AWS Prescriptive Guidance

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

Table of Contents

This disclaimer applies to the APG Library and all other documents, information, data, and responses (written or verbal) provided by Amazon Web Services, Inc. ("AWS") in connection with the APG Library and your use ("You") of the APG Library services. All use of AWS`s Service Offerings will be governed by the AWS Customer Agreement available at http://aws.amazon.com/agreement/(or other definitive written agreement as may be agreed between the parties governing the use of AWS's Service Offerings) (as applicable, the "Agreement"). Pricing information (if any) provided in this APG Library is subject to change in accordance with the Agreement. Nothing in the APG Library will modify or supplement the terms of the Agreement or any Non-Disclosure Agreement ("NDA") between AWS and You. No part of the APG Library may be disclosed without AWS's prior written consent.

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

Created by Sandeep Gawande (AWS), Mason Cahill (AWS), Sandip Gangapadhyay (AWS), Siamak Heshmati (AWS) and RAJNEESH TYAGI (AWS)

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.

Targeted business outcomes

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.
- **Increased speed** – Take advantage of familiar features in the AWS CDK to accelerate your development process. These features include objects, loops, and conditions.

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

This disclaimer applies to the APG Library and all other documents, information, data, and responses (written or verbal) provided by Amazon Web Services, Inc. ("AWS") in connection with the APG Library and your use ("You") of the APG Library services. All use of AWS`s Service Offerings will be governed by the AWS Customer Agreement available at http://aws.amazon.com/agreement/(or other definitive written agreement as may be agreed between the parties governing the use of AWS's Service Offerings) (as applicable, the "Agreement"). Pricing information (if any) provided in this APG Library is subject to change in accordance with the Agreement. Nothing in the APG Library will modify or supplement the terms of the Agreement or any Non-Disclosure Agreement ("NDA") between AWS and You. No part of the APG Library may be disclosed without AWS's prior written consent.

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.

