that callers provide the expected parameter when using this class.

```
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
```

```
})
```

Some properties can only be modified when an object is first created. You can specify this by putting `readonly` before the name of the property, as the following example shows.

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

Extend interfaces

Extending interfaces reduces duplication, because you don't have to copy the properties between interfaces. Also, the reader of your code can easily understand the relationships in your application.

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

Avoid empty interfaces

We recommend that you avoid empty interfaces due to the potential risks they create. In the following example, there's an empty interface called `BucketProps`. The `myS3Bucket1` and `myS3Bucket2` objects are both valid, but they follow different standards because the interface doesn't enforce any contracts. The following code will compile and print the properties but this introduces inconsistency in your application.

```
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",
});
```

Use factories

In an Abstract Factory pattern, an interface is responsible for creating a factory of related objects without explicitly specifying their classes. For example, you can create a Lambda factory for creating Lambda functions. Instead of creating a new Lambda function within your construct, you're delegating the creation process to the factory. For more information on this design pattern, see Abstract Factory in TypeScript in the Refactoring.Guru documentation.

Use destructuring on properties

Destructuring, introduced in ECMAScript 6 (ES6), is a JavaScript feature that gives you the ability to extract multiple pieces of data from an array or object and assign them to their own variables.

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

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

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

Define standard naming conventions

Enforcing a naming convention keeps the code base consistent and reduces overhead when thinking about how to name a variable. We recommend the following:

- Use camelCase for variable and function names.

- Use PascalCase for class names and interface names.
- Use camelCase for interface members.
- Use PascalCase for type names and enum names.
- Name files with camelCase (for example, `ebsVolumes.tsx` or `storage.tsb`)

Don't use the var keyword

The `let` statement is used to declare a local variable in TypeScript. It's similar to the `var` keyword, but it has some restrictions in scoping compared to the `var` keyword. A variable declared in a block with `let` is only available for use within that block. The `var` keyword cannot be block-scoped, which means it can be accessed outside a particular block (represented by `{}`) but not outside of the function it's defined in. You can redeclare and update `var` variables. It's a best practice to avoid using the `var` keyword.

Consider using ESLint and Prettier

ESLint statically analyzes your code to quickly find issues. You can use ESLint to create a series of assertions (called *lint rules*) that define how your code should look or behave. ESLint also has auto-fixer suggestions to help you improve your code. Finally, you can use ESLint to load in lint rules from shared plugins.

Prettier is a well-known code formatter that supports a variety of different programming languages. You can use Prettier to set your code style so that you can avoid manually formatting your code. After installation, you can update your `package.json` file and run the `npm run format` and `npm run lint` commands.

The following example shows you how to enable ESLint and the Prettier formatter for your AWS CDK project.

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

Use access modifiers

The private modifier in TypeScript limits visibility to the same class only. When you add the private modifier to a property or method, you can access that property or method within the same class.

The public modifier allows class properties and methods to be accessible from all locations. If you don't specify any access modifiers for properties and methods, they will take the public modifier by default.

The protected modifier allows properties and methods of a class to be accessible within the same class and within subclasses. Use the protected modifier when you expect to create subclasses in your AWS CDK application.

Use utility types

Utility types in TypeScript are predefined type functions that perform transformations and operations on existing types. This helps you create new types based on existing types. For example, you can change or extract properties, make properties optional or required, or create immutable versions of types. By using utility types, you can define more precise types and catch potential errors at compile time.

Partial<Type>

`Partial` marks all members of an input type `Type` as optional. This utility returns a type that represents all subsets of a given type. The following is an example of `Partial`.

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

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

Required<Type>

`Required` does the opposite of `Partial`. It makes all members of an input type `Type` non-optional (in other words, required). The following is an example of `Required`.

```
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
};
```

Scan for security vulnerabilities and formatting errors

Infrastructure as code (IaC) and automation have become essential for enterprises. With IaC being so robust, you have a large responsibility to manage security risks. Common IaC security risks can include the following:

- Over-permissive AWS Identity and Access Management (IAM) privileges
- Open security groups
- Unencrypted resources
- Access logs not turned on

Security approaches and tools

We recommend that you implement the following security approaches:

- **Vulnerability detection in development** – Remediating vulnerabilities in production is expensive and time-consuming due to the complexity of developing and distributing software patches. Additionally, vulnerabilities in production carry the risk of exploitation. We recommend that you use code scanning on your IaC resources so that vulnerabilities can be detected and remediated prior to release into production.
- **Compliance and auto-remediation** – AWS Config offers AWS managed rules. These rules help you enforce compliance and enable you to attempt auto-remediation by using AWS Systems

Manager automation. You can also create and associate custom automation documents by using AWS Config rules.

Common development tools

The tools covered in this section help you to extend their built-in functionality with your own custom rules. We recommend that you align your custom rules with your organization's standards. Here are some common development tools to consider:

- Use cfn-nag to identify infrastructure security issues, such as permissive IAM rules or password literals, in CloudFormation templates. For more information, see the GitHub cfn-nag repository from Stelligent.
- Use cdk-nag, inspired by cfn-nag, to validate that constructs within a given scope comply with a defined set of rules. You can also use cdk-nag for rule suppression and compliance reporting. The cdk-nag tool validates constructs by extending aspects in the AWS CDK. For more information, see Manage application security and compliance with the AWS Cloud Development Kit (AWS CDK) and cdk-nag in the AWS DevOps Blog.
- Use the open-source tool Checkov to perform static analysis on your IaC environment. Checkov helps identify cloud misconfigurations by scanning your infrastructure code in Kubernetes, Terraform, or CloudFormation. You can use Checkov to get outputs in different formats, including JSON, JUnit XML, or CLI. Checkov can handle variables effectively by building a graph that shows dynamic code dependency. For more information, see the GitHub Checkov repository from Bridgecrew.
- Use TFLint to check for errors and deprecated syntax and to help you enforce best practices. Note that TFLint may not validate provider-specific issues. For more information on TFLint, see the GitHub TFLint repository from Terraform Linters.
- Use Amazon Q Developer to perform security scans. When used in an integrated development environment (IDE), Amazon Q Developer provides AI-powered software development assistance. It can chat about code, provide inline code completions, generate net new code, scan your code for security vulnerabilities, and make code upgrades and improvements.

Develop and refine documentation

Documentation is critical to the success of your project. Not only does documentation explain how your code works but it also helps developers better understand the features and functionality

of your applications. Developing and refining high-quality documentation can strengthen the software development process, maintain high-quality software, and help with knowledge transfer between developers.

There are two categories of documentation: documentation inside the code and supporting documentation about the code. Documentation inside the code is in the form of comments. Supporting documentation about the code can be README files and external documents. It's not uncommon for developers to think of documentation as overhead, as the code itself is easy to understand. This could be true for small projects, but documentation is crucial for large-scale projects where multiple teams are involved.

It's a best practice for the author of the code to write the documentation since they have a good understanding of its functionalities. Developers can struggle with the additional overhead of maintaining separate supporting documentation. To overcome this challenge, developers can add the comments in the code and those comments can be extracted automatically so every version of code and documentation will be in sync.

There are a variety of different tools to help developers extract comments from code and generate the documentation for it. This guide focuses on TypeDoc as the preferred tool for AWS CDK constructs.

Why code documentation is required for AWS CDK constructs

AWS CDK common constructs are created by multiple teams in an organization and shared across different teams for consumption. Good documentation helps the consumers of the construct library easily integrate constructs and build their infrastructure with minimum effort. Keeping all documents in sync is a big task. We recommend that you maintain the document inside the code, which will be extracted using the TypeDoc library.

Using TypeDoc with the AWS Construct Library

TypeDoc is a document generator for TypeScript. You can use TypeDoc to read your TypeScript source files, parse the comments in those files, and then generate a static site that contains documentation for your code.

