Developer Guide

AWS Cloud Development Kit (AWS CDK) v2

Version 2

AWS Cloud Development Kit (AWS CDK) v2: Developer Guide

Table of Contents

This is the AWS CDK v2 Developer Guide. The older CDK v1 entered maintenance on June 1, 2022 and ended support on June 1, 2023.

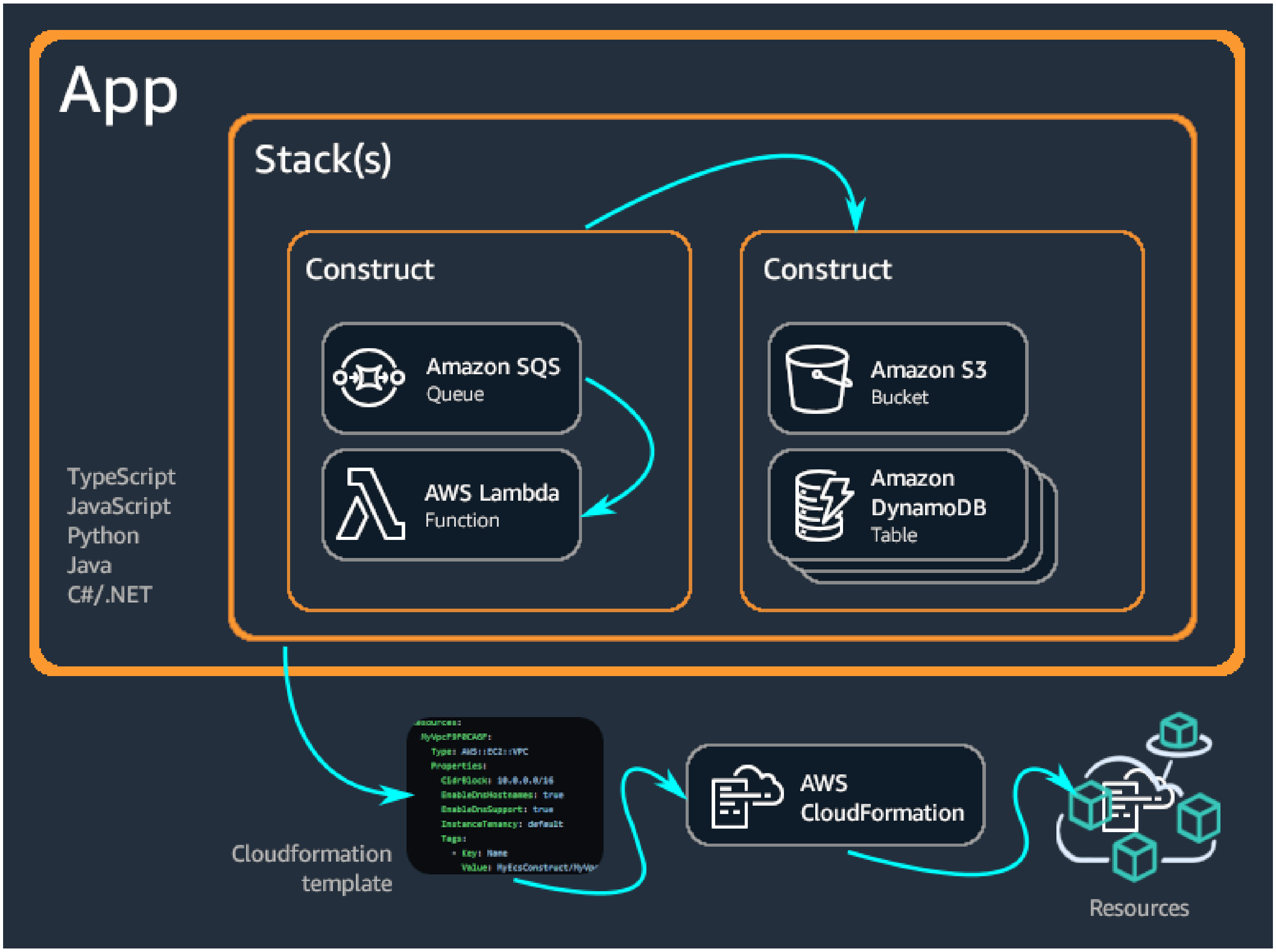
What is the AWS CDK?

The AWS Cloud Development Kit (AWS CDK) is an open-source software development framework for defining cloud infrastructure in code and provisioning it through AWS CloudFormation.

The AWS CDK consists of two primary parts:

- **AWS CDK Construct Library** – A collection of pre-written modular and reusable pieces of code, called constructs, that you can use, modify, and integrate to develop your infrastructure quickly. The goal of the AWS CDK Construct Library is to reduce the complexity required to define and integrate AWS services together when building applications on AWS.
- AWS CDK Toolkit - Tools that you can use to manage and interact with your CDK apps, such as performing synthesis or deployment. The CDK Toolkit consists of a command line tool (CDK CLI) and a programmatic library (CDK Toolkit Library).

The AWS CDK supports TypeScript, JavaScript, Python, Java, C#/.Net, and Go. You can use any of these supported programming languages to define reusable cloud components known as constructs. You compose these together into stacks and apps. Then, you deploy your CDK applications through AWS CloudFormation to provision or update your resources.



Benefits of the AWS CDK

Use the AWS CDK to develop reliable, scalable, cost-effective applications in the cloud with the considerable expressive power of a programming language. This approach yields many benefits, including:

Develop and manage your infrastructure as code (IaC)

Practice *infrastructure as code* to create, deploy, and maintain infrastructure in a programmatic, descriptive, and declarative way. With IaC, you treat infrastructure the same way developers treat code. This results in a scalable and structured approach to managing infrastructure. To learn more about IaC, see [Infrastructure as code] in the *Introduction to DevOps on AWS Whitepaper*.

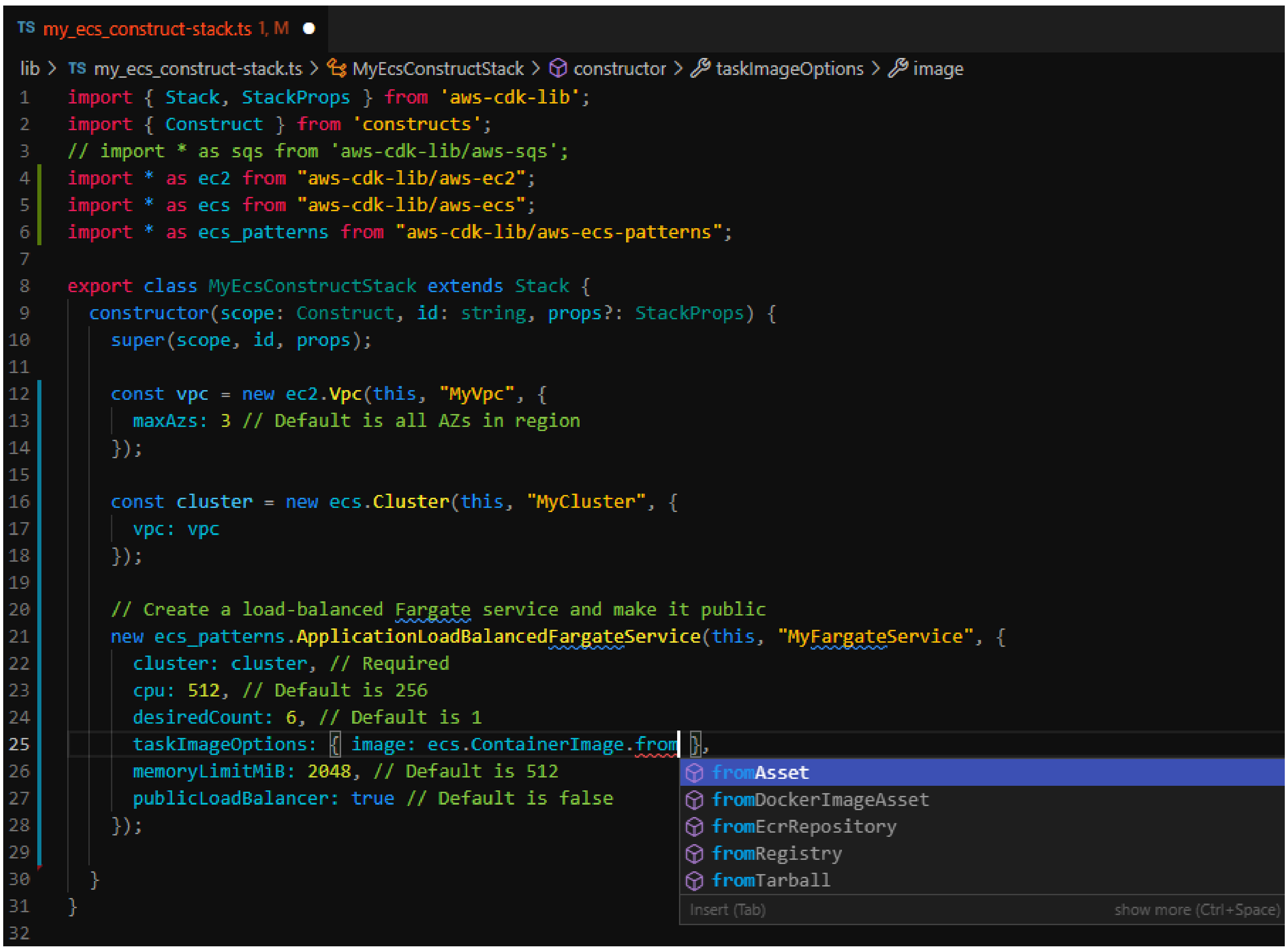
With the AWS CDK, you can put your infrastructure, application code, and configuration all in one place, ensuring that you have a complete, cloud-deployable system at every milestone. Employ software engineering best practices such as code reviews, unit tests, and source control to make your infrastructure more robust.

Define your cloud infrastructure using general-purpose programming languages

With the AWS CDK, you can use any of the following programming languages to define your cloud infrastructure: TypeScript, JavaScript, Python, Java, C#/.Net, and Go. Choose your preferred language and use programming elements like parameters, conditionals, loops, composition, and inheritance to define the desired outcome of your infrastructure.

Use the same programming language to define your infrastructure and your application logic.

Receive the benefits of developing infrastructure in your preferred IDE (Integrated Development Environment), such as syntax highlighting and intelligent code completion.



Deploy infrastructure through AWS CloudFormation

AWS CDK integrates with AWS CloudFormation to deploy and provision your infrastructure on AWS. AWS CloudFormation is a managed AWS service that offers extensive support of resource and property configurations for provisioning services on AWS. With AWS CloudFormation, you can perform infrastructure deployments predictably and repeatedly, with rollback on error. If you are already familiar with AWS CloudFormation, you don't have to learn a new IaC management service when getting started with the AWS CDK.

Get started developing your application quickly with constructs

Develop faster by using and sharing reusable components called constructs. Use low-level constructs to define individual AWS CloudFormation resources and their properties. Use high-level constructs to quickly define larger components of your application, with sensible, secure defaults for your AWS resources, defining more infrastructure with less code.

Create your own constructs that are customized for your unique use cases and share them across your organization or even with the public.

Example of the AWS CDK

The following is an example of using the AWS CDK Constructs Library to create an Amazon Elastic Container Service (Amazon ECS) service with AWS Fargate launch type. For more details of this example, see [Example: Create an AWS Fargate service using the AWS CDK](#).

Example

TypeScript

```
export class MyEcsConstructStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });
```

```
    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is false
    });
  }
}
```

JavaScript

```
class MyEcsConstructStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is false
    });
  }
}
```

```
module.exports = { MyEcsConstructStack }
```

Python

```
class MyEcsConstructStack(Stack):

  def __init__(self, scope: Construct, id: str, **kwargs) -> None:
    super().__init__(scope, id, **kwargs)

    vpc = ec2.Vpc(self, "MyVpc", max_azs=3)     # default is all AZs in region

    cluster = ecs.Cluster(self, "MyCluster", vpc=vpc)

    ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
      cluster=cluster,            # Required
      cpu=512,                    # Default is 256
      desired_count=6,            # Default is 1
      task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
          image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
      memory_limit_mib=2048,      # Default is 512
      public_load_balancer=True)  # Default is False
```

Java

```
public class MyEcsConstructStack extends Stack {

  public MyEcsConstructStack(final Construct scope, final String id) {
    this(scope, id, null);
  }

  public MyEcsConstructStack(final Construct scope, final String id,
      StackProps props) {
    super(scope, id, props);

    Vpc vpc = Vpc.Builder.create(this, "MyVpc").maxAzs(3).build();

    Cluster cluster = Cluster.Builder.create(this, "MyCluster")
            .vpc(vpc).build();

    ApplicationLoadBalancedFargateService.Builder.create(this, "MyFargateService")
            .cluster(cluster)
            .cpu(512)
            .desiredCount(6)
```

```
            .taskImageOptions(
                    ApplicationLoadBalancedTaskImageOptions.builder()
                            .image(ContainerImage
                                    .fromRegistry("amazon/amazon-ecs-sample"))
                            .build()).memoryLimitMiB(2048)
            .publicLoadBalancer(true).build();
  }
}
```

C#

```
public class MyEcsConstructStack : Stack
{
    public MyEcsConstructStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
    {
        var vpc = new Vpc(this, "MyVpc", new VpcProps
        {
            MaxAzs = 3
        });

        var cluster = new Cluster(this, "MyCluster", new ClusterProps
        {
            Vpc = vpc
        });

        new ApplicationLoadBalancedFargateService(this, "MyFargateService",
            new ApplicationLoadBalancedFargateServiceProps
        {
            Cluster = cluster,
            Cpu = 512,
            DesiredCount = 6,
            TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
            {
                Image = ContainerImage.FromRegistry("amazon/amazon-ecs-sample")
            },
            MemoryLimitMiB = 2048,
            PublicLoadBalancer = true,
        });
    }
}
```

Go

```
func NewMyEcsConstructStack(scope constructs.Construct, id string, props
 *MyEcsConstructStackProps) awscdk.Stack {

 var sprops awscdk.StackProps

 if props != nil {
  sprops = props.StackProps
 }

 stack := awscdk.NewStack(scope, &id, &sprops)

 vpc := awsec2.NewVpc(stack, jsii.String("MyVpc"), &awsec2.VpcProps{
  MaxAzs: jsii.Number(3), // Default is all AZs in region
 })

 cluster := awsecs.NewCluster(stack, jsii.String("MyCluster"), &awsecs.ClusterProps{
  Vpc: vpc,
 })

 awsecspatterns.NewApplicationLoadBalancedFargateService(stack,
 jsii.String("MyFargateService"),
  &awsecspatterns.ApplicationLoadBalancedFargateServiceProps{
   Cluster:        cluster,           // required
   Cpu:            jsii.Number(512),  // default is 256
   DesiredCount:   jsii.Number(5),    // default is 1
   MemoryLimitMiB: jsii.Number(2048), // Default is 512
   TaskImageOptions: &awsecspatterns.ApplicationLoadBalancedTaskImageOptions{
    Image: awsecs.ContainerImage_FromRegistry(jsii.String("amazon/amazon-ecs-
sample"), nil),
   },
   PublicLoadBalancer: jsii.Bool(true), // Default is false
  })

 return stack

}
```

This class produces an AWS CloudFormation template of more than 500 lines. Deploying the AWS CDK app produces more than 50 resources of the following types:

- `AWS::EC2::EIP`
- `AWS::EC2::InternetGateway`
- `AWS::EC2::NatGateway`
- `AWS::EC2::Route`
- `AWS::EC2::RouteTable`
- `AWS::EC2::SecurityGroup`
- `AWS::EC2::Subnet`
- `AWS::EC2::SubnetRouteTableAssociation`
- `AWS::EC2::VPCGatewayAttachment`
- `AWS::EC2::VPC`
- `AWS::ECS::Cluster`
- `AWS::ECS::Service`
- `AWS::ECS::TaskDefinition`
- `AWS::ElasticLoadBalancingV2::Listener`
- `AWS::ElasticLoadBalancingV2::LoadBalancer`
- `AWS::ElasticLoadBalancingV2::TargetGroup`
- `AWS::IAM::Policy`
- `AWS::IAM::Role`
- `AWS::Logs::LogGroup`

AWS CDK features

The AWS CDK GitHub repository

For the official AWS CDK GitHub repository, see aws-cdk. Here, you can submit issues, view our license, track releases, and more.

Because the AWS CDK is open-source, the team encourages you to contribute to make it an even better tool. For details, see Contributing to the AWS Cloud Development Kit (AWS CDK).

The AWS CDK API reference

The AWS CDK Construct Library provides APIs to define your CDK application and add CDK constructs to the application. For more information, see the AWS CDK API Reference.

The Construct Programming Model

The Construct Programming Model (CPM) extends the concepts behind the AWS CDK into additional domains. Other tools using the CPM include:

- CDK for Terraform (CDKtf)
- CDK for Kubernetes (CDK8s)
- Projen, for building project configurations

The Construct Hub

The Construct Hub is an online registry where you can find, publish, and share open-source AWS CDK libraries.

Next steps

To get started with using the AWS CDK, see Getting started with the AWS CDK.

Learn more

To continue learning about the AWS CDK, see the following:

- **Learn AWS CDK core concepts** – Important concepts and terms for the AWS CDK.
- **AWS CDK Workshop** – Hands-on workshop to learn and use the AWS CDK.
- **AWS CDK Patterns** – Open-source collection of AWS serverless architecture patterns, built for the AWS CDK by AWS experts.
- **AWS CDK code examples** – GitHub repository of example AWS CDK projects.
- **cdk.dev** – Community-driven hub for the AWS CDK, including a community Slack workspace.
- **Awesome CDK** – GitHub repository containing a curated list of AWS CDK open-source projects, guides, blogs, and other resources.

- **AWS Solutions Constructs** – Vetted, configuration infrastructure as code (IaC) patterns that can easily be assembled into production-ready applications.
- **AWS Developer Tools Blog** – Blog posts filtered for the AWS CDK.
- **AWS CDK on Stack Overflow** – Questions tagged with **aws-cdk** on Stack Overflow.
- **AWS CDK tutorial for AWS Cloud9** – Tutorial on using the AWS CDK with the AWS Cloud9 development environment.

To learn more about related topics to the AWS CDK, see the following:

- **AWS CloudFormation concepts** – Since the AWS CDK is built to work with AWS CloudFormation, we recommend that you learn and understand key AWS CloudFormation concepts.
- **AWS Glossary** – Definitions of key terms used across AWS.

To learn more about tools related to the AWS CDK that can be used to simplify serverless application development and deployment, see the following:

- **AWS Serverless Application Model** – An open-source developer tool that simplifies and improves the experience of building and running serverless applications on AWS.
- **AWS Chalice** – A framework for writing serverless apps in Python.

Learn AWS CDK core concepts

Learn core concepts behind the AWS Cloud Development Kit (AWS CDK).

AWS CDK and IaC

The AWS CDK is an open-source framework that you can use to manage your AWS infrastructure using code. This approach is known as *infrastructure as code (IaC)*. By managing and provisioning your infrastructure as code, you treat your infrastructure in the same way that developers treat code. This provides many benefits, such as version control and scalability. To learn more about IaC, see [What is Infrastructure as Code?](#)

AWS CDK and AWS CloudFormation

The AWS CDK is tightly integrated with AWS CloudFormation. AWS CloudFormation is a fully managed service that you can use to manage and provision your infrastructure on AWS. With AWS CloudFormation, you define your infrastructure in templates and deploy them to AWS CloudFormation. The AWS CloudFormation service then provisions your infrastructure according to the configuration defined on your templates.

AWS CloudFormation templates are *declarative*, meaning they declare the desired state or outcome of your infrastructure. Using JSON or YAML, you declare your AWS infrastructure by defining AWS *resources* and *properties*. Resources represent the many services on AWS and properties represent your desired configuration of those services. When you deploy your template to AWS CloudFormation, your resources and their configured properties are provisioned as described on your template.

With the AWS CDK, you can manage your infrastructure *imperatively*, using general-purpose programming languages. Instead of just defining a desired state declaratively, you can define the logic or sequence necessary to reach the desired state. For example, you can use `if` statements or conditional loops that determine how to reach a desired end state for your infrastructure.

Infrastructure created with the AWS CDK is eventually translated, or *synthesized* into AWS CloudFormation templates and deployed using the AWS CloudFormation service. So while the AWS CDK offers a different approach to creating your infrastructure, you still receive the benefits of AWS CloudFormation, such as extensive AWS resource configuration support and robust deployment processes.

To learn more about AWS CloudFormation, see What is AWS CloudFormation? in the *AWS CloudFormation User Guide*.

AWS CDK and abstractions

With AWS CloudFormation, you must define every detail of how your resources are configured. This provides the benefit of having complete control over your infrastructure. However, this requires you to learn, understand, and create robust templates that contain resource configuration details and relationships between resources, such as permissions and event-driven interactions.

With the AWS CDK, you can have the same control over your resource configurations. However, the AWS CDK also offers powerful abstractions, which can speed up and simplify the infrastructure development process. For example, the AWS CDK includes constructs that provide sensible default configurations and helper methods that generate boilerplate code for you. The AWS CDK also offers tools, such as the AWS CDK Command Line Interface (AWS CDK CLI), that perform infrastructure management actions for you.

Learn more about core AWS CDK concepts

Interacting with the AWS CDK

When using with the AWS CDK, you will primarily interact with the AWS Construct Library and the AWS CDK CLI.

Developing with the AWS CDK

The AWS CDK can be written in any supported programming language. You start with a CDK project, which contains a structure of folders and files, including assets. Within the project, you create a CDK application. Within the app, you define a stack, which directly represents a CloudFormation stack. Within the stack, you define your AWS resources and properties using constructs.

Deploying with the AWS CDK

You deploy CDK apps into an AWS environment. Before deploying, you must perform a one-time bootstrapping to prepare your environment.

Learn more

To learn more about AWS CDK core concepts, see the topics in this section.

Supported programming languages for the AWS CDK

The AWS Cloud Development Kit (AWS CDK) has first-class support for the following general-purpose programming languages:

- TypeScript
- JavaScript
- Python
- Java
- C#
- Go

Other JVM and .NET CLR languages may also be used in theory, but we do not offer official support at this time.

The AWS CDK is developed in one language, TypeScript. To support the other languages, the AWS CDK utilizes a tool called JSII to generate language bindings.

We attempt to offer each language's usual conventions to make development with the AWS CDK as natural and intuitive as possible. For example, we distribute AWS Construct Library modules using your preferred language's standard repository, and you install them using the language's standard package manager. Methods and properties are also named using your language's recommended naming patterns.

The following are a few code examples:

Example

TypeScript

```
const bucket = new s3.Bucket(this, 'amzn-s3-demo-bucket', {
  bucketName: 'amzn-s3-demo-bucket',
  versioned: true,
  websiteRedirect: {hostName: 'aws.amazon.com'}});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'amzn-s3-demo-bucket', {
  bucketName: 'amzn-s3-demo-bucket',
  versioned: true,
  websiteRedirect: {hostName: 'aws.amazon.com'}});
```

Python

```
bucket = s3.Bucket("amzn-s3-demo-bucket", bucket_name="amzn-s3-demo-bucket",
 versioned=True,
            website_redirect=s3.RedirectTarget(host_name="aws.amazon.com"))
```

Java

```
Bucket bucket = Bucket.Builder.create(self, "amzn-s3-demo-bucket")
                      .bucketName("amzn-s3-demo-bucket")
                      .versioned(true)
                      .websiteRedirect(new RedirectTarget.Builder()
                          .hostName("aws.amazon.com").build())
                      .build();
```

C#

```
var bucket = new Bucket(this, "amzn-s3-demo-bucket", new BucketProps {
                      BucketName = "amzn-s3-demo-bucket",
                      Versioned  = true,
                      WebsiteRedirect = new RedirectTarget {
                              HostName = "aws.amazon.com"
                      }});
```

Go

```
bucket := awss3.NewBucket(scope, jsii.String("amzn-s3-demo-bucket"),
 &awss3.BucketProps {
 BucketName: jsii.String("amzn-s3-demo-bucket"),
 Versioned: jsii.Bool(true),
 WebsiteRedirect: &awss3.RedirectTarget {
  HostName: jsii.String("aws.amazon.com"),
 },
})
```

> **Note**
>
> These code snippets are intended for illustration only. They are incomplete and won't run as they are.

The AWS Construct Library is distributed using each language's standard package management tools, including NPM, PyPi, Maven, and NuGet. We also provide a version of the AWS CDK API Reference for each language.

To help you use the AWS CDK in your preferred language, this guide includes the following topics for supported languages:

- Working with the AWS CDK in TypeScript
- Working with the AWS CDK in JavaScript
- Working with the AWS CDK in Python
- Working with the AWS CDK in Java
- Working with the AWS CDK in C#
- Working with the AWS CDK in Go

TypeScript was the first language supported by the AWS CDK, and much of the AWS CDK example code is written in TypeScript. This guide includes a topic specifically to show how to adapt TypeScript AWS CDK code for use with the other supported languages. For more information, see Comparing AWS CDK in TypeScript with other languages.

The AWS CDK libraries

Learn about the core libraries that you will use with the AWS Cloud Development Kit (AWS CDK).

The AWS CDK Library

The AWS CDK Library, also referred to as `aws-cdk-lib`, is the main library that you will use to develop applications with the AWS CDK. It is developed and maintained by AWS. This library contains base classes, such as App and `Stack`. It also contains the libraries you will use to define your infrastructure through constructs.

The AWS Construct Library

The AWS Construct Library is a part of the AWS CDK Library. It contains a collection of constructs that are developed and maintained by AWS. It is organized into various modules for each AWS service. Each module includes constructs that you can use to define your AWS resources and properties.

The Constructs library

The Constructs library, commonly referred to as `constructs`, is a library for defining and composing cloud infrastructure components. It contains the core `Construct` class, which represents the construct building block. This class is the foundational base class of all constructs from the AWS Construct Library. The Constructs library is a separate general-purpose library that is used by other construct-based tools such as *CDK for Terraform* and *CDK for Kubernetes*.

The AWS CDK API reference

The AWS CDK API reference contains official reference documentation for the AWS CDK Library, including the AWS Construct Library and Constructs library. A version of the API reference is provided for each supported programming language.

- For AWS CDK Library (`aws-cdk-lib`) documentation, see aws-cdk-lib module.
- Documentation for constructs in the AWS Construct Library are organized by AWS service in the following format: `aws-cdk-lib.<service>`. For example, construct documentation for Amazon Simple Storage Service (Amazon S3), can be found at aws-cdk-lib.aws_s3 module.
- For Constructs library (constructs) documentation, see constructs module.

Contribute to the AWS CDK API reference

The AWS CDK is open-source and we welcome you to contribute. Community contributions positively impact and improve the AWS CDK. For instructions on contributing specifically to AWS CDK API reference documentation, see Documentation in the *aws-cdk GitHub repository*.

Learn more

For instructions on importing and using the CDK Library, see Work with the CDK library.

AWS CDK projects

An AWS Cloud Development Kit (AWS CDK) project represents the files and folders that contain your CDK code. Contents will vary based on your programming language.

You can create your AWS CDK project manually or with the AWS CDK Command Line Interface (AWS CDK CLI) `cdk init` command. In this topic, we will refer to the project structure and naming conventions of files and folders created by the AWS CDK CLI. You can customize and organize your CDK projects to fit your needs.

> **Note**
>
> Project structure created by the AWS CDK CLI may vary across versions over time.

Universal files and folders

`.git`

If you have `git` installed, the AWS CDK CLI automatically initializes a Git repository for your project. The `.git` directory contains information about the repository.

`.gitignore`

Text file used by Git to specify files and folders to ignore.

`README.md`

Text file that provides you with basic guidance and important information for managing your AWS CDK project. Modify this file as necessary to document important information regarding your CDK project.

`cdk.json`

Configuration file for the AWS CDK. This file provides instruction to the AWS CDK CLI regarding how to run your app.

Language-specific files and folders

The following files and folders are unique to each supported programming language.

Example

TypeScript

The following is an example project created in the `my-cdk-ts-project` directory using the `cdk init --language typescript` command:

```
my-cdk-ts-project
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### my-cdk-ts-project.ts
### cdk.json
### jest.config.js
### lib
#   ### my-cdk-ts-project-stack.ts
### node_modules
### package-lock.json
### package.json
### test
#   ### my-cdk-ts-project.test.ts
### tsconfig.json
```

`.npmignore`

File that specifies which files and folders to ignore when publishing a package to the `npm` registry. This file is similar to `.gitignore`, but is specific to `npm` packages.

`bin/my-cdk-ts-project.ts`

The *application file* defines your CDK app. CDK projects can contain one or more application files. Application files are stored in the `bin` folder.

The following is an example of a basic application file that defines a CDK app:

```
#!/usr/bin/env node
import 'source-map-support/register';
```

```
import * as cdk from 'aws-cdk-lib';
import { MyCdkTsProjectStack } from '../lib/my-cdk-ts-project-stack';

const app = new cdk.App();
new MyCdkTsProjectStack(app, 'MyCdkTsProjectStack');
```

`jest.config.js`

Configuration file for Jest. *Jest* is a popular JavaScript testing framework.

`lib/my-cdk-ts-project-stack.ts`

The *stack file* defines your CDK stack. Within your stack, you define AWS resources and properties using constructs.