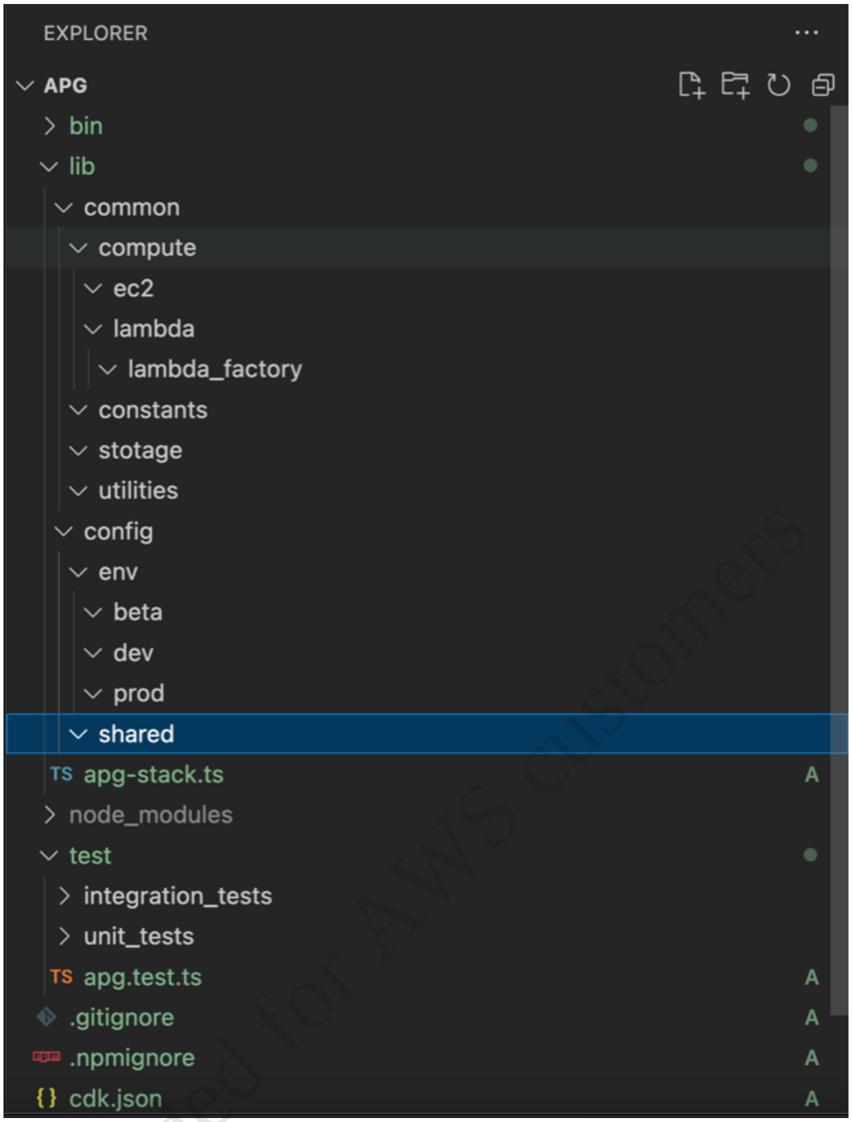
How to organize your code for scale

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.



```
EXPLORER                          ...
v APG
  > bin
  v lib
    v common
      v compute
        v ec2
        v lambda
          v lambda_factory
      v constants
      v stotage
      v utilities
    v config
      v env
        v beta
        v dev
        v prod
      v shared
    TS apg-stack.ts                 A
  > node_modules
  v test
    > integration_tests
    > unit_tests
    TS apg.test.ts                  A
  .gitignore                        A
  .npmignore                        A
  {} cdk.json                       A
```

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

This disclaimer applies to the APG Library and all other documents, information, data, and responses (either written or verbal) provided by Amazon Web Services, Inc. ("AWS") in connection with the APG Library and your use ("You") of the APG Library services. All use of AWS`s Service Offerings will be governed by the AWS Customer Agreement available at http://aws.amazon.com/agreement/(or other definitive written agreement as may be agreed between the parties governing the use of AWS's Service Offerings) (as applicable, the "Agreement"). Pricing information (if any) provided in this APG Library is subject to change in accordance with the Agreement. Nothing in the APG Library will modify or supplement the terms of the Agreement or any Non-Disclosure Agreement ("NDA") between AWS and You. No part of the APG Library may be disclosed without AWS's prior written consent.

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 the 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: 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: "my-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. This 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: "my-bucket",
    region: "us-east-1",
    encryption: false,
});

const myS3Bucket2 = new S3Bucket({
    name: "my-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 has global scope, which means that it's available and can be accessed only within that function. You can re-declare 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.

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 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.

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`.