The following code shows you how to integrate TypeDoc with the AWS Construct Library, and then add the following packages in your `package.json` file in `devDependencies`.

```
{
```

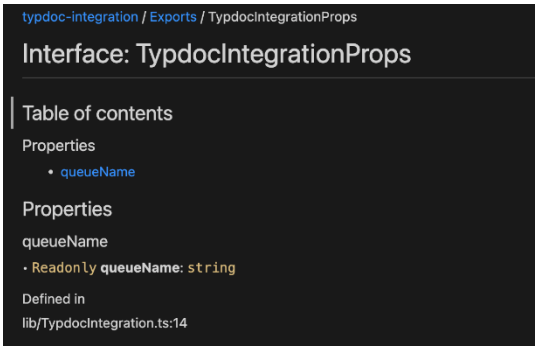
```
  "devDependencies": {

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

}
```

To add `typedoc.json` in the CDK library folder, use the following code.

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

To generate the README files, run the `npx typedoc` command in the root directory of the AWS CDK construct library project.

The following sample document is generated by TypeDoc.



For more information about TypeDoc integration options, see Doc Comments in the TypeDoc documentation.

Adopt a test-driven development approach

We recommend that you follow a test-driven development (TDD) approach with the AWS CDK. TDD is a software development approach where you develop test cases to specify and validate your code. In simple terms, first you create test cases for each functionality and if the test fails, then you write the new code to pass the test and make the code simple and bug-free.

You can use TDD to write the test case first. This helps you validate the infrastructure with different design constraints in terms of enforcing security policy for the resources and following a unique

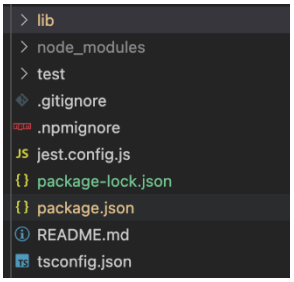
naming convention for the project. The standard approach to testing AWS CDK applications is to use the AWS CDK assertions module and popular test frameworks, such as Jest for TypeScript and JavaScript or pytest for Python.

There are two categories of tests that you can write for your AWS CDK applications:

- Use **fine-grained assertions** to test a specific aspect of the generated CloudFormation template, such as "this resource has this property with this value." These tests can detect regressions and are also useful when you're developing new features using TDD (write a test first, then make it pass by writing a correct implementation). Fine-grained assertions are the tests you'll write the most.
- Use **snapshot tests** to test the synthesized CloudFormation template against a previously-stored baseline template. Snapshot tests make it possible to refactor freely, since you can be sure that the refactored code works exactly the same way as the original. If the changes were intentional, you can accept a new baseline for future tests. However, AWS CDK upgrades can also cause synthesized templates to change, so you can't rely only on snapshots to make sure your implementation is correct.

Unit test

This guide focuses on unit test integration for TypeScript specifically. To enable testing, make sure that your `package.json` file has the following libraries: `@types/jest`, `jest`, and `ts-jest` in `devDependencies`. To add these packages, run the `cdk init lib --language=typescript` command. After you run the preceding command, you see the following structure.



The following code is an example of a `package.json` file that's enabled with the Jest library.

```
{
  ...
  "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",
    ...
  }
}
```

Under the **Test** folder, you can write the test case. The following example shows a test case for an AWS CodePipeline construct.

```
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);
});
);
```

To run a test, run the `npm run test` command in your project. The test returns the following results.

```
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
```

For more information on test cases, see Testing constructs in the *AWS Cloud Development Kit (AWS CDK) Developer Guide*.

Integration test

Integration tests for AWS CDK constructs can also be included by using an `integ-tests` module. An integration test should be defined as an AWS CDK application. There should be a one-to-one relationship between an integration test and an AWS CDK application. For more information, visit integ-tests-alpha module in the *AWS CDK API Reference*.

Use release and version control for constructs

Version control for the AWS CDK

AWS CDK common constructs can be created by multiple teams and shared across an organization for consumption. Typically, developers release new features or bug fixes in their common AWS CDK constructs. These constructs are used by AWS CDK applications or any other existing AWS CDK constructs as part of a dependency. For this reason, it's crucial that developers update and release their construct with proper semantic versions independently. Downstream AWS CDK applications or other AWS CDK constructs can update their dependency to use the newly released AWS CDK construct version.

Semantic versioning (Semver) is a set of rules, or method, for providing unique software numbers to computer software. Versions are defined as follows:

- A MAJOR version consists of incompatible API changes or a breaking change.
- A MINOR version consists of functionality that's added in a backwards-compatible manner.
- A PATCH version consists of backwards-compatible bug fixes.

For more information on semantic versioning, see Semantic Versioning Specification (SemVer) in the Semantic Versioning documentation.

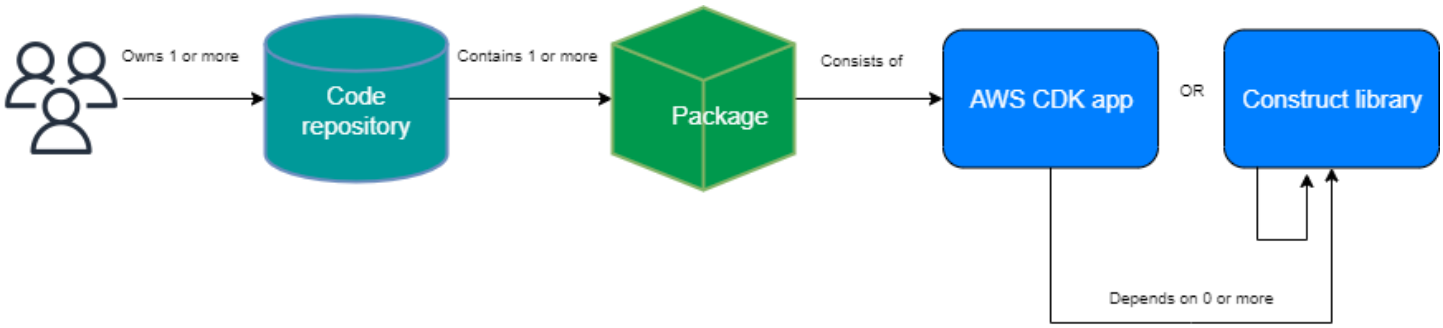
Repository and packaging for AWS CDK constructs

As AWS CDK constructs are developed by different teams and are used by multiple AWS CDK applications, you can use a separate repository for each AWS CDK construct. This also can help you enforce access control. Each repository could contain all the source code related to the same AWS CDK construct along with all of its dependencies. By keeping a single application (that is, an AWS CDK construct) in a single repository, you can decrease the scope of impact of changes during deployment.

The AWS CDK not only generates CloudFormation templates for deploying infrastructure, but it also bundles runtime assets like Lambda functions and Docker images and deploys them alongside your infrastructure. It's not only possible to combine the code that defines your infrastructure and the code that implements your runtime logic into a single construct—it's a best practice. These two kinds of code don't need to live in separate repositories or even in separate packages.

To consume packages across repository boundaries, you must have a private package repository —similar to npm, PyPi, or Maven Central, but internal to your organization. You must also have a release process that builds, tests, and publishes the package to the private package repository. You can create private repositories, such as PyPi server, by using a local virtual machine (VM) or Amazon S3. When you design or create a private package registry, it’s crucial to consider the risk of service disruption due to high availability and scalability. A serverless managed service that's hosted in the cloud to store packages can greatly decrease the maintenance overhead. For example, you can use AWS CodeArtifact to host packages for most popular programming languages. You can also use CodeArtifact to set external repository connections and replicate them within CodeArtifact.