The following is an example of a basic stack file that defines a CDK stack:

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

export class MyCdkTsProjectStack extends cdk.Stack {
 constructor(scope: Construct, id: string, props?: cdk.StackProps) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}
```

`node_modules`

Common folder in Node.js projects that contain dependencies for your project.

`package-lock.json`

Metadata file that works with the `package.json` file to manage versions of dependencies.

`package.json`

Metadata file that is commonly used in Node.js projects. This file contains information about your CDK project such as the project name, script definitions, dependencies, and other import project-level information.

`test/my-cdk-ts-project.test.ts`

A test folder is created to organize tests for your CDK project. A sample test file is also created.

You can write tests in TypeScript and use Jest to compile your TypeScript code before running tests.

`tsconfig.json`

Configuration file used in TypeScript projects that specifies compiler options and project settings.

JavaScript

The following is an example project created in the `my-cdk-js-project` directory using the `cdk init --language javascript` command:

```
my-cdk-js-project
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### my-cdk-js-project.js
### cdk.json
### jest.config.js
### lib
#   ### my-cdk-js-project-stack.js
### node_modules
### package-lock.json
### package.json
### test
    ### my-cdk-js-project.test.js
```

`.npmignore`

File that specifies which files and folders to ignore when publishing a package to the `npm` registry. This file is similar to `.gitignore`, but is specific to `npm` packages.

`bin/my-cdk-js-project.js`

The *application file* defines your CDK app. CDK projects can contain one or more application files. Application files are stored in the `bin` folder.

The following is an example of a basic application file that defines a CDK app:

```
#!/usr/bin/env node
```

```
const cdk = require('aws-cdk-lib');
const { MyCdkJsProjectStack } = require('../lib/my-cdk-js-project-stack');

const app = new cdk.App();
new MyCdkJsProjectStack(app, 'MyCdkJsProjectStack');
```

`jest.config.js`

Configuration file for Jest. *Jest* is a popular JavaScript testing framework.

`lib/my-cdk-js-project-stack.js`

The *stack file* defines your CDK stack. Within your stack, you define AWS resources and properties using constructs.

The following is an example of a basic stack file that defines a CDK stack:

```
const { Stack, Duration } = require('aws-cdk-lib');

class MyCdkJsProjectStack extends Stack {
 constructor(scope, id, props) {
  super(scope, id, props);

  // code that defines your resources and properties go here
 }
}

module.exports = { MyCdkJsProjectStack }
```

`node_modules`

Common folder in Node.js projects that contain dependencies for your project.

`package-lock.json`

Metadata file that works with the `package.json` file to manage versions of dependencies.

`package.json`

Metadata file that is commonly used in Node.js projects. This file contains information about your CDK project such as the project name, script definitions, dependencies, and other import project-level information.

`test/my-cdk-js-project.test.js`

A test folder is created to organize tests for your CDK project. A sample test file is also created.

You can write tests in JavaScript and use Jest to compile your JavaScript code before running tests.

Python

The following is an example project created in the `my-cdk-py-project` directory using the `cdk init --language python` command:

```
my-cdk-py-project
### .git
### .gitignore
### .venv
### README.md
### app.py
### cdk.json
### my_cdk_py_project
#   ### __init__.py
#   ### my_cdk_py_project_stack.py
### requirements-dev.txt
### requirements.txt
### source.bat
### tests
    ### __init__.py
    ### unit
```

`.venv`

The CDK CLI automatically creates a virtual environment for your project. The `.venv` directory refers to this virtual environment.

`app.py`

The *application file* defines your CDK app. CDK projects can contain one or more application files.

The following is an example of a basic application file that defines a CDK app:

```
#!/usr/bin/env python3
import os

import aws_cdk as cdk

from my_cdk_py_project.my_cdk_py_project_stack import MyCdkPyProjectStack

app = cdk.App()
MyCdkPyProjectStack(app, "MyCdkPyProjectStack")

app.synth()
```

`my_cdk_py_project`

Directory that contains your *stack files*. The CDK CLI creates the following here:

- __init__.py – An empty Python package definition file.
- `my_cdk_py_project` – File that defines your CDK stack. You then define AWS resources and properties within the stack using constructs.

The following is an example of a stack file:

```
from aws_cdk import Stack

from constructs import Construct

class MyCdkPyProjectStack(Stack):
 def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
  super().__init__(scope, construct_id, **kwargs)

  # code that defines your resources and properties go here
```

`requirements-dev.txt`

File similar to `requirements.txt`, but used to manage dependencies specifically for development purposes rather than production.

`requirements.txt`

Common file used in Python projects to specify and manage project dependencies.

`source.bat`

Batch file for Windows that is used to set up the Python virtual environment.

`tests`

Directory that contains tests for your CDK project.

The following is an example of a unit test:

```
import aws_cdk as core
import aws_cdk.assertions as assertions

from my_cdk_py_project.my_cdk_py_project_stack import MyCdkPyProjectStack

def test_sqs_queue_created():
 app = core.App()
 stack = MyCdkPyProjectStack(app, "my-cdk-py-project")
 template = assertions.Template.from_stack(stack)

 template.has_resource_properties("AWS::SQS::Queue", {
  "VisibilityTimeout": 300
 })
```

Java

The following is an example project created in the `my-cdk-java-project` directory using the `cdk init --language java` command:

```
my-cdk-java-project
### .git
### .gitignore
### README.md
### cdk.json
### pom.xml
### src
    ### main
    ### test
```

`pom.xml`

File that contains configuration information and metadata about your CDK project. This file is a part of Maven.

`src/main`

Directory containing your *application* and *stack* files.

The following is an example application file:

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class MyCdkJavaProjectApp {
 public static void main(final String[] args) {
  App app = new App();

  new MyCdkJavaProjectStack(app, "MyCdkJavaProjectStack", StackProps.builder()
   .build());

  app.synth();
 }
}
```

The following is an example stack file:

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class MyCdkJavaProjectStack extends Stack {
 public MyCdkJavaProjectStack(final Construct scope, final String id) {
  this(scope, id, null);
 }

 public MyCdkJavaProjectStack(final Construct scope, final String id, final
 StackProps props) {
  super(scope, id, props);

  // code that defines your resources and properties go here
```

```
 }
}
```

`src/test`

Directory containing your test files. The following is an example:

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.assertions.Template;
import java.io.IOException;

import java.util.HashMap;

import org.junit.jupiter.api.Test;

public class MyCdkJavaProjectTest {

 @Test
 public void testStack() throws IOException {
  App app = new App();
  MyCdkJavaProjectStack stack = new MyCdkJavaProjectStack(app, "test");

  Template template = Template.fromStack(stack);

  template.hasResourceProperties("AWS::SQS::Queue", new HashMap<String, Number>()
 {{
   put("VisibilityTimeout", 300);
  }});
 }
}
```

C#

The following is an example project created in the `my-cdk-csharp-project` directory using the `cdk init --language csharp` command:

```
my-cdk-csharp-project
### .git
### .gitignore
### README.md
```

```
### cdk.json
### src
    ### MyCdkCsharpProject
    ### MyCdkCsharpProject.sln
```

`src/MyCdkCsharpProject`

Directory containing your *application* and *stack* files.

The following is an example application file:

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace MyCdkCsharpProject
{
 sealed class Program
 {
  public static void Main(string[] args)
  {
   var app = new App();
   new MyCdkCsharpProjectStack(app, "MyCdkCsharpProjectStack", new StackProps{});
   app.Synth();
  }
 }
}
```

The following is an example stack file:

```
using Amazon.CDK;
using Constructs;

namespace MyCdkCsharpProject
{
 public class MyCdkCsharpProjectStack : Stack
 {
  internal MyCdkCsharpProjectStack(Construct scope, string id, IStackProps props
 = null) : base(scope, id, props)
  {
   // code that defines your resources and properties go here
  }
```

```
 }
}
```

This directory also contains the following:

- `GlobalSuppressions.cs` – File used to suppress specific compiler warnings or errors across your project.
- `.csproj` – XML-based file used to define project settings, dependencies, and build configurations. ---

`src/MyCdkCsharpProject.sln`

Microsoft Visual Studio Solution File used to organize and manage related projects.

Go

The following is an example project created in the `my-cdk-go-project` directory using the `cdk init --language go` command:

```
my-cdk-go-project
### .git
### .gitignore
### README.md
### cdk.json
### go.mod
### my-cdk-go-project.go
### my-cdk-go-project_test.go
```

`go.mod`

File that contains module information and is used to manage dependencies and versioning for your Go project.

`my-cdk-go-project.go`

File that defines your CDK application and stacks.

The following is an example:

```
package main
import (
```

```
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type MyCdkGoProjectStackProps struct {
 awscdk.StackProps
}

func NewMyCdkGoProjectStack(scope constructs.Construct, id string, props
 *MyCdkGoProjectStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)
 // The code that defines your resources and properties go here

  return stack
}

func main() {
 defer jsii.Close()
 app := awscdk.NewApp(nil)
 NewMyCdkGoProjectStack(app, "MyCdkGoProjectStack", &MyCdkGoProjectStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })
 app.Synth(nil)
}

func env() *awscdk.Environment {

 return nil
}
```

`my-cdk-go-project_test.go`

File that defines a sample test.

The following is an example:

```
package main
```

```
import (
 "testing"

 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/assertions"
 "github.com/aws/jsii-runtime-go"
)

func TestMyCdkGoProjectStack(t *testing.T) {

 // GIVEN
 app := awscdk.NewApp(nil)

 // WHEN
 stack := NewMyCdkGoProjectStack(app, "MyStack", nil)

 // THEN
 template := assertions.Template_FromStack(stack, nil)
 template.HasResourceProperties(jsii.String("AWS::SQS::Queue"),
 map[string]interface{}{
  "VisibilityTimeout": 300,
 })
}
```

AWS CDK apps

The AWS Cloud Development Kit (AWS CDK) application or *app* is a collection of one or more CDK stacks. Stacks are a collection of one or more constructs, which define AWS resources and properties. Therefore, the overall grouping of your stacks and constructs are known as your CDK app.

How to create a CDK app

You create an app by defining an app instance in the application file of your project. To do this, you import and use the App construct from the AWS Construct Library. The App construct doesn't require any initialization arguments. It is the only construct that can be used as the root.

The `App` and `Stack` classes from the AWS Construct Library are unique constructs. Compared to other constructs, they don't configure AWS resources on their own. Instead, they are used to provide context for your other constructs. All constructs that represent AWS resources must be defined, directly or indirectly, within the scope of a `Stack` construct. `Stack` constructs are defined within the scope of an App construct.

Apps are then synthesized to create AWS CloudFormation templates for your stacks. The following is an example:

Example

TypeScript

```
const app = new App();
new MyFirstStack(app, 'hello-cdk');
app.synth();
```

JavaScript

```
const app = new App();
new MyFirstStack(app, 'hello-cdk');
app.synth();
```

Python

```
app = App()
MyFirstStack(app, "hello-cdk")
app.synth()
```

Java

```
App app = new App();
new MyFirstStack(app, "hello-cdk");
app.synth();
```

C#

```
var app = new App();
new MyFirstStack(app, "hello-cdk");
app.Synth();
```

Go

```
app := awscdk.NewApp(nil)
```

```
MyFirstStack(app, "MyFirstStack", &MyFirstStackProps{
  awscdk.StackProps{
    Env: env(),
  },
})

app.Synth(nil)
```

Stacks within a single app can easily refer to each other's resources and properties. The AWS CDK infers dependencies between stacks so that they can be deployed in the correct order. You can deploy any or all of the stacks within an app with a single `cdk deploy` command.

The construct tree

Constructs are defined inside of other constructs using the `scope` argument that is passed to every construct, with the `App` class as the root. In this way, an AWS CDK app defines a hierarchy of constructs known as the *construct tree*.

The root of this tree is your app, which is an instance of the `App` class. Within the app, you instantiate one or more stacks. Within stacks, you instantiate constructs, which may themselves instantiate resources or other constructs, and so on down the tree.

Constructs are *always* explicitly defined within the scope of another construct, which creates relationships between constructs. Almost always, you should pass `this` (in Python, `self`) as the scope, indicating that the new construct is a child of the current construct. The intended pattern is that you derive your construct from `Construct`, then instantiate the constructs it uses in its constructor.

Passing the scope explicitly allows each construct to add itself to the tree, with this behavior entirely contained within the `Construct` base class. It works the same way in every language supported by the AWS CDK and does not require additional customization.

> **Important**
>
> Technically, it's possible to pass some scope other than `this` when instantiating a construct. You can add constructs anywhere in the tree, or even in another stack in the same app. For example, you could write a mixin-style function that adds constructs to a scope passed in as an argument. The practical difficulty here is that you can't easily ensure

that the IDs you choose for your constructs are unique within someone else's scope. This practice also makes your code more difficult to understand, maintain, and reuse. Therefore, we recommend that you use the general structure of the construct tree.

The AWS CDK uses the IDs of all constructs in the path from the tree's root to each child construct to generate the unique IDs required by AWS CloudFormation. This approach means that construct IDs only need to be unique within their scope, rather than within the entire stack as in native AWS CloudFormation. However, if you move a construct to a different scope, its generated stack-unique ID changes, and AWS CloudFormation won't consider it the same resource.

The construct tree is separate from the constructs that you define in your AWS CDK code. However, it's accessible through any construct's `node` attribute, which is a reference to the node that represents that construct in the tree. Each node is a Node instance, the attributes of which provide access to the tree's root and to the node's parent scopes and children.

1. `node.children` – The direct children of the construct.
2. `node.id` – The identifier of the construct within its scope.
3. `node.path` – The full path of the construct including the IDs of all of its parents.
4. `node.root` – The root of the construct tree (the app).
5. `node.scope` – The scope (parent) of the construct, or undefined if the node is the root.
6. `node.scopes` – All parents of the construct, up to the root.
7. `node.uniqueId` – The unique alphanumeric identifier for this construct within the tree (by default, generated from `node.path` and a hash).

The construct tree defines an implicit order in which constructs are synthesized to resources in the final AWS CloudFormation template. Where one resource must be created before another, AWS CloudFormation or the AWS Construct Library generally infers the dependency. They then make sure that the resources are created in the right order.

You can also add an explicit dependency between two nodes by using `node.addDependency()`. For more information, see Dependencies in the *AWS CDK API Reference*.

The AWS CDK provides a simple way to visit every node in the construct tree and perform an operation on each one. For more information, see Aspects and the AWS CDK.

Introduction to AWS CDK stacks

An AWS CDK stack is the smallest single unit of deployment. It represents a collection of AWS resources that you define using CDK constructs. When you deploy CDK apps, the resources within a CDK stack are deployed together as an AWS CloudFormation stack. To learn more about AWS CloudFormation stacks, see [Managing AWS resources as a single unit with AWS CloudFormation stacks] in the *AWS CloudFormation User Guide*.

You define a stack by extending or inheriting from the `Stack` construct. The following example is a common pattern for defining a CDK stack on a separate file, known as a *stack file*. Here, we extend or inherit the `Stack` class and define a constructor that accepts `scope`, `id`, and `props`. Then, we invoke the base `Stack` class constructor using `super` with the received `scope`, `id`, and `props`:

Example

TypeScript

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

export class MyCdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define your constructs here

  }
}
```

JavaScript

```
const { Stack } = require('aws-cdk-lib');

class MyCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define your constructs here

  }
}
```

```
module.exports = { MyCdkStack }
```

Python

```
from aws_cdk import (
  Stack,
)
from constructs import Construct

class MyCdkStack(Stack):

  def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
    super().__init__(scope, construct_id, **kwargs)

    # Define your constructs here
```

Java

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class MyCdkStack extends Stack {
  public MyCdkStack(final Construct scope, final String id) {
    this(scope, id, null);
  }

  public MyCdkStack(final Construct scope, final String id, final StackProps props)
 {
    super(scope, id, props);

    // Define your constructs here
  }
}
```

C#

```
using Amazon.CDK;
using Constructs;
```

```
namespace MyCdk
{
  public class MyCdkStack : Stack
  {
    internal MyCdkStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
    {
      // Define your constructs here
    }
  }
}
```

Go

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type CdkDemoAppStackProps struct {
 awscdk.StackProps
}

func NewCdkDemoAppStack(scope constructs.Construct, id string, props
 *CdkDemoAppStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 // The code that defines your stack goes here

 return stack
}

func main() {
 defer jsii.Close()
```

```
 app := awscdk.NewApp(nil)

 NewCdkDemoAppStack(app, "CdkDemoAppStack", &CdkDemoAppStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })

 app.Synth(nil)
}

//...
```

The previous example has only defined a stack. To create the stack, it must be instantiated within the context of your CDK app. A common pattern is to define your CDK app and initialize your stack on a separate file, known as an *application file*.

The following is an example that creates a CDK stack named `MyCdkStack`. Here, the CDK app is created and `MyCdkStack` is instantiated in the context of the app:

Example

TypeScript

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MyCdkStack } from '../lib/my-cdk-stack';

const app = new cdk.App();
new MyCdkStack(app, 'MyCdkStack', {
});
```

JavaScript

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { MyCdkStack } = require('../lib/my-cdk-stack');

const app = new cdk.App();
new MyCdkStack(app, 'MyCdkStack', {
```

```
});
```

Python

Located in `app.py`:

```
#!/usr/bin/env python3
import os

import aws_cdk as cdk

from my_cdk.my_cdk_stack import MyCdkStack


app = cdk.App()
MyCdkStack(app, "MyCdkStack",)

app.synth()
```

Java

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class MyCdkApp {
  public static void main(final String[] args) {
    App app = new App();

    new MyCdkStack(app, "MyCdkStack", StackProps.builder()
      .build());

    app.synth();
  }
}
```

C#

```
using Amazon.CDK;
```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace MyCdk
{
  sealed class Program
  {
    public static void Main(string[] args)
    {
      var app = new App();
      new MyCdkStack(app, "MyCdkStack", new StackProps
      {});
      app.Synth();
    }
  }
}
```

Go

```
package main

import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
)

// ...

func main() {
  defer jsii.Close()

  app := awscdk.NewApp(nil)

  NewMyCdkStack(app, "MyCdkStack", &MyCdkStackProps{
    awscdk.StackProps{
      Env: env(),
    },
  })

  app.Synth(nil)
}
```

```
// ...
```

The following example creates a CDK app that contains two stacks:

Example

TypeScript

```
const app = new App();

new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

JavaScript

```
const app = new App();

new MyFirstStack(app, 'stack1');
new MySecondStack(app, 'stack2');

app.synth();
```

Python

```
app = App()

MyFirstStack(app, 'stack1')
MySecondStack(app, 'stack2')

app.synth()
```

Java

```
App app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");
```

```
app.synth();
```

C#

```
var app = new App();

new MyFirstStack(app, "stack1");
new MySecondStack(app, "stack2");

app.Synth();
```

Go

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type MyFirstStackProps struct {
 awscdk.StackProps
}

func NewMyFirstStack(scope constructs.Construct, id string, props
 *MyFirstStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 myFirstStack := awscdk.NewStack(scope, &id, &sprops)

 // The code that defines your stack goes here

 return myFirstStack
}

type MySecondStackProps struct {
 awscdk.StackProps
}
```

```
func NewMySecondStack(scope constructs.Construct, id string, props
 *MySecondStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 mySecondStack := awscdk.NewStack(scope, &id, &sprops)

 // The code that defines your stack goes here

 return mySecondStack
}

func main() {
 defer jsii.Close()

 app := awscdk.NewApp(nil)

 NewMyFirstStack(app, "MyFirstStack", &MyFirstStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })

 NewMySecondStack(app, "MySecondStack", &MySecondStackProps{
  awscdk.StackProps{
   Env: env(),
  },
 })

 app.Synth(nil)
}

// ...
```

About the stack API

The `Stack` object provides a rich API, including the following:

- `Stack.of(construct)` – A static method that returns the **Stack** in which a construct is defined. This is useful if you need to interact with a stack from within a reusable construct. The call fails if a stack cannot be found in scope.

- `stack.stackName` (Python: `stack_name`) – Returns the physical name of the stack. As mentioned previously, all AWS CDK stacks have a physical name that the AWS CDK can resolve during synthesis.
- `stack.region` and `stack.account` – Return the AWS Region and account, respectively, into which this stack will be deployed. These properties return one of the following:
  - The account or Region explicitly specified when the stack was defined
  - A string-encoded token that resolves to the AWS CloudFormation pseudo parameters for account and Region to indicate that this stack is environment agnostic

    For information about how environments are determined for stacks, see Environments for the AWS CDK.
- `stack.addDependency(stack)` (Python: `stack.add_dependency(stack)`) – Can be used to explicitly define dependency order between two stacks. This order is respected by the `cdk deploy` command when deploying multiple stacks at once.
- `stack.tags` – Returns a TagManager that you can use to add or remove stack-level tags. This tag manager tags all resources within the stack, and also tags the stack itself when it's created through AWS CloudFormation.
- `stack.partition`, `stack.urlSuffix` (Python: `url_suffix`), `stack.stackId` (Python: `stack_id`), and `stack.notificationArn` (Python: `notification_arn`) – Return tokens that resolve to the respective AWS CloudFormation pseudo parameters, such as `{ "Ref": "AWS::Partition" }`. These tokens are associated with the specific stack object so that the AWS CDK framework can identify cross-stack references.
- `stack.availabilityZones` (Python: `availability_zones`) – Returns the set of Availability Zones available in the environment in which this stack is deployed. For environment-agnostic stacks, this always returns an array with two Availability Zones. For environment-specific stacks, the AWS CDK queries the environment and returns the exact set of Availability Zones available in the Region that you specified.
- `stack.parseArn(arn)` and `stack.formatArn(comps)` (Python: `parse_arn`, `format_arn`) – Can be used to work with Amazon Resource Names (ARNs).
- `stack.toJsonString(obj)` (Python: `to_json_string`) – Can be used to format an arbitrary object as a JSON string that can be embedded in an AWS CloudFormation template. The object can include tokens, attributes, and references, which are only resolved during deployment.
- `stack.templateOptions` (Python: `template_options`) – Use to specify AWS CloudFormation template options, such as Transform, Description, and Metadata, for your stack.

Working with stacks

Stacks are deployed as an AWS CloudFormation stack into an AWS environment. The environment covers a specific AWS account and AWS Region.

When you run the `cdk synth` command for an app with multiple stacks, the cloud assembly includes a separate template for each stack instance. Even if the two stacks are instances of the same class, the AWS CDK emits them as two individual templates.

You can synthesize each template by specifying the stack name in the `cdk synth` command. The following example synthesizes the template for `stack1`:

```
$ cdk synth <stack1>
```

This approach is conceptually different from how AWS CloudFormation templates are normally used, where a template can be deployed multiple times and parameterized through AWS CloudFormation parameters. Although AWS CloudFormation parameters can be defined in the AWS CDK, they are generally discouraged because AWS CloudFormation parameters are resolved only during deployment. This means that you cannot determine their value in your code.

For example, to conditionally include a resource in your app based on a parameter value, you must set up an AWS CloudFormation condition and tag the resource with it. The AWS CDK takes an approach where concrete templates are resolved at synthesis time. Therefore, you can use an `if` statement to check the value to determine whether a resource should be defined or some behavior should be applied.

> **Note**
>
> The AWS CDK provides as much resolution as possible during synthesis time to enable idiomatic and natural usage of your programming language.

Like any other construct, stacks can be composed together into groups. The following code shows an example of a service that consists of three stacks: a control plane, a data plane, and monitoring stacks. The service construct is defined twice: once for the beta environment and once for the production environment.

Example

TypeScript

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

interface EnvProps {
  prod: boolean;
}

// imagine these stacks declare a bunch of related resources
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}

class MyService extends Construct {

  constructor(scope: Construct, id: string, props?: EnvProps) {

    super(scope, id);

    // we might use the prod argument to change how the service is configured
    new ControlPlane(this, "cp");
    new DataPlane(this, "data");
    new Monitoring(this, "mon");  }
}

const app = new App();
new MyService(app, "beta");
new MyService(app, "prod", { prod: true });

app.synth();
```

JavaScript

```
const { App, Stack } = require('aws-cdk-lib');
const { Construct } = require('constructs');

// imagine these stacks declare a bunch of related resources
class ControlPlane extends Stack {}
class DataPlane extends Stack {}
class Monitoring extends Stack {}
```

```
class MyService extends Construct {

  constructor(scope, id, props) {

    super(scope, id);

    // we might use the prod argument to change how the service is configured
    new ControlPlane(this, "cp");
    new DataPlane(this, "data");
    new Monitoring(this, "mon");
  }
}

const app = new App();
new MyService(app, "beta");
new MyService(app, "prod", { prod: true });

app.synth();
```

Python

```
from aws_cdk import App, Stack
from constructs import Construct

# imagine these stacks declare a bunch of related resources
class ControlPlane(Stack): pass
class DataPlane(Stack): pass
class Monitoring(Stack): pass

class MyService(Construct):

  def __init__(self, scope: Construct, id: str, *, prod=False):

    super().__init__(scope, id)

    # we might use the prod argument to change how the service is configured
    ControlPlane(self, "cp")
    DataPlane(self, "data")
    Monitoring(self, "mon")

app = App();
MyService(app, "beta")
```

```
MyService(app, "prod", prod=True)

app.synth()
```

Java

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.constructs.Construct;

public class MyApp {

    // imagine these stacks declare a bunch of related resources
    static class ControlPlane extends Stack {
        ControlPlane(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class DataPlane extends Stack {
        DataPlane(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class Monitoring extends Stack {
        Monitoring(Construct scope, String id) {
            super(scope, id);
        }
    }

    static class MyService extends Construct {
        MyService(Construct scope, String id) {
            this(scope, id, false);
        }

        MyService(Construct scope, String id, boolean prod) {
            super(scope, id);

            // we might use the prod argument to change how the service is
 configured
```

```
            new ControlPlane(this, "cp");
            new DataPlane(this, "data");
            new Monitoring(this, "mon");
        }
    }

    public static void main(final String argv[]) {
        App app = new App();

        new MyService(app, "beta");
        new MyService(app, "prod", true);

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;

// imagine these stacks declare a bunch of related resources
public class ControlPlane : Stack {
    public ControlPlane(Construct scope, string id=null) : base(scope, id) { }
}

public class DataPlane : Stack {
    public DataPlane(Construct scope, string id=null) : base(scope, id) { }
}

public class Monitoring : Stack
{
    public Monitoring(Construct scope, string id=null) : base(scope, id) { }
}

public class MyService : Construct
{
    public MyService(Construct scope, string id, Boolean prod=false) : base(scope,
 id)
    {
        // we might use the prod argument to change how the service is configured
        new ControlPlane(this, "cp");
        new DataPlane(this, "data");
```

```
        new Monitoring(this, "mon");
    }
}

class Program
{
    static void Main(string[] args)
    {

        var app = new App();
        new MyService(app, "beta");
        new MyService(app, "prod", prod: true);
        app.Synth();
    }
}
```

Go

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type ControlPlaneStackProps struct {
 awscdk.StackProps
}

func NewControlPlaneStack(scope constructs.Construct, id string, props
 *ControlPlaneStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 ControlPlaneStack := awscdk.NewStack(scope, jsii.String(id), &sprops)

 // The code that defines your stack goes here

 return ControlPlaneStack
}
```

```
type DataPlaneStackProps struct {
 awscdk.StackProps
}

func NewDataPlaneStack(scope constructs.Construct, id string, props
 *DataPlaneStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 DataPlaneStack := awscdk.NewStack(scope, jsii.String(id), &sprops)

 // The code that defines your stack goes here

 return DataPlaneStack
}

type MonitoringStackProps struct {
 awscdk.StackProps
}

func NewMonitoringStack(scope constructs.Construct, id string, props
 *MonitoringStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 MonitoringStack := awscdk.NewStack(scope, jsii.String(id), &sprops)

 // The code that defines your stack goes here

 return MonitoringStack
}

type MyServiceStackProps struct {
 awscdk.StackProps
 Prod bool
}

func NewMyServiceStack(scope constructs.Construct, id string, props
 *MyServiceStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
```

```
 }
 MyServiceStack := awscdk.NewStack(scope, jsii.String(id), &sprops)

 NewControlPlaneStack(MyServiceStack, "cp", &ControlPlaneStackProps{
  StackProps: sprops,
 })
 NewDataPlaneStack(MyServiceStack, "data", &DataPlaneStackProps{
  StackProps: sprops,
 })
 NewMonitoringStack(MyServiceStack, "mon", &MonitoringStackProps{
  StackProps: sprops,
 })

 return MyServiceStack
}

func main() {
 defer jsii.Close()

 app := awscdk.NewApp(nil)

 betaProps := MyServiceStackProps{
  StackProps: awscdk.StackProps{
   Env: env(),
  },
  Prod: false,
 }

 NewMyServiceStack(app, "beta", &betaProps)

 prodProps := MyServiceStackProps{
  StackProps: awscdk.StackProps{
   Env: env(),
  },
  Prod: true,
 }

 NewMyServiceStack(app, "prod", &prodProps)

 app.Synth(nil)
}

// ...
```

This AWS CDK app eventually consists of six stacks, three for each environment:

```
$ cdk ls

betacpDA8372D3
betadataE23DB2BA
betamon632BD457
prodcp187264CE
proddataF7378CE5
prodmon631A1083
```

The physical names of the AWS CloudFormation stacks are automatically determined by the AWS CDK based on the stack's construct path in the tree. By default, a stack's name is derived from the construct ID of the `Stack` object. However, you can specify an explicit name by using the `stackName` prop (in Python, `stack_name`), as follows.

Example

TypeScript

```
new MyStack(this, 'not:a:stack:name', { stackName: 'this-is-stack-name' });
```

JavaScript

```
new MyStack(this, 'not:a:stack:name', { stackName: 'this-is-stack-name' });
```

Python

```
MyStack(self, "not:a:stack:name", stack_name="this-is-stack-name")
```

Java

```
new MyStack(this, "not:a:stack:name", StackProps.builder()
    .StackName("this-is-stack-name").build());
```

C#

```
new MyStack(this, "not:a:stack:name", new StackProps
{
    StackName = "this-is-stack-name"
});
```

Working with nested stacks

A *nested stack* is a CDK stack that you create inside another stack, known as the parent stack. You create nested stacks using the `NestedStack` construct.

By using nested stacks, you can organize resources across multiple stacks. Nested stacks also offer a way around the AWS CloudFormation 500-resource limit for stacks. A nested stack counts as only one resource in the stack that contains it. However, it can contain up to 500 resources, including additional nested stacks.

The scope of a nested stack must be a `Stack` or `NestedStack` construct. The nested stack doesn't need to be declared lexically inside its parent stack. It is necessary only to pass the parent stack as the first parameter (`scope`) when instantiating the nested stack. Aside from this restriction, defining constructs in a nested stack works exactly the same as in an ordinary stack.

At synthesis time, the nested stack is synthesized to its own AWS CloudFormation template, which is uploaded to the AWS CDK staging bucket at deployment. Nested stacks are bound to their parent stack and are not treated as independent deployment artifacts. They aren't listed by `cdk list`, and they can't be deployed by `cdk deploy`.

References between parent stacks and nested stacks are automatically translated to stack parameters and outputs in the generated AWS CloudFormation templates, as with any cross-stack reference.

> **Warning**
>
> Changes in security posture are not displayed before deployment for nested stacks. This information is displayed only for top-level stacks.

Introduction to AWS CDK stages

An AWS Cloud Development Kit (AWS CDK) *stage* represents a group of one or more CDK stacks that are configured to deploy together. Use stages to deploy the same grouping of stacks to multiple environments, such as development, testing, and production.

To configure a CDK stage, import and use the `Stage` construct.

The following is a basic example that defines a CDK stage named `MyAppStage`. We add two CDK stacks, named `AppStack` and `DatabaseStack` to our stage. For this example, `AppStack` contains application resources and `DatabaseStack` contains database resources. We then create two instances of `MyAppStage`, for development and production environments:

Example

TypeScript

In `cdk-demo-app/lib/app-stack.ts`:

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

// Define the app stack
export class AppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
    // The code that defines your application goes here
  }
}
```

In `cdk-demo-app/lib/database-stack.ts`:

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

// Define the database stack
export class DatabaseStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
    // The code that defines your database goes here
  }
}
```

In `cdk-demo-app/lib/my-stage.ts`:

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { Stage } from 'aws-cdk-lib';
import { AppStack } from './app-stack';
import { DatabaseStack } from './database-stack';

// Define the stage
export class MyAppStage extends Stage {
```

```
  constructor(scope: Construct, id: string, props?: cdk.StageProps) {
    super(scope, id, props);

    // Add both stacks to the stage
    new AppStack(this, 'AppStack');
    new DatabaseStack(this, 'DatabaseStack');
  }
}
```