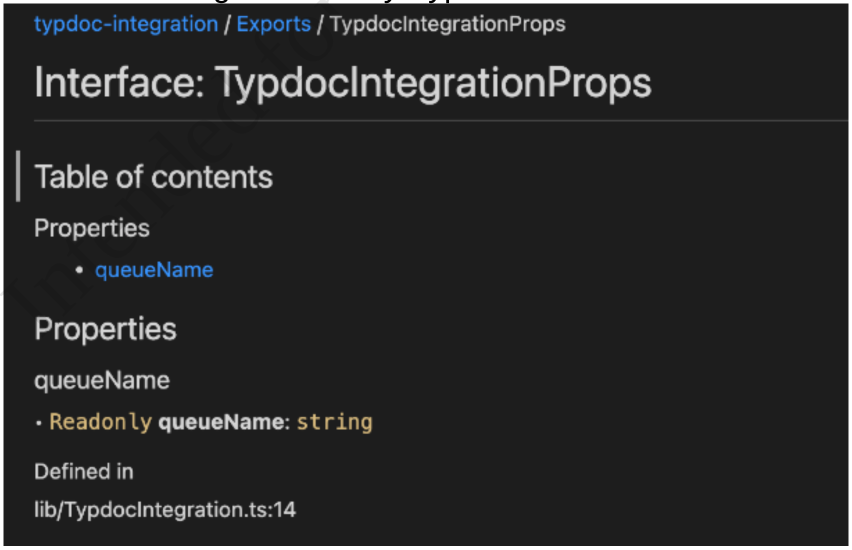
```
{


  "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.



typdoc-integration / Exports / TypdocIntegrationProps

Interface: TypdocIntegrationProps

Table of contents

Properties

- queueName

Properties

queueName

• `Readonly` **queueName**: `string`

Defined in

lib/TypdocIntegration.ts:14

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.

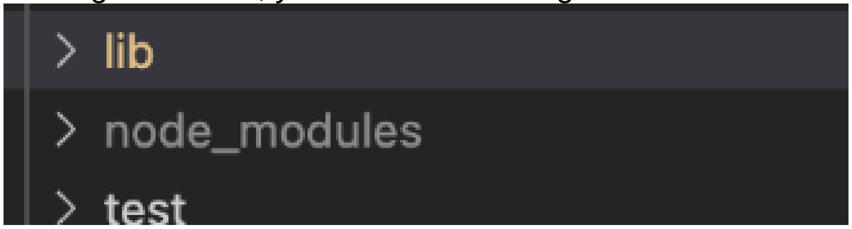
There are two categories of tests that you can write for your AWS CDK applications:

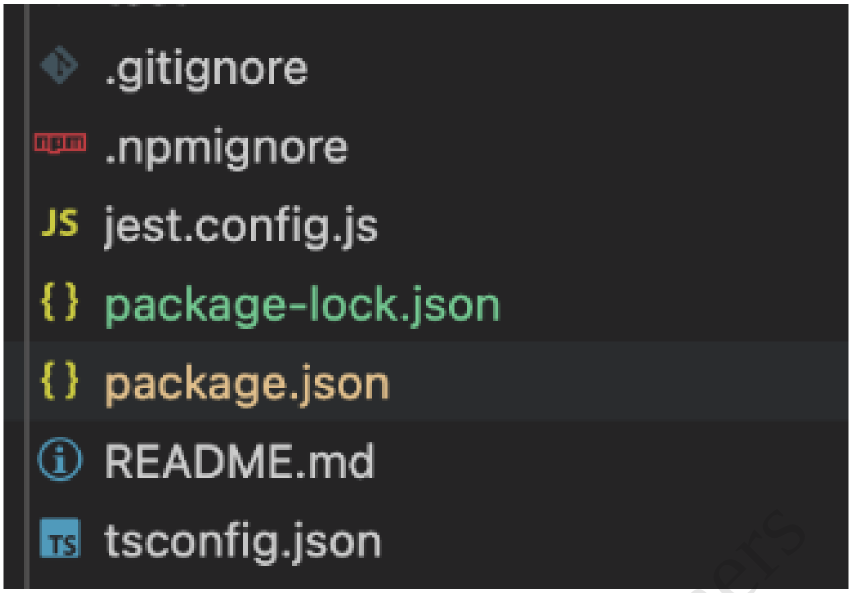
- Use **fine-grained assertions** to test a specific aspect of the generated CloudFormation template, such as "this resource has this property with this value." These tests can detect regressions and are also useful when you're developing new features using TDD (write a test first, then make it pass 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 integration

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 {App,Stack} from 'aws-cdk-lib';
import { Template } from 'aws-cdk-lib/assertions';
import * as CodepipelineModule from '../lib/index';
import { Role, ServicePrincipal } from 'aws-cdk-lib/aws-iam';
import { Repository } from 'aws-cdk-lib/aws-codecommit';
import { PipelineProject } from 'aws-cdk-lib/aws-codebuild';

const testData:CodepipelineModule.CodepipelineModuleProps = {

    pipelineName: "validate-test-pipeline",
    serviceRoleARN: "",
    codeCommitRepositoryARN: "",
    branchName: "master",
    buildStages:[]
}


test('Code Pipeline Created', () => {
    const app = new App();
    const stack = new Stack(app, "TestStack");
    // WHEN
    const serviceRole = new Role(stack, "testRole", { assumedBy: new ServicePrincipal
('codepipeline.amazonaws.com') })
    const codeCommit = new Repository(stack, "testRepo", {
        repositoryName: "validate-codeCommit-repo"
    });
    const codeBuildProject=new PipelineProject(stack,"TestCodeBuildProject",{});
    testData.serviceRoleARN = serviceRole.roleArn;
    testData.codeCommitRepositoryARN = codeCommit.repositoryArn;
    testData["buildStages"].push({
        stageName:"Deploy",
        codeBuildProject:codeBuildProject
    })
    new CodepipelineModule.CodepipelineModule(stack, 'MyTestConstruct', testData);
    // THEN
    const template = Template.fromStack(stack);

    template.hasResourceProperties('AWS::CodePipeline::Pipeline', {
        Name:testData.pipelineName
    });
});
```

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 Developer Guide.