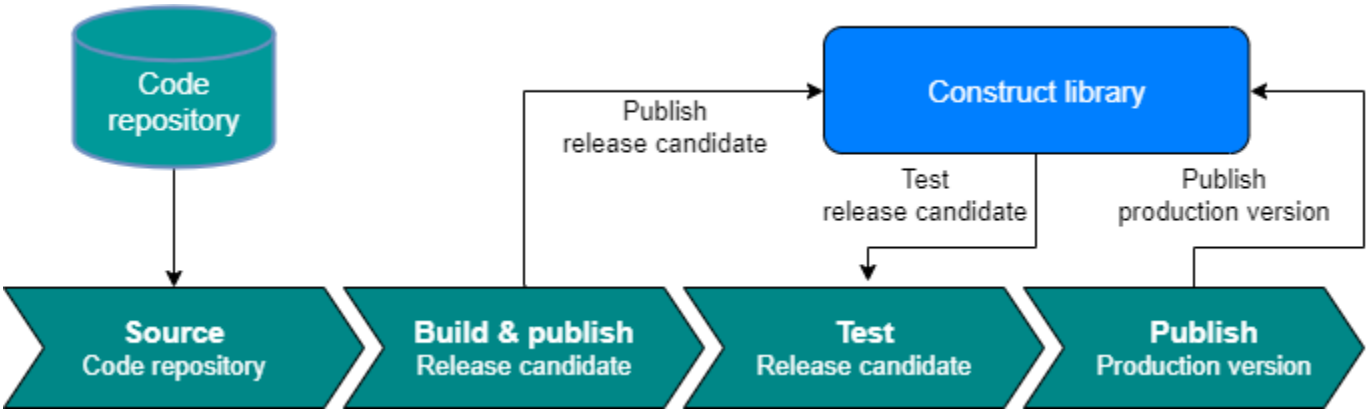
Dependencies on packages in the package repository are managed by your language's package manager (for example, npm for TypeScript or JavaScript applications). Your package manager makes sure that builds are repeatable by recording the specific versions of every package your application depends on and then lets you upgrade those dependencies in a controlled manner, as the following diagram shows.



Construct releasing for the AWS CDK

We recommend that you create your own automated pipeline to build and release new AWS CDK construct versions. If you put a proper pull request approval process in place, then once you commit and push your source code into the main branch of the repository, the pipeline can build and create a release candidate version. That version can be pushed to CodeArtifact and tested before releasing the production-ready version. Optionally, you can test your new AWS CDK construct version locally before merging the code with the main branch. This causes the pipeline to release the production-ready version. Take into consideration that shared constructs and packages must be tested independently of the consuming application, as if they were being released to the public.

The following diagram shows a sample AWS CDK version release pipeline.



You can use the following sample commands to build, test, and publish npm packages. First, sign in to the artifact repository by running the following command.

```
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>
```

Then, complete the following steps:

1. Install the required packages based on the `package.json` file: `npm install`
2. Create the release candidate version: `npm version prerelease --preid rc`
3. Build the npm package: `npm run build`
4. Test the npm package: `npm run test`
5. Publish the npm package: `npm publish`

Enforce library version management

Lifecycle management is a significant challenge when you're maintaining AWS CDK code bases. For example, assume that you start an AWS CDK project with version 1.97 and then version 1.169 becomes available later on. Version 1.169 offers new features and bug fixes, but you have deployed your infrastructure by using the old version. Now, updating the constructs becomes challenging as this gap increases because of the breaking changes that could be introduced in new versions. This can be a challenge if you have many resources in your environment. The pattern introduced in this section can help you manage your AWS CDK library version using automation. Here's the workflow of this pattern:

1. When you launch a new CodeArtifact Service Catalog product, the AWS CDK library versions and its dependencies are stored in the `package.json` file.
2. You deploy a common pipeline that keeps track of all the repositories so that you can apply automatic upgrades to them if there are no breaking changes.
3. An AWS CodeBuild stage checks for the dependency tree and looks for the breaking changes.
4. The pipeline creates a feature branch and then runs `cdk synth` with the new version to confirm there are no errors.
5. The new version is deployed in the test environment and finally runs an integration test to make sure the deployment is healthy.
6. You can use two Amazon Simple Queue Service (Amazon SQS) queues to keep track of the stacks. Users can review the stacks manually in the exception queue and address breaking changes. Items that pass the integration test are allowed to be merged and released.

FAQ

What problems can TypeScript solve?

Typically, you can eliminate bugs in your code by writing automated tests, manually verifying that the code works as expected, and then finally having another person validate your code. Validating the connections between every part of your project is time consuming. To speed up the validation process, you can use a type-checked language like TypeScript to automate code validation and provide instant feedback during development.

Why should I use TypeScript?

TypeScript is an open-source language that simplifies JavaScript code, making it easier to read and debug. TypeScript also provides highly productive development tools for JavaScript IDEs and practices, such as static checking. Additionally, TypeScript offers the benefits of ECMAScript 6 (ES6) and can increase your productivity. Finally, TypeScript can help you avoid painful bugs that developers commonly run into when writing JavaScript by type checking the code.

Should I use the AWS CDK or CloudFormation?

We recommend that you use the AWS Cloud Development Kit (AWS CDK) instead of AWS CloudFormation, if your organization has the development expertise to take advantage of the AWS CDK. This is because the AWS CDK is more flexible than CloudFormation, since you can use a programming language and OOP concepts. Keep in mind that you can use CloudFormation to create AWS resources in an orderly and predictable manner. In CloudFormation, resources are written in text files by using the JSON or YAML format.

What if the AWS CDK doesn't support a newly launched AWS service?

You can use a raw override or a CloudFormation custom resource.

What are the different programming languages supported by the AWS CDK?

AWS CDK is generally available in JavaScript, TypeScript, Python, Java, C#, and Go (in Developer Preview).

How much does the AWS CDK cost?

There is no additional charge for the AWS CDK. You pay for the AWS resources (such as Amazon EC2 instances or Elastic Load Balancing load balancers) that are created when you use AWS CDK in the same way as if you created them manually. You only pay for what you use, as you use it. There are no minimum fees and no required upfront commitments.

Next steps

We recommend that you start building with AWS Cloud Development Kit (AWS CDK) in TypeScript. For more information, see the AWS CDK Immersion Day Workshop.

Resources

References

- AWS Solution Constructs (AWS Solutions)
- AWS Cloud Development Kit (AWS CDK) (GitHub)
- AWS Construct Library API reference (AWS CDK Reference Documentation)
- AWS CDK Reference Documentation (AWS CDK Reference Documentation)
- AWS CDK Immersion Day Workshop (AWS Workshop Studio)

Tools

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

Guides and patterns

- AWS Solutions Constructs patterns (AWS documentation)

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Update code	We updated the code samples in the Follow TypeScript best practices section.	February 16, 2024
Add sections	We added the Use utility types and Integration test sections.	January 10, 2024
Minor update	Updated code example for creating an L3 construct.	June 16, 2023
Initial publication	—	December 8, 2022

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Numbers

7 Rs

Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
- Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. You migrate servers from an on-premises platform to a cloud service for the same platform. Example: Migrate a Microsoft Hyper-V application to AWS.
- Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there's no business justification for migrating them.

- Retire – Decommission or remove applications that are no longer needed in your source environment.

A

ABAC

See attribute-based access control.

abstracted services

See managed services.

ACID

See atomicity, consistency, isolation, durability.

active-active migration

A database migration method in which the source and target databases are kept in sync (by using a bidirectional replication tool or dual write operations), and both databases handle transactions from connecting applications during migration. This method supports migration in small, controlled batches instead of requiring a one-time cutover. It's more flexible but requires more work than active-passive migration.

active-passive migration

A database migration method in which in which the source and target databases are kept in sync, but only the source database handles transactions from connecting applications while data is replicated to the target database. The target database doesn't accept any transactions during migration.

aggregate function

A SQL function that operates on a group of rows and calculates a single return value for the group. Examples of aggregate functions include `SUM` and `MAX`.

AI

See artificial intelligence.