In `cdk-demo-app/bin/cdk-demo-app.ts`:

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MyAppStage } from '../lib/my-stage';

// Create a CDK app
const app = new cdk.App();

// Create the development stage
new MyAppStage(app, 'Dev', {
  env: {
    account: '123456789012',
    region: 'us-east-1'
  }
});

// Create the production stage
new MyAppStage(app, 'Prod', {
  env: {
    account: '098765432109',
    region: 'us-east-1'
  }
});
```

JavaScript

In `cdk-demo-app/lib/app-stack.js`:

```
const { Stack } = require('aws-cdk-lib');

class AppStack extends Stack {
  constructor(scope, id, props) {
```

```
    super(scope, id, props);

    // The code that defines your application goes here
  }
}

module.exports = { AppStack }
```

In `cdk-demo-app/lib/database-stack.js`:

```
const { Stack } = require('aws-cdk-lib');

class DatabaseStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // The code that defines your database goes here
  }
}

module.exports = { DatabaseStack }
```

In `cdk-demo-app/lib/my-stage.js`:

```
const { Stage } = require('aws-cdk-lib');
const { AppStack } = require('./app-stack');
const { DatabaseStack } = require('./database-stack');

// Define the stage
class MyAppStage extends Stage {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Add both stacks to the stage
    new AppStack(this, 'AppStack');
    new DatabaseStack(this, 'DatabaseStack');
  }
}

module.exports = { MyAppStage };
```

In `cdk-demo-app/bin/cdk-demo-app.js`:

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { MyAppStage } = require('../lib/my-stage');

// Create the CDK app
const app = new cdk.App();

// Create the development stage
new MyAppStage(app, 'Dev', {
  env: {
    account: '123456789012',
    region: 'us-east-1',
  },
});

// Create the production stage
new MyAppStage(app, 'Prod', {
  env: {
    account: '098765432109',
    region: 'us-east-1',
  },
});
```

Python

In `cdk-demo-app/cdk_demo_app/app_stack.py`:

```
from aws_cdk import Stack
from constructs import Construct

# Define the app stack
class AppStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your application goes here
```

In `cdk-demo-app/cdk_demo_app/database_stack.py`:

```
from aws_cdk import Stack
from constructs import Construct
```

```
# Define the database stack
class DatabaseStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your database goes here
```

In `cdk-demo-app/cdk_demo_app/my_stage.py`:

```
from aws_cdk import Stage
from constructs import Construct
from .app_stack import AppStack
from .database_stack import DatabaseStack

# Define the stage
class MyAppStage(Stage):
    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Add both stacks to the stage
        AppStack(self, "AppStack")
        DatabaseStack(self, "DatabaseStack")
```

In `cdk-demo-app/app.py`:

```
#!/usr/bin/env python3
import os

import aws_cdk as cdk

from cdk_demo_app.my_stage import MyAppStage

#  Create a CDK app
app = cdk.App()

# Create the development stage
MyAppStage(app, 'Dev',
           env=cdk.Environment(account='123456789012', region='us-east-1'),
           )

# Create the production stage
MyAppStage(app, 'Prod',
```

```
            env=cdk.Environment(account='098765432109', region='us-east-1'),
            )

app.synth()
```

Java

In `cdk-demo-app/src/main/java/com/myorg/AppStack.java`:

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class AppStack extends Stack {
    public AppStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public AppStack(final Construct scope, final String id, final StackProps props)
 {
        super(scope, id, props);

        // The code that defines your application goes here
    }
}
```

In `cdk-demo-app/src/main/java/com/myorg/DatabaseStack.java`:

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class DatabaseStack extends Stack {
    public DatabaseStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public DatabaseStack(final Construct scope, final String id, final StackProps
 props) {
```

```
        super(scope, id, props);

        // The code that defines your database goes here
    }
}
```

In `cdk-demo-app/src/main/java/com/myorg/MyAppStage.java`:

```
package com.myorg;

import software.amazon.awscdk.Stage;
import software.amazon.awscdk.StageProps;
import software.constructs.Construct;

// Define the stage
public class MyAppStage extends Stage {
    public MyAppStage(final Construct scope, final String id, final
 software.amazon.awscdk.Environment env) {
        super(scope, id, StageProps.builder().env(env).build());

        // Add both stacks to the stage
        new AppStack(this, "AppStack");
        new DatabaseStack(this, "DatabaseStack");
    }
}
```

In `cdk-demo-app/src/main/java/com/myorg/CdkDemoAppApp.java`:

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class CdkDemoAppApp {
    public static void main(final String[] args) {

        // Create a CDK app
        App app = new App();

        // Create the development stage
```

```
        new MyAppStage(app, "Dev", Environment.builder()
                .account("123456789012")
                .region("us-east-1")
                .build());

        // Create the production stage
        new MyAppStage(app, "Prod", Environment.builder()
        .account("098765432109")
        .region("us-east-1")
        .build());

        app.synth();
    }
}
```

C#

In `cdk-demo-app/src/CdkDemoApp/AppStack.cs`:

```
using Amazon.CDK;
using Constructs;

namespace CdkDemoApp
{
    public class AppStack : Stack
    {
        internal AppStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
        {
            // The code that defines your application goes here
        }
    }
}
```

In `cdk-demo-app/src/CdkDemoApp/DatabaseStack.cs`:

```
using Amazon.CDK;
using Constructs;

namespace CdkDemoApp
{
    public class DatabaseStack : Stack
    {
```

```
        internal DatabaseStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // The code that defines your database goes here
        }
    }
}
```

In `cdk-demo-app/src/CdkDemoApp/MyAppStage.cs`:

```
using Amazon.CDK;
using Constructs;

namespace CdkDemoApp
{
    // Define the stage
    public class MyAppStage : Stage
    {
        internal MyAppStage(Construct scope, string id, Environment env) :
 base(scope, id, new StageProps { Env = env })
        {
            // Add both stacks to the stage
            new AppStack(this, "AppStack");
            new DatabaseStack(this, "DatabaseStack");
        }
    }
}
```

In `cdk-demo-app/src/CdkDemoApp/program.cs`:

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace CdkDemoApp
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            // Create a CDK app
            var app = new App();
```

```
            // Create the development stage
            new MyAppStage(app, "Dev", new Amazon.CDK.Environment
            {
                Account = "123456789012",
                Region = "us-east-1"
            });

            // Create the production stage
            new MyAppStage(app, "Prod", new Amazon.CDK.Environment
            {
                Account = "098765432109",
                Region = "us-east-1"
            });

            app.Synth();
        }
    }
}
```

Go

In `cdk-demo-app/cdk-demo-app.go`:

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

// Define the app stack
type AppStackProps struct {
 awscdk.StackProps
}

func NewAppStack(scope constructs.Construct, id string, props *AppStackProps)
 awscdk.Stack {
 stack := awscdk.NewStack(scope, &id, &props.StackProps)

 // The code that defines your application goes here

 return stack
```

```
}

// Define the database stack
type DatabaseStackProps struct {
 awscdk.StackProps
}

func NewDatabaseStack(scope constructs.Construct, id string, props
 *DatabaseStackProps) awscdk.Stack {
 stack := awscdk.NewStack(scope, &id, &props.StackProps)

 // The code that defines your database goes here

 return stack
}

// Define the stage
type MyAppStageProps struct {
 awscdk.StageProps
}

func NewMyAppStage(scope constructs.Construct, id string, props *MyAppStageProps)
 awscdk.Stage {
 stage := awscdk.NewStage(scope, &id, &props.StageProps)

 // Add both stacks to the stage
 NewAppStack(stage, "AppStack", &AppStackProps{
  StackProps: awscdk.StackProps{
   Env: props.Env,
  },
 })

 NewDatabaseStack(stage, "DatabaseStack", &DatabaseStackProps{
  StackProps: awscdk.StackProps{
   Env: props.Env,
  },
 })

 return stage
}

func main() {
 defer jsii.Close()
```

```
 // Create a CDK app
 app := awscdk.NewApp(nil)

 // Create the development stage
 NewMyAppStage(app, "Dev", &MyAppStageProps{
  StageProps: awscdk.StageProps{
   Env: &awscdk.Environment{
    Account: jsii.String("123456789012"),
    Region:  jsii.String("us-east-1"),
   },
  },
 })

 // Create the production stage
 NewMyAppStage(app, "Prod", &MyAppStageProps{
  StageProps: awscdk.StageProps{
   Env: &awscdk.Environment{
    Account: jsii.String("098765432109"),
    Region:  jsii.String("us-east-1"),
   },
  },
 })

 app.Synth(nil)
}

func env() *awscdk.Environment {
 return nil
}
```

When we run `cdk synth`, two cloud assemblies are created in `cdk.out`. These two cloud assemblies contain the synthesized AWS CloudFormation template and assets for each stage. The following is snippet of our project directory:

Example

TypeScript

```
cdk-demo-app
### bin
#   ### cdk-demo-app.ts
### cdk.out
```

```
#   ### assembly-Dev
#   #   ### DevAppStack<unique-hash>.assets.json
#   #   ### DevAppStack<unique-hash>.template.json
#   #   ### DevDatabaseStack<unique-hash>.assets.json
#   #   ### DevDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
#   ### assembly-Prod
#   #   ### ProdAppStack<unique-hash>.assets.json
#   #   ### ProdAppStack<unique-hash>.template.json
#   #   ### ProdDatabaseStack<unique-hash>.assets.json
#   #   ### ProdDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
### lib
    ### app-stack.ts
    ### database-stack.ts
    ### my-stage.ts
```

JavaScript

```
cdk-demo-app
### bin
#   ### cdk-demo-app.js
### cdk.out
#   ### assembly-Dev
#   #   ### DevAppStack<unique-hash>.assets.json
#   #   ### DevAppStack<unique-hash>.template.json
#   #   ### DevDatabaseStack<unique-hash>.assets.json
#   #   ### DevDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
#   ### assembly-Prod
#   #   ### ProdAppStack<unique-hash>.assets.json
#   #   ### ProdAppStack<unique-hash>.template.json
#   #   ### ProdDatabaseStack<unique-hash>.assets.json
#   #   ### ProdDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
### lib
    ### app-stack.js
    ### database-stack.js
    ### my-stage.js
```

Python

```
cdk-demo-app
### app.py
### cdk.out
#   ### assembly-Dev
#   #   ### DevAppStack<unique-hash>.assets.json
#   #   ### DevAppStack<unique-hash>.template.json
#   #   ### DevDatabaseStack<unique-hash>.assets.json
#   #   ### DevDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
#   ### assembly-Prod
#   #   ### ProdAppStack<unique-hash>.assets.json
#   #   ### ProdAppStack<unique-hash>.template.json
#   #   ### ProdDatabaseStack<unique-hash>.assets.json
#   #   ### ProdDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
#   ### cdk.out
#   ### manifest.json
#   ### tree.json
### cdk_demo_app
    ### __init__.py
    ### app_stack.py
    ### database_stack.py
    ### my_stage.py
```

Java

```
cdk-demo-app
### cdk.out
#   ### assembly-Dev
#   #   ### DevAppStack<unique-hash>.assets.json
#   #   ### DevAppStack<unique-hash>.template.json
#   #   ### DevDatabaseStack<unique-hash>.assets.json
#   #   ### DevDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
#   ### assembly-Prod
#   #   ### ProdAppStack<unique-hash>.assets.json
#   #   ### ProdAppStack<unique-hash>.template.json
#   #   ### ProdDatabaseStack<unique-hash>.assets.json
```

```
#   #   ### ProdDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
#   ### cdk.out
#   ### manifest.json
#   ### tree.json
### src
    ### main
        ### java
            ### com
                ### myorg
                    ### AppStack.java
                    ### CdkDemoAppApp.java
                    ### DatabaseStack.java
                    ### MyAppStage.java
```

C#

```
cdk-demo-app
### cdk.out
#   ### assembly-Dev
#   #   ### DevAppStack<unique-hash>.assets.json
#   #   ### DevAppStack<unique-hash>.template.json
#   #   ### DevDatabaseStack<unique-hash>.assets.json
#   #   ### DevDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
#   ### assembly-Prod
#   #   ### ProdAppStack<unique-hash>.assets.json
#   #   ### ProdAppStack<unique-hash>.template.json
#   #   ### ProdDatabaseStack<unique-hash>.assets.json
#   #   ### ProdDatabaseStack<unique-hash>.template.json
#   #   ### cdk.out
#   #   ### manifest.json
#   ### cdk.out
#   ### manifest.json
#   ### tree.json
### src
    ### CdkDemoApp
        ### AppStack.cs
        ### DatabaseStack.cs
        ### MyAppStage.cs
        ### Program.cs
```

Go

```
cdk-demo-app
### cdk-demo-app.go
### cdk.out
    ### assembly-Dev
    #   ### DevAppStack<unique-hash>.assets.json
    #   ### DevAppStack<unique-hash>.template.json
    #   ### DevDatabaseStack<unique-hash>.assets.json
    #   ### DevDatabaseStack<unique-hash>.template.json
    #   ### cdk.out
    #   ### manifest.json
    ### assembly-Prod
    #   ### ProdAppStack<unique-hash>.assets.json
    #   ### ProdAppStack<unique-hash>.template.json
    #   ### ProdDatabaseStack<unique-hash>.assets.json
    #   ### ProdDatabaseStack<unique-hash>.template.json
    #   ### cdk.out
    #   ### manifest.json
    ### cdk.out
    ### manifest.json
    ### tree.json
```

When we list our stacks with `cdk list`, we see a total of four stacks:

```
$ cdk list
Dev/AppStack (Dev-AppStack)
Dev/DatabaseStack (Dev-DatabaseStack)
Prod/AppStack (Prod-AppStack)
Prod/DatabaseStack (Prod-DatabaseStack)
```

To deploy a specific stage, we run `cdk deploy` and provide the stacks to deploy. The following is an example that uses the `/*` wildcard to deploy both stacks in our `Dev` stage:

```
$ cdk deploy <"Dev/*">

#  Synthesis time: 3.18s

Dev/AppStack (Dev-AppStack)
Dev/AppStack (Dev-AppStack): deploying... [1/2]
```

```
 #  Dev/AppStack (Dev-AppStack)

#  Deployment time: 1.11s

Stack ARN:
...

#  Total time: 4.29s

Dev/DatabaseStack (Dev-DatabaseStack)
Dev/DatabaseStack (Dev-DatabaseStack): deploying... [2/2]

 #  Dev/DatabaseStack (Dev-DatabaseStack)

#  Deployment time: 1.09s

Stack ARN:
...

#  Total time: 4.27s
```

AWS CDK Constructs

Constructs are the basic building blocks of AWS Cloud Development Kit (AWS CDK) applications. A construct is a component within your application that represents one or more AWS CloudFormation resources and their configuration. You build your application, piece by piece, by importing and configuring constructs.

Import and use constructs

Constructs are classes that you import into your CDK applications from the AWS Construct Library. You can also create and distribute your own constructs, or use constructs created by third-party developers.

Constructs are part of the Construct Programming Model (CPM). They are available to use with other tools such as CDK for Terraform (CDKtf), CDK for Kubernetes (CDK8s), and Projen.

Numerous third parties have also published constructs compatible with the AWS CDK. Visit Construct Hub to explore the AWS CDK construct partner ecosystem.

Construct levels

Constructs from the AWS Construct Library are categorized into three levels. Each level offers an increasing level of abstraction. The higher the abstraction, the easier to configure, requiring less expertise. The lower the abstraction, the more customization available, requiring more expertise.

Level 1 (L1) constructs

L1 constructs, also known as *CFN resources*, are the lowest-level construct and offer no abstraction. Each L1 construct maps directly to a single AWS CloudFormation resource. With L1 constructs, you import a construct that represents a specific AWS CloudFormation resource. You then define the resource's properties within your construct instance.

L1 constructs are great to use when you are familiar with AWS CloudFormation and need complete control over defining your AWS resource properties.

In the AWS Construct Library, L1 constructs are named starting with `Cfn`, followed by an identifier for the AWS CloudFormation resource that it represents. For example, the `CfnBucket` construct is an L1 construct that represents an `AWS::S3::Bucket` AWS CloudFormation resource.

L1 constructs are generated from the AWS CloudFormation resource specification. If a resource exists in AWS CloudFormation, it'll be available in the AWS CDK as an L1 construct. New resources or properties may take up to a week to become available in the AWS Construct Library. For more information, see AWS resource and property types reference in the *AWS CloudFormation User Guide*.

Level 2 (L2) constructs

L2 constructs, also known as *curated* constructs, are thoughtfully developed by the CDK team and are usually the most widely used construct type. L2 constructs map directly to single AWS CloudFormation resources, similar to L1 constructs. Compared to L1 constructs, L2 constructs provide a higher-level abstraction through an intuitive intent-based API. L2 constructs include sensible default property configurations, best practice security policies, and generate a lot of the boilerplate code and glue logic for you.

L2 constructs also provide helper methods for most resources that make it simpler and quicker to define properties, permissions, event-based interactions between resources, and more.

The `s3.Bucket` class is an example of an L2 construct for an Amazon Simple Storage Service (Amazon S3) bucket resource.

The AWS Construct Library contains L2 constructs that are designated stable and ready for production use. For L2 constructs under development, they are designated as experimental and offered in a separate module.

Level 3 (L3) constructs

L3 constructs, also known as *patterns*, are the highest-level of abstraction. Each L3 construct can contain a collection of resources that are configured to work together to accomplish a specific task or service within your application. L3 constructs are used to create entire AWS architectures for particular use cases in your application.

To provide complete system designs, or substantial parts of a larger system, L3 constructs offer opinionated default property configurations. They are built around a particular approach toward solving a problem and providing a solution. With L3 constructs, you can create and configure multiple resources quickly, with the fewest amount of input and code.

The `ecsPatterns.ApplicationLoadBalancedFargateService` class is an example of an L3 construct that represents an AWS Fargate service running on an Amazon Elastic Container Service (Amazon ECS) cluster and fronted by an application load balancer.

Similar to L2 constructs, L3 constructs that are ready for production use are included in the AWS Construct Library. Those under development are offered in separate modules.

Defining constructs

Composition

Composition is the key pattern for defining higher-level abstractions through constructs. A high-level construct can be composed from any number of lower-level constructs. From a bottom-up perspective, you use constructs to organize the individual AWS resources that you want to deploy. You use whatever abstractions are convenient for your purpose, with as many levels as you need.

With composition, you define reusable components and share them like any other code. For example, a team can define a construct that implements the company's best practice for an Amazon DynamoDB table, including backup, global replication, automatic scaling, and monitoring. The team can share the construct internally with other teams, or publicly.

Teams can use constructs like any other library package. When the library is updated, developers get access to the new version's improvements and bug fixes, similar to any other code library.

Initialization

Constructs are implemented in classes that extend the `Construct` base class. You define a construct by instantiating the class. All constructs take three parameters when they are initialized:

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

  Identifiers need only be unique within a scope. This lets you instantiate and reuse constructs without concern for the constructs and identifiers they might contain, and enables composing constructs into higher-level abstractions. In addition, scopes make it possible to refer to groups of constructs all at once. Examples include for tagging, or specifying where the constructs will be deployed.
- **props** – A set of properties or keyword arguments, depending on the language, that define the construct's initial configuration. Higher-level constructs provide more defaults, and if all prop elements are optional, you can omit the props parameter completely.

Configuration

Most constructs accept `props` as their third argument (or in Python, keyword arguments), a name/value collection that defines the construct's configuration. The following example defines a bucket with AWS Key Management Service (AWS KMS) encryption and static website hosting enabled. Since it does not explicitly specify an encryption key, the `Bucket` construct defines a new `kms.Key` and associates it with the bucket.

Example

TypeScript

```
new s3.Bucket(this, 'MyEncryptedBucket', {
  encryption: s3.BucketEncryption.KMS,
  websiteIndexDocument: 'index.html'
```

```
});
```

JavaScript

```
new s3.Bucket(this, 'MyEncryptedBucket', {
  encryption: s3.BucketEncryption.KMS,
  websiteIndexDocument: 'index.html'
});
```

Python

```
s3.Bucket(self, "MyEncryptedBucket", encryption=s3.BucketEncryption.KMS,
    website_index_document="index.html")
```

Java

```
Bucket.Builder.create(this, "MyEncryptedBucket")
        .encryption(BucketEncryption.KMS_MANAGED)
        .websiteIndexDocument("index.html").build();
```

C#

```
new Bucket(this, "MyEncryptedBucket", new BucketProps
{
    Encryption = BucketEncryption.KMS_MANAGED,
    WebsiteIndexDocument = "index.html"
});
```

Go

```
 awss3.NewBucket(stack, jsii.String("MyEncryptedBucket"), &awss3.BucketProps{
  Encryption: awss3.BucketEncryption_KMS,
  WebsiteIndexDocument: jsii.String("index.html"),
 })
```

Interacting with constructs

Constructs are classes that extend the base Construct class. After you instantiate a construct, the construct object exposes a set of methods and properties that let you interact with the construct and pass it around as a reference to other parts of the system.

The AWS CDK framework doesn't put any restrictions on the APIs of constructs. Authors can define any API they want. However, the AWS constructs that are included with the AWS Construct Library, such as `s3.Bucket`, follow guidelines and common patterns. This provides a consistent experience across all AWS resources.

Most AWS constructs have a set of grant methods that you can use to grant AWS Identity and Access Management (IAM) permissions on that construct to a principal. The following example grants the IAM group `data-science` permission to read from the Amazon S3 bucket `raw-data`.

Example

TypeScript

```
const rawData = new s3.Bucket(this, 'raw-data');
const dataScience = new iam.Group(this, 'data-science');
rawData.grantRead(dataScience);
```

JavaScript

```
const rawData = new s3.Bucket(this, 'raw-data');
const dataScience = new iam.Group(this, 'data-science');
rawData.grantRead(dataScience);
```

Python

```
raw_data = s3.Bucket(self, 'raw-data')
data_science = iam.Group(self, 'data-science')
raw_data.grant_read(data_science)
```

Java

```
Bucket rawData = new Bucket(this, "raw-data");
Group dataScience = new Group(this, "data-science");
rawData.grantRead(dataScience);
```

C#

```
var rawData = new Bucket(this, "raw-data");
var dataScience = new Group(this, "data-science");
rawData.GrantRead(dataScience);
```

Go

```
rawData := awss3.NewBucket(stack, jsii.String("raw-data"), nil)
dataScience := awsiam.NewGroup(stack, jsii.String("data-science"), nil)
rawData.GrantRead(dataScience, nil)
```

Another common pattern is for AWS constructs to set one of the resource's attributes from data supplied elsewhere. Attributes can include Amazon Resource Names (ARNs), names, or URLs.

The following code defines an AWS Lambda function and associates it with an Amazon Simple Queue Service (Amazon SQS) queue through the queue's URL in an environment variable.

Example

TypeScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

JavaScript

```
const jobsQueue = new sqs.Queue(this, 'jobs');
const createJobLambda = new lambda.Function(this, 'create-job', {
  runtime: lambda.Runtime.NODEJS_18_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset('./create-job-lambda-code'),
  environment: {
    QUEUE_URL: jobsQueue.queueUrl
  }
});
```

Python

```
jobs_queue = sqs.Queue(self, "jobs")
```

```
create_job_lambda = lambda_.Function(self, "create-job",
    runtime=lambda_.Runtime.NODEJS_18_X,
    handler="index.handler",
    code=lambda_.Code.from_asset("./create-job-lambda-code"),
    environment=dict(
        QUEUE_URL=jobs_queue.queue_url
    )
)
```

Java

```
final Queue jobsQueue = new Queue(this, "jobs");
Function createJobLambda = Function.Builder.create(this, "create-job")
                .handler("index.handler")
                .code(Code.fromAsset("./create-job-lambda-code"))
                .environment(java.util.Map.of(   // Map.of is Java 9 or later
                    "QUEUE_URL", jobsQueue.getQueueUrl()))
                .build();
```

C#

```
var jobsQueue = new Queue(this, "jobs");
var createJobLambda = new Function(this, "create-job", new FunctionProps
{
    Runtime = Runtime.NODEJS_18_X,
    Handler = "index.handler",
    Code = Code.FromAsset(@".\create-job-lambda-code"),
    Environment = new Dictionary<string, string>
    {
        ["QUEUE_URL"] = jobsQueue.QueueUrl
    }
});
```

Go

```
 createJobLambda := awslambda.NewFunction(stack, jsii.String("create-job"),
 &awslambda.FunctionProps{
  Runtime: awslambda.Runtime_NODEJS_18_X(),
  Handler: jsii.String("index.handler"),
  Code:    awslambda.Code_FromAsset(jsii.String(".\\create-job-lambda-code"), nil),
  Environment: &map[string]*string{
   "QUEUE_URL": jsii.String(*jobsQueue.QueueUrl()),
```

```
  },
 })
```

For information about the most common API patterns in the AWS Construct Library, see Resources and the AWS CDK.

The app and stack construct

The `App` and `Stack` classes from the AWS Construct Library are unique constructs. Compared to other constructs, they don't configure AWS resources on their own. Instead, they are used to provide context for your other constructs. All constructs that represent AWS resources must be defined, directly or indirectly, within the scope of a `Stack` construct. `Stack` constructs are defined within the scope of an App construct.

To learn more about CDK apps, see AWS CDK apps. To learn more about CDK stacks, see Introduction to AWS CDK stacks.

The following example defines an app with a single stack. Within the stack, an L2 construct is used to configure an Amazon S3 bucket resource.

Example

TypeScript

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import * as s3 from 'aws-cdk-lib/aws-s3';

class HelloCdkStack extends Stack {
  constructor(scope: App, id: string, props?: StackProps) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

JavaScript

```
const { App , Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyFirstBucket', {
      versioned: true
    });
  }
}

const app = new App();
new HelloCdkStack(app, "HelloCdkStack");
```

Python

```
from aws_cdk import App, Stack
import aws_cdk.aws_s3 as s3
from constructs import Construct

class HelloCdkStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        s3.Bucket(self, "MyFirstBucket", versioned=True)

app = App()
HelloCdkStack(app, "HelloCdkStack")
```

Java

Stack defined in `HelloCdkStack.java` file:

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.s3.*;
```

```
public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
            .versioned(true).build();
    }
}
```

App defined in `HelloCdkApp.java` file:

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.StackProps;

public class HelloCdkApp {
    public static void main(final String[] args) {
        App app = new App();

        new HelloCdkStack(app, "HelloCdkStack", StackProps.builder()
                .build());

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace HelloCdkApp
{
    internal static class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
```

```
            new HelloCdkStack(app, "HelloCdkStack");
            app.Synth();
        }
    }

    public class HelloCdkStack : Stack
    {
        public HelloCdkStack(Construct scope, string id, IStackProps props=null) :
 base(scope, id, props)
        {
            new Bucket(this, "MyFirstBucket", new BucketProps { Versioned = true });
        }
    }
}
```

Go

```
func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })

 return stack
}
```

Working with constructs

Working with L1 constructs

L1 constructs map directly to individual AWS CloudFormation resources. You must provide the resource’s required configuration.

In this example, we create a `bucket` object using the `CfnBucket` L1 construct:

Example

TypeScript

```
const bucket = new s3.CfnBucket(this, "amzn-s3-demo-bucket", {
  bucketName: "amzn-s3-demo-bucket"
});
```

JavaScript

```
const bucket = new s3.CfnBucket(this, "amzn-s3-demo-bucket", {
  bucketName: "amzn-s3-demo-bucket"
});
```

Python

```
bucket = s3.CfnBucket(self, "amzn-s3-demo-bucket", bucket_name="amzn-s3-demo-
bucket")
```

Java

```
CfnBucket bucket = new CfnBucket.Builder().bucketName("amzn-s3-demo-
bucket").build();
```

C#

```
var bucket = new CfnBucket(this, "amzn-s3-demo-bucket", new CfnBucketProps
{
    BucketName= "amzn-s3-demo-bucket"
});
```

Go

```
 awss3.NewCfnBucket(stack, jsii.String("amzn-s3-demo-bucket"),
 &awss3.CfnBucketProps{
  BucketName: jsii.String("amzn-s3-demo-bucket"),
 })
```

Construct properties that aren't simple Booleans, strings, numbers, or containers are handled differently in the supported languages.

Example

TypeScript

```
const bucket = new s3.CfnBucket(this, "amzn-s3-demo-bucket", {
  bucketName: "amzn-s3-demo-bucket",
  corsConfiguration: {
    corsRules: [{
          allowedOrigins: ["*"],
          allowedMethods: ["GET"]
    }]
  }
});
```

JavaScript

```
const bucket = new s3.CfnBucket(this, "amzn-s3-demo-bucket", {
  bucketName: "amzn-s3-demo-bucket",
  corsConfiguration: {
    corsRules: [{
          allowedOrigins: ["*"],
          allowedMethods: ["GET"]
    }]
  }
});
```

Python

In Python, these properties are represented by types defined as inner classes of the L1 construct. For example, the optional property `cors_configuration` of a `CfnBucket` requires a wrapper of type `CfnBucket.CorsConfigurationProperty`. Here we are defining `cors_configuration` on a `CfnBucket` instance.

```
bucket = CfnBucket(self, "amzn-s3-demo-bucket", bucket_name="amzn-s3-demo-bucket",
    cors_configuration=CfnBucket.CorsConfigurationProperty(
        cors_rules=[CfnBucket.CorsRuleProperty(
            allowed_origins=["*"],
            allowed_methods=["GET"]
        )]
    )
)
```

Java

In Java, these properties are represented by types defined as inner classes of the L1 construct. For example, the optional property `corsConfiguration` of a `CfnBucket` requires a wrapper of type `CfnBucket.CorsConfigurationProperty`. Here we are defining `corsConfiguration` on a `CfnBucket` instance.

```
CfnBucket bucket = CfnBucket.Builder.create(this, "amzn-s3-demo-bucket")
                        .bucketName("amzn-s3-demo-bucket")
                        .corsConfiguration(new
 CfnBucket.CorsConfigurationProperty.Builder()
                            .corsRules(Arrays.asList(new
 CfnBucket.CorsRuleProperty.Builder()
                                .allowedOrigins(Arrays.asList("*"))
                                .allowedMethods(Arrays.asList("GET"))
                                .build()))
                            .build())
                        .build();
```

C#

In C#, these properties are represented by types defined as inner classes of the L1 construct. For example, the optional property `CorsConfiguration` of a `CfnBucket` requires a wrapper of type `CfnBucket.CorsConfigurationProperty`. Here we are defining `CorsConfiguration` on a `CfnBucket` instance.

```
var bucket = new CfnBucket(this, "amzn-s3-demo-bucket", new CfnBucketProps
{
    BucketName = "amzn-s3-demo-bucket",
    CorsConfiguration = new CfnBucket.CorsConfigurationProperty
    {
        CorsRules = new object[] {
            new CfnBucket.CorsRuleProperty
            {
                AllowedOrigins = new string[] { "*" },
                AllowedMethods = new string[] { "GET" },
            }
        }
    }
});
```

Go

In Go, these types are named using the name of the L1 construct, an underscore, and the property name. For example, the optional property `CorsConfiguration` of a `CfnBucket` requires a wrapper of type `CfnBucket_CorsConfigurationProperty`. Here we are defining `CorsConfiguration` on a `CfnBucket` instance.