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.

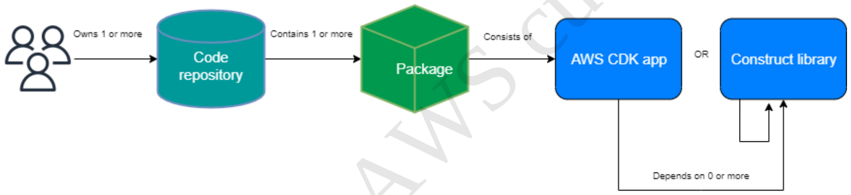
Repository and packaging for AWS CDK constructs

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.

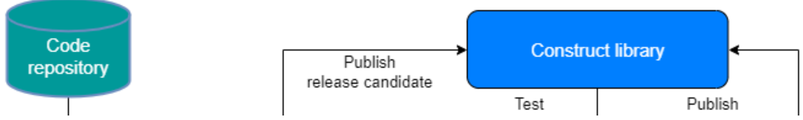
Dependencies on packages in the package repository are managed by your language's package manager (for example, npm for TypeScript or JavaScript applications). Your package manager makes sure that builds are repeatable by recording the specific versions of every package your application depends on and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

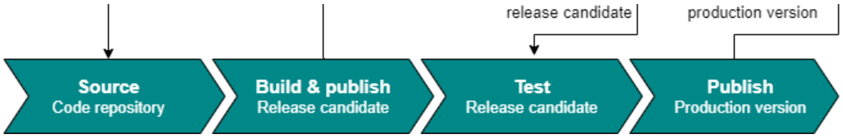


Construct releasing for the AWS CDK

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 AWS 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 CDK instead of 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?

The 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 AWS resources (such as Amazon EC2 instances or Elastic Load Balancing load balancers) that are created when you use the 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 CDK in TypeScript. For more information, see the AWS CDK Workshop in the AWS CDK Workshop documentation.

This disclaimer applies to the APG Library and all other documents, information, data, and responses (written or verbal) provided by Amazon Web Services, Inc. ("AWS") in connection with the APG Library and your use ("You") of the APG Library services. All use of AWS`s Service Offerings will be governed by the AWS Customer Agreement available at http://aws.amazon.com/agreement/(or other definitive written agreement as may be agreed between the parties governing the use of AWS's Service Offerings) (as applicable, the "Agreement"). Pricing information (if any) provided in this APG Library is subject to change in accordance with the Agreement. Nothing in the APG Library will modify or supplement the terms of the Agreement or any Non-Disclosure Agreement ("NDA") between AWS and You. No part of the APG Library may be disclosed without AWS's prior written consent.

Resources

References

- AWS Solution Constructs (AWS Solutions)
- Build infrastructure with AWS CloudFormation and the AWS CDK (AWS CDK Workshop)
- AWS Cloud Development Kits (AWS CDK) (GitHub)
- AWS Construct Library API reference (AWS CDK Reference Documentation)
- AWS CDK Reference Documentation (AWS CDK Reference Documentation)
- AWS CDK Workshop (AWS CDK Workshop)

Tools

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

Guides and patterns

- AWS Solutions Constructs patterns (AWS documentation)

This disclaimer applies to the APG Library and all other documents, information, data, and responses (written or verbal) provided by Amazon Web Services, Inc. ("AWS") in connection with the APG Library and your use ("You") of the APG Library services. All use of AWS`s Service Offerings will be governed by the AWS Customer Agreement available at http://aws.amazon.com/agreement/(or other definitive written agreement as may be agreed between the parties governing the use of AWS's Service Offerings) (as applicable, the "Agreement"). Pricing information (if any) provided in this APG Library is subject to change in accordance with the Agreement. Nothing in the APG Library will modify or supplement the terms of the Agreement or any Non-Disclosure Agreement ("NDA") between AWS and You. No part of the APG Library may be disclosed without AWS's prior written consent.

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
Initial publication	—	December 8, 2022
Update	Updated code example for creating an L3 construct	June 16, 2023