AIOps

See artificial intelligence operations.

anonymization

The process of permanently deleting personal information in a dataset. Anonymization can help protect personal privacy. Anonymized data is no longer considered to be personal data.

anti-pattern

A frequently used solution for a recurring issue where the solution is counter-productive, ineffective, or less effective than an alternative.

application control

A security approach that allows the use of only approved applications in order to help protect a system from malware.

application portfolio

A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence (AI)

The field of computer science that is dedicated to using computing technologies to perform cognitive functions that are typically associated with humans, such as learning, solving problems, and recognizing patterns. For more information, see What is Artificial Intelligence?

artificial intelligence operations (AIOps)

The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

asymmetric encryption

An encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption. You can share the public key because it isn't used for decryption, but access to the private key should be highly restricted.

atomicity, consistency, isolation, durability (ACID)

A set of software properties that guarantee the data validity and operational reliability of a database, even in the case of errors, power failures, or other problems.

attribute-based access control (ABAC)

The practice of creating fine-grained permissions based on user attributes, such as department, job role, and team name. For more information, see ABAC for AWS in the AWS Identity and Access Management (IAM) documentation.

authoritative data source

A location where you store the primary version of data, which is considered to be the most reliable source of information. You can copy data from the authoritative data source to other locations for the purposes of processing or modifying the data, such as anonymizing, redacting, or pseudonymizing it.

Availability Zone

A distinct location within an AWS Region that is insulated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same Region.

AWS Cloud Adoption Framework (AWS CAF)

A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS Workload Qualification Framework (AWS WQF)

A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

B

bad bot

A bot that is intended to disrupt or cause harm to individuals or organizations.

BCP

See business continuity planning.

behavior graph

A unified, interactive view of resource behavior and interactions over time. You can use a behavior graph with Amazon Detective to examine failed logon attempts, suspicious API calls, and similar actions. For more information, see Data in a behavior graph in the Detective documentation.

big-endian system

A system that stores the most significant byte first. See also endianness.

binary classification

A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?" or "Is this product a book or a car?"

bloom filter

A probabilistic, memory-efficient data structure that is used to test whether an element is a member of a set.

blue/green deployment

A deployment strategy where you create two separate but identical environments. You run the current application version in one environment (blue) and the new application version in the other environment (green). This strategy helps you quickly roll back with minimal impact.

bot

A software application that runs automated tasks over the internet and simulates human activity or interaction. Some bots are useful or beneficial, such as web crawlers that index information on the internet. Some other bots, known as *bad bots*, are intended to disrupt or cause harm to individuals or organizations.

botnet

Networks of bots that are infected by malware and are under the control of a single party, known as a *bot herder* or *bot operator*. Botnets are the best-known mechanism to scale bots and their impact.

branch

A contained area of a code repository. The first branch created in a repository is the *main branch*. You can create a new branch from an existing branch, and you can then develop features or fix bugs in the new branch. A branch you create to build a feature is commonly referred to as a *feature branch*. When the feature is ready for release, you merge the feature branch back into the main branch. For more information, see About branches (GitHub documentation).

break-glass access

In exceptional circumstances and through an approved process, a quick means for a user to gain access to an AWS account that they don't typically have permissions to access. For more information, see the Implement break-glass procedures indicator in the AWS Well-Architected guidance.

brownfield strategy

The existing infrastructure in your environment. When adopting a brownfield strategy for a system architecture, you design the architecture around the constraints of the current systems and infrastructure. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

buffer cache

The memory area where the most frequently accessed data is stored.

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

business continuity planning (BCP)

A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

C

CAF

See AWS Cloud Adoption Framework.

canary deployment

The slow and incremental release of a version to end users. When you are confident, you deploy the new version and replace the current version in its entirety.

CCoE

See Cloud Center of Excellence.

CDC

See change data capture.

change data capture (CDC)

The process of tracking changes to a data source, such as a database table, and recording metadata about the change. You can use CDC for various purposes, such as auditing or replicating changes in a target system to maintain synchronization.

chaos engineering

Intentionally introducing failures or disruptive events to test a system's resilience. You can use AWS Fault Injection Service (AWS FIS) to perform experiments that stress your AWS workloads and evaluate their response.

CI/CD

See continuous integration and continuous delivery.

classification

A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

client-side encryption

Encryption of data locally, before the target AWS service receives it.

Cloud Center of Excellence (CCoE)

A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.

cloud computing

The cloud technology that is typically used for remote data storage and IoT device management. Cloud computing is commonly connected to edge computing technology.

cloud operating model

In an IT organization, the operating model that is used to build, mature, and optimize one or more cloud environments. For more information, see Building your Cloud Operating Model.

cloud stages of adoption

The four phases that organizations typically go through when they migrate to the AWS Cloud:

- Project – Running a few cloud-related projects for proof of concept and learning purposes
- Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)
- Migration – Migrating individual applications
- Re-invention – Optimizing products and services, and innovating in the cloud

These stages were defined by Stephen Orban in the blog post The Journey Toward Cloud-First & the Stages of Adoption on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the migration readiness guide.

CMDB

See configuration management database.

code repository

A location where source code and other assets, such as documentation, samples, and scripts, are stored and updated through version control processes. Common cloud repositories include GitHub or Bitbucket Cloud. Each version of the code is called a *branch*. In a microservice structure, each repository is devoted to a single piece of functionality. A single CI/CD pipeline can use multiple repositories.

cold cache

A buffer cache that is empty, not well populated, or contains stale or irrelevant data. This affects performance because the database instance must read from the main memory or disk, which is slower than reading from the buffer cache.

cold data

Data that is rarely accessed and is typically historical. When querying this kind of data, slow queries are typically acceptable. Moving this data to lower-performing and less expensive storage tiers or classes can reduce costs.

computer vision (CV)

A field of AI that uses machine learning to analyze and extract information from visual formats such as digital images and videos. For example, AWS Panorama offers devices that add CV to on-premises camera networks, and Amazon SageMaker AI provides image processing algorithms for CV.

configuration drift

For a workload, a configuration change from the expected state. It might cause the workload to become noncompliant, and it's typically gradual and unintentional.

configuration management database (CMDB)

A repository that stores and manages information about a database and its IT environment, including both hardware and software components and their configurations. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration.

conformance pack

A collection of AWS Config rules and remediation actions that you can assemble to customize your compliance and security checks. You can deploy a conformance pack as a single entity in an AWS account and Region, or across an organization, by using a YAML template. For more information, see Conformance packs in the AWS Config documentation.

continuous integration and continuous delivery (CI/CD)

The process of automating the source, build, test, staging, and production stages of the software release process. CI/CD is commonly described as a pipeline. CI/CD can help you automate processes, improve productivity, improve code quality, and deliver faster. For more information, see Benefits of continuous delivery. CD can also stand for *continuous deployment*. For more information, see Continuous Delivery vs. Continuous Deployment.

CV

See computer vision.

D

data at rest

Data that is stationary in your network, such as data that is in storage.

data classification

A process for identifying and categorizing the data in your network based on its criticality and sensitivity. It is a critical component of any cybersecurity risk management strategy because it helps you determine the appropriate protection and retention controls for the data. Data classification is a component of the security pillar in the AWS Well-Architected Framework. For more information, see Data classification.

data drift

A meaningful variation between the production data and the data that was used to train an ML model, or a meaningful change in the input data over time. Data drift can reduce the overall quality, accuracy, and fairness in ML model predictions.

data in transit

Data that is actively moving through your network, such as between network resources.

data mesh

An architectural framework that provides distributed, decentralized data ownership with centralized management and governance.

data minimization

The principle of collecting and processing only the data that is strictly necessary. Practicing data minimization in the AWS Cloud can reduce privacy risks, costs, and your analytics carbon footprint.

data perimeter

A set of preventive guardrails in your AWS environment that help make sure that only trusted identities are accessing trusted resources from expected networks. For more information, see Building a data perimeter on AWS.

data preprocessing

To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

data provenance

The process of tracking the origin and history of data throughout its lifecycle, such as how the data was generated, transmitted, and stored.

data subject

An individual whose data is being collected and processed.

data warehouse

A data management system that supports business intelligence, such as analytics. Data warehouses commonly contain large amounts of historical data, and they are typically used for queries and analysis.

database definition language (DDL)

Statements or commands for creating or modifying the structure of tables and objects in a database.

database manipulation language (DML)

Statements or commands for modifying (inserting, updating, and deleting) information in a database.