```
awss3.NewCfnBucket(stack, jsii.String("amzn-s3-demo-bucket"),
&awss3.CfnBucketProps{
 BucketName: jsii.String("amzn-s3-demo-bucket"),
 CorsConfiguration: &awss3.CfnBucket_CorsConfigurationProperty{
  CorsRules: []awss3.CorsRule{
   awss3.CorsRule{
    AllowedOrigins: jsii.Strings("*"),
    AllowedMethods: &[]awss3.HttpMethods{"GET"},
   },
  },
 },
})
```

Important

You can't use L2 property types with L1 constructs, or vice versa. When working with L1 constructs, always use the types defined for the L1 construct you're using. Do not use types from other L1 constructs (some may have the same name, but they are not the same type). Some of our language-specific API references currently have errors in the paths to L1 property types, or don't document these classes at all. We hope to fix this soon. In the meantime, remember that such types are always inner classes of the L1 construct they are used with.

Working with L2 constructs

In the following example, we define an Amazon S3 bucket by creating an object from the [`Bucket`] L2 construct:

Example

TypeScript

```
import * as s3 from 'aws-cdk-lib/aws-s3';

// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true
});
```

JavaScript

```
const s3 = require('aws-cdk-lib/aws-s3');

// "this" is HelloCdkStack
new s3.Bucket(this, 'MyFirstBucket', {
  versioned: true
});
```

Python

```
import aws_cdk.aws_s3 as s3

# "self" is HelloCdkStack
s3.Bucket(self, "MyFirstBucket", versioned=True)
```

Java

```
import software.amazon.awscdk.services.s3.*;

public class HelloCdkStack extends Stack {
    public HelloCdkStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "MyFirstBucket")
                .versioned(true).build();
```

```
    }
}
```

C#

```
using Amazon.CDK.AWS.S3;

// "this" is HelloCdkStack
new Bucket(this, "MyFirstBucket", new BucketProps
{
    Versioned = true
});
```

Go

```
import (
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/jsii-runtime-go"
)

// stack is HelloCdkStack
awss3.NewBucket(stack, jsii.String("MyFirstBucket"), &awss3.BucketProps{
  Versioned: jsii.Bool(true),
 })
```

`MyFirstBucket` is not the name of the bucket that AWS CloudFormation creates. It is a logical identifier given to the new construct within the context of your CDK app. The physicalName value will be used to name the AWS CloudFormation resource.

Working with third-party constructs

Construct Hub is a resource to help you discover additional constructs from AWS, third parties, and the open-source CDK community.

Writing your own constructs

In addition to using existing constructs, you can also write your own constructs and let anyone use them in their apps. All constructs are equal in the AWS CDK. Constructs from the AWS Construct Library are treated the same as a construct from a third-party library published via NPM, Maven, or PyPI. Constructs published to your company's internal package repository are also treated in the same way.

To declare a new construct, create a class that extends the Construct base class, in the `constructs` package, then follow the pattern for initializer arguments.

The following example shows how to declare a construct that represents an Amazon S3 bucket. The S3 bucket sends an Amazon Simple Notification Service (Amazon SNS) notification every time someone uploads a file into it.

Example

TypeScript

```
export interface NotifyingBucketProps {
  prefix?: string;
}

export class NotifyingBucket extends Construct {
  constructor(scope: Construct, id: string, props: NotifyingBucketProps = {}) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    const topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic),
      { prefix: props.prefix });
  }
}
```

JavaScript

```
class NotifyingBucket extends Construct {
  constructor(scope, id, props = {}) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    const topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(topic),
      { prefix: props.prefix });
  }
}

module.exports = { NotifyingBucket }
```

Python

```
class NotifyingBucket(Construct):

    def __init__(self, scope: Construct, id: str, *, prefix=None):
        super().__init__(scope, id)
        bucket = s3.Bucket(self, "bucket")
        topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(s3notify.SnsDestination(topic),
            s3.NotificationKeyFilter(prefix=prefix))
```

Java

```
public class NotifyingBucket extends Construct {

    public NotifyingBucket(final Construct scope, final String id) {
        this(scope, id, null, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props) {
        this(scope, id, props, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final String
 prefix) {
        this(scope, id, null, prefix);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props, final String prefix) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "bucket");
        Topic topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
     }
}
```

C#

```
public class NotifyingBucketProps : BucketProps
{
    public string Prefix { get; set; }
}

public class NotifyingBucket : Construct
{
    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props =
 null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        var topic = new Topic(this, "topic");
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new
 NotificationKeyFilter
        {
            Prefix = props?.Prefix
        });
    }
}
```

Go

```
type NotifyingBucketProps struct {
 awss3.BucketProps
 Prefix *string
}

func NewNotifyingBucket(scope constructs.Construct, id *string, props
 *NotifyingBucketProps) awss3.Bucket {
 var bucket awss3.Bucket
 if props == nil {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), nil)
 } else {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), &props.BucketProps)
 }
 topic := awssns.NewTopic(scope, jsii.String(*id+"Topic"), nil)
 if props == nil {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic))
 } else {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic),
 &awss3.NotificationKeyFilter{
```

```
    Prefix: props.Prefix,
   })
  }
  return bucket
}
```

> **Note**
>
> Our `NotifyingBucket` construct inherits not from `Bucket` but rather from `Construct`. We are using composition, not inheritance, to bundle an Amazon S3 bucket and an Amazon SNS topic together. In general, composition is preferred over inheritance when developing AWS CDK constructs.

The `NotifyingBucket` constructor has a typical construct signature: `scope`, `id`, and `props`. The last argument, `props`, is optional (gets the default value `{}`) because all props are optional. (The base `Construct` class does not take a `props` argument.) You could define an instance of this construct in your app without `props`, for example:

Example

TypeScript

```
new NotifyingBucket(this, 'MyNotifyingBucket');
```

JavaScript

```
new NotifyingBucket(this, 'MyNotifyingBucket');
```

Python

```
NotifyingBucket(self, "MyNotifyingBucket")
```

Java

```
new NotifyingBucket(this, "MyNotifyingBucket");
```

C#

```
new NotifyingBucket(this, "MyNotifyingBucket");
```

Go

```
NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"), nil)
```

Or you could use `props` (in Java, an additional parameter) to specify the path prefix to filter on, for example:

Example

TypeScript

```
new NotifyingBucket(this, 'MyNotifyingBucket', { prefix: 'images/' });
```

JavaScript

```
new NotifyingBucket(this, 'MyNotifyingBucket', { prefix: 'images/' });
```

Python

```
NotifyingBucket(self, "MyNotifyingBucket", prefix="images/")
```

Java

```
new NotifyingBucket(this, "MyNotifyingBucket", "/images");
```

C#

```
new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
    Prefix = "/images"
});
```

Go

```
NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"), &NotifyingBucketProps{
 Prefix: jsii.String("images/"),
```

```
})
```

Typically, you would also want to expose some properties or methods on your constructs. It's not very useful to have a topic hidden behind your construct, because users of your construct aren't able to subscribe to it. Adding a `topic` property lets consumers access the inner topic, as shown in the following example:

Example

TypeScript

```
export class NotifyingBucket extends Construct {
  public readonly topic: sns.Topic;

  constructor(scope: Construct, id: string, props: NotifyingBucketProps) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    this.topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic),
 { prefix: props.prefix });
  }
}
```

JavaScript

```
class NotifyingBucket extends Construct {

  constructor(scope, id, props) {
    super(scope, id);
    const bucket = new s3.Bucket(this, 'bucket');
    this.topic = new sns.Topic(this, 'topic');
    bucket.addObjectCreatedNotification(new s3notify.SnsDestination(this.topic),
 { prefix: props.prefix });
  }
}

module.exports = { NotifyingBucket };
```

Python

```
class NotifyingBucket(Construct):
```

```
    def __init__(self, scope: Construct, id: str, *, prefix=None, **kwargs):
        super().__init__(scope, id)
        bucket = s3.Bucket(self, "bucket")
        self.topic = sns.Topic(self, "topic")
        bucket.add_object_created_notification(s3notify.SnsDestination(self.topic),
            s3.NotificationKeyFilter(prefix=prefix))
```

Java

```
public class NotifyingBucket extends Construct {

    public Topic topic = null;

    public NotifyingBucket(final Construct scope, final String id) {
        this(scope, id, null, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props) {
        this(scope, id, props, null);
    }

    public NotifyingBucket(final Construct scope, final String id, final String
 prefix) {
        this(scope, id, null, prefix);
    }

    public NotifyingBucket(final Construct scope, final String id, final BucketProps
 props, final String prefix) {
        super(scope, id);

        Bucket bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
        if (prefix != null)
            bucket.addObjectCreatedNotification(new SnsDestination(topic),
                NotificationKeyFilter.builder().prefix(prefix).build());
     }
}
```

C#

```
public class NotifyingBucket : Construct
```

```
{
    public readonly Topic topic;

    public NotifyingBucket(Construct scope, string id, NotifyingBucketProps props =
 null) : base(scope, id)
    {
        var bucket = new Bucket(this, "bucket");
        topic = new Topic(this, "topic");
        bucket.AddObjectCreatedNotification(new SnsDestination(topic), new
 NotificationKeyFilter
        {
            Prefix = props?.Prefix
        });
    }
}
```

Go

To do this in Go, we'll need a little extra plumbing. Our original `NewNotifyingBucket` function returned an `awss3.Bucket`. We'll need to extend `Bucket` to include a `topic` member by creating a `NotifyingBucket` struct. Our function will then return this type.

```
type NotifyingBucket struct {
 awss3.Bucket
 topic awssns.Topic
}

func NewNotifyingBucket(scope constructs.Construct, id *string, props
 *NotifyingBucketProps) NotifyingBucket {
 var bucket awss3.Bucket
 if props == nil {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), nil)
 } else {
  bucket = awss3.NewBucket(scope, jsii.String(*id+"Bucket"), &props.BucketProps)
 }
 topic := awssns.NewTopic(scope, jsii.String(*id+"Topic"), nil)
 if props == nil {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic))
 } else {
  bucket.AddObjectCreatedNotification(awss3notifications.NewSnsDestination(topic),
 &awss3.NotificationKeyFilter{
   Prefix: props.Prefix,
  })
```

```
 }
 var nbucket NotifyingBucket
 nbucket.Bucket = bucket
 nbucket.topic = topic
 return nbucket
}
```

Now, consumers can subscribe to the topic, for example:

Example

TypeScript

```
const queue = new sqs.Queue(this, 'NewImagesQueue');
const images = new NotifyingBucket(this, '/images');
images.topic.addSubscription(new sns_sub.SqsSubscription(queue));
```

JavaScript

```
const queue = new sqs.Queue(this, 'NewImagesQueue');
const images = new NotifyingBucket(this, '/images');
images.topic.addSubscription(new sns_sub.SqsSubscription(queue));
```

Python

```
queue = sqs.Queue(self, "NewImagesQueue")
images = NotifyingBucket(self, prefix="Images")
images.topic.add_subscription(sns_sub.SqsSubscription(queue))
```

Java

```
NotifyingBucket images = new NotifyingBucket(this, "MyNotifyingBucket", "/images");
images.topic.addSubscription(new SqsSubscription(queue));
```

C#

```
var queue = new Queue(this, "NewImagesQueue");
var images = new NotifyingBucket(this, "MyNotifyingBucket", new NotifyingBucketProps
{
    Prefix = "/images"
});
```

```
images.topic.AddSubscription(new SqsSubscription(queue));
```

Go

```
queue := awssqs.NewQueue(stack, jsii.String("NewImagesQueue"), nil)
images := NewNotifyingBucket(stack, jsii.String("MyNotifyingBucket"),
&NotifyingBucketProps{
 Prefix: jsii.String("/images"),
})
images.topic.AddSubscription(awssnssubscriptions.NewSqsSubscription(queue, nil))
```

Learn more

The following video provides a comprehensive overview of CDK constructs, and explains how you can use them in your CDK apps.

Environments for the AWS CDK

An environment consists of the AWS account and AWS Region that you deploy an AWS Cloud Development Kit (AWS CDK) stack to.

AWS account

When you create an AWS account, you receive an account ID. This ID is a 12-digit number, such as **012345678901**, that uniquely identifies your account. To learn more, see View AWS account identifiers in the *AWS Account Management Reference Guide*.

AWS Region

AWS Regions are named by using a combination of geographical location and a number that represents an Availability Zone in the Region. For example, **us-east-1** represents an Availability Zone in the US East (N. Virginia) Region. To learn more about AWS Regions, see Regions and Availability Zones. For a list of Region codes, see Regional endpoints in the *AWS General Reference* Reference Guide.

The AWS CDK can determine environments from your credentials and configuration files. These files can be created and managed with the AWS Command Line Interface (AWS CLI). The following is a basic example of these files:

Credentials file

```
[default]
aws_access_key_id=ASIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
aws_session_token =
 IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZVERYLONGSTRINGEXAMPLE

[user1]
aws_access_key_id=ASIAI44QH8DHBEXAMPLE
aws_secret_access_key=je7MtGbClwBF/2Zp9Utk/h3yCo8nvbEXAMPLEKEY
aws_session_token =
 fcZib3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZ2luX2IQoJb3JpZVERYLONGSTRINGEXAMPLE
```

Configuration file

```
[default]
region=us-west-2
output=json

[profile user1]
region=us-east-1
output=text
```

You can pass environment information from these files in your CDK code through environment variables that are provided by the CDK. When you run a CDK CLI command, such as `cdk deploy`, you then provide the profile from your credentials and configuration files to gather environment information from.

The following is an example of specifying these environment variables in your CDK code:

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

The following is an example of passing values associated with the `user1` profile from your credentials and configuration files to the CDK CLI using the `--profile` option. Values from these files will be passed to your environment variables:

```
$ cdk deploy <myStack> --profile <user1>
```

Instead of using values from the credentials and configuration files, you can also hard-code environment values in your CDK code. The following is an example:

```
const envEU = { account: '238383838383', region: 'eu-west-1' };
const envUSA = { account: '837873873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

Learn more

To get started with using environments with the AWS CDK, see Configure environments to use with the AWS CDK.

AWS CDK bootstrapping

Bootstrapping is the process of preparing your AWS environment for usage with the AWS Cloud Development Kit (AWS CDK). Before you deploy a CDK stack into an AWS environment, the environment must first be bootstrapped.

What is bootstrapping?

Bootstrapping prepares your AWS environment by provisioning specific AWS resources in your environment that are used by the AWS CDK. These resources are commonly referred to as your *bootstrap resources*. They include the following:

- **Amazon Simple Storage Service (Amazon S3) bucket** – Used to store your CDK project files, such as AWS Lambda function code and assets.
- **Amazon Elastic Container Registry (Amazon ECR) repository** – Used primarily to store Docker images.
- **AWS Identity and Access Management (IAM) roles** – Configured to grant permissions needed by the AWS CDK to perform deployments. For more information about the IAM roles created during bootstrapping, see IAM roles created during bootstrapping.

How does bootstrapping work?

Resources and their configuration that are used by the CDK are defined in an AWS CloudFormation template. This template is created and managed by the CDK team. For the latest version of this template, see `bootstrap-template.yaml` in the *aws-cdk-cli GitHub repository*.

To bootstrap an environment, you use the AWS CDK Command Line Interface (AWS CDK CLI) `cdk bootstrap` command. The CDK CLI retrieves the template and deploys it to AWS CloudFormation as a stack, known as the *bootstrap stack*. By default, the stack name is `CDKToolkit`. By deploying this template, CloudFormation provisions the resources in your environment. After deployment, the bootstrap stack will appear in the AWS CloudFormation console of your environment.

You can also customize bootstrapping by modifying the template or by using CDK CLI options with the `cdk bootstrap` command.

AWS environments are independent. Each environment that you want to use with the AWS CDK must first be bootstrapped.

Learn more

For instructions on bootstrapping your environment, see Bootstrap your environment for use with the AWS CDK.

Resources and the AWS CDK

Resources are what you configure to use AWS services in your applications. Resources are a feature of AWS CloudFormation. By configuring resources and their properties in a AWS CloudFormation template, you can deploy to AWS CloudFormation to provision your resources. With the AWS Cloud Development Kit (AWS CDK), you can configure resources through constructs. You then deploy your CDK app, which involves synthesizing a AWS CloudFormation template and deploying to AWS CloudFormation to provision your resources.

Configuring resources using constructs

As described in AWS CDK Constructs, the AWS CDK provides a rich class library of constructs, called *constructs*, that represent all AWS resources.

To create an instance of a resource using its corresponding construct, pass in the scope as the first argument, the logical ID of the construct, and a set of configuration properties (props).

For example, here's how to create an Amazon SQS queue with AWS KMS encryption using the `sqs.Queue` construct from the AWS Construct Library.

Example

TypeScript

```
import * as sqs from '@aws-cdk/aws-sqs';

new sqs.Queue(this, 'MyQueue', {
    encryption: sqs.QueueEncryption.KMS_MANAGED
});
```

JavaScript

```
const sqs = require('@aws-cdk/aws-sqs');

new sqs.Queue(this, 'MyQueue', {
    encryption: sqs.QueueEncryption.KMS_MANAGED
});
```

Python

```
import aws_cdk.aws_sqs as sqs

sqs.Queue(self, "MyQueue", encryption=sqs.QueueEncryption.KMS_MANAGED)
```

Java

```
import software.amazon.awscdk.services.sqs.*;

Queue.Builder.create(this, "MyQueue").encryption(
        QueueEncryption.KMS_MANAGED).build();
```

C#

```
using Amazon.CDK.AWS.SQS;

new Queue(this, "MyQueue", new QueueProps
{
    Encryption = QueueEncryption.KMS_MANAGED
```

```
});
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  sqs "github.com/aws/aws-cdk-go/awscdk/v2/awssqs"
)

sqs.NewQueue(stack, jsii.String("MyQueue"), &sqs.QueueProps{
  Encryption: sqs.QueueEncryption_KMS_MANAGED,
})
```

Some configuration props are optional, and in many cases have default values. In some cases, all props are optional, and the last argument can be omitted entirely.

Resource attributes

Most resources in the AWS Construct Library expose attributes, which are resolved at deployment time by AWS CloudFormation. Attributes are exposed in the form of properties on the resource classes with the type name as a prefix. The following example shows how to get the URL of an Amazon SQS queue using the `queueUrl` (Python: `queue_url`) property.

Example

TypeScript

```
import * as sqs from '@aws-cdk/aws-sqs';

const queue = new sqs.Queue(this, 'MyQueue');
const url = queue.queueUrl; // => A string representing a deploy-time value
```

JavaScript

```
const sqs = require('@aws-cdk/aws-sqs');

const queue = new sqs.Queue(this, 'MyQueue');
const url = queue.queueUrl; // => A string representing a deploy-time value
```

Python

```
import aws_cdk.aws_sqs as sqs

queue = sqs.Queue(self, "MyQueue")
url = queue.queue_url # => A string representing a deploy-time value
```

Java

```
Queue queue = new Queue(this, "MyQueue");
String url = queue.getQueueUrl();    // => A string representing a deploy-time value
```

C#

```
var queue = new Queue(this, "MyQueue");
var url = queue.QueueUrl; // => A string representing a deploy-time value
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  sqs "github.com/aws/aws-cdk-go/awscdk/v2/awssqs"
)

queue := sqs.NewQueue(stack, jsii.String("MyQueue"), &sqs.QueueProps{})
url := queue.QueueUrl() // => A string representing a deploy-time value
```

See Tokens and the AWS CDK for information about how the AWS CDK encodes deploy-time attributes as strings.

Referencing resources

When configuring resources, you will often have to reference properties of another resource. The following are examples:

- An Amazon Elastic Container Service (Amazon ECS) resource requires a reference to the cluster on which it runs.
- An Amazon CloudFront distribution requires a reference to the Amazon Simple Storage Service (Amazon S3) bucket containing the source code.

You can reference resources in any of the following ways:

- By passing a resource defined in your CDK app, either in the same stack or in a different one
- By passing a proxy object referencing a resource defined in your AWS account, created from a unique identifier of the resource (such as an ARN)

If the property of a construct represents a construct for another resource, its type is that of the interface type of the construct. For example, the Amazon ECS construct takes a property `cluster` of type `ecs.ICluster`. Another example, is the CloudFront distribution construct that takes a property `sourceBucket` (Python: `source_bucket`) of type `s3.IBucket`.

You can directly pass any resource object of the proper type defined in the same AWS CDK app. The following example defines an Amazon ECS cluster and then uses it to define an Amazon ECS service.

Example

TypeScript

```
const cluster = new ecs.Cluster(this, 'Cluster', { /*...*/ });

const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster });
```

JavaScript

```
const cluster = new ecs.Cluster(this, 'Cluster', { /*...*/ });

const service = new ecs.Ec2Service(this, 'Service', { cluster: cluster });
```

Python

```
cluster = ecs.Cluster(self, "Cluster")

service = ecs.Ec2Service(self, "Service", cluster=cluster)
```

Java

```
Cluster cluster = new Cluster(this, "Cluster");
Ec2Service service = new Ec2Service(this, "Service",
        new Ec2ServiceProps.Builder().cluster(cluster).build());
```

C#

```
var cluster = new Cluster(this, "Cluster");
var service = new Ec2Service(this, "Service", new Ec2ServiceProps { Cluster =
 cluster });
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  ecs "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
)

cluster := ecs.NewCluster(stack, jsii.String("MyCluster"), &ecs.ClusterProps{})
service := ecs.NewEc2Service(stack, jsii.String("MyService"), &ecs.Ec2ServiceProps{
  Cluster: cluster,
})
```

Referencing resources in a different stack

You can refer to resources in a different stack as long as they are defined in the same app and are in the same AWS environment. The following pattern is generally used:

- Store a reference to the construct as an attribute of the stack that produces the resource. (To get a reference to the current construct's stack, use `Stack.of(this)`.)
- Pass this reference to the constructor of the stack that consumes the resource as a parameter or a property. The consuming stack then passes it as a property to any construct that needs it.

The following example defines a stack `stack1`. This stack defines an Amazon S3 bucket and stores a reference to the bucket construct as an attribute of the stack. Then the app defines a second stack, `stack2`, which accepts a bucket at instantiation. `stack2` might, for example, define an AWS Glue Table that uses the bucket for data storage.

Example

TypeScript

```
const prod = { account: '123456789012', region: 'us-east-1' };
```

```
const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

// stack2 will take a property { bucket: IBucket }
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket,
  env: prod
});
```

JavaScript

```
const prod = { account: '123456789012', region: 'us-east-1' };

const stack1 = new StackThatProvidesABucket(app, 'Stack1', { env: prod });

// stack2 will take a property { bucket: IBucket }
const stack2 = new StackThatExpectsABucket(app, 'Stack2', {
  bucket: stack1.bucket,
  env: prod
});
```

Python

```
prod = core.Environment(account="123456789012", region="us-east-1")

stack1 = StackThatProvidesABucket(app, "Stack1", env=prod)

# stack2 will take a property "bucket"
stack2 = StackThatExpectsABucket(app, "Stack2", bucket=stack1.bucket, env=prod)
```

Java

```
// Helper method to build an environment
static Environment makeEnv(String account, String region) {
    return Environment.builder().account(account).region(region)
            .build();
}

App app = new App();

Environment prod = makeEnv("123456789012", "us-east-1");
```

```
StackThatProvidesABucket stack1 = new StackThatProvidesABucket(app, "Stack1",
        StackProps.builder().env(prod).build());

// stack2 will take an argument "bucket"
StackThatExpectsABucket stack2 = new StackThatExpectsABucket(app, "Stack,",
        StackProps.builder().env(prod).build(), stack1.bucket);
```

C#

```
Amazon.CDK.Environment makeEnv(string account, string region)
{
    return new Amazon.CDK.Environment { Account = account, Region = region };
}

var prod = makeEnv(account: "123456789012", region: "us-east-1");

var stack1 = new StackThatProvidesABucket(app, "Stack1", new StackProps { Env =
 prod });

// stack2 will take a property "bucket"
var stack2 = new StackThatExpectsABucket(app, "Stack2", new StackProps { Env = prod,
    bucket = stack1.Bucket});
```

If the AWS CDK determines that the resource is in the same environment, but in a different stack, it automatically synthesizes AWS CloudFormation exports in the producing stack and an `Fn::ImportValue` in the consuming stack to transfer that information from one stack to the other.

Resolving dependency deadlocks

Referencing a resource from one stack in a different stack creates a dependency between the two stacks. This makes sure that they're deployed in the right order. After the stacks are deployed, this dependency is concrete. After that, removing the use of the shared resource from the consuming stack can cause an unexpected deployment failure. This happens if there is another dependency between the two stacks that force them to be deployed in the same order. It can also happen without a dependency if the producing stack is simply chosen by the CDK Toolkit to be deployed first. The AWS CloudFormation export is removed from the producing stack because it's no longer needed, but the exported resource is still being used in the consuming stack because its update is not yet deployed. Therefore, deploying the producer stack fails.

To break this deadlock, remove the use of the shared resource from the consuming stack. (This removes the automatic export from the producing stack.) Next, manually add the same export to the producing stack using exactly the same logical ID as the automatically generated export. Remove the use of the shared resource in the consuming stack and deploy both stacks. Then, remove the manual export (and the shared resource if it's no longer needed) and deploy both stacks again. The stack's `exportValue()` method is a convenient way to create the manual export for this purpose. (See the example in the linked method reference.)

Referencing resources in your AWS account

Suppose you want to use a resource already available in your AWS account in your AWS CDK app. This might be a resource that was defined through the console, an AWS SDK, directly with AWS CloudFormation, or in a different AWS CDK application. You can turn the resource's ARN (or another identifying attribute, or group of attributes) into a proxy object. The proxy object serves as a reference to the resource by calling a static factory method on the resource's class.

When you create such a proxy, the external resource **does not** become a part of your AWS CDK app. Therefore, changes you make to the proxy in your AWS CDK app do not affect the deployed resource. The proxy can, however, be passed to any AWS CDK method that requires a resource of that type.

The following example shows how to reference a bucket based on an existing bucket with the ARN `arn:aws:s3:::amzn-s3-demo-bucket1`, and an Amazon Virtual Private Cloud based on an existing VPC having a specific ID.

Example

TypeScript

```
// Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.fromBucketName(this, 'MyBucket', 'amzn-s3-demo-bucket1');

// Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.fromBucketArn(this, 'MyBucket', 'arn:aws:s3:::amzn-s3-demo-bucket1');

// Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde',
});
```

JavaScript

```
// Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.fromBucketName(this, 'MyBucket', 'amzn-s3-demo-bucket1');

// Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.fromBucketArn(this, 'MyBucket', 'arn:aws:s3:::amzn-s3-demo-bucket1');

// Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.fromVpcAttributes(this, 'MyVpc', {
  vpcId: 'vpc-1234567890abcde'
});
```

Python

```
# Construct a proxy for a bucket by its name (must be same account)
s3.Bucket.from_bucket_name(self, "MyBucket", "amzn-s3-demo-bucket1")

# Construct a proxy for a bucket by its full ARN (can be another account)
s3.Bucket.from_bucket_arn(self, "MyBucket", "arn:aws:s3:::amzn-s3-demo-bucket1")

# Construct a proxy for an existing VPC from its attribute(s)
ec2.Vpc.from_vpc_attributes(self, "MyVpc", vpc_id="vpc-1234567890abcdef")
```

Java

```
// Construct a proxy for a bucket by its name (must be same account)
Bucket.fromBucketName(this, "MyBucket", "amzn-s3-demo-bucket1");

// Construct a proxy for a bucket by its full ARN (can be another account)
Bucket.fromBucketArn(this, "MyBucket",
        "arn:aws:s3:::amzn-s3-demo-bucket1");

// Construct a proxy for an existing VPC from its attribute(s)
Vpc.fromVpcAttributes(this, "MyVpc", VpcAttributes.builder()
        .vpcId("vpc-1234567890abcdef").build());
```

C#

```
// Construct a proxy for a bucket by its name (must be same account)
Bucket.FromBucketName(this, "MyBucket", "amzn-s3-demo-bucket1");

```

```
// Construct a proxy for a bucket by its full ARN (can be another account)
Bucket.FromBucketArn(this, "MyBucket", "arn:aws:s3:::amzn-s3-demo-bucket1");

// Construct a proxy for an existing VPC from its attribute(s)
Vpc.FromVpcAttributes(this, "MyVpc", new VpcAttributes
{
    VpcId = "vpc-1234567890abcdef"
});
```

Go

```
// Define a proxy for a bucket by its name (must be same account)
s3.Bucket_FromBucketName(stack, jsii.String("MyBucket"), jsii.String("amzn-s3-demo-
bucket1"))

// Define a proxy for a bucket by its full ARN (can be another account)
s3.Bucket_FromBucketArn(stack, jsii.String("MyBucket"),
 jsii.String("arn:aws:s3:::amzn-s3-demo-bucket1"))

// Define a proxy for an existing VPC from its attributes
ec2.Vpc_FromVpcAttributes(stack, jsii.String("MyVpc"), &ec2.VpcAttributes{
  VpcId: jsii.String("vpc-1234567890abcde"),
})
```

Let's take a closer look at the `Vpc.fromLookup()` method. Because the `ec2.Vpc` construct is complex, there are many ways you might want to select the VPC to be used with your CDK app. To address this, the VPC construct has a `fromLookup` static method (Python: `from_lookup`) that lets you look up the desired Amazon VPC by querying your AWS account at synthesis time.

To use `Vpc.fromLookup()`, the system that synthesizes the stack must have access to the account that owns the Amazon VPC. This is because the CDK Toolkit queries the account to find the right Amazon VPC at synthesis time.

Furthermore, `Vpc.fromLookup()` works only in stacks that are defined with an explicit **account** and **region** (see Environments for the AWS CDK). If the AWS CDK tries to look up an Amazon VPC from an environment-agnostic stack, the CDK Toolkit doesn't know which environment to query to find the VPC.

You must provide `Vpc.fromLookup()` attributes sufficient to uniquely identify a VPC in your AWS account. For example, there can only ever be one default VPC, so it's sufficient to specify the VPC as the default.

Example

TypeScript

```
ec2.Vpc.fromLookup(this, 'DefaultVpc', {
  isDefault: true
});
```

JavaScript

```
ec2.Vpc.fromLookup(this, 'DefaultVpc', {
  isDefault: true
});
```

Python

```
ec2.Vpc.from_lookup(self, "DefaultVpc", is_default=True)
```

Java

```
Vpc.fromLookup(this, "DefaultVpc", VpcLookupOptions.builder()
        .isDefault(true).build());
```

C#

```
Vpc.FromLookup(this, id = "DefaultVpc", new VpcLookupOptions { IsDefault = true });
```

Go

```
ec2.Vpc_FromLookup(this, jsii.String("DefaultVpc"), &ec2.VpcLookupOptions{
  IsDefault: jsii.Bool(true),
})
```

You can also use the `tags` property to query for VPCs by tag. You can add tags to the Amazon VPC at the time of its creation by using AWS CloudFormation or the AWS CDK. You can edit tags at any time after creation by using the AWS Management Console, the AWS CLI, or an AWS SDK. In addition to any tags you add yourself, the AWS CDK automatically adds the following tags to all VPCs it creates.