DDL

See database definition language.

deep ensemble

To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

deep learning

An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

defense-in-depth

An information security approach in which a series of security mechanisms and controls are thoughtfully layered throughout a computer network to protect the confidentiality, integrity, and availability of the network and the data within. When you adopt this strategy on AWS, you add multiple controls at different layers of the AWS Organizations structure to help secure resources. For example, a defense-in-depth approach might combine multi-factor authentication, network segmentation, and encryption.

delegated administrator

In AWS Organizations, a compatible service can register an AWS member account to administer the organization's accounts and manage permissions for that service. This account is called the *delegated administrator* for that service. For more information and a list of compatible services, see Services that work with AWS Organizations in the AWS Organizations documentation.

deployment

The process of making an application, new features, or code fixes available in the target environment. Deployment involves implementing changes in a code base and then building and running that code base in the application's environments.

development environment

See environment.

detective control

A security control that is designed to detect, log, and alert after an event has occurred. These controls are a second line of defense, alerting you to security events that bypassed the preventative controls in place. For more information, see Detective controls in *Implementing security controls on AWS*.

development value stream mapping (DVSM)

A process used to identify and prioritize constraints that adversely affect speed and quality in a software development lifecycle. DVSM extends the value stream mapping process originally designed for lean manufacturing practices. It focuses on the steps and teams required to create and move value through the software development process.

digital twin

A virtual representation of a real-world system, such as a building, factory, industrial equipment, or production line. Digital twins support predictive maintenance, remote monitoring, and production optimization.

dimension table

In a star schema, a smaller table that contains data attributes about quantitative data in a fact table. Dimension table attributes are typically text fields or discrete numbers that behave like text. These attributes are commonly used for query constraining, filtering, and result set labeling.

disaster

An event that prevents a workload or system from fulfilling its business objectives in its primary deployed location. These events can be natural disasters, technical failures, or the result of human actions, such as unintentional misconfiguration or a malware attack.

disaster recovery (DR)

The strategy and process you use to minimize downtime and data loss caused by a disaster. For more information, see Disaster Recovery of Workloads on AWS: Recovery in the Cloud in the AWS Well-Architected Framework.

DML

See database manipulation language.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

DR

See disaster recovery.

drift detection

Tracking deviations from a baselined configuration. For example, you can use AWS CloudFormation to detect drift in system resources, or you can use AWS Control Tower to detect changes in your landing zone that might affect compliance with governance requirements.

DVSM

See development value stream mapping.

E

EDA

See exploratory data analysis.

EDI

See electronic data interchange.

edge computing

The technology that increases the computing power for smart devices at the edges of an IoT network. When compared with cloud computing, edge computing can reduce communication latency and improve response time.

electronic data interchange (EDI)

The automated exchange of business documents between organizations. For more information, see What is Electronic Data Interchange.

encryption

A computing process that transforms plaintext data, which is human-readable, into ciphertext.

encryption key

A cryptographic string of randomized bits that is generated by an encryption algorithm. Keys can vary in length, and each key is designed to be unpredictable and unique.

endianness

The order in which bytes are stored in computer memory. Big-endian systems store the most significant byte first. Little-endian systems store the least significant byte first.

endpoint

See service endpoint.

endpoint service

A service that you can host in a virtual private cloud (VPC) to share with other users. You can create an endpoint service with AWS PrivateLink and grant permissions to other AWS accounts or to AWS Identity and Access Management (IAM) principals. These accounts or principals can connect to your endpoint service privately by creating interface VPC endpoints. For more information, see Create an endpoint service in the Amazon Virtual Private Cloud (Amazon VPC) documentation.

enterprise resource planning (ERP)

A system that automates and manages key business processes (such as accounting, MES, and project management) for an enterprise.

envelope encryption

The process of encrypting an encryption key with another encryption key. For more information, see Envelope encryption in the AWS Key Management Service (AWS KMS) documentation.

environment

An instance of a running application. The following are common types of environments in cloud computing:

- development environment – An instance of a running application that is available only to the core team responsible for maintaining the application. Development environments are used to test changes before promoting them to upper environments. This type of environment is sometimes referred to as a *test environment*.
- lower environments – All development environments for an application, such as those used for initial builds and tests.
- production environment – An instance of a running application that end users can access. In a CI/CD pipeline, the production environment is the last deployment environment.
- upper environments – All environments that can be accessed by users other than the core development team. This can include a production environment, preproduction environments, and environments for user acceptance testing.

epic

In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the program implementation guide.

ERP

See enterprise resource planning.

exploratory data analysis (EDA)

The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

F

fact table

The central table in a star schema. It stores quantitative data about business operations. Typically, a fact table contains two types of columns: those that contain measures and those that contain a foreign key to a dimension table.

fail fast

A philosophy that uses frequent and incremental testing to reduce the development lifecycle. It is a critical part of an agile approach.

fault isolation boundary

In the AWS Cloud, a boundary such as an Availability Zone, AWS Region, control plane, or data plane that limits the effect of a failure and helps improve the resilience of workloads. For more information, see AWS Fault Isolation Boundaries.

feature branch

See branch.

features

The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

feature importance

How significant a feature is for a model's predictions. This is usually expressed as a numerical score that can be calculated through various techniques, such as Shapley Additive Explanations (SHAP) and integrated gradients. For more information, see Machine learning model interpretability with AWS.

feature transformation

To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the "2021-05-27 00:15:37" date into "2021", "May", "Thu", and "15", you can help the learning algorithm learn nuanced patterns associated with different data components.

few-shot prompting

Providing an LLM with a small number of examples that demonstrate the task and desired output before asking it to perform a similar task. This technique is an application of in-context learning, where models learn from examples (*shots*) that are embedded in prompts. Few-shot prompting can be effective for tasks that require specific formatting, reasoning, or domain knowledge. See also zero-shot prompting.

FGAC

See fine-grained access control.

fine-grained access control (FGAC)

The use of multiple conditions to allow or deny an access request.

flash-cut migration

A database migration method that uses continuous data replication through change data capture to migrate data in the shortest time possible, instead of using a phased approach. The objective is to keep downtime to a minimum.

FM

See foundation model.

foundation model (FM)

A large deep-learning neural network that has been training on massive datasets of generalized and unlabeled data. FMs are capable of performing a wide variety of general tasks, such as understanding language, generating text and images, and conversing in natural language. For more information, see What are Foundation Models.

G

generative AI

A subset of AI models that have been trained on large amounts of data and that can use a simple text prompt to create new content and artifacts, such as images, videos, text, and audio. For more information, see What is Generative AI.

geo blocking

See geographic restrictions.

geographic restrictions (geo blocking)

In Amazon CloudFront, an option to prevent users in specific countries from accessing content distributions. You can use an allow list or block list to specify approved and banned countries. For more information, see Restricting the geographic distribution of your content in the CloudFront documentation.

Gitflow workflow

An approach in which lower and upper environments use different branches in a source code repository. The Gitflow workflow is considered legacy, and the trunk-based workflow is the modern, preferred approach.

golden image

A snapshot of a system or software that is used as a template to deploy new instances of that system or software. For example, in manufacturing, a golden image can be used to provision software on multiple devices and helps improve speed, scalability, and productivity in device manufacturing operations.

greenfield strategy

The absence of existing infrastructure in a new environment. When adopting a greenfield strategy for a system architecture, you can select all new technologies without the restriction

of compatibility with existing infrastructure, also known as brownfield. If you are expanding the existing infrastructure, you might blend brownfield and greenfield strategies.

guardrail

A high-level rule that helps govern resources, policies, and compliance across organizational units (OUs). *Preventive guardrails* enforce policies to ensure alignment to compliance standards. They are implemented by using service control policies and IAM permissions boundaries. *Detective guardrails* detect policy violations and compliance issues, and generate alerts for remediation. They are implemented by using AWS Config, AWS Security Hub, Amazon GuardDuty, AWS Trusted Advisor, Amazon Inspector, and custom AWS Lambda checks.

H

HA

See high availability.

heterogeneous database migration

Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides AWS SCT that helps with schema conversions.

high availability (HA)

The ability of a workload to operate continuously, without intervention, in the event of challenges or disasters. HA systems are designed to automatically fail over, consistently deliver high-quality performance, and handle different loads and failures with minimal performance impact.

historian modernization

An approach used to modernize and upgrade operational technology (OT) systems to better serve the needs of the manufacturing industry. A *historian* is a type of database that is used to collect and store data from various sources in a factory.

holdout data

A portion of historical, labeled data that is withheld from a dataset that is used to train a machine learning model. You can use holdout data to evaluate the model performance by comparing the model predictions against the holdout data.

homogeneous database migration

Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema.

hot data

Data that is frequently accessed, such as real-time data or recent translational data. This data typically requires a high-performance storage tier or class to provide fast query responses.

hotfix

An urgent fix for a critical issue in a production environment. Due to its urgency, a hotfix is usually made outside of the typical DevOps release workflow.

hypercare period

Immediately following cutover, the period of time when a migration team manages and monitors the migrated applications in the cloud in order to address any issues. Typically, this period is 1–4 days in length. At the end of the hypercare period, the migration team typically transfers responsibility for the applications to the cloud operations team.

I

IaC

See infrastructure as code.

identity-based policy

A policy attached to one or more IAM principals that defines their permissions within the AWS Cloud environment.

idle application

An application that has an average CPU and memory usage between 5 and 20 percent over a period of 90 days. In a migration project, it is common to retire these applications or retain them on premises.

IIoT

See industrial Internet of Things.

immutable infrastructure

A model that deploys new infrastructure for production workloads instead of updating, patching, or modifying the existing infrastructure. Immutable infrastructures are inherently more consistent, reliable, and predictable than mutable infrastructure. For more information, see the Deploy using immutable infrastructure best practice in the AWS Well-Architected Framework.

inbound (ingress) VPC

In an AWS multi-account architecture, a VPC that accepts, inspects, and routes network connections from outside an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

incremental migration

A cutover strategy in which you migrate your application in small parts instead of performing a single, full cutover. For example, you might move only a few microservices or users to the new system initially. After you verify that everything is working properly, you can incrementally move additional microservices or users until you can decommission your legacy system. This strategy reduces the risks associated with large migrations.

Industry 4.0

A term that was introduced by Klaus Schwab in 2016 to refer to the modernization of manufacturing processes through advances in connectivity, real-time data, automation, analytics, and AI/ML.

infrastructure

All of the resources and assets contained within an application's environment.

infrastructure as code (IaC)

The process of provisioning and managing an application's infrastructure through a set of configuration files. IaC is designed to help you centralize infrastructure management, standardize resources, and scale quickly so that new environments are repeatable, reliable, and consistent.

industrial Internet of Things (IIoT)

The use of internet-connected sensors and devices in the industrial sectors, such as manufacturing, energy, automotive, healthcare, life sciences, and agriculture. For more information, see Building an industrial Internet of Things (IIoT) digital transformation strategy.

inspection VPC

In an AWS multi-account architecture, a centralized VPC that manages inspections of network traffic between VPCs (in the same or different AWS Regions), the internet, and on-premises networks. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

Internet of Things (IoT)

The network of connected physical objects with embedded sensors or processors that communicate with other devices and systems through the internet or over a local communication network. For more information, see What is IoT?

interpretability

A characteristic of a machine learning model that describes the degree to which a human can understand how the model's predictions depend on its inputs. For more information, see Machine learning model interpretability with AWS.

IoT

See Internet of Things.

IT information library (ITIL)

A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM.

IT service management (ITSM)

Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the operations integration guide.

ITIL

See IT information library.

ITSM

See IT service management.

L

label-based access control (LBAC)

An implementation of mandatory access control (MAC) where the users and the data itself are each explicitly assigned a security label value. The intersection between the user security label and data security label determines which rows and columns can be seen by the user.

landing zone

A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

large language model (LLM)

A deep learning AI model that is pretrained on a vast amount of data. An LLM can perform multiple tasks, such as answering questions, summarizing documents, translating text into other languages, and completing sentences. For more information, see What are LLMs.

large migration

A migration of 300 or more servers.

LBAC

See label-based access control.

least privilege

The security best practice of granting the minimum permissions required to perform a task. For more information, see Apply least-privilege permissions in the IAM documentation.

lift and shift

See 7 Rs.

little-endian system

A system that stores the least significant byte first. See also endianness.

LLM

See large language model.

lower environments

See environment.

M

machine learning (ML)

A type of artificial intelligence that uses algorithms and techniques for pattern recognition and learning. ML analyzes and learns from recorded data, such as Internet of Things (IoT) data, to generate a statistical model based on patterns. For more information, see Machine Learning.

main branch

See branch.

malware

Software that is designed to compromise computer security or privacy. Malware might disrupt computer systems, leak sensitive information, or gain unauthorized access. Examples of malware include viruses, worms, ransomware, Trojan horses, spyware, and keyloggers.

managed services

AWS services for which AWS operates the infrastructure layer, the operating system, and platforms, and you access the endpoints to store and retrieve data. Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB are examples of managed services. These are also known as *abstracted services*.

manufacturing execution system (MES)

A software system for tracking, monitoring, documenting, and controlling production processes that convert raw materials to finished products on the shop floor.

MAP

See Migration Acceleration Program.

mechanism

A complete process in which you create a tool, drive adoption of the tool, and then inspect the results in order to make adjustments. A mechanism is a cycle that reinforces and improves itself as it operates. For more information, see Building mechanisms in the AWS Well-Architected Framework.

member account

All AWS accounts other than the management account that are part of an organization in AWS Organizations. An account can be a member of only one organization at a time.

MES

See manufacturing execution system.

Message Queuing Telemetry Transport (MQTT)

A lightweight, machine-to-machine (M2M) communication protocol, based on the publish/subscribe pattern, for resource-constrained IoT devices.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed,

and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

Migration Acceleration Program (MAP)

An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios.

migration at scale

The process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a *migration factory* of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

migration factory

Cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the Cloud Migration Factory guide in this content set.

migration metadata

The information about the application and server that is needed to complete the migration. Each migration pattern requires a different set of migration metadata. Examples of migration metadata include the target subnet, security group, and AWS account.

migration pattern

A repeatable migration task that details the migration strategy, the migration destination, and the migration application or service used. Example: Rehost migration to Amazon EC2 with AWS Application Migration Service.

Migration Portfolio Assessment (MPA)

An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO

comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The MPA tool (requires login) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA)

The process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

migration strategy

The approach used to migrate a workload to the AWS Cloud. For more information, see the 7 Rs entry in this glossary and see Mobilize your organization to accelerate large-scale migrations.

ML

See machine learning.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can

use a microservices architecture. For more information, see Decomposing monoliths into microservices.

MPA

See Migration Portfolio Assessment.

MQTT

See Message Queuing Telemetry Transport.

multiclass classification

A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

mutable infrastructure

A model that updates and modifies the existing infrastructure for production workloads. For improved consistency, reliability, and predictability, the AWS Well-Architected Framework recommends the use of immutable infrastructure as a best practice.

O

OAC

See origin access control.