- **Name –** The name of the VPC.
- **aws-cdk:subnet-name –** The name of the subnet.
- **aws-cdk:subnet-type –** The type of the subnet: Public, Private, or Isolated.

Example

TypeScript

```
ec2.Vpc.fromLookup(this, 'PublicVpc',
    {tags: {'aws-cdk:subnet-type': "Public"}});
```

JavaScript

```
ec2.Vpc.fromLookup(this, 'PublicVpc',
    {tags: {'aws-cdk:subnet-type': "Public"}});
```

Python

```
ec2.Vpc.from_lookup(self, "PublicVpc",
    tags={"aws-cdk:subnet-type": "Public"})
```

Java

```
Vpc.fromLookup(this, "PublicVpc", VpcLookupOptions.builder()
        .tags(java.util.Map.of("aws-cdk:subnet-type", "Public"))  // Java 9 or later
        .build());
```

C#

```
Vpc.FromLookup(this, id: "PublicVpc", new VpcLookupOptions
{
    Tags = new Dictionary<string, string> { ["aws-cdk:subnet-type"] = "Public" }
});
```

Go

```
ec2.Vpc_FromLookup(this, jsii.String("DefaultVpc"), &ec2.VpcLookupOptions{
  Tags: &map[string]*string{"aws-cdk:subnet-type": jsii.String("Public")},
```

```
})
```

Results of `Vpc.fromLookup()` are cached in the project's `cdk.context.json` file. (See Context values and the AWS CDK.) Commit this file to version control so that your app will continue to refer to the same Amazon VPC. This works even if you later change the attributes of your VPCs in a way that would result in a different VPC being selected. This is particularly important if you're deploying the stack in an environment that doesn't have access to the AWS account that defines the VPC, such as CDK Pipelines.

Although you can use an external resource anywhere you'd use a similar resource defined in your AWS CDK app, you cannot modify it. For example, calling `addToResourcePolicy` (Python: `add_to_resource_policy`) on an external `s3.Bucket` does nothing.

Resource physical names

The logical names of resources in AWS CloudFormation are different from the names of resources that are shown in the AWS Management Console after they're deployed by AWS CloudFormation. The AWS CDK calls these final names *physical names*.

For example, AWS CloudFormation might create the Amazon S3 bucket with the logical ID `Stack2MyBucket4DD88B4F` and the physical name `stack2MyBucket4dd88b4f-iuv1rbv9z3to`.

You can specify a physical name when creating constructs that represent resources by using the property `<resourceType>Name`. The following example creates an Amazon S3 bucket with the physical name `amzn-s3-demo-bucket`.

Example

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'amzn-s3-demo-bucket',
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: 'amzn-s3-demo-bucket'
```

```
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket", bucket_name="amzn-s3-demo-bucket")
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
        .bucketName("amzn-s3-demo-bucket").build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps { BucketName = "amzn-s3-
demo-bucket" });
```

Go

```
bucket := s3.NewBucket(this, jsii.String("MyBucket"), &s3.BucketProps{
  BucketName: jsii.String("amzn-s3-demo-bucket"),
})
```

Assigning physical names to resources has some disadvantages in AWS CloudFormation. Most importantly, any changes to deployed resources that require a resource replacement, such as changes to a resource's properties that are immutable after creation, will fail if a resource has a physical name assigned. If you end up in that state, the only solution is to delete the AWS CloudFormation stack, then deploy the AWS CDK app again. See the AWS CloudFormation documentation for details.

In some cases, such as when creating an AWS CDK app with cross-environment references, physical names are required for the AWS CDK to function correctly. In those cases, if you don't want to bother with coming up with a physical name yourself, you can let the AWS CDK name it for you. To do so, use the special value `PhysicalName.GENERATE_IF_NEEDED`, as follows.

Example

TypeScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
```

```
  bucketName: core.PhysicalName.GENERATE_IF_NEEDED,
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'MyBucket', {
  bucketName: core.PhysicalName.GENERATE_IF_NEEDED
});
```

Python

```
bucket = s3.Bucket(self, "MyBucket",
                         bucket_name=core.PhysicalName.GENERATE_IF_NEEDED)
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "MyBucket")
        .bucketName(PhysicalName.GENERATE_IF_NEEDED).build();
```

C#

```
var bucket = new Bucket(this, "MyBucket", new BucketProps
    { BucketName = PhysicalName.GENERATE_IF_NEEDED });
```

Go

```
bucket := s3.NewBucket(this, jsii.String("MyBucket"), &s3.BucketProps{
  BucketName: awscdk.PhysicalName_GENERATE_IF_NEEDED(),
})
```

Passing unique resource identifiers

Whenever possible, you should pass resources by reference, as described in the previous section. However, there are cases where you have no other choice but to refer to a resource by one of its attributes. Example use cases include the following:

- When you are using low-level AWS CloudFormation resources.
- When you need to expose resources to the runtime components of an AWS CDK application, such as when referring to Lambda functions through environment variables.

These identifiers are available as attributes on the resources, such as the following.

Example

TypeScript

```
bucket.bucketName
lambdaFunc.functionArn
securityGroup.groupArn
```

JavaScript

```
bucket.bucketName
lambdaFunc.functionArn
securityGroup.groupArn
```

Python

```
bucket.bucket_name
lambda_func.function_arn
security_group_arn
```

Java

The Java AWS CDK binding uses getter methods for attributes.

```
bucket.getBucketName()
lambdaFunc.getFunctionArn()
securityGroup.getGroupArn()
```

C#

```
bucket.BucketName
lambdaFunc.FunctionArn
securityGroup.GroupArn
```

Go

```
bucket.BucketName()
```

```
fn.FunctionArn()
```

The following example shows how to pass a generated bucket name to an AWS Lambda function.

Example

TypeScript

```
const bucket = new s3.Bucket(this, 'Bucket');

new lambda.Function(this, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName,
  },
});
```

JavaScript

```
const bucket = new s3.Bucket(this, 'Bucket');

new lambda.Function(this, 'MyLambda', {
  // ...
  environment: {
    BUCKET_NAME: bucket.bucketName
  }
});
```

Python

```
bucket = s3.Bucket(self, "Bucket")

lambda.Function(self, "MyLambda", environment=dict(BUCKET_NAME=bucket.bucket_name))
```

Java

```
final Bucket bucket = new Bucket(this, "Bucket");

Function.Builder.create(this, "MyLambda")
        .environment(java.util.Map.of(    // Java 9 or later
                "BUCKET_NAME", bucket.getBucketName()))
```

```
        .build();
```

C#

```
var bucket = new Bucket(this, "Bucket");

new Function(this, "MyLambda", new FunctionProps
{
    Environment = new Dictionary<string, string>
    {
        ["BUCKET_NAME"] = bucket.BucketName
    }
});
```

Go

```
bucket := s3.NewBucket(this, jsii.String("Bucket"), &s3.BucketProps{})
lambda.NewFunction(this, jsii.String("MyLambda"), &lambda.FunctionProps{
  Environment: &map[string]*string{"BUCKET_NAME": bucket.BucketName()},
})
```

Granting permissions between resources

Higher-level constructs make least-privilege permissions achievable by offering simple, intent-based APIs to express permission requirements. For example, many L2 constructs offer grant methods that you can use to grant an entity (such as an IAM role or user) permission to work with the resource, without having to manually create IAM permission statements.

The following example creates the permissions to allow a Lambda function's execution role to read and write objects to a particular Amazon S3 bucket. If the Amazon S3 bucket is encrypted with an AWS KMS key, this method also grants permissions to the Lambda function's execution role to decrypt with the key.

Example

TypeScript

```
if (bucket.grantReadWrite(func).success) {
  // ...
```

```
}
```

JavaScript

```
if ( bucket.grantReadWrite(func).success) {
  // ...
}
```

Python

```
if bucket.grant_read_write(func).success:
    # ...
```

Java

```
if (bucket.grantReadWrite(func).getSuccess()) {
    // ...
}
```

C#

```
if (bucket.GrantReadWrite(func).Success)
{
    // ...
}
```

Go

```
if *bucket.GrantReadWrite(function, nil).Success() {
  // ...
}
```

The grant methods return an `iam.Grant` object. Use the `success` attribute of the `Grant` object to determine whether the grant was effectively applied (for example, it may not have been applied on external resources). You can also use the `assertSuccess` (Python: `assert_success`) method of the `Grant` object to enforce that the grant was successfully applied.

If a specific grant method isn't available for the particular use case, you can use a generic grant method to define a new grant with a specified list of actions.

The following example shows how to grant a Lambda function access to the Amazon DynamoDB `CreateBackup` action.

Example

TypeScript

```
table.grant(func, 'dynamodb:CreateBackup');
```

JavaScript

```
table.grant(func, 'dynamodb:CreateBackup');
```

Python

```
table.grant(func, "dynamodb:CreateBackup")
```

Java

```
table.grant(func, "dynamodb:CreateBackup");
```

C#

```
table.Grant(func, "dynamodb:CreateBackup");
```

Go

```
table := dynamodb.NewTable(this, jsii.String("MyTable"), &dynamodb.TableProps{})
table.Grant(function, jsii.String("dynamodb:CreateBackup"))
```

Many resources, such as Lambda functions, require a role to be assumed when executing code. A configuration property enables you to specify an `iam.IRole`. If no role is specified, the function automatically creates a role specifically for this use. You can then use grant methods on the resources to add statements to the role.

The grant methods are built using lower-level APIs for handling with IAM policies. Policies are modeled as [PolicyDocument](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.aws_iam.PolicyDocument.html) objects. Add statements directly to roles (or a construct's attached role) using the addToRolePolicy method (Python: add_to_role_policy), or to a resource's policy (such as a Bucket policy) using the addToResourcePolicy (Python: add_to_resource_policy) method.

Resource metrics and alarms

Many resources emit CloudWatch metrics that can be used to set up monitoring dashboards and alarms. Higher-level constructs have metric methods that let you access the metrics without looking up the correct name to use.

The following example shows how to define an alarm when the ApproximateNumberOfMessagesNotVisible of an Amazon SQS queue exceeds 100.

Example

TypeScript

```
import * as cw from '@aws-cdk/aws-cloudwatch';
import * as sqs from '@aws-cdk/aws-sqs';
import { Duration } from '@aws-cdk/core';

const queue = new sqs.Queue(this, 'MyQueue');

const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5),
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100,
  // ...
});
```

JavaScript

```
const cw = require('@aws-cdk/aws-cloudwatch');
const sqs = require('@aws-cdk/aws-sqs');
const { Duration } = require('@aws-cdk/core');

const queue = new sqs.Queue(this, 'MyQueue');
```

```
const metric = queue.metricApproximateNumberOfMessagesNotVisible({
  label: 'Messages Visible (Approx)',
  period: Duration.minutes(5)
  // ...
});
metric.createAlarm(this, 'TooManyMessagesAlarm', {
  comparisonOperator: cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
  threshold: 100
  // ...
});
```

Python

```
import aws_cdk.aws_cloudwatch as cw
import aws_cdk.aws_sqs as sqs
from aws_cdk.core import Duration

queue = sqs.Queue(self, "MyQueue")
metric = queue.metric_approximate_number_of_messages_not_visible(
    label="Messages Visible (Approx)",
    period=Duration.minutes(5),
    # ...
)
metric.create_alarm(self, "TooManyMessagesAlarm",
    comparison_operator=cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
    threshold=100,
    # ...
)
```

Java

```
import software.amazon.awscdk.core.Duration;
import software.amazon.awscdk.services.sqs.Queue;
import software.amazon.awscdk.services.cloudwatch.Metric;
import software.amazon.awscdk.services.cloudwatch.MetricOptions;
import software.amazon.awscdk.services.cloudwatch.CreateAlarmOptions;
import software.amazon.awscdk.services.cloudwatch.ComparisonOperator;

Queue queue = new Queue(this, "MyQueue");

Metric metric = queue
        .metricApproximateNumberOfMessagesNotVisible(MetricOptions.builder()
```

```
                .label("Messages Visible (Approx)")
                .period(Duration.minutes(5)).build());

metric.createAlarm(this, "TooManyMessagesAlarm", CreateAlarmOptions.builder()
                .comparisonOperator(ComparisonOperator.GREATER_THAN_THRESHOLD)
                .threshold(100)
                // ...
                .build());
```

C#

```
using cdk = Amazon.CDK;
using cw = Amazon.CDK.AWS.CloudWatch;
using sqs = Amazon.CDK.AWS.SQS;

var queue = new sqs.Queue(this, "MyQueue");
var metric = queue.MetricApproximateNumberOfMessagesNotVisible(new cw.MetricOptions
{
    Label = "Messages Visible (Approx)",
    Period = cdk.Duration.Minutes(5),
    // ...
});
metric.CreateAlarm(this, "TooManyMessagesAlarm", new cw.CreateAlarmOptions
{
    ComparisonOperator = cw.ComparisonOperator.GREATER_THAN_THRESHOLD,
    Threshold = 100,
    // ..
});
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  cw "github.com/aws/aws-cdk-go/awscdk/v2/awscloudwatch"
  sqs "github.com/aws/aws-cdk-go/awscdk/v2/awssqs"
)

queue := sqs.NewQueue(this, jsii.String("MyQueue"), &sqs.QueueProps{})
metric := queue.MetricApproximateNumberOfMessagesNotVisible(&cw.MetricOptions{
  Label: jsii.String("Messages Visible (Approx)"),
  Period: awscdk.Duration_Minutes(jsii.Number(5)),
})
```

```
metric.CreateAlarm(this, jsii.String("TooManyMessagesAlarm"),
 &cw.CreateAlarmOptions{
  ComparisonOperator: cw.ComparisonOperator_GREATER_THAN_THRESHOLD,
  Threshold: jsii.Number(100),
})
```

If there is no method for a particular metric, you can use the general metric method to specify the metric name manually.

Metrics can also be added to CloudWatch dashboards. See CloudWatch.

Network traffic

In many cases, you must enable permissions on a network for an application to work, such as when the compute infrastructure needs to access the persistence layer. Resources that establish or listen for connections expose methods that enable traffic flows, including setting security group rules or network ACLs.

IConnectable resources have a `connections` property that is the gateway to network traffic rules configuration.

You enable data to flow on a given network path by using `allow` methods. The following example enables HTTPS connections to the web and incoming connections from the Amazon EC2 Auto Scaling group `fleet2`.

Example

TypeScript

```
import * as asg from '@aws-cdk/aws-autoscaling';
import * as ec2 from '@aws-cdk/aws-ec2';

const fleet1: asg.AutoScalingGroup = asg.AutoScalingGroup(/*...*/);

// Allow surfing the (secure) web
fleet1.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443,
 toPort: 443 }));

const fleet2: asg.AutoScalingGroup = asg.AutoScalingGroup(/*...*/);
```

```
fleet1.connections.allowFrom(fleet2, ec2.Port.AllTraffic());
```

JavaScript

```
const asg = require('@aws-cdk/aws-autoscaling');
const ec2 = require('@aws-cdk/aws-ec2');

const fleet1 = asg.AutoScalingGroup();

// Allow surfing the (secure) web
fleet1.connections.allowTo(new ec2.Peer.anyIpv4(), new ec2.Port({ fromPort: 443,
 toPort: 443 }));

const fleet2 = asg.AutoScalingGroup();
fleet1.connections.allowFrom(fleet2, ec2.Port.AllTraffic());
```

Python

```
import aws_cdk.aws_autoscaling as asg
import aws_cdk.aws_ec2 as ec2

fleet1 = asg.AutoScalingGroup( ... )

# Allow surfing the (secure) web
fleet1.connections.allow_to(ec2.Peer.any_ipv4(),
  ec2.Port(PortProps(from_port=443, to_port=443)))

fleet2 = asg.AutoScalingGroup( ... )
fleet1.connections.allow_from(fleet2, ec2.Port.all_traffic())
```

Java

```
import software.amazon.awscdk.services.autoscaling.AutoScalingGroup;
import software.amazon.awscdk.services.ec2.Peer;
import software.amazon.awscdk.services.ec2.Port;

AutoScalingGroup fleet1 = AutoScalingGroup.Builder.create(this, "MyFleet")
        /* ... */.build();

// Allow surfing the (secure) Web
fleet1.getConnections().allowTo(Peer.anyIpv4(),
        Port.Builder.create().fromPort(443).toPort(443).build());
```

```
AutoScalingGroup fleet2 = AutoScalingGroup.Builder.create(this, "MyFleet2")
        /* ... */.build();
fleet1.getConnections().allowFrom(fleet2, Port.allTraffic());
```

C#

```
using cdk = Amazon.CDK;
using asg = Amazon.CDK.AWS.AutoScaling;
using ec2 = Amazon.CDK.AWS.EC2;

// Allow surfing the (secure) Web
var fleet1 = new asg.AutoScalingGroup(this, "MyFleet", new asg.AutoScalingGroupProps
 { /* ... */ });
fleet1.Connections.AllowTo(ec2.Peer.AnyIpv4(), new ec2.Port(new ec2.PortProps
  { FromPort = 443, ToPort = 443 }));

var fleet2 = new asg.AutoScalingGroup(this, "MyFleet2", new
 asg.AutoScalingGroupProps { /* ... */ });
fleet1.Connections.AllowFrom(fleet2, ec2.Port.AllTraffic());
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  autoscaling "github.com/aws/aws-cdk-go/awscdk/v2/awsautoscaling"
  ec2 "github.com/aws/aws-cdk-go/awscdk/v2/awsec2"
)

fleet1 := autoscaling.NewAutoScalingGroup(this, jsii.String("MyFleet1"),
 &autoscaling.AutoScalingGroupProps{})
fleet1.Connections().AllowTo(ec2.Peer_AnyIpv4(),ec2.NewPort(&ec2.PortProps{ FromPort:
 jsii.Number(443), ToPort: jsii.Number(443) }),jsii.String("secure web"))

fleet2 := autoscaling.NewAutoScalingGroup(this, jsii.String("MyFleet2"),
 &autoscaling.AutoScalingGroupProps{})
fleet1.Connections().AllowFrom(fleet2, ec2.Port_AllTraffic(),jsii.String("all
 traffic"))
```

Certain resources have default ports associated with them. Examples include the listener of a load balancer on the public port, and the ports on which the database engine accepts connections for instances of an Amazon RDS database. In such cases, you can enforce tight network control without having to manually specify the port. To do so, use the `allowDefaultPortFrom` and `allowToDefaultPort` methods (Python: `allow_default_port_from`, `allow_to_default_port`).

The following example shows how to enable connections from any IPV4 address, and a connection from an Auto Scaling group to access a database.

Example

TypeScript

```
listener.connections.allowDefaultPortFromAnyIpv4('Allow public access');

fleet.connections.allowToDefaultPort(rdsDatabase, 'Fleet can access database');
```

JavaScript

```
listener.connections.allowDefaultPortFromAnyIpv4('Allow public access');

fleet.connections.allowToDefaultPort(rdsDatabase, 'Fleet can access database');
```

Python

```
listener.connections.allow_default_port_from_any_ipv4("Allow public access")

fleet.connections.allow_to_default_port(rds_database, "Fleet can access database")
```

Java

```
listener.getConnections().allowDefaultPortFromAnyIpv4("Allow public access");

fleet.getConnections().AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

C#

```
listener.Connections.AllowDefaultPortFromAnyIpv4("Allow public access");

fleet.Connections.AllowToDefaultPort(rdsDatabase, "Fleet can access database");
```

Go

```
listener.Connections().AllowDefaultPortFromAnyIpv4(jsii.String("Allow public
 Access"))
fleet.Connections().AllowToDefaultPort(rdsDatabase, jsii.String("Fleet can access
 database"))
```

Event handling

Some resources can act as event sources. Use the `addEventNotification` method (Python: `add_event_notification`) to register an event target to a particular event type emitted by the resource. In addition to this, `addXxxNotification` methods offer a simple way to register a handler for common event types.

The following example shows how to trigger a Lambda function when an object is added to an Amazon S3 bucket.

Example

TypeScript

```
import * as s3nots from '@aws-cdk/aws-s3-notifications';

const handler = new lambda.Function(this, 'Handler', { /*…*/ });
const bucket = new s3.Bucket(this, 'Bucket');
bucket.addObjectCreatedNotification(new s3nots.LambdaDestination(handler));
```

JavaScript

```
const s3nots = require('@aws-cdk/aws-s3-notifications');

const handler = new lambda.Function(this, 'Handler', { /*…*/ });
const bucket = new s3.Bucket(this, 'Bucket');
bucket.addObjectCreatedNotification(new s3nots.LambdaDestination(handler));
```

Python

```
import aws_cdk.aws_s3_notifications as s3_nots

handler = lambda_.Function(self, "Handler", ...)
```

```
bucket = s3.Bucket(self, "Bucket")
bucket.add_object_created_notification(s3_nots.LambdaDestination(handler))
```

Java

```
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.s3.notifications.LambdaDestination;

Function handler = Function.Builder.create(this, "Handler")/* ... */.build();
Bucket bucket = new Bucket(this, "Bucket");
bucket.addObjectCreatedNotification(new LambdaDestination(handler));
```

C#

```
using lambda = Amazon.CDK.AWS.Lambda;
using s3 = Amazon.CDK.AWS.S3;
using s3Nots = Amazon.CDK.AWS.S3.Notifications;

var handler = new lambda.Function(this, "Handler", new lambda.FunctionProps { .. });
var bucket = new s3.Bucket(this, "Bucket");
bucket.AddObjectCreatedNotification(new s3Nots.LambdaDestination(handler));
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  s3 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
  s3nots "github.com/aws/aws-cdk-go/awscdk/v2/awss3notifications"
)

handler := lambda.NewFunction(this, jsii.String("MyFunction"),
 &lambda.FunctionProps{})
bucket := s3.NewBucket(this, jsii.String("Bucket"), &s3.BucketProps{})
bucket.AddObjectCreatedNotification(s3nots.NewLambdaDestination(handler), nil)
```

Removal policies

Resources that maintain persistent data, such as databases, Amazon S3 buckets, and Amazon ECR registries, have a *removal policy*. The removal policy indicates whether to delete persistent objects when the AWS CDK stack that contains them is destroyed. The values specifying the removal policy are available through the `RemovalPolicy` enumeration in the AWS CDK `core` module.

> **Note**
>
> Resources besides those that store data persistently might also have a `removalPolicy` that is used for a different purpose. For example, a Lambda function version uses a `removalPolicy` attribute to determine whether a given version is retained when a new version is deployed. These have different meanings and defaults compared to the removal policy on an Amazon S3 bucket or DynamoDB table.

Value	Meaning
`RemovalPolicy.RETAIN`	Keep the contents of the resource when destroying the stack (default). The resource is orphaned from the stack and must be deleted manually. If you attempt to re-deploy the stack while the resource still exists, you will receive an error message due to a name conflict.
`RemovalPolicy.DESTROY`	The resource will be destroyed along with the stack.

AWS CloudFormation does not remove Amazon S3 buckets that contain files even if their removal policy is set to `DESTROY`. Attempting to do so is an AWS CloudFormation error. To have the AWS CDK delete all files from the bucket before destroying it, set the bucket's `autoDeleteObjects` property to `true`.

Following is an example of creating an Amazon S3 bucket with `RemovalPolicy` of `DESTROY` and `autoDeleteOjbects` set to `true`.

Example

TypeScript

```
import * as cdk from '@aws-cdk/core';
```

```
import * as s3 from '@aws-cdk/aws-s3';

export class CdkTestStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true
    });
  }
}
```

JavaScript

```
const cdk = require('@aws-cdk/core');
const s3 = require('@aws-cdk/aws-s3');

class CdkTestStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      autoDeleteObjects: true
    });
  }
}

module.exports = { CdkTestStack }
```

Python

```
import aws_cdk.core as cdk
import aws_cdk.aws_s3 as s3

class CdkTestStack(cdk.stack):
    def __init__(self, scope: cdk.Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        bucket = s3.Bucket(self, "Bucket",
            removal_policy=cdk.RemovalPolicy.DESTROY,
            auto_delete_objects=True)
```

Java

```
software.amazon.awscdk.core.*;
import software.amazon.awscdk.services.s3.*;

public class CdkTestStack extends Stack {
    public CdkTestStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkTestStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "Bucket")
                .removalPolicy(RemovalPolicy.DESTROY)
                .autoDeleteObjects(true).build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY,
        AutoDeleteObjects = true
    });
}
```

Go

```
import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/jsii-runtime-go"
  s3 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
)

s3.NewBucket(this, jsii.String("Bucket"), &s3.BucketProps{
```

```
  RemovalPolicy: awscdk.RemovalPolicy_DESTROY,
  AutoDeleteObjects: jsii.Bool(true),
})
```

You can also apply a removal policy directly to the underlying AWS CloudFormation resource via the `applyRemovalPolicy()` method. This method is available on some stateful resources that do not have a `removalPolicy` property in their L2 resource's props. Examples include the following:

- AWS CloudFormation stacks
- Amazon Cognito user pools
- Amazon DocumentDB database instances
- Amazon EC2 volumes
- Amazon OpenSearch Service domains
- Amazon FSx file systems
- Amazon SQS queues

Example

TypeScript

```
const resource = bucket.node.findChild('Resource') as cdk.CfnResource;
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

JavaScript

```
const resource = bucket.node.findChild('Resource');
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

Python

```
resource = bucket.node.find_child('Resource')
resource.apply_removal_policy(cdk.RemovalPolicy.DESTROY);
```

Java

```
CfnResource resource = (CfnResource)bucket.node.findChild("Resource");
resource.applyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

C#

```
var resource = (CfnResource)bucket.node.findChild('Resource');
resource.ApplyRemovalPolicy(cdk.RemovalPolicy.DESTROY);
```

Note

The AWS CDK's `RemovalPolicy` translates to AWS CloudFormation's `DeletionPolicy`. However, the default in AWS CDK is to retain the data, which is the opposite of the AWS CloudFormation default.

Identifiers and the AWS CDK

When building AWS Cloud Development Kit (AWS CDK) apps, you will use many types of identifiers and names. To use the AWS CDK effectively and avoid errors, it is important to understand the types of identifiers.

Identifiers must be unique within the scope in which they are created; they do not need to be globally unique in your AWS CDK application.

If you attempt to create an identifier with the same value within the same scope, the AWS CDK throws an exception.

Construct IDs

The most common identifier, `id`, is the identifier passed as the second argument when instantiating a construct object. This identifier, like all identifiers, only needs to be unique within the scope in which it is created, which is the first argument when instantiating a construct object.

Note

The `id` of a stack is also the identifier that you use to refer to it in the AWS CDK CLI reference.

Let's look at an example where we have two constructs with the identifier `MyBucket` in our app. The first is defined in the scope of the stack with the identifier `Stack1`. The second is defined in the scope of a stack with the identifier `Stack2`. Because they're defined in different scopes, this doesn't cause any conflict, and they can coexist in the same app without issues.

Example

TypeScript

```
import { App, Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

class MyStack extends Stack {
  constructor(scope: Construct, id: string, props: StackProps = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

JavaScript

```
const { App , Stack } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class MyStack extends Stack {
  constructor(scope, id, props = {}) {
    super(scope, id, props);

    new s3.Bucket(this, 'MyBucket');
  }
}

const app = new App();
new MyStack(app, 'Stack1');
new MyStack(app, 'Stack2');
```

Python

```
from aws_cdk import App, Construct, Stack, StackProps
```

```
from constructs import Construct
from aws_cdk import aws_s3 as s3

class MyStack(Stack):

    def __init__(self, scope: Construct, id: str, **kwargs):

        super().__init__(scope, id, **kwargs)
        s3.Bucket(self, "MyBucket")

app = App()
MyStack(app, 'Stack1')
MyStack(app, 'Stack2')
```

Java

```
// MyStack.java
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;
import software.amazon.awscdk.services.s3.Bucket;

public class MyStack extends Stack {
    public MyStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);
        new Bucket(this, "MyBucket");
    }
}

// Main.java
package com.myorg;

import software.amazon.awscdk.App;

public class Main {
    public static void main(String[] args) {
```

```
        App app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

C#

```
using Amazon.CDK;
using constructs;
using Amazon.CDK.AWS.S3;

public class MyStack : Stack
{
    public MyStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
    {
        new Bucket(this, "MyBucket");
    }
}

class Program
{
    static void Main(string[] args)
    {
        var app = new App();
        new MyStack(app, "Stack1");
        new MyStack(app, "Stack2");
    }
}
```

Paths

The constructs in an AWS CDK application form a hierarchy rooted in the App class. We refer to the collection of IDs from a given construct, its parent construct, its grandparent, and so on to the root of the construct tree, as a *path*.

The AWS CDK typically displays paths in your templates as a string. The IDs from the levels are separated by slashes, starting at the node immediately under the root App instance, which is usually a stack. For example, the paths of the two Amazon S3 bucket resources in the previous code example are `Stack1/MyBucket` and `Stack2/MyBucket`.

You can access the path of any construct programmatically, as shown in the following example. This gets the path of `myConstruct` (or `my_construct`, as Python developers would write it). Since IDs must be unique within the scope they are created, their paths are always unique within an AWS CDK application.

Example

TypeScript

```
const path: string = myConstruct.node.path;
```

JavaScript

```
const path = myConstruct.node.path;
```

Python

```
path = my_construct.node.path
```

Java

```
String path = myConstruct.getNode().getPath();
```

C#

```
string path = myConstruct.Node.Path;
```

Unique IDs

AWS CloudFormation requires that all logical IDs in a template be unique. Because of this, the AWS CDK must be able to generate a unique identifier for each construct in an application. Resources have paths that are globally unique (the names of all scopes from the stack to a specific resource). Therefore, the AWS CDK generates the necessary unique identifiers by concatenating the elements of the path and adding an 8-digit hash. (The hash is necessary to distinguish distinct paths, such as `A/B/C` and `A/BC`, that would result in the same AWS CloudFormation identifier. AWS CloudFormation identifiers are alphanumeric and cannot contain slashes or other separator characters.) The AWS CDK calls this string the *unique ID* of the construct.

In general, your AWS CDK app should not need to know about unique IDs. You can, however, access the unique ID of any construct programmatically, as shown in the following example.

Example

TypeScript

```
const uid: string = Names.uniqueId(myConstruct);
```

JavaScript

```
const uid = Names.uniqueId(myConstruct);
```

Python

```
uid = Names.unique_id(my_construct)
```

Java

```
String uid = Names.uniqueId(myConstruct);
```

C#

```
string uid = Names.Uniqueid(myConstruct);
```

The *address* is another kind of unique identifier that uniquely distinguishes CDK resources. Derived from the SHA-1 hash of the path, it is not human-readable. However, its constant, relatively short length (always 42 hexadecimal characters) makes it useful in situations where the "traditional" unique ID might be too long. Some constructs may use the address in the synthesized AWS CloudFormation template instead of the unique ID. Again, your app generally should not need to know about its constructs' addresses, but you can retrieve a construct’s address as follows.