OAI

See origin access identity.

OCM

See organizational change management.

offline migration

A migration method in which the source workload is taken down during the migration process. This method involves extended downtime and is typically used for small, non-critical workloads.

OI

See operations integration.

OLA

See operational-level agreement.

online migration

A migration method in which the source workload is copied to the target system without being taken offline. Applications that are connected to the workload can continue to function during the migration. This method involves zero to minimal downtime and is typically used for critical production workloads.

OPC-UA

See Open Process Communications - Unified Architecture.

Open Process Communications - Unified Architecture (OPC-UA)

A machine-to-machine (M2M) communication protocol for industrial automation. OPC-UA provides an interoperability standard with data encryption, authentication, and authorization schemes.

operational-level agreement (OLA)

An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

operational readiness review (ORR)

A checklist of questions and associated best practices that help you understand, evaluate, prevent, or reduce the scope of incidents and possible failures. For more information, see Operational Readiness Reviews (ORR) in the AWS Well-Architected Framework.

operational technology (OT)

Hardware and software systems that work with the physical environment to control industrial operations, equipment, and infrastructure. In manufacturing, the integration of OT and information technology (IT) systems is a key focus for Industry 4.0 transformations.

operations integration (OI)

The process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

organization trail

A trail that's created by AWS CloudTrail that logs all events for all AWS accounts in an organization in AWS Organizations. This trail is created in each AWS account that's part of the

organization and tracks the activity in each account. For more information, see Creating a trail for an organization in the CloudTrail documentation.

organizational change management (OCM)

A framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called *people acceleration*, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

origin access control (OAC)

In CloudFront, an enhanced option for restricting access to secure your Amazon Simple Storage Service (Amazon S3) content. OAC supports all S3 buckets in all AWS Regions, server-side encryption with AWS KMS (SSE-KMS), and dynamic PUT and DELETE requests to the S3 bucket.

origin access identity (OAI)

In CloudFront, an option for restricting access to secure your Amazon S3 content. When you use OAI, CloudFront creates a principal that Amazon S3 can authenticate with. Authenticated principals can access content in an S3 bucket only through a specific CloudFront distribution. See also OAC, which provides more granular and enhanced access control.

ORR

See operational readiness review.

OT

See operational technology.

outbound (egress) VPC

In an AWS multi-account architecture, a VPC that handles network connections that are initiated from within an application. The AWS Security Reference Architecture recommends setting up your Network account with inbound, outbound, and inspection VPCs to protect the two-way interface between your application and the broader internet.

P

permissions boundary

An IAM management policy that is attached to IAM principals to set the maximum permissions that the user or role can have. For more information, see Permissions boundaries in the IAM documentation.

personally identifiable information (PII)

Information that, when viewed directly or paired with other related data, can be used to reasonably infer the identity of an individual. Examples of PII include names, addresses, and contact information.

PII

See personally identifiable information.

playbook

A set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

PLC

See programmable logic controller.

PLM

See product lifecycle management.

policy

An object that can define permissions (see identity-based policy), specify access conditions (see resource-based policy), or define the maximum permissions for all accounts in an organization in AWS Organizations (see service control policy).

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store

best adapted to their requirements. For more information, see Enabling data persistence in microservices.

portfolio assessment

A process of discovering, analyzing, and prioritizing the application portfolio in order to plan the migration. For more information, see Evaluating migration readiness.

predicate

A query condition that returns `true` or `false`, commonly located in a `WHERE` clause.

predicate pushdown

A database query optimization technique that filters the data in the query before transfer. This reduces the amount of data that must be retrieved and processed from the relational database, and it improves query performance.

preventative control

A security control that is designed to prevent an event from occurring. These controls are a first line of defense to help prevent unauthorized access or unwanted changes to your network. For more information, see Preventative controls in *Implementing security controls on AWS*.

principal

An entity in AWS that can perform actions and access resources. This entity is typically a root user for an AWS account, an IAM role, or a user. For more information, see *Principal* in Roles terms and concepts in the IAM documentation.

privacy by design

A system engineering approach that takes privacy into account through the whole development process.

private hosted zones

A container that holds information about how you want Amazon Route 53 to respond to DNS queries for a domain and its subdomains within one or more VPCs. For more information, see Working with private hosted zones in the Route 53 documentation.

proactive control

A security control designed to prevent the deployment of noncompliant resources. These controls scan resources before they are provisioned. If the resource is not compliant with the control, then it isn't provisioned. For more information, see the Controls reference guide in the

AWS Control Tower documentation and see Proactive controls in *Implementing security controls on AWS*.

product lifecycle management (PLM)

The management of data and processes for a product throughout its entire lifecycle, from design, development, and launch, through growth and maturity, to decline and removal.

production environment

See environment.

programmable logic controller (PLC)

In manufacturing, a highly reliable, adaptable computer that monitors machines and automates manufacturing processes.

prompt chaining

Using the output of one LLM prompt as the input for the next prompt to generate better responses. This technique is used to break down a complex task into subtasks, or to iteratively refine or expand a preliminary response. It helps improve the accuracy and relevance of a model's responses and allows for more granular, personalized results.

pseudonymization

The process of replacing personal identifiers in a dataset with placeholder values. Pseudonymization can help protect personal privacy. Pseudonymized data is still considered to be personal data.

publish/subscribe (pub/sub)

A pattern that enables asynchronous communications among microservices to improve scalability and responsiveness. For example, in a microservices-based MES, a microservice can publish event messages to a channel that other microservices can subscribe to. The system can add new microservices without changing the publishing service.

Q

query plan

A series of steps, like instructions, that are used to access the data in a SQL relational database system.

query plan regression

When a database service optimizer chooses a less optimal plan than it did before a given change to the database environment. This can be caused by changes to statistics, constraints, environment settings, query parameter bindings, and updates to the database engine.

R

RACI matrix

See responsible, accountable, consulted, informed (RACI).

RAG

See Retrieval Augmented Generation.

ransomware

A malicious software that is designed to block access to a computer system or data until a payment is made.

RASCI matrix

See responsible, accountable, consulted, informed (RACI).

RCAC

See row and column access control.

read replica

A copy of a database that's used for read-only purposes. You can route queries to the read replica to reduce the load on your primary database.

re-architect

See 7 Rs.

recovery point objective (RPO)

The maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

recovery time objective (RTO)

The maximum acceptable delay between the interruption of service and restoration of service.

refactor

See 7 Rs.

Region

A collection of AWS resources in a geographic area. Each AWS Region is isolated and independent of the others to provide fault tolerance, stability, and resilience. For more information, see Specify which AWS Regions your account can use.

regression

An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

rehost

See 7 Rs.

release

In a deployment process, the act of promoting changes to a production environment.

relocate

See 7 Rs.

replatform

See 7 Rs.

repurchase

See 7 Rs.

resiliency

An application's ability to resist or recover from disruptions. High availability and disaster recovery are common considerations when planning for resiliency in the AWS Cloud. For more information, see AWS Cloud Resilience.

resource-based policy

A policy attached to a resource, such as an Amazon S3 bucket, an endpoint, or an encryption key. This type of policy specifies which principals are allowed access, supported actions, and any other conditions that must be met.

responsible, accountable, consulted, informed (RACI) matrix

A matrix that defines the roles and responsibilities for all parties involved in migration activities and cloud operations. The matrix name is derived from the responsibility types defined in the matrix: responsible (R), accountable (A), consulted (C), and informed (I). The support (S) type is optional. If you include support, the matrix is called a *RASCI matrix*, and if you exclude it, it's called a *RACI matrix*.

responsive control

A security control that is designed to drive remediation of adverse events or deviations from your security baseline. For more information, see Responsive controls in *Implementing security controls on AWS*.

retain

See 7 Rs.

retire

See 7 Rs.

Retrieval Augmented Generation (RAG)

A generative AI technology in which an LLM references an authoritative data source that is outside of its training data sources before generating a response. For example, a RAG model might perform a semantic search of an organization's knowledge base or custom data. For more information, see What is RAG.

rotation

The process of periodically updating a secret to make it more difficult for an attacker to access the credentials.

row and column access control (RCAC)

The use of basic, flexible SQL expressions that have defined access rules. RCAC consists of row permissions and column masks.

RPO

See recovery point objective.

RTO

See recovery time objective.

runbook

A set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

S

SAML 2.0

An open standard that many identity providers (IdPs) use. This feature enables federated single sign-on (SSO), so users can log into the AWS Management Console or call the AWS API operations without you having to create user in IAM for everyone in your organization. For more information about SAML 2.0-based federation, see About SAML 2.0-based federation in the IAM documentation.

SCADA

See supervisory control and data acquisition.

SCP

See service control policy.

secret

In AWS Secrets Manager, confidential or restricted information, such as a password or user credentials, that you store in encrypted form. It consists of the secret value and its metadata. The secret value can be binary, a single string, or multiple strings. For more information, see What's in a Secrets Manager secret? in the Secrets Manager documentation.

security by design

A system engineering approach that takes security into account through the whole development process.

security control

A technical or administrative guardrail that prevents, detects, or reduces the ability of a threat actor to exploit a security vulnerability. There are four primary types of security controls: preventative, detective, responsive, and proactive.

security hardening

The process of reducing the attack surface to make it more resistant to attacks. This can include actions such as removing resources that are no longer needed, implementing the security best practice of granting least privilege, or deactivating unnecessary features in configuration files.

security information and event management (SIEM) system

Tools and services that combine security information management (SIM) and security event management (SEM) systems. A SIEM system collects, monitors, and analyzes data from servers, networks, devices, and other sources to detect threats and security breaches, and to generate alerts.

security response automation

A predefined and programmed action that is designed to automatically respond to or remediate a security event. These automations serve as detective or responsive security controls that help you implement AWS security best practices. Examples of automated response actions include modifying a VPC security group, patching an Amazon EC2 instance, or rotating credentials.

server-side encryption

Encryption of data at its destination, by the AWS service that receives it.

service control policy (SCP)

A policy that provides centralized control over permissions for all accounts in an organization in AWS Organizations. SCPs define guardrails or set limits on actions that an administrator can delegate to users or roles. You can use SCPs as allow lists or deny lists, to specify which services or actions are permitted or prohibited. For more information, see Service control policies in the AWS Organizations documentation.

service endpoint

The URL of the entry point for an AWS service. You can use the endpoint to connect programmatically to the target service. For more information, see AWS service endpoints in *AWS General Reference*.

service-level agreement (SLA)

An agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

service-level indicator (SLI)

A measurement of a performance aspect of a service, such as its error rate, availability, or throughput.

service-level objective (SLO)

A target metric that represents the health of a service, as measured by a service-level indicator.

shared responsibility model

A model describing the responsibility you share with AWS for cloud security and compliance. AWS is responsible for security *of* the cloud, whereas you are responsible for security *in* the cloud. For more information, see Shared responsibility model.

SIEM

See security information and event management system.

single point of failure (SPOF)

A failure in a single, critical component of an application that can disrupt the system.

SLA

See service-level agreement.

SLI

See service-level indicator.

SLO

See service-level objective.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid

innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

SPOF

See single point of failure.

star schema

A database organizational structure that uses one large fact table to store transactional or measured data and uses one or more smaller dimensional tables to store data attributes. This structure is designed for use in a data warehouse or for business intelligence purposes.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

subnet

A range of IP addresses in your VPC. A subnet must reside in a single Availability Zone.

supervisory control and data acquisition (SCADA)

In manufacturing, a system that uses hardware and software to monitor physical assets and production operations.

symmetric encryption

An encryption algorithm that uses the same key to encrypt and decrypt the data.

synthetic testing

Testing a system in a way that simulates user interactions to detect potential issues or to monitor performance. You can use Amazon CloudWatch Synthetics to create these tests.

system prompt

A technique for providing context, instructions, or guidelines to an LLM to direct its behavior. System prompts help set context and establish rules for interactions with users.

T

tags

Key-value pairs that act as metadata for organizing your AWS resources. Tags can help you manage, identify, organize, search for, and filter resources. For more information, see Tagging your AWS resources.

target variable

The value that you are trying to predict in supervised ML. This is also referred to as an *outcome variable*. For example, in a manufacturing setting the target variable could be a product defect.

task list

A tool that is used to track progress through a runbook. A task list contains an overview of the runbook and a list of general tasks to be completed. For each general task, it includes the estimated amount of time required, the owner, and the progress.

test environment

See environment.

training

To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

transit gateway

A network transit hub that you can use to interconnect your VPCs and on-premises networks. For more information, see What is a transit gateway in the AWS Transit Gateway documentation.

trunk-based workflow

An approach in which developers build and test features locally in a feature branch and then merge those changes into the main branch. The main branch is then built to the development, preproduction, and production environments, sequentially.

trusted access

Granting permissions to a service that you specify to perform tasks in your organization in AWS Organizations and in its accounts on your behalf. The trusted service creates a service-linked role in each account, when that role is needed, to perform management tasks for you. For more information, see Using AWS Organizations with other AWS services in the AWS Organizations documentation.

tuning

To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.

U

uncertainty

A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: *Epistemic uncertainty* is caused by limited, incomplete data, whereas *aleatoric uncertainty* is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

undifferentiated tasks

Also known as *heavy lifting*, work that is necessary to create and operate an application but that doesn't provide direct value to the end user or provide competitive advantage. Examples of undifferentiated tasks include procurement, maintenance, and capacity planning.

upper environments

See environment.

V

vacuuming

A database maintenance operation that involves cleaning up after incremental updates to reclaim storage and improve performance.

version control

Processes and tools that track changes, such as changes to source code in a repository.

VPC peering

A connection between two VPCs that allows you to route traffic by using private IP addresses. For more information, see What is VPC peering in the Amazon VPC documentation.

vulnerability

A software or hardware flaw that compromises the security of the system.

W

warm cache

A buffer cache that contains current, relevant data that is frequently accessed. The database instance can read from the buffer cache, which is faster than reading from the main memory or disk.

warm data

Data that is infrequently accessed. When querying this kind of data, moderately slow queries are typically acceptable.

window function

A SQL function that performs a calculation on a group of rows that relate in some way to the current record. Window functions are useful for processing tasks, such as calculating a moving average or accessing the value of rows based on the relative position of the current row.

workload

A collection of resources and code that delivers business value, such as a customer-facing application or backend process.

workstream

Functional groups in a migration project that are responsible for a specific set of tasks. Each workstream is independent but supports the other workstreams in the project. For example, the portfolio workstream is responsible for prioritizing applications, wave planning, and collecting migration metadata. The portfolio workstream delivers these assets to the migration workstream, which then migrates the servers and applications.

WORM

See write once, read many.

WQF

See AWS Workload Qualification Framework.

write once, read many (WORM)

A storage model that writes data a single time and prevents the data from being deleted or modified. Authorized users can read the data as many times as needed, but they cannot change it. This data storage infrastructure is considered immutable.

Z

zero-day exploit

An attack, typically malware, that takes advantage of a zero-day vulnerability.

zero-day vulnerability

An unmitigated flaw or vulnerability in a production system. Threat actors can use this type of vulnerability to attack the system. Developers frequently become aware of the vulnerability as a result of the attack.

zero-shot prompting

Providing an LLM with instructions for performing a task but no examples (*shots*) that can help guide it. The LLM must use its pre-trained knowledge to handle the task. The effectiveness of zero-shot prompting depends on the complexity of the task and the quality of the prompt. See also few-shot prompting.

zombie application

An application that has an average CPU and memory usage below 5 percent. In a migration project, it is common to retire these applications.