Example

TypeScript

```
const addr: string = myConstruct.node.addr;
```

JavaScript

```
const addr = myConstruct.node.addr;
```

Python

```
addr = my_construct.node.addr
```

Java

```
String addr = myConstruct.getNode().getAddr();
```

C#

```
string addr = myConstruct.Node.Addr;
```

Logical IDs

Unique IDs serve as the *logical identifiers* (or *logical names*) of resources in the generated AWS CloudFormation templates for constructs that represent AWS resources.

For example, the Amazon S3 bucket in the previous example that is created within `Stack2` results in an `AWS::S3::Bucket` resource. The resource's logical ID is `Stack2MyBucket4DD88B4F` in the resulting AWS CloudFormation template. (For details on how this identifier is generated, see Unique IDs.)

Logical ID stability

Avoid changing the logical ID of a resource after it has been created. AWS CloudFormation identifies resources by their logical ID. Therefore, if you change the logical ID of a resource, AWS CloudFormation creates a new resource with the new logical ID, then deletes the existing one. Depending on the type of resource, this might cause service interruption, data loss, or both.

Tokens and the AWS CDK

In the AWS Cloud Development Kit (AWS CDK), *tokens* are placeholders for values that aren't known when defining constructs or synthesizing stacks. These values will be fully resolved at deployment, when your actual infrastructure is created. When developing AWS CDK applications, you will work with tokens to manage these values across your application.

Token example

The following is an example of a CDK stack that defines a construct for an Amazon Simple Storage Service (Amazon S3) bucket. Since the name of our bucket is not yet known, the value for `bucketName` is stored as a token:

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

export class CdkDemoAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define an S3 bucket
    const myBucket = new s3.Bucket(this, 'myBucket');

    // Store value of the S3 bucket name
    const myBucketName = myBucket.bucketName;

    // Print the current value for the S3 bucket name at synthesis
    console.log("myBucketName: " + bucketName);
  }
}
```

JavaScript

```
const { Stack, Duration } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class CdkDemoAppStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define an S3 bucket
    const myBucket = new s3.Bucket(this, 'myBucket');

    // Store value of the S3 bucket name
    const myBucketName = myBucket.bucketName;
```

```
    // Print the current value for the S3 bucket name at synthesis
    console.log("myBucketName: " + myBucketName);
  }
}

module.exports = { CdkDemoAppStack }
```

Python

```
from aws_cdk import (
    Stack
)
from constructs import Construct
from aws_cdk import aws_s3 as s3

class CdkDemoAppStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define an S3 bucket
        my_bucket = s3.Bucket(self, "myBucket")

        # Store the value of the S3 bucket name
        my_bucket_name = my_bucket.bucket_name

        # Print the current value for the S3 bucket name at synthesis
        print(f"myBucketName: {my_bucket_name}")
```

Java

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.s3.Bucket;

import java.util.Map;

public class CdkDemoAppStack extends Stack {
    public CdkDemoAppStack(final Construct scope, final String id) {
```

```
        this(scope, id, null);
    }

    public CdkDemoAppStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // Define an S3 bucket
        Bucket myBucket = Bucket.Builder.create(this, "myBucket")
            .build();

        // Store the token for the bucket name
        String myBucketName = myBucket.getBucketName();

        // Print the token at synthesis
        System.out.println("myBucketName: " + myBucketName);
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.S3;

namespace CdkDemoApp
{
    public class CdkDemoAppStack : Stack
    {
        internal CdkDemoAppStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Define an S3 bucket
            var myBucket = new Bucket(this, "myBucket");

            // Store the token for the bucket name
            var myBucketName = myBucket.BucketName;

            // Print the token at synthesis
            System.Console.WriteLine($"myBucketName: {myBucketName}");
        }
    }
}
```

Go

```
package main

import (
 "fmt"

 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type CdkDemoAppStackProps struct {
 awscdk.StackProps
}

func NewCdkDemoAppStack(scope constructs.Construct, id string, props
 *CdkDemoAppStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 // Define an S3 bucket
 myBucket := awss3.NewBucket(stack, jsii.String("myBucket"), &awss3.BucketProps{})

 // Store the token for the bucket name
 myBucketName := myBucket.BucketName()

 // Print the token at synthesis
 fmt.Println("myBucketName: ", *myBucketName)

 return stack
}

// ...
```

When we run `cdk synth` to synthesize our stack, the value for `myBucketName` will be displayed in the token format of `${Token[TOKEN.<1234>]}`. This token format is a result of how the AWS CDK encodes tokens. In this example, the token is encoded as a string:

```
$ cdk synth --quiet
myBucketName: ${Token[TOKEN.21]}
```

Since the value for our bucket name is not known at synthesis, the token is rendered as `myBucket<unique-hash>`. Our AWS CloudFormation template uses the `Ref` intrinsic function to reference its value, which will be known at deployment:

```
Resources:
  myBucket<5AF9C99B>:
    # ...
Outputs:
  bucketNameOutput:
    Description: The name of the S3 bucket
    Value:
      Ref: myBucket<5AF9C99B>
```

For more information on how the unique hash is generated, see Generated logical IDs in your AWS CloudFormation template.

Passing tokens

Tokens can be passed around as if they were the actual value they represent. The following is an example that passes the token for our bucket name to a construct for an AWS Lambda function:

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';
import * as lambda from 'aws-cdk-lib/aws-lambda';

export class CdkDemoAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define an S3 bucket
    const myBucket = new s3.Bucket(this, 'myBucket');

    // ...
```

```
    // Define a Lambda function
    const myFunction = new lambda.Function(this, "myFunction", {
      runtime: lambda.Runtime.NODEJS_20_X,
      handler: "index.handler",
      code: lambda.Code.fromInline(`
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello World!'),
          };
        };
      `),
      functionName: myBucketName + "Function", // Pass token for the S3 bucket name
      environment: {
        BUCKET_NAME: myBucketName, // Pass token for the S3 bucket name
      }
    });
  }
}
```

JavaScript

```
const { Stack, Duration } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');
const lambda = require('aws-cdk-lib/aws-lambda');

class CdkDemoAppStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define an S3 bucket
    const myBucket = new s3.Bucket(this, 'myBucket');

    // ...

    // Define a Lambda function
    const myFunction = new lambda.Function(this, 'myFunction', {
      runtime: lambda.Runtime.NODEJS_20_X,
      handler: 'index.handler',
      code: lambda.Code.fromInline(`
        exports.handler = async function(event) {
          return {
```

```
            statusCode: 200,
            body: JSON.stringify('Hello World!'),
          };
        };
      `),
      functionName: myBucketName + 'Function', // Pass token for the S3 bucket name
      environment: {
        BUCKET_NAME: myBucketName, // Pass token for the S3 bucket name
      }
    });
  }
}

module.exports = { CdkDemoAppStack }
```

Python

```
from aws_cdk import (
    Stack
)
from constructs import Construct
from aws_cdk import aws_s3 as s3
from aws_cdk import aws_lambda as _lambda

class CdkDemoAppStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define an S3 bucket
        my_bucket = s3.Bucket(self, "myBucket")

        # ...

        # Define a Lambda function
        my_function = _lambda.Function(self, "myFunction",
            runtime=_lambda.Runtime.NODEJS_20_X,
            handler="index.handler",
            code=_lambda.Code.from_inline("""
                exports.handler = async function(event) {
                  return {
                    statusCode: 200,
                    body: JSON.stringify('Hello World!'),
```

```
                };
            };
        """),
        function_name=f"{my_bucket_name}Function",  # Pass token for the S3
 bucket name
        environment={
            "BUCKET_NAME": my_bucket_name  # Pass token for the S3 bucket name
        }
    )
```

Java

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

import java.util.Map;

public class CdkDemoAppStack extends Stack {
    public CdkDemoAppStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkDemoAppStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // Define an S3 bucket
        Bucket myBucket = Bucket.Builder.create(this, "myBucket")
            .build();

        // ...

        // Define a Lambda function
        Function myFunction = Function.Builder.create(this, "myFunction")
            .runtime(Runtime.NODEJS_20_X)
            .handler("index.handler")
```

```
            .code(Code.fromInline(
                "exports.handler = async function(event) {" +
                "return {" +
                "statusCode: 200," +
                "body: JSON.stringify('Hello World!')," +
                "};" +
                "};"
            ))
            .functionName(myBucketName + "Function") // Pass the token for the s3
 bucket to the function construct
            .environment(Map.of("BUCKET_NAME", myBucketName))  // Pass the bucket
 name as environment variable
            .build();
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.S3;
using Amazon.CDK.AWS.Lambda;
using System;
using System.Collections.Generic;

namespace CdkDemoApp
{
    public class CdkDemoAppStack : Stack
    {
        internal CdkDemoAppStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Define an S3 bucket
            var myBucket = new Bucket(this, "myBucket");

            // ...

            // Define a Lambda function
            var myFunction = new Function(this, "myFunction", new FunctionProps
            {
                 Runtime = Runtime.NODEJS_20_X,
                 Handler = "index.handler",
                 Code = Code.FromInline(@"
```

```
                      exports.handler = async function(event) {
                        return {
                          statusCode: 200,
                          body: JSON.stringify('Hello World!'),
                        };
                      };
                  "),
                  // Pass the token for the S3 bucket name
                  Environment = new Dictionary<string, string>
                  {
                      { "BUCKET_NAME", myBucketName }
                  },
                  FunctionName = $"{myBucketName}Function" // Pass the token for the
 s3 bucket to the function construct
             });
         }
     }
}
```

Go

```
package main

import (
 "fmt"

 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type CdkDemoAppStackProps struct {
 awscdk.StackProps
}

func NewCdkDemoAppStack(scope constructs.Construct, id string, props
 *CdkDemoAppStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
```

```
 stack := awscdk.NewStack(scope, &id, &sprops)

 // Define an S3 bucket
 myBucket := awss3.NewBucket(stack, jsii.String("myBucket"), &awss3.BucketProps{})

 // ...

 // Define a Lambda function
 myFunction := awslambda.NewFunction(stack, jsii.String("myFunction"),
 &awslambda.FunctionProps{
  Runtime: awslambda.Runtime_NODEJS_20_X(),
  Handler: jsii.String("index.handler"),
  Code: awslambda.Code_FromInline(jsii.String(`
   exports.handler = async function(event) {
    return {
     statusCode: 200,
     body: JSON.stringify('Hello World!'),
    };
   };
  `)),
  FunctionName: jsii.String(fmt.Sprintf("%sFunction", *myBucketName)), // Pass the
 token for the S3 bucket to the function name
  Environment: &map[string]*string{
   "BUCKET_NAME": myBucketName,
  },
 })

 return stack
}
// ...
```

When we synthesize our template, the `Ref` and `Fn::Join` intrinsic functions are used to specify the values, which will be known at deployment:

```
Resources:
  myBucket<5AF9C99B>:
    Type: AWS::S3::Bucket
    # ...
  myFunction<884E1557>:
    Type: AWS::Lambda::Function
    Properties:
      # ...
```

```
      Environment:
        Variables:
          BUCKET_NAME:
            Ref: myBucket<5AF9C99B>
      FunctionName:
        Fn::Join:
          - ""
          - - Ref: myBucket<5AF9C99B>
            - Function
      # ...
```

How token encodings work

Tokens are objects that implement the `IResolvable` interface, which contains a single `resolve` method. During synthesis, the AWS CDK calls this method to produce the final value for tokens in your CloudFormation template.

> **Note**
>
> You'll rarely work directly with the `IResolvable` interface. You will most likely only see string-encoded versions of tokens.

Token encoding types

Tokens participate in the synthesis process to produce arbitrary values of any type. Other functions typically only accept arguments of basic types, such as `string` or `number`. To use tokens in these cases, you can encode them into one of three types by using static methods on the `cdk.Token` class.

- `Token.asString` to generate a string encoding (or call `.toString()` on the token object).
- `Token.asList` to generate a list encoding.
- `Token.asNumber` to generate a numeric encoding.

These take an arbitrary value, which can be an `IResolvable`, and encode them into a primitive value of the indicated type.

Important

Because any one of the previous types can potentially be an encoded token, be careful when you parse or try to read their contents. For example, if you attempt to parse a string to extract a value from it, and the string is an encoded token, your parsing fails. Similarly, if you try to query the length of an array or perform math operations with a number, you must first verify that they aren't encoded tokens.

How to check for tokens in your app

To check whether a value has an unresolved token in it, call the `Token.isUnresolved` (Python: `is_unresolved`) method. The following is an example that checks if the value for our Amazon S3 bucket name is a token. If its not a token, we then validate the length of the bucket name:

Example

TypeScript

```
// ...

export class CdkDemoAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define an S3 bucket
    const myBucket = new s3.Bucket(this, 'myBucket');
    // ...

    // Check if bucket name is a token. If not, check if length is less than 10
 characters
    if (cdk.Token.isUnresolved(myBucketName)) {
      console.log("Token identified.");
    } else if (!cdk.Token.isUnresolved(myBucketName) && myBucketName.length > 10) {
      throw new Error('Maximum length for name is 10 characters.');
    };

    // ...
  }
}
```

JavaScript

```
const { Stack, Duration, Token, CfnOutput } = require('aws-cdk-lib');
// ...

class CdkDemoAppStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define an S3 bucket
    const myBucket = new s3.Bucket(this, 'myBucket');

    // ...

    // Check if bucket name is a token. If not, check if length is less than 10
 characters
    if (Token.isUnresolved(myBucketName)) {
      console.log("Token identified.");
    } else if (!Token.isUnresolved(myBucketName) && myBucketName.length > 10) {
      throw new Error('Maximum length for name is 10 characters.');
    };

    // ...
  }
}
```

Python

```
from aws_cdk import (
    Stack,
    Token
)
# ...

class CdkDemoAppStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define an S3 bucket
        my_bucket = s3.Bucket(self, "myBucket")

        # ...
```

```
        # Check if bucket name is a token. If not, check if length is less than 10
 characters
        if Token.is_unresolved(my_bucket_name):
            print("Token identified.")
        elif not Token.is_unresolved(my_bucket_name) and len(my_bucket_name) < 10:
            raise ValueError("Maximum length for name is 10 characters.")

        # ...
```

Java

```
// ...
import software.amazon.awscdk.Token;
import software.amazon.awscdk.services.s3.Bucket;
// ...

public class CdkDemoAppStack extends Stack {
    public CdkDemoAppStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkDemoAppStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // Define an S3 bucket
        Bucket myBucket = Bucket.Builder.create(this, "myBucket")
            .build();

        // ...

        // Get the bucket name
        String myBucketName = myBucket.getBucketName();

        // Check if the bucket name is a token. If not, check if length is less than
 10 characters
        if (Token.isUnresolved(myBucketName)) {
            System.out.println("Token identified.");
        } else if (!Token.isUnresolved(myBucketName) && myBucketName.length() > 10)
 {
            throw new IllegalArgumentException("Maximum length for name is 10
 characters.");
```

```
        }

        // ...
      }
    }
  }
```

C#

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.S3;
using Amazon.CDK.AWS.Lambda;
using System;
using System.Collections.Generic;

namespace CdkDemoApp
{
    public class CdkDemoAppStack : Stack
    {
        internal CdkDemoAppStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Define an S3 bucket
            var myBucket = new Bucket(this, "myBucket");

            // ...

            // Get the bucket name
            var myBucketName = myBucket.BucketName;

            // Check if bucket name is a token. If not, check if length is less than
 10 characters
            if (Token.IsUnresolved(myBucketName))
            {
                System.Console.WriteLine("Token identified.");
            }
            else if (!Token.IsUnresolved(myBucketName) && myBucketName.Length > 10)
            {
                throw new System.Exception("Maximum length for name is 10
 characters.");
            }
```

```
            // ...
        }
    }
}
```

Go

```
// ...

func NewCdkDemoAppStack(scope constructs.Construct, id string, props
 *CdkDemoAppStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 // Define an S3 bucket
 myBucket := awss3.NewBucket(stack, jsii.String("myBucket"), &awss3.BucketProps{})

 // ...

 // Check if the bucket name is unresolved (a token)
 if tokenUnresolved := awscdk.Token_IsUnresolved(myBucketName); tokenUnresolved !=
 nil && *tokenUnresolved {
  fmt.Println("Token identified.")
 } else if tokenUnresolved != nil && !*tokenUnresolved && len(*myBucketName) > 10 {
  panic("Maximum length for name is 10 characters.")
 }

 // ...
}
```

When we run `cdk synth`, `myBucketName` is identified as a token:

```
$ cdk synth --quiet
Token identified.
```

> **Note**
>
> You can use token encodings to escape the type system. For example, you could string-encode a token that produces a number value at synthesis time. If you use these functions, it's your responsibility to make sure that your template resolves to a usable state after synthesis.

Working with string-encoded tokens

String-encoded tokens look like the following.

```
${TOKEN[Bucket.Name.1234]}
```

They can be passed around like regular strings, and can be concatenated, as shown in the following example.

Example

TypeScript

```
const functionName = bucket.bucketName + 'Function';
```

JavaScript

```
const functionName = bucket.bucketName + 'Function';
```

Python

```
function_name = bucket.bucket_name + "Function"
```

Java

```
String functionName = bucket.getBucketName().concat("Function");
```

C#

```
string functionName = bucket.BucketName + "Function";
```

Go

```
functionName := *bucket.BucketName() + "Function"
```

You can also use string interpolation, if your language supports it, as shown in the following example.

Example

TypeScript

```
const functionName = `${bucket.bucketName}Function`;
```

JavaScript

```
const functionName = `${bucket.bucketName}Function`;
```

Python

```
function_name = f"{bucket.bucket_name}Function"
```

Java

```
String functionName = String.format("%sFunction". bucket.getBucketName());
```

C#

```
string functionName = $"${bucket.bucketName}Function";
```

Go

Use `fmt.Sprintf` for similar functionality:

```
functionName := fmt.Sprintf("%sFunction", *bucket.BucketName())
```

Avoid manipulating the string in other ways. For example, taking a substring of a string is likely to break the string token.

Working with list-encoded tokens

List-encoded tokens look like the following:

```
["#{TOKEN[Stack.NotificationArns.1234]}"]
```

The only safe thing to do with these lists is pass them directly to other constructs. Tokens in string list form cannot be concatenated, nor can an element be taken from the token. The only safe way to manipulate them is by using AWS CloudFormation intrinsic functions like `Fn.select`.

Working with number-encoded tokens

Number-encoded tokens are a set of tiny negative floating-point numbers that look like the following.

```
-1.8881545897087626e+289
```

As with list tokens, you cannot modify the number value, as doing so is likely to break the number token.

The following is an example of a construct that contains a token encoded as a number:

Example

TypeScript

```
import { Stack, Duration, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as rds from 'aws-cdk-lib/aws-rds';
import * as ec2 from 'aws-cdk-lib/aws-ec2';

export class CdkDemoAppStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    // Define a new VPC
    const vpc = new ec2.Vpc(this, 'MyVpc', {
      maxAzs: 3,  // Maximum number of availability zones to use
    });

    // Define an RDS database cluster
```

```
    const dbCluster = new rds.DatabaseCluster(this, 'MyRDSCluster', {
      engine: rds.DatabaseClusterEngine.AURORA,
      instanceProps: {
        vpc,
      },
    });

    // Get the port token (this is a token encoded as a number)
    const portToken = dbCluster.clusterEndpoint.port;

    // Print the value for our token at synthesis
    console.log("portToken: " + portToken);
  }
}
```

JavaScript

```
const { Stack, Duration } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const rds = require('aws-cdk-lib/aws-rds');
const ec2 = require('aws-cdk-lib/aws-ec2');

class CdkDemoAppStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define a new VPC
    const vpc = new ec2.Vpc(this, 'MyVpc', {
      maxAzs: 3,  // Maximum number of availability zones to use
    });

    // Define an RDS database cluster
    const dbCluster = new rds.DatabaseCluster(this, 'MyRDSCluster', {
      engine: rds.DatabaseClusterEngine.AURORA,
      instanceProps: {
        vpc,
      },
    });

    // Get the port token (this is a token encoded as a number)
    const portToken = dbCluster.clusterEndpoint.port;

    // Print the value for our token at synthesis
```

```
    console.log("portToken: " + portToken);
  }
}

module.exports = { CdkDemoAppStack }
```

Python

```
from aws_cdk import (
    Duration,
    Stack,
)
from aws_cdk import aws_rds as rds
from aws_cdk import aws_ec2 as ec2
from constructs import Construct

class CdkDemoAppStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define a new VPC
        vpc = ec2.Vpc(self, 'MyVpc',
            max_azs=3  # Maximum number of availability zones to use
        )

        # Define an RDS database cluster
        db_cluster = rds.DatabaseCluster(self, 'MyRDSCluster',
            engine=rds.DatabaseClusterEngine.AURORA,
            instance_props=rds.InstanceProps(
                vpc=vpc
            )
        )

        # Get the port token (this is a token encoded as a number)
        port_token = db_cluster.cluster_endpoint.port

        # Print the value for our token at synthesis
        print(f"portToken: {port_token}")
```

Java

```
package com.myorg;
```

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.ec2.Vpc;
import software.amazon.awscdk.services.rds.DatabaseCluster;
import software.amazon.awscdk.services.rds.DatabaseClusterEngine;
import software.amazon.awscdk.services.rds.InstanceProps;

public class CdkDemoAppStack extends Stack {
    public CdkDemoAppStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkDemoAppStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // Define a new VPC
        Vpc vpc = Vpc.Builder.create(this, "MyVpc")
            .maxAzs(3) // Maximum number of availability zones to use
            .build();

        // Define an RDS database cluster
        DatabaseCluster dbCluster = DatabaseCluster.Builder.create(this,
 "MyRDSCluster")
            .engine(DatabaseClusterEngine.AURORA)
            .instanceProps(InstanceProps.builder()
                .vpc(vpc)
                .build())
            .build();

        // Get the port token (this is a token encoded as a number)
        Number portToken = dbCluster.getClusterEndpoint().getPort();

        // Print the value for our token at synthesis
        System.out.println("portToken: " + portToken);
    }
}
```

C#

```
using Amazon.CDK;
```

```
using Constructs;
using Amazon.CDK.AWS.EC2;
using Amazon.CDK.AWS.RDS;
using System;
using System.Collections.Generic;

namespace CdkDemoApp
{
    public class CdkDemoAppStack : Stack
    {
        internal CdkDemoAppStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Define a new VPC
            var vpc = new Vpc(this, "MyVpc", new VpcProps
            {
                MaxAzs = 3  // Maximum number of availability zones to use
            });

            // Define an RDS database cluster
            var dbCluster = new DatabaseCluster(this, "MyRDSCluster", new
 DatabaseClusterProps
            {
                Engine = DatabaseClusterEngine.AURORA,  // Remove parentheses
                InstanceProps = new Amazon.CDK.AWS.RDS.InstanceProps // Specify RDS
 InstanceProps
                {
                    Vpc = vpc
                }
            });

            // Get the port token (this is a token encoded as a number)
            var portToken = dbCluster.ClusterEndpoint.Port;

            // Print the value for our token at synthesis
            System.Console.WriteLine($"portToken: {portToken}");
        }
    }
}
```

Go

```
package main
```

```
import (
 "fmt"

 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/awsec2"
 "github.com/aws/aws-cdk-go/awscdk/v2/awsrds"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

type CdkDemoAppStackProps struct {
 awscdk.StackProps
}

func NewCdkDemoAppStack(scope constructs.Construct, id string, props
 *CdkDemoAppStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 // Define a new VPC
 vpc := awsec2.NewVpc(stack, jsii.String("MyVpc"), &awsec2.VpcProps{
  MaxAzs: jsii.Number(3), // Maximum number of availability zones to use
 })

 // Define an RDS database cluster
 dbCluster := awsrds.NewDatabaseCluster(stack, jsii.String("MyRDSCluster"),
 &awsrds.DatabaseClusterProps{
  Engine: awsrds.DatabaseClusterEngine_AURORA(),
  InstanceProps: &awsrds.InstanceProps{
   Vpc: vpc,
  },
 })

 // Get the port token (this is a token encoded as a number)
 portToken := dbCluster.ClusterEndpoint().Port()

 // Print the value for our token at synthesis
 fmt.Println("portToken: ", portToken)

 return stack
```

```
}

// ...
```

When we run `cdk synth`, the value for `portToken` is displayed as a number-encoded token:

```
$ cdk synth --quiet
portToken: -1.8881545897087968e+289
```

Pass number-encoded tokens

When you pass number-encoded tokens to other constructs, it may make sense to convert them to strings first. For example, if you want to use the value of a number-encoded string as part of a concatenated string, converting it helps with readability.

In the following example,`portToken` is a number-encoded token that we want to pass to our Lambda function as part of `connectionString`:

Example

TypeScript

```
import { Stack, Duration, CfnOutput, StackProps } from 'aws-cdk-lib';
// ...
import * as lambda from 'aws-cdk-lib/aws-lambda';

export class CdkDemoAppStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    // Define a new VPC
    // ...

    // Define an RDS database cluster
    // ...

    // Get the port token (this is a token encoded as a number)
    const portToken = dbCluster.clusterEndpoint.port;

    // ...
```

```
    // Example connection string with the port token as a number
    const connectionString = `jdbc:mysql://mydb.cluster.amazonaws.com:${portToken}/
mydatabase`;

    // Use the connection string as an environment variable in a Lambda function
    const myFunction = new lambda.Function(this, 'MyLambdaFunction', {
      runtime: lambda.Runtime.NODEJS_20_X,
      handler: 'index.handler',
      code: lambda.Code.fromInline(`
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello World!'),
          };
        };
      `),
      environment: {
        DATABASE_CONNECTION_STRING: connectionString,  // Using the port token as
 part of the string
      },
    });

    // Output the value of our connection string at synthesis
    console.log("connectionString: " + connectionString);

    // Output the connection string
    new CfnOutput(this, 'ConnectionString', {
      value: connectionString,
    });
  }
}
```

JavaScript

```
const { Stack, Duration, CfnOutput } = require('aws-cdk-lib');
// ...
const lambda = require('aws-cdk-lib/aws-lambda');

class CdkDemoAppStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define a new VPC
```

```
    // ...

    // Define an RDS database cluster
    // ...

    // Get the port token (this is a token encoded as a number)
    const portToken = dbCluster.clusterEndpoint.port;

    // ...

    // Example connection string with the port token as a number
    const connectionString = `jdbc:mysql://mydb.cluster.amazonaws.com:${portToken}/
mydatabase`;

    // Use the connection string as an environment variable in a Lambda function
    const myFunction = new lambda.Function(this, 'MyLambdaFunction', {
      runtime: lambda.Runtime.NODEJS_20_X,
      handler: 'index.handler',
      code: lambda.Code.fromInline(`
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello World!'),
          };
        };
      `),
      environment: {
        DATABASE_CONNECTION_STRING: connectionString,  // Using the port token as
 part of the string
      },
    });

    // Output the value of our connection string at synthesis
    console.log("connectionString: " + connectionString);

    // Output the connection string
    new CfnOutput(this, 'ConnectionString', {
      value: connectionString,
    });
  }
}

module.exports = { CdkDemoAppStack }
```

Python

```
from aws_cdk import (
    Duration,
    Stack,
    CfnOutput,
)
from aws_cdk import aws_lambda as _lambda
# ...

class CdkDemoAppStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define a new VPC
        # ...

        # Define an RDS database cluster
        # ...

        # Get the port token (this is a token encoded as a number)
        port_token = db_cluster.cluster_endpoint.port

        # ...

        # Example connection string with the port token as a number
        connection_string = f"jdbc:mysql://mydb.cluster.amazonaws.com:{port_token}/
mydatabase"

        # Use the connection string as an environment variable in a Lambda function
        my_function = _lambda.Function(self, 'MyLambdaFunction',
            runtime=_lambda.Runtime.NODEJS_20_X,
            handler='index.handler',
            code=_lambda.Code.from_inline("""
                exports.handler = async function(event) {
                    return {
                        statusCode: 200,
                        body: JSON.stringify('Hello World!'),
                    };
                };
            """),
            environment={
```

```
                'DATABASE_CONNECTION_STRING': connection_string  # Using the port
 token as part of the string
            }
        )

        # Output the value of our connection string at synthesis
        print(f"connectionString: {connection_string}")

        # Output the connection string
        CfnOutput(self, 'ConnectionString',
            value=connection_string
        )
```

Java

```
// ...
import software.amazon.awscdk.CfnOutput;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Code;

import java.util.Map;

public class CdkDemoAppStack extends Stack {
    public CdkDemoAppStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkDemoAppStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // Define a new VPC
        // ...

        // Define an RDS database cluster
        // ...

        // Get the port token (this is a token encoded as a number)
        Number portToken = dbCluster.getClusterEndpoint().getPort();

        // ...
```

```
        // Example connection string with the port token as a number
        String connectionString = "jdbc:mysql://mydb.cluster.amazonaws.com:" +
 portToken + "/mydatabase";

        // Use the connection string as an environment variable in a Lambda function
        Function myFunction = Function.Builder.create(this, "MyLambdaFunction")
            .runtime(Runtime.NODEJS_20_X)
            .handler("index.handler")
            .code(Code.fromInline(
                "exports.handler = async function(event) {\n" +
                "  return {\n" +
                "    statusCode: 200,\n" +
                "    body: JSON.stringify('Hello World!'),\n" +
                "  };\n" +
                "};"))
            .environment(Map.of(
                "DATABASE_CONNECTION_STRING", connectionString // Using the port
 token as part of the string
            ))
            .build();

        // Output the value of our connection string at synthesis
        System.out.println("connectionString: " + connectionString);

        // Output the connection string
        CfnOutput.Builder.create(this, "ConnectionString")
            .value(connectionString)
            .build();
    }
}
```

C#

```
// ...
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.RDS;
using Amazon.CDK;
using Constructs;
using System;
using System.Collections.Generic;

namespace CdkDemoApp
{
```

```
    public class CdkDemoAppStack : Stack
    {
        internal CdkDemoAppStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Define a new VPC
            // ...

            // Define an RDS database cluster
            var dbCluster = new DatabaseCluster(this, "MyRDSCluster", new
 DatabaseClusterProps
            {
                // ... properties would go here
            });

            // Get the port token (this is a token encoded as a number)
            var portToken = dbCluster.ClusterEndpoint.Port;

            // ...

            // Example connection string with the port token as a number
            var connectionString = $"jdbc:mysql://mydb.cluster.amazonaws.com:
{portToken}/mydatabase";

            // Use the connection string as an environment variable in a Lambda
 function
            var myFunction = new Function(this, "MyLambdaFunction", new
 FunctionProps
            {
                Runtime = Runtime.NODEJS_20_X,
                Handler = "index.handler",
                Code = Code.FromInline(@"
                    exports.handler = async function(event) {
                        return {
                            statusCode: 200,
                            body: JSON.stringify('Hello World!'),
                        };
                    };
                "),
                Environment = new Dictionary<string, string>
                {
                    { "DATABASE_CONNECTION_STRING", connectionString }  // Using the
 port token as part of the string
                }
```

```
            });

            // Output the value of our connection string at synthesis
            Console.WriteLine($"connectionString: {connectionString}");

            // Output the connection string
            new CfnOutput(this, "ConnectionString", new CfnOutputProps
            {
                Value = connectionString
            });
        }
    }
}
```

Go

```
// ...
 "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
)

type CdkDemoAppStackProps struct {
 awscdk.StackProps
}

func NewCdkDemoAppStack(scope constructs.Construct, id string, props
 *CdkDemoAppStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 // Define a new VPC
 // ...

 // Define an RDS database cluster
 // ...

 // Get the port token (this is a token encoded as a number)
 portToken := dbCluster.ClusterEndpoint().Port()

 // ...
```

```
 // Example connection string with the port token as a number
  connectionString := fmt.Sprintf("jdbc:mysql://mydb.cluster.amazonaws.com:%s/
mydatabase", portToken)

 // Use the connection string as an environment variable in a Lambda function
 myFunction := awslambda.NewFunction(stack, jsii.String("MyLambdaFunction"),
 &awslambda.FunctionProps{
  Runtime: awslambda.Runtime_NODEJS_20_X(),
  Handler: jsii.String("index.handler"),
  Code: awslambda.Code_FromInline(jsii.String(`
   exports.handler = async function(event) {
    return {
     statusCode: 200,
     body: JSON.stringify('Hello World!'),
    };
   };
  `)),
  Environment: &map[string]*string{
   "DATABASE_CONNECTION_STRING": jsii.String(connectionString), // Using the port
 token as part of the string
  },
 })

 // Output the value of our connection string at synthesis
 fmt.Println("connectionString: ", connectionString)

 // Output the connection string
 awscdk.NewCfnOutput(stack, jsii.String("ConnectionString"), &awscdk.CfnOutputProps{
  Value: jsii.String(connectionString),
 })

 return stack
}

// ...
```

If we pass this value to `connectionString`, the output value when we run `cdk synth` may be confusing due to the number-encoded string:

```
$ cdk synth --quiet
connectionString: jdbc:mysql://mydb.cluster.amazonaws.com:-1.888154589708796e+289/
mydatabase
```

To convert a number-encoded token to a string, use `cdk.Tokenization.stringifyNumber(<token>)`. In the following example, we convert the number-encoded token to a string before defining our connection string:

Example

TypeScript

```
import { Stack, Duration, Tokenization, CfnOutput, StackProps } from 'aws-cdk-lib';
// ...

export class CdkDemoAppStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    // Define a new VPC
    // ...

    // Define an RDS database cluster
    // ...

    // Get the port token (this is a token encoded as a number)
    const portToken = dbCluster.clusterEndpoint.port;

    // ...

    // Convert the encoded number to an encoded string for use in the connection
 string
    const portAsString = Tokenization.stringifyNumber(portToken);

    // Example connection string with the port token as a string
    const connectionString = `jdbc:mysql://mydb.cluster.amazonaws.com:
${portAsString}/mydatabase`;

    // Use the connection string as an environment variable in a Lambda function
    const myFunction = new lambda.Function(this, 'MyLambdaFunction', {
      // ...
      environment: {
        DATABASE_CONNECTION_STRING: connectionString,  // Using the port token as
 part of the string
      },
    });
```

```
    // Output the value of our connection string at synthesis
    console.log("connectionString: " + connectionString);

    // Output the connection string
    new CfnOutput(this, 'ConnectionString', {
      value: connectionString,
    });
  }
}
```

JavaScript

```
const { Stack, Duration, Tokenization, CfnOutput } = require('aws-cdk-lib');
// ...

class CdkDemoAppStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define a new VPC
    // ...

    // Define an RDS database cluster
    // ...

    // Get the port token (this is a token encoded as a number)
    const portToken = dbCluster.clusterEndpoint.port;

    // ...

    // Convert the encoded number to an encoded string for use in the connection
 string
    const portAsString = Tokenization.stringifyNumber(portToken);

    // Example connection string with the port token as a string
    const connectionString = `jdbc:mysql://mydb.cluster.amazonaws.com:
${portAsString}/mydatabase`;

    // Use the connection string as an environment variable in a Lambda function
    const myFunction = new lambda.Function(this, 'MyLambdaFunction', {
      // ...
      environment: {
```

```
        DATABASE_CONNECTION_STRING: connectionString,  // Using the port token as
 part of the string
      },
    });

    // Output the value of our connection string at synthesis
    console.log("connectionString: " + connectionString);

    // Output the connection string
    new CfnOutput(this, 'ConnectionString', {
      value: connectionString,
    });
  }
}

module.exports = { CdkDemoAppStack }
```

Python

```
from aws_cdk import (
    Duration,
    Stack,
    Tokenization,
    CfnOutput,
)
# ...

class CdkDemoAppStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define a new VPC
        # ...

        # Define an RDS database cluster
        # ...

        # Get the port token (this is a token encoded as a number)
        port_token = db_cluster.cluster_endpoint.port

        # Convert the encoded number to an encoded string for use in the connection
 string
```

```
        port_as_string = Tokenization.stringify_number(port_token)

        # Example connection string with the port token as a string
        connection_string = f"jdbc:mysql://mydb.cluster.amazonaws.com:
{port_as_string}/mydatabase"

        # Use the connection string as an environment variable in a Lambda function
        my_function = _lambda.Function(self, 'MyLambdaFunction',
            # ...
            environment={
                'DATABASE_CONNECTION_STRING': connection_string  # Using the port
 token as part of the string
            }
        )

        # Output the value of our connection string at synthesis
        print(f"connectionString: {connection_string}")

        # Output the connection string
        CfnOutput(self, 'ConnectionString',
            value=connection_string
        )
```

Java

```
// ...
import software.amazon.awscdk.Tokenization;

public class CdkDemoAppStack extends Stack {
    public CdkDemoAppStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkDemoAppStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // Define a new VPC
        // ...

        // Define an RDS database cluster
        // ...
```

```
        // Get the port token (this is a token encoded as a number)
        Number portToken = dbCluster.getClusterEndpoint().getPort();

        // ...

        // Convert the encoded number to an encoded string for use in the connection
 string
        String portAsString = Tokenization.stringifyNumber(portToken);

        // Example connection string with the port token as a string
        String connectionString = "jdbc:mysql://mydb.cluster.amazonaws.com:" +
 portAsString + "/mydatabase";

        // Use the connection string as an environment variable in a Lambda function
        Function myFunction = Function.Builder.create(this, "MyLambdaFunction")
            // ...
            .environment(Map.of(
                "DATABASE_CONNECTION_STRING", connectionString // Using the port
 token as part of the string
            ))
            .build();

        // Output the value of our connection string at synthesis
        System.out.println("connectionString: " + connectionString);

        // Output the connection string
        CfnOutput.Builder.create(this, "ConnectionString")
            .value(connectionString)
            .build();
    }
}
```

C#

```
// ...

namespace CdkDemoApp
{
    public class CdkDemoAppStack : Stack
    {
        internal CdkDemoAppStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
```

```
            // Define a new VPC
            // ...

            // Define an RDS database cluster
            // ...

            // Get the port token (this is a token encoded as a number)
            var portToken = dbCluster.ClusterEndpoint.Port;

            // ...

            // Convert the encoded number to an encoded string for use in the
 connection string
            var portAsString = Tokenization.StringifyNumber(portToken);

            // Example connection string with the port token as a string
            var connectionString = $"jdbc:mysql://mydb.cluster.amazonaws.com:
{portAsString}/mydatabase";

            // Use the connection string as an environment variable in a Lambda
 function
            var myFunction = new Function(this, "MyLambdaFunction", new
 FunctionProps
            {
                // ...
                Environment = new Dictionary<string, string>
                {
                    { "DATABASE_CONNECTION_STRING", connectionString }  // Using the
 port token as part of the string
                }
            });

            // Output the value of our connection string at synthesis
            Console.WriteLine($"connectionString: {connectionString}");

            // Output the connection string
            new CfnOutput(this, "ConnectionString", new CfnOutputProps
            {
                Value = connectionString
            });
        }
    }
}
```

Go

```
// ...

func NewCdkDemoAppStack(scope constructs.Construct, id string, props
 *CdkDemoAppStackProps) awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 // Define a new VPC
 // ...

 // Define an RDS database cluster
 // ...

 // Get the port token (this is a token encoded as a number)
 portToken := dbCluster.ClusterEndpoint().Port()

 // ...

 // Convert the encoded number to an encoded string for use in the connection string
 portAsString := awscdk.Tokenization_StringifyNumber(portToken)

 // Example connection string with the port token as a string
 connectionString := fmt.Sprintf("jdbc:mysql://mydb.cluster.amazonaws.com:%s/
mydatabase", portAsString)

 // Use the connection string as an environment variable in a Lambda function
 myFunction := awslambda.NewFunction(stack, jsii.String("MyLambdaFunction"),
 &awslambda.FunctionProps{
  // ...
  Environment: &map[string]*string{
   "DATABASE_CONNECTION_STRING": jsii.String(connectionString), // Using the port
 token as part of the string
  },
 })

 // Output the value of our connection string at synthesis
 fmt.Println("connectionString: ", connectionString)

 // Output the connection string
```

```
	awscdk.NewCfnOutput(stack, jsii.String("ConnectionString"), &awscdk.CfnOutputProps{
		Value: jsii.String(connectionString),
	})

	fmt.Println(myFunction)

	return stack
}

// ...
```

When we run `cdk synth`, the value for our connection string is represented in a cleaner and clearer format:

```
$ cdk synth --quiet
connectionString: jdbc:mysql://mydb.cluster.amazonaws.com:${Token[TOKEN.242]}/
mydatabase
```

Lazy values

In addition to representing deploy-time values, such as AWS CloudFormation [parameters](), tokens are also commonly used to represent synthesis-time lazy values. These are values for which the final value will be determined before synthesis has completed, but not at the point where the value is constructed. Use tokens to pass a literal string or number value to another construct, while the actual value at synthesis time might depend on some calculation that has yet to occur.

You can construct tokens representing synth-time lazy values using static methods on the `Lazy` class, such as [`Lazy.string`]() and [`Lazy.number`](). These methods accept an object whose `produce` property is a function that accepts a context argument and returns the final value when called.

The following example creates an Auto Scaling group whose capacity is determined after its creation.

Example

TypeScript

```
let actualValue: number;

new AutoScalingGroup(this, 'Group', {
```

```
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return actualValue;
    }
  })
});

// At some later point
actualValue = 10;
```

JavaScript

```
let actualValue;

new AutoScalingGroup(this, 'Group', {
  desiredCapacity: Lazy.numberValue({
    produce(context) {
      return (actualValue);
    }
  })
});

// At some later point
actualValue = 10;
```

Python

```
class Producer:
    def __init__(self, func):
        self.produce = func

actual_value = None

AutoScalingGroup(self, "Group",
    desired_capacity=Lazy.number_value(Producer(lambda context: actual_value))
)

# At some later point
actual_value = 10
```

Java

```
double actualValue = 0;
```

```
class ProduceActualValue implements INumberProducer {

    @Override
    public Number produce(IResolveContext context) {
        return actualValue;
    }
}

AutoScalingGroup.Builder.create(this, "Group")
    .desiredCapacity(Lazy.numberValue(new ProduceActualValue())).build();

// At some later point
actualValue = 10;
```

C#

```
public class NumberProducer : INumberProducer
{
    Func<Double> function;

    public NumberProducer(Func<Double> function)
    {
        this.function = function;
    }

    public Double Produce(IResolveContext context)
    {
        return function();
    }
}

double actualValue = 0;

new AutoScalingGroup(this, "Group", new AutoScalingGroupProps
{
    DesiredCapacity = Lazy.NumberValue(new NumberProducer(() => actualValue))
});

// At some later point
actualValue = 10;
```

Converting to JSON

Sometimes you want to produce a JSON string of arbitrary data, and you may not know whether the data contains tokens. To properly JSON-encode any data structure, regardless of whether it contains tokens, use the method `stack.toJsonString`, as shown in the following example.

Example

TypeScript

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

JavaScript

```
const stack = Stack.of(this);
const str = stack.toJsonString({
  value: bucket.bucketName
});
```

Python

```
stack = Stack.of(self)
string = stack.to_json_string(dict(value=bucket.bucket_name))
```

Java

```
Stack stack = Stack.of(this);
String stringVal = stack.toJsonString(java.util.Map.of(    // Map.of requires Java
 9+
        put("value", bucket.getBucketName())));
```

C#

```
var stack = Stack.Of(this);
var stringVal = stack.ToJsonString(new Dictionary<string, string>
{
    ["value"] = bucket.BucketName
```

```
});
```

Parameters and the AWS CDK

Parameters are custom values that are supplied at deployment time. Parameters are a feature of AWS CloudFormation. Since the AWS Cloud Development Kit (AWS CDK) synthesizes AWS CloudFormation templates, it also offers support for deployment-time parameters.

About parameters

Using the AWS CDK, you can define parameters, which can then be used in the properties of constructs you create. You can also deploy stacks that contain parameters.

When deploying the AWS CloudFormation template using the AWS CDK CLI, you provide the parameter values on the command line. If you deploy the template through the AWS CloudFormation console, you are prompted for the parameter values.

In general, we recommend against using AWS CloudFormation parameters with the AWS CDK. The usual ways to pass values into AWS CDK apps are context values and environment variables. Because they are not available at synthesis time, parameter values cannot be easily used for flow control and other purposes in your CDK app.

> **Note**
>
> To do control flow with parameters, you can use `CfnCondition` constructs, although this is awkward compared to native `if` statements.

Using parameters requires you to be mindful of how the code you're writing behaves at deployment time, and also at synthesis time. This makes it harder to understand and reason about your AWS CDK application, in many cases for little benefit.

Generally, it's better to have your CDK app accept necessary information in a well-defined way and use it directly to declare constructs in your CDK app. An ideal AWS CDK–generated AWS CloudFormation template is concrete, with no values remaining to be specified at deployment time.

There are, however, use cases to which AWS CloudFormation parameters are uniquely suited. If you have separate teams defining and deploying infrastructure, for example, you can use parameters to make the generated templates more widely useful. Also, because the AWS CDK supports AWS CloudFormation parameters, you can use the AWS CDK with AWS services that use AWS CloudFormation templates (such as Service Catalog). These AWS services use parameters to configure the template that's being deployed.

Learn more

For instructions on developing CDK apps with parameters, see Use CloudFormation parameters to get a CloudFormation value.

Tags and the AWS CDK

Tags are informational key-value elements that you can add to constructs in your AWS CDK app. A tag applied to a given construct also applies to all of its taggable children. Tags are included in the AWS CloudFormation template synthesized from your app and are applied to the AWS resources it deploys. You can use tags to identify and categorize resources for the following purposes:

- Simplifying management
- Cost allocation
- Access control
- Any other purposes that you devise

> **Tip**
>
> For more information about how you can use tags with your AWS resources, see Best Practices for Tagging AWS Resources in the *AWS Whitepaper*.

Using tags

The Tags class includes the static method `of()`, through which you can add tags to, or remove tags from, the specified construct.

- `Tags.of(<SCOPE>).add()` applies a new tag to the given construct and all of its children.
- `Tags.of(<SCOPE>).remove()` removes a tag from the given construct and any of its children, including tags a child construct may have applied to itself.

> **Note**
>
> Tagging is implemented using [Aspects and the AWS CDK](). Aspects are a way to apply an operation (such as tagging) to all constructs in a given scope.

The following example applies the tag **key** with the value **value** to a construct.

Example

TypeScript

```
Tags.of(myConstruct).add('key', 'value');
```

JavaScript

```
Tags.of(myConstruct).add('key', 'value');
```

Python

```
Tags.of(my_construct).add("key", "value")
```

Java

```
Tags.of(myConstruct).add("key", "value");
```

C#

```
Tags.Of(myConstruct).Add("key", "value");
```

Go

```
awscdk.Tags_Of(myConstruct).Add(jsii.String("key"), jsii.String("value"),
 &awscdk.TagProps{})
```

The following example deletes the tag **key** from a construct.

Example

TypeScript

```
Tags.of(myConstruct).remove('key');
```

JavaScript

```
Tags.of(myConstruct).remove('key');
```

Python

```
Tags.of(my_construct).remove("key")
```

Java

```
Tags.of(myConstruct).remove("key");
```

C#

```
Tags.Of(myConstruct).Remove("key");
```

Go

```
awscdk.Tags_Of(myConstruct).Remove(jsii.String("key"), &awscdk.TagProps{})
```

If you are using `Stage` constructs, apply the tag at the `Stage` level or below. Tags are not applied across `Stage` boundaries.

Tag priorities

The AWS CDK applies and removes tags recursively. If there are conflicts, the tagging operation with the highest priority wins. (Priorities are set using the optional `priority` property.) If the priorities of two operations are the same, the tagging operation closest to the bottom of the construct tree wins. By default, applying a tag has a priority of 100 (except for tags added directly to an AWS CloudFormation resource, which has a priority of 50). The default priority for removing a tag is 200.

The following applies a tag with a priority of 300 to a construct.

Example

TypeScript

```
Tags.of(myConstruct).add('key', 'value', {
  priority: 300
});
```

JavaScript

```
Tags.of(myConstruct).add('key', 'value', {
  priority: 300
});
```

Python

```
Tags.of(my_construct).add("key", "value", priority=300)
```

Java

```
Tags.of(myConstruct).add("key", "value", TagProps.builder()
        .priority(300).build());
```

C#

```
Tags.Of(myConstruct).Add("key", "value", new TagProps { Priority = 300 });
```

Go

```
awscdk.Tags_Of(myConstruct).Add(jsii.String("key"), jsii.String("value"),
 &awscdk.TagProps{
  Priority: jsii.Number(300),
})
```

Optional properties

Tags support `properties` that fine-tune how tags are applied to, or removed from, resources. All properties are optional.

`applyToLaunchedInstances` (Python: `apply_to_launched_instances`)

Available for add() only. By default, tags are applied to instances launched in an Auto Scaling group. Set this property to **false** to ignore instances launched in an Auto Scaling group.

`includeResourceTypes`/`excludeResourceTypes` (Python: `include_resource_types`/`exclude_resource_types`)

Use these to manipulate tags only on a subset of resources, based on AWS CloudFormation resource types. By default, the operation is applied to all resources in the construct subtree, but this can be changed by including or excluding certain resource types. Exclude takes precedence over include, if both are specified.

`priority`

Use this to set the priority of this operation with respect to other `Tags.add()` and `Tags.remove()` operations. Higher values take precedence over lower values. The default is 100 for add operations (50 for tags applied directly to AWS CloudFormation resources) and 200 for remove operations.

The following example applies the tag **tagname** with the value **value** and priority **100** to resources of type **AWS::Xxx::Yyy** in the construct. It doesn't apply the tag to instances launched in an Amazon EC2 Auto Scaling group or to resources of type **AWS::Xxx::Zzz**. (These are placeholders for two arbitrary but different AWS CloudFormation resource types.)

Example

TypeScript

```
Tags.of(myConstruct).add('tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 100,
});
```

JavaScript

```
Tags.of(myConstruct).add('tagname', 'value', {
  applyToLaunchedInstances: false,
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
```

```
  priority: 100
});
```

Python

```
Tags.of(my_construct).add("tagname", "value",
    apply_to_launched_instances=False,
    include_resource_types=["AWS::Xxx::Yyy"],
    exclude_resource_types=["AWS::Xxx::Zzz"],
    priority=100)
```

Java

```
Tags.of(myConstruct).add("tagname", "value", TagProps.builder()
                .applyToLaunchedInstances(false)
                .includeResourceTypes(Arrays.asList("AWS::Xxx::Yyy"))
                .excludeResourceTypes(Arrays.asList("AWS::Xxx::Zzz"))
                .priority(100).build());
```

C#

```
Tags.Of(myConstruct).Add("tagname", "value", new TagProps
{
    ApplyToLaunchedInstances = false,
    IncludeResourceTypes = ["AWS::Xxx::Yyy"],
    ExcludeResourceTypes = ["AWS::Xxx::Zzz"],
    Priority = 100
});
```

Go

```
awscdk.Tags_Of(myConstruct).Add(jsii.String("tagname"), jsii.String("value"),
 &awscdk.TagProps{
  ApplyToLaunchedInstances: jsii.Bool(false),
  IncludeResourceTypes:     &[]*string{jsii.String("AWS::Xxx:Yyy")},
  ExcludeResourceTypes:     &[]*string{jsii.String("AWS::Xxx:Zzz")},
  Priority:                 jsii.Number(100),
})
```

The following example removes the tag **tagname** with priority **200** from resources of type **AWS::Xxx::Yyy** in the construct, but not from resources of type **AWS::Xxx::Zzz**.

Example

TypeScript

```
Tags.of(myConstruct).remove('tagname', {
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 200,
});
```

JavaScript

```
Tags.of(myConstruct).remove('tagname', {
  includeResourceTypes: ['AWS::Xxx::Yyy'],
  excludeResourceTypes: ['AWS::Xxx::Zzz'],
  priority: 200
});
```

Python

```
Tags.of(my_construct).remove("tagname",
    include_resource_types=["AWS::Xxx::Yyy"],
    exclude_resource_types=["AWS::Xxx::Zzz"],
    priority=200,)
```

Java

```
Tags.of((myConstruct).remove("tagname", TagProps.builder()
        .includeResourceTypes(Arrays.asList("AWS::Xxx::Yyy"))
        .excludeResourceTypes(Arrays.asList("AWS::Xxx::Zzz"))
        .priority(100).build());
        )
```

C#

```
Tags.Of(myConstruct).Remove("tagname", new TagProps
{
    IncludeResourceTypes = ["AWS::Xxx::Yyy"],
    ExcludeResourceTypes = ["AWS::Xxx::Zzz"],
    Priority = 100
});
```

Go

```
awscdk.Tags_Of(myConstruct).Remove(jsii.String("tagname"), &awscdk.TagProps{
  IncludeResourceTypes: &[]*string{jsii.String("AWS::Xxx:Yyy")},
  ExcludeResourceTypes: &[]*string{jsii.String("AWS::Xxx:Zzz")},
  Priority:             jsii.Number(200),
})
```

Example

The following example adds the tag key **StackType** with value **TheBest** to any resource created within the `Stack` named `MarketingSystem`. Then it removes it again from all resources except Amazon EC2 VPC subnets. The result is that only the subnets have the tag applied.

Example

TypeScript

```
import { App, Stack, Tags } from 'aws-cdk-lib';

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add('StackType', 'TheBest');

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove('StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

JavaScript

```
const { App, Stack, Tags } = require('aws-cdk-lib');

const app = new App();
const theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add('StackType', 'TheBest');
```

```
// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove('StackType', {
  excludeResourceTypes: ['AWS::EC2::Subnet']
});
```

Python

```
from aws_cdk import App, Stack, Tags

app = App();
the_best_stack = Stack(app, 'MarketingSystem')

# Add a tag to all constructs in the stack
Tags.of(the_best_stack).add("StackType", "TheBest")

# Remove the tag from all resources except subnet resources
Tags.of(the_best_stack).remove("StackType",
    exclude_resource_types=["AWS::EC2::Subnet"])
```

Java

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Tags;

// Add a tag to all constructs in the stack
Tags.of(theBestStack).add("StackType", "TheBest");

// Remove the tag from all resources except subnet resources
Tags.of(theBestStack).remove("StackType", TagProps.builder()
        .excludeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
        .build());
```

C#

```
using Amazon.CDK;

var app = new App();
var theBestStack = new Stack(app, 'MarketingSystem');

// Add a tag to all constructs in the stack
Tags.Of(theBestStack).Add("StackType", "TheBest");
```

```
// Remove the tag from all resources except subnet resources
Tags.Of(theBestStack).Remove("StackType", new TagProps
{
    ExcludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

Go

```
import "github.com/aws/aws-cdk-go/awscdk/v2"
app := awscdk.NewApp(nil)
theBestStack := awscdk.NewStack(app, jsii.String("MarketingSystem"),
 &awscdk.StackProps{})

// Add a tag to all constructs in the stack
awscdk.Tags_Of(theBestStack).Add(jsii.String("StackType"), jsii.String("TheBest"),
 &awscdk.TagProps{})

// Remove the tag from all resources except subnet resources
awscdk.Tags_Of(theBestStack).Add(jsii.String("StackType"), jsii.String("TheBest"),
 &awscdk.TagProps{
  ExcludeResourceTypes: &[]*string{jsii.String("AWS::EC2::Subnet")},
})
```

The following code achieves the same result. Consider which approach (inclusion or exclusion) makes your intent clearer.

Example

TypeScript

```
Tags.of(theBestStack).add('StackType', 'TheBest',
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

JavaScript

```
Tags.of(theBestStack).add('StackType', 'TheBest',
  { includeResourceTypes: ['AWS::EC2::Subnet']});
```

Python

```
Tags.of(the_best_stack).add("StackType", "TheBest",
```

```
    include_resource_types=["AWS::EC2::Subnet"])
```

Java

```
Tags.of(theBestStack).add("StackType", "TheBest", TagProps.builder()
        .includeResourceTypes(Arrays.asList("AWS::EC2::Subnet"))
        .build());
```

C#

```
Tags.Of(theBestStack).Add("StackType", "TheBest", new TagProps {
    IncludeResourceTypes = ["AWS::EC2::Subnet"]
});
```

Go

```
awscdk.Tags_Of(theBestStack).Add(jsii.String("StackType"), jsii.String("TheBest"),
 &awscdk.TagProps{
  IncludeResourceTypes: &[]*string{jsii.String("AWS::EC2::Subnet")},
})
```

Tagging single constructs

`Tags.of(scope).add(key, value)` is the standard way to add tags to constructs in the AWS CDK. Its tree-walking behavior, which recursively tags all taggable resources under the given scope, is almost always what you want. Sometimes, however, you need to tag a specific, arbitrary construct (or constructs).

One such case involves applying tags whose value is derived from some property of the construct being tagged. The standard tagging approach recursively applies the same key and value to all matching resources in the scope. However, here the value could be different for each tagged construct.

Tags are implemented using aspects, and the CDK calls the tag's `visit()` method for each construct under the scope you specified using `Tags.of(scope)`. We can call `Tag.visit()` directly to apply a tag to a single construct.

Example

TypeScript

```
new cdk.Tag(key, value).visit(scope);
```

JavaScript

```
new cdk.Tag(key, value).visit(scope);
```

Python

```
cdk.Tag(key, value).visit(scope)
```

Java

```
Tag.Builder.create(key, value).build().visit(scope);
```

C#

```
new Tag(key, value).Visit(scope);
```

Go

```
awscdk.NewTag(key, value, &awscdk.TagProps{}).Visit(scope)
```

You can tag all constructs under a scope but let the values of the tags derive from properties of each construct. To do so, write an aspect and apply the tag in the aspect's `visit()` method as shown in the preceding example. Then, add the aspect to the desired scope using `Aspects.of(scope).add(aspect)`.

The following example applies a tag to each resource in a stack containing the resource's path.

Example

TypeScript

```
class PathTagger implements cdk.IAspect {
  visit(node: IConstruct) {
    new cdk.Tag("aws-cdk-path", node.node.path).visit(node);
```

```
  }
}

stack = new MyStack(app);
cdk.Aspects.of(stack).add(new PathTagger())
```

JavaScript

```
class PathTagger {
  visit(node) {
    new cdk.Tag("aws-cdk-path", node.node.path).visit(node);
  }
}

stack = new MyStack(app);
cdk.Aspects.of(stack).add(new PathTagger())
```

Python

```
@jsii.implements(cdk.IAspect)
class PathTagger:
    def visit(self, node: IConstruct):
        cdk.Tag("aws-cdk-path", node.node.path).visit(node)

stack = MyStack(app)
cdk.Aspects.of(stack).add(PathTagger())
```

Java

```
final class PathTagger implements IAspect {
 public void visit(IConstruct node) {
  Tag.Builder.create("aws-cdk-path", node.getNode().getPath()).build().visit(node);
 }
}

stack stack = new MyStack(app);
Aspects.of(stack).add(new PathTagger());
```

C#

```
public class PathTagger : IAspect
{
```

```
    public void Visit(IConstruct node)
    {
        new Tag("aws-cdk-path", node.Node.Path).Visit(node);
    }
}

var stack = new MyStack(app);
Aspects.Of(stack).Add(new PathTagger);
```

Tip

The logic of conditional tagging, including priorities, resource types, and so on, is built into the `Tag` class. You can use these features when applying tags to arbitrary resources; the tag is not applied if the conditions aren't met. Also, the `Tag` class only tags taggable resources, so you don't need to test whether a construct is taggable before applying a tag.

Assets and the AWS CDK

Assets are local files, directories, or Docker images that can be bundled into AWS CDK libraries and apps. For example, an asset might be a directory that contains the handler code for an AWS Lambda function. Assets can represent any artifact that the app needs to operate.

The following tutorial video provides a comprehensive overview of CDK assets, and explains how you can use them in your infrastructure as code (IaC).

You add assets through APIs that are exposed by specific AWS constructs. For example, when you define a [`lambda.Function`] construct, the [`code`] property lets you pass an [`asset`] (directory). `Function` uses assets to bundle the contents of the directory and use it for the function's code. Similarly, [`ecs.ContainerImage.fromAsset`] uses a Docker image built from a local directory when defining an Amazon ECS task definition.

Assets in detail

When you refer to an asset in your app, the [cloud assembly] that's synthesized from your application includes metadata information with instructions for the AWS CDK CLI. The instructions include where to find the asset on the local disk and what type of bundling to perform based on the asset type, such as a directory to compress (zip) or a Docker image to build.

The AWS CDK generates a source hash for assets. This can be used at construction time to determine whether the contents of an asset have changed.

By default, the AWS CDK creates a copy of the asset in the cloud assembly directory, which defaults to `cdk.out`, under the source hash. This way, the cloud assembly is self-contained, so if it moved over to a different host for deployment, it can still be deployed. See Cloud assemblies for details.

When the AWS CDK deploys an app that references assets (either directly by the app code or through a library), the AWS CDK CLI first prepares and publishes the assets to an Amazon S3 bucket or Amazon ECR repository. (The S3 bucket or repository is created during bootstrapping.) Only then are the resources defined in the stack deployed.

This section describes the low-level APIs available in the framework.

Asset types

The AWS CDK supports the following types of assets:

Amazon S3 assets

These are local files and directories that the AWS CDK uploads to Amazon S3.

Docker Image

These are Docker images that the AWS CDK uploads to Amazon ECR.

These asset types are explained in the following sections.

Amazon S3 assets

You can define local files and directories as assets, and the AWS CDK packages and uploads them to Amazon S3 through the `aws-s3-assets` module.

The following example defines a local directory asset and a file asset.

Example

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
```

```
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');

// Archived and uploaded to Amazon S3 as a .zip file
const directoryAsset = new Asset(this, "SampleZippedDirAsset", {
  path: path.join(__dirname, "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
const fileAsset = new Asset(this, 'SampleSingleFileAsset', {
  path: path.join(__dirname, 'file-asset.txt')
});
```

Python

```
import os.path
dirname = os.path.dirname(__file__)

from aws_cdk.aws_s3_assets import Asset

# Archived and uploaded to Amazon S3 as a .zip file
directory_asset = Asset(self, "SampleZippedDirAsset",
  path=os.path.join(dirname, "sample-asset-directory")
)

# Uploaded to Amazon S3 as-is
file_asset = Asset(self, 'SampleSingleFileAsset',
  path=os.path.join(dirname, 'file-asset.txt')
)
```

Java

```
import java.io.File;
```

```
import software.amazon.awscdk.services.s3.assets.Asset;

// Directory where app was started
File startDir = new File(System.getProperty("user.dir"));

// Archived and uploaded to Amazon S3 as a .zip file
Asset directoryAsset = Asset.Builder.create(this, "SampleZippedDirAsset")
                .path(new File(startDir, "sample-asset-
directory").toString()).build();

// Uploaded to Amazon S3 as-is
Asset fileAsset = Asset.Builder.create(this, "SampleSingleFileAsset")
                .path(new File(startDir, "file-asset.txt").toString()).build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.S3.Assets;

// Archived and uploaded to Amazon S3 as a .zip file
var directoryAsset = new Asset(this, "SampleZippedDirAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "sample-asset-directory")
});

// Uploaded to Amazon S3 as-is
var fileAsset = new Asset(this, "SampleSingleFileAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), "file-asset.txt")
});
```

Go

```
dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

awss3assets.NewAsset(stack, jsii.String("SampleZippedDirAsset"),
 awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "sample-asset-directory")),
})
```

```
awss3assets.NewAsset(stack, jsii.String("SampleSingleFileAsset"),
 awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "file-asset.txt")),
})
```

In most cases, you don't need to directly use the APIs in the `aws-s3-assets` module. Modules that support assets, such as `aws-lambda`, have convenience methods so that you can use assets. For Lambda functions, the [`fromAsset()`]() static method enables you to specify a directory or a .zip file in the local file system.

Lambda function example

A common use case is creating Lambda functions with the handler code as an Amazon S3 asset.

The following example uses an Amazon S3 asset to define a Python handler in the local directory `handler`. It also creates a Lambda function with the local directory asset as the `code` property. Following is the Python code for the handler.

```
def lambda_handler(event, context):
  message = 'Hello World!'
  return {
    'message': message
  }
```

The code for the main AWS CDK app should look like the following.

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Constructs } from 'constructs';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as path from 'path';

export class HelloAssetStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    new lambda.Function(this, 'myLambdaFunction', {
      code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
```

```
      runtime: lambda.Runtime.PYTHON_3_6,
      handler: 'index.lambda_handler'
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const path = require('path');

class HelloAssetStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    new lambda.Function(this, 'myLambdaFunction', {
      code: lambda.Code.fromAsset(path.join(__dirname, 'handler')),
      runtime: lambda.Runtime.PYTHON_3_6,
      handler: 'index.lambda_handler'
    });
  }
}

module.exports = { HelloAssetStack }
```

Python

```
from aws_cdk import Stack
from constructs import Construct
from aws_cdk import aws_lambda as lambda_

import os.path
dirname = os.path.dirname(__file__)

class HelloAssetStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        lambda_.Function(self, 'myLambdaFunction',
            code=lambda_.Code.from_asset(os.path.join(dirname, 'handler')),
            runtime=lambda_.Runtime.PYTHON_3_6,
            handler="index.lambda_handler")
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class HelloAssetStack extends Stack {

    public HelloAssetStack(final App scope, final String id) {
        this(scope, id, null);
    }

    public HelloAssetStack(final App scope, final String id, final StackProps props)
 {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Function.Builder.create(this, "myLambdaFunction")
                .code(Code.fromAsset(new File(startDir, "handler").toString()))
                .runtime(Runtime.PYTHON_3_6)
                .handler("index.lambda_handler").build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using System.IO;

public class HelloAssetStack : Stack
{
    public HelloAssetStack(Construct scope, string id, StackProps props) :
 base(scope, id, props)
    {
        new Function(this, "myLambdaFunction", new FunctionProps
        {
            Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(),
 "handler")),
```

```
            Runtime = Runtime.PYTHON_3_6,
            Handler = "index.lambda_handler"
        });
    }
}
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
)

func HelloAssetStack(scope constructs.Construct, id string, props
 *HelloAssetStackProps) awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, id, sprops)

  dirName, err := os.Getwd()
  if err != nil {
    panic(err)
  }

  awslambda.NewFunction(stack, jsii.String("myLambdaFunction"),
 awslambda.FunctionProps{
    Code: awslambda.AssetCode_FromAsset(jsii.String(path.Join(dirName, "handler")),
 awss3assets.AssetOptions{}),
    Runtime: awslambda.Runtime_PYTHON_3_6(),
    Handler: jsii.String("index.lambda_handler"),
  })

  return stack
}
```

The `Function` method uses assets to bundle the contents of the directory and use it for the function's code.

> **Tip**
>
> Java `.jar` files are ZIP files with a different extension. These are uploaded as-is to Amazon S3, but when deployed as a Lambda function, the files they contain are extracted, which you might not want. To avoid this, place the `.jar` file in a directory and specify that directory as the asset.

Deploy-time attributes example

Amazon S3 asset types also expose deploy-time attributes that can be referenced in AWS CDK libraries and apps. The AWS CDK CLI command `cdk synth` displays asset properties as AWS CloudFormation parameters.

The following example uses deploy-time attributes to pass the location of an image asset into a Lambda function as environment variables. (The kind of file doesn't matter; the PNG image used here is only an example.)

Example

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';
import * as path from 'path';

const imageAsset = new Asset(this, "SampleAsset", {
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {
  code: lambda.Code.asset(path.join(__dirname, "handler")),
  runtime: lambda.Runtime.PYTHON_3_6,
  handler: "index.lambda_handler",
  environment: {
    'S3_BUCKET_NAME': imageAsset.s3BucketName,
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,
    'S3_OBJECT_URL': imageAsset.s3ObjectUrl
  }
```

```
});
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');
const path = require('path');

const imageAsset = new Asset(this, "SampleAsset", {
  path: path.join(__dirname, "images/my-image.png")
});

new lambda.Function(this, "myLambdaFunction", {
  code: lambda.Code.asset(path.join(__dirname, "handler")),
  runtime: lambda.Runtime.PYTHON_3_6,
  handler: "index.lambda_handler",
  environment: {
    'S3_BUCKET_NAME': imageAsset.s3BucketName,
    'S3_OBJECT_KEY': imageAsset.s3ObjectKey,
    'S3_OBJECT_URL': imageAsset.s3ObjectUrl
  }
});
```

Python

```
import os.path

import aws_cdk.aws_lambda as lambda_
from aws_cdk.aws_s3_assets import Asset

dirname = os.path.dirname(__file__)

image_asset = Asset(self, "SampleAsset",
    path=os.path.join(dirname, "images/my-image.png"))

lambda_.Function(self, "myLambdaFunction",
    code=lambda_.Code.asset(os.path.join(dirname, "handler")),
    runtime=lambda_.Runtime.PYTHON_3_6,
    handler="index.lambda_handler",
    environment=dict(
        S3_BUCKET_NAME=image_asset.s3_bucket_name,
        S3_OBJECT_KEY=image_asset.s3_object_key,
        S3_OBJECT_URL=image_asset.s3_object_url))
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.s3.assets.Asset;

public class FunctionStack extends Stack {
    public FunctionStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset imageAsset = Asset.Builder.create(this, "SampleAsset")
                .path(new File(startDir, "images/my-image.png").toString()).build())

        Function.Builder.create(this, "myLambdaFunction")
                .code(Code.fromAsset(new File(startDir, "handler").toString()))
                .runtime(Runtime.PYTHON_3_6)
                .handler("index.lambda_handler")
                .environment(java.util.Map.of(    // Java 9 or later
                    "S3_BUCKET_NAME", imageAsset.getS3BucketName(),
                    "S3_OBJECT_KEY", imageAsset.getS3ObjectKey(),
                    "S3_OBJECT_URL", imageAsset.getS3ObjectUrl()))
                .build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;
using System.Collections.Generic;

var imageAsset = new Asset(this, "SampleAsset", new AssetProps
{
    Path = Path.Combine(Directory.GetCurrentDirectory(), @"images\my-image.png")
});
```

```
new Function(this, "myLambdaFunction", new FunctionProps
{
    Code = Code.FromAsset(Path.Combine(Directory.GetCurrentDirectory(), "handler")),
    Runtime = Runtime.PYTHON_3_6,
    Handler = "index.lambda_handler",
    Environment = new Dictionarystring, string
    {
        ["S3_BUCKET_NAME"] = imageAsset.S3BucketName,
        ["S3_OBJECT_KEY"] = imageAsset.S3ObjectKey,
        ["S3_OBJECT_URL"] = imageAsset.S3ObjectUrl
    }
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

imageAsset := awss3assets.NewAsset(stack, jsii.String("SampleAsset"),
 awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "images/my-image.png")),
})

awslambda.NewFunction(stack, jsii.String("myLambdaFunction"),
 awslambda.FunctionProps{
  Code: awslambda.AssetCode_FromAsset(jsii.String(path.Join(dirName, "handler"))),
  Runtime: awslambda.Runtime_PYTHON_3_6(),
  Handler: jsii.String("index.lambda_handler"),
  Environment: map[string]*string{
    "S3_BUCKET_NAME": imageAsset.S3BucketName(),
    "S3_OBJECT_KEY": imageAsset.S3ObjectKey(),
```

```
    "S3_URL": imageAsset.S3ObjectUrl(),
  },
})
```

Permissions

If you use Amazon S3 assets directly through the `aws-s3-assets` module, IAM roles, users, or groups, and you need to read assets in runtime, then grant those assets IAM permissions through the `asset.grantRead` method.

The following example grants an IAM group read permissions on a file asset.

Example

TypeScript

```
import { Asset } from 'aws-cdk-lib/aws-s3-assets';
import * as path from 'path';

const asset = new Asset(this, 'MyFile', {
  path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

JavaScript

```
const { Asset } = require('aws-cdk-lib/aws-s3-assets');
const path = require('path');

const asset = new Asset(this, 'MyFile', {
  path: path.join(__dirname, 'my-image.png')
});

const group = new iam.Group(this, 'MyUserGroup');
asset.grantRead(group);
```

Python

```
from aws_cdk.aws_s3_assets import Asset
import aws_cdk.aws_iam as iam
```

```
import os.path
dirname = os.path.dirname(__file__)

        asset = Asset(self, "MyFile",
            path=os.path.join(dirname, "my-image.png"))

        group = iam.Group(self, "MyUserGroup")
        asset.grant_read(group)
```

Java

```
import java.io.File;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.iam.Group;
import software.amazon.awscdk.services.s3.assets.Asset;

public class GrantStack extends Stack {
    public GrantStack(final App scope, final String id, final StackProps props) {
        super(scope, id, props);

        File startDir = new File(System.getProperty("user.dir"));

        Asset asset = Asset.Builder.create(this, "SampleAsset")
                .path(new File(startDir, "images/my-image.png").toString()).build();

        Group group = new Group(this, "MyUserGroup");
        asset.grantRead(group);   }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.IAM;
using Amazon.CDK.AWS.S3.Assets;
using System.IO;

var asset = new Asset(this, "MyFile", new AssetProps {
    Path = Path.Combine(Path.Combine(Directory.GetCurrentDirectory(), @"images\my-
image.png"))
});
```

```
var group = new Group(this, "MyUserGroup");
asset.GrantRead(group);
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsiam"
  "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awss3assets.NewAsset(stack, jsii.String("MyFile"), awss3assets.AssetProps{
  Path: jsii.String(path.Join(dirName, "my-image.png")),
})

group := awsiam.NewGroup(stack, jsii.String("MyUserGroup"), awsiam.GroupProps{})

asset.GrantRead(group)
```

Docker image assets

The AWS CDK supports bundling local Docker images as assets through the `aws-ecr-assets` module.

The following example defines a Docker image that is built locally and pushed to Amazon ECR. Images are built from a local Docker context directory (with a Dockerfile) and uploaded to Amazon ECR by the AWS CDK CLI or your app's CI/CD pipeline. The images can be naturally referenced in your AWS CDK app.

Example

TypeScript

```
import { DockerImageAsset } from 'aws-cdk-lib/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

JavaScript

```
const { DockerImageAsset } = require('aws-cdk-lib/aws-ecr-assets');

const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});
```

Python

```
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'MyBuildImage',
    directory=os.path.join(dirname, 'my-image'))
```

Java

```
import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
            .directory(new File(startDir, "my-image").toString()).build();
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECR.Assets;
```

```
var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
{
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image")
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
})
```

The `my-image` directory must include a Dockerfile. The AWS CDK CLI builds a Docker image from `my-image`, pushes it to an Amazon ECR repository, and specifies the name of the repository as an AWS CloudFormation parameter to your stack. Docker image asset types expose [deploy-time attributes](#) that can be referenced in AWS CDK libraries and apps. The AWS CDK CLI command `cdk synth` displays asset properties as AWS CloudFormation parameters.

Amazon ECS task definition example

A common use case is to create an Amazon ECS [`TaskDefinition`](#) to run Docker containers. The following example specifies the location of a Docker image asset that the AWS CDK builds locally and pushes to Amazon ECR.

Example

TypeScript

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as ecr_assets from 'aws-cdk-lib/aws-ecr-assets';
import * as path from 'path';

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

const asset = new ecr_assets.DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromDockerImageAsset(asset)
});
```

JavaScript

```
const ecs = require('aws-cdk-lib/aws-ecs');
const ecr_assets = require('aws-cdk-lib/aws-ecr-assets');
const path = require('path');

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

const asset = new ecr_assets.DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image')
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromDockerImageAsset(asset)
});
```

Python

```
import aws_cdk.aws_ecs as ecs
```

```
import aws_cdk.aws_ecr_assets as ecr_assets

import os.path
dirname = os.path.dirname(__file__)

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024,
    cpu=512)

asset = ecr_assets.DockerImageAsset(self, 'MyBuildImage',
    directory=os.path.join(dirname, 'my-image'))

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_docker_image_asset(asset))
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

File startDir = new File(System.getProperty("user.dir"));

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "MyBuildImage")
            .directory(new File(startDir, "my-image").toString()).build();

taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder()
            .image(ContainerImage.fromDockerImageAsset(asset))
            .build();
)
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;
```

```
using Amazon.CDK.AWS.Ecr.Assets;

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps
{
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image")
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromDockerImageAsset(asset)
});
```

Go

```
import (
  "os"
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

taskDefinition := awsecs.NewTaskDefinition(stack, jsii.String("TaskDef"),
 awsecs.TaskDefinitionProps{
  MemoryMiB: jsii.String("1024"),
  Cpu: jsii.String("512"),
})

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 awsecrassets.DockerImageAssetProps{
```

```
  Directory: jsii.String(path.Join(dirName, "my-image")),
})

taskDefinition.AddContainer(jsii.String("MyOtherContainer"),
 awsecs.ContainerDefinitionOptions{
  Image: awsecs.ContainerImage_FromDockerImageAsset(asset),
})
```

Deploy-time attributes example

The following example shows how to use the deploy-time attributes `repository` and `imageUri` to create an Amazon ECS task definition with the AWS Fargate launch type. Note that the Amazon ECR repo lookup requires the image's tag, not its URI, so we snip it from the end of the asset's URI.

Example

TypeScript

```
import * as ecs from 'aws-cdk-lib/aws-ecs';
import * as path from 'path';
import { DockerImageAsset } from 'aws-cdk-lib/aws-ecr-assets';

const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, "..", "demo-image")
});

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository,
 asset.imageUri.split(":").pop())
});
```

JavaScript

```
const ecs = require('aws-cdk-lib/aws-ecs');
const path = require('path');
const { DockerImageAsset } = require('aws-cdk-lib/aws-ecr-assets');
```

```
const asset = new DockerImageAsset(this, 'my-image', {
  directory: path.join(__dirname, "..", "demo-image")
});

const taskDefinition = new ecs.FargateTaskDefinition(this, "TaskDef", {
  memoryLimitMiB: 1024,
  cpu: 512
});

taskDefinition.addContainer("my-other-container", {
  image: ecs.ContainerImage.fromEcrRepository(asset.repository,
 asset.imageUri.split(":").pop())
});
```

Python

```
import aws_cdk.aws_ecs as ecs
from aws_cdk.aws_ecr_assets import DockerImageAsset

import os.path
dirname = os.path.dirname(__file__)

asset = DockerImageAsset(self, 'my-image',
    directory=os.path.join(dirname, "..", "demo-image"))

task_definition = ecs.FargateTaskDefinition(self, "TaskDef",
    memory_limit_mib=1024, cpu=512)

task_definition.add_container("my-other-container",
    image=ecs.ContainerImage.from_ecr_repository(
        asset.repository, asset.image_uri.rpartition(":")[-1]))
```

Java

```
import java.io.File;

import software.amazon.awscdk.services.ecr.assets.DockerImageAsset;

import software.amazon.awscdk.services.ecs.FargateTaskDefinition;
import software.amazon.awscdk.services.ecs.ContainerDefinitionOptions;
import software.amazon.awscdk.services.ecs.ContainerImage;
```

```
File startDir = new File(System.getProperty("user.dir"));

DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image")
            .directory(new File(startDir, "demo-image").toString()).build();

FargateTaskDefinition taskDefinition = FargateTaskDefinition.Builder.create(
        this, "TaskDef").memoryLimitMiB(1024).cpu(512).build();

// extract the tag from the asset's image URI for use in ECR repo lookup
String imageUri = asset.getImageUri();
String imageTag = imageUri.substring(imageUri.lastIndexOf(":") + 1);

taskDefinition.addContainer("my-other-container",
        ContainerDefinitionOptions.builder().image(ContainerImage.fromEcrRepository(
                asset.getRepository(), imageTag)).build());
```

C#

```
using System.IO;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECR.Assets;

var asset = new DockerImageAsset(this, "my-image", new DockerImageAssetProps {
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "demo-image")
});

var taskDefinition = new FargateTaskDefinition(this, "TaskDef", new
 FargateTaskDefinitionProps
{
    MemoryLimitMiB = 1024,
    Cpu = 512
});

taskDefinition.AddContainer("my-other-container", new ContainerDefinitionOptions
{
    Image = ContainerImage.FromEcrRepository(asset.Repository,
 asset.ImageUri.Split(":").Last())
});
```

Go

```
import (
  "os"
```

```
  "path"

  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecrassets"
  "github.com/aws/aws-cdk-go/awscdk/v2/awsecs"
)

dirName, err := os.Getwd()
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyImage"),
 awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "demo-image")),
})

taskDefinition := awsecs.NewFargateTaskDefinition(stack, jsii.String("TaskDef"),
 awsecs.FargateTaskDefinitionProps{
  MemoryLimitMiB: jsii.Number(1024),
  Cpu: jsii.Number(512),
})

taskDefinition.AddContainer(jsii.String("MyOtherContainer"),
 awsecs.ContainerDefinitionOptions{
  Image: awsecs.ContainerImage_FromEcrRepository(asset.Repository(),
 asset.ImageTag()),
})
```

Build arguments example

You can provide customized build arguments for the Docker build step through the `buildArgs` (Python: `build_args`) property option when the AWS CDK CLI builds the image during deployment.

Example

TypeScript

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image'),
  buildArgs: {
```

```
    HTTP_PROXY: 'http://10.20.30.2:1234'
  }
});
```

JavaScript

```
const asset = new DockerImageAsset(this, 'MyBuildImage', {
  directory: path.join(__dirname, 'my-image'),
  buildArgs: {
    HTTP_PROXY: 'http://10.20.30.2:1234'
  }
});
```

Python

```
asset = DockerImageAsset(self, "MyBuildImage",
    directory=os.path.join(dirname, "my-image"),
    build_args=dict(HTTP_PROXY="http://10.20.30.2:1234"))
```

Java

```
DockerImageAsset asset = DockerImageAsset.Builder.create(this, "my-image"),
            .directory(new File(startDir, "my-image").toString())
            .buildArgs(java.util.Map.of(    // Java 9 or later
                "HTTP_PROXY", "http://10.20.30.2:1234"))
            .build();
```

C#

```
var asset = new DockerImageAsset(this, "MyBuildImage", new DockerImageAssetProps {
    Directory = Path.Combine(Directory.GetCurrentDirectory(), "my-image"),
    BuildArgs = new Dictionarystring, string
    {
        ["HTTP_PROXY"] = "http://10.20.30.2:1234"
    }
});
```

Go

```
dirName, err := os.Getwd()
```

```
if err != nil {
  panic(err)
}

asset := awsecrassets.NewDockerImageAsset(stack, jsii.String("MyBuildImage"),
 awsecrassets.DockerImageAssetProps{
  Directory: jsii.String(path.Join(dirName, "my-image")),
  BuildArgs: map[string]*string{
    "HTTP_PROXY": jsii.String("http://10.20.30.2:1234"),
  },
})
```

Permissions

If you use a module that supports Docker image assets, such as `aws-ecs`, the AWS CDK manages permissions for you when you use assets directly or through `ContainerImage.fromEcrRepository` (Python: `from_ecr_repository`). If you use Docker image assets directly, make sure that the consuming principal has permissions to pull the image.

In most cases, you should use `asset.repository.grantPull` method (Python: `grant_pull`). This modifies the IAM policy of the principal to enable it to pull images from this repository. If the principal that is pulling the image is not in the same account, or if it's an AWS service that doesn't assume a role in your account (such as AWS CodeBuild), you must grant pull permissions on the resource policy and not on the principal's policy. Use the `asset.repository.addToResourcePolicy` method (Python: `add_to_resource_policy`) to grant the appropriate principal permissions.

AWS CloudFormation resource metadata

> **Note**
>
> This section is relevant only for construct authors. In certain situations, tools need to know that a certain CFN resource is using a local asset. For example, you can use the AWS SAM CLI to invoke Lambda functions locally for debugging purposes. See AWS SAM integration for details.

To enable such use cases, external tools consult a set of metadata entries on AWS CloudFormation resources:

- `aws:asset:path` – Points to the local path of the asset.
- `aws:asset:property` – The name of the resource property where the asset is used.

Using these two metadata entries, tools can identify that assets are used by a certain resource, and enable advanced local experiences.

To add these metadata entries to a resource, use the `asset.addResourceMetadata` (Python: `add_resource_metadata`) method.

Permissions and the AWS CDK

The AWS Construct Library uses a few common, widely implemented idioms to manage access and permissions. The IAM module provides you with the tools you need to use these idioms.

AWS CDK uses AWS CloudFormation to deploy changes. Every deployment involves an actor (either a developer, or an automated system) that starts a AWS CloudFormation deployment. In the course of doing this, the actor will assume one or more IAM Identities (user or roles) and optionally pass a role to AWS CloudFormation.

If you use AWS IAM Identity Center to authenticate as a user, then the single sign-on provider supplies short-lived session credentials that authorize you to act as a pre-defined IAM role. To learn how the AWS CDK obtains AWS credentials from IAM Identity Center authentication, see Understand IAM Identity Center authentication in the *AWS SDKs and Tools Reference Guide*.

Principals

An IAM principal is an authenticated AWS entity representing a user, service, or application that can call AWS APIs. The AWS Construct Library supports specifying principals in several flexible ways to grant them access your AWS resources.

In security contexts, the term "principal" refers specifically to authenticated entities such as users. Objects like groups and roles do not *represent* users (and other authenticated entities) but rather *identify* them indirectly for the purpose of granting permissions.

For example, if you create an IAM group, you can grant the group (and thus its members) write access to an Amazon RDS table. However, the group itself is not a principal because it doesn't represent a single entity (also, you cannot log in to a group).

In the CDK's IAM library, classes that directly or indirectly identify principals implement the `IPrincipal` interface, allowing these objects to be used interchangeably in access policies. However, not all of them are principals in the security sense. These objects include:

1. IAM resources such as `Role`, `User`, and `Group`
2. Service principals (`new iam.ServicePrincipal('service.amazonaws.com')`)
3. Federated principals (`new iam.FederatedPrincipal('cognito-identity.amazonaws.com')`)
4. Account principals (`new iam.AccountPrincipal('0123456789012')`)
5. Canonical user principals (`new iam.CanonicalUserPrincipal('79a59d[…]7ef2be')`)
6. AWS Organizations principals (`new iam.OrganizationPrincipal('org-id')`)
7. Arbitrary ARN principals (`new iam.ArnPrincipal(res.arn)`)
8. An `iam.CompositePrincipal(principal1, principal2, …)` to trust multiple principals

Grants

Every construct that represents a resource that can be accessed, such as an Amazon S3 bucket or Amazon DynamoDB table, has methods that grant access to another entity. All such methods have names starting with **grant**.

For example, Amazon S3 buckets have the methods `grantRead` and `grantReadWrite` (Python: `grant_read`, `grant_read_write`) to enable read and read/write access, respectively, from an entity to the bucket. The entity doesn't have to know exactly which Amazon S3 IAM permissions are required to perform these operations.

The first argument of a **grant** method is always of type `IGrantable`. This interface represents entities that can be granted permissions. That is, it represents resources with roles, such as the IAM objects `Role`, `User`, and `Group`.

Other entities can also be granted permissions. For example, later in this topic, we show how to grant a CodeBuild project access to an Amazon S3 bucket. Generally, the associated role is obtained via a `role` property on the entity being granted access.

Resources that use execution roles, such as `lambda.Function`, also implement `IGrantable`, so you can grant them access directly instead of granting access to their role. For example, if `bucket` is an Amazon S3 bucket, and `function` is a Lambda function, the following code grants the function read access to the bucket.

Example

TypeScript

```
bucket.grantRead(function);
```

JavaScript

```
bucket.grantRead(function);
```

Python

```
bucket.grant_read(function)
```

Java

```
bucket.grantRead(function);
```

C#

```
bucket.GrantRead(function);
```

Sometimes permissions must be applied while your stack is being deployed. One such case is when you grant an AWS CloudFormation custom resource access to some other resource. The custom resource will be invoked during deployment, so it must have the specified permissions at deployment time.

Another case is when a service verifies that the role you pass to it has the right policies applied. (A number of AWS services do this to make sure that you didn't forget to set the policies.) In those cases, the deployment might fail if the permissions are applied too late.

To force the grant's permissions to be applied before another resource is created, you can add a dependency on the grant itself, as shown here. Though the return value of grant methods is commonly discarded, every grant method in fact returns an `iam.Grant` object.

Example

TypeScript

```
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

JavaScript

```
const grant = bucket.grantRead(lambda);
const custom = new CustomResource(...);
custom.node.addDependency(grant);
```

Python

```
grant = bucket.grant_read(function)
custom = CustomResource(...)
custom.node.add_dependency(grant)
```

Java

```
Grant grant = bucket.grantRead(function);
CustomResource custom = new CustomResource(...);
custom.node.addDependency(grant);
```

C#

```
var grant = bucket.GrantRead(function);
var custom = new CustomResource(...);
custom.node.AddDependency(grant);
```

Roles

The IAM package contains a `Role` construct that represents IAM roles. The following code creates a new role, trusting the Amazon EC2 service.

Example

TypeScript

```
import * as iam from 'aws-cdk-lib/aws-iam';

const role = new iam.Role(this, 'Role', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),   // required
});
```

JavaScript

```
const iam = require('aws-cdk-lib/aws-iam');

const role = new iam.Role(this, 'Role', {
  assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com')   // required
});
```

Python

```
import aws_cdk.aws_iam as iam

role = iam.Role(self, "Role",
          assumed_by=iam.ServicePrincipal("ec2.amazonaws.com")) # required
```

Java

```
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.iam.ServicePrincipal;

Role role = Role.Builder.create(this, "Role")
        .assumedBy(new ServicePrincipal("ec2.amazonaws.com")).build();
```

C#

```
using Amazon.CDK.AWS.IAM;

var role = new Role(this, "Role", new RoleProps
{
    AssumedBy = new ServicePrincipal("ec2.amazonaws.com"),   // required
});
```

You can add permissions to a role by calling the role's `addToPolicy` method (Python: `add_to_policy`), passing in a `PolicyStatement` that defines the rule to be added. The statement is added to the role's default policy; if it has none, one is created.

The following example adds a `Deny` policy statement to the role for the actions `ec2:SomeAction` and `s3:AnotherAction` on the resources `bucket` and `otherRole` (Python: `other_role`), under the condition that the authorized service is AWS CodeBuild.

Example

TypeScript

```
role.addToPolicy(new iam.PolicyStatement({
  effect: iam.Effect.DENY,
  resources: [bucket.bucketArn, otherRole.roleArn],
  actions: ['ec2:SomeAction', 's3:AnotherAction'],
  conditions: {StringEquals: {
    'ec2:AuthorizedService': 'codebuild.amazonaws.com',
}}}));
```

JavaScript

```
role.addToPolicy(new iam.PolicyStatement({
  effect: iam.Effect.DENY,
  resources: [bucket.bucketArn, otherRole.roleArn],
  actions: ['ec2:SomeAction', 's3:AnotherAction'],
  conditions: {StringEquals: {
    'ec2:AuthorizedService': 'codebuild.amazonaws.com'
}}}));
```

Python

```
role.add_to_policy(iam.PolicyStatement(
    effect=iam.Effect.DENY,
    resources=[bucket.bucket_arn, other_role.role_arn],
    actions=["ec2:SomeAction", "s3:AnotherAction"],
    conditions={"StringEquals": {
        "ec2:AuthorizedService": "codebuild.amazonaws.com"}}
))
```

Java

```
role.addToPolicy(PolicyStatement.Builder.create()
        .effect(Effect.DENY)
        .resources(Arrays.asList(bucket.getBucketArn(), otherRole.getRoleArn()))
        .actions(Arrays.asList("ec2:SomeAction", "s3:AnotherAction"))
        .conditions(java.util.Map.of(    // Map.of requires Java 9 or later
            "StringEquals", java.util.Map.of(
                "ec2:AuthorizedService", "codebuild.amazonaws.com")))
        .build());
```

C#

```
role.AddToPolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.DENY,
    Resources = new string[] { bucket.BucketArn, otherRole.RoleArn },
    Actions = new string[] { "ec2:SomeAction", "s3:AnotherAction" },
    Conditions = new Dictionary<string, object>
    {
        ["StringEquals"] = new Dictionary<string, string>
        {
            ["ec2:AuthorizedService"] = "codebuild.amazonaws.com"
        }
    }
}));
```

In the preceding example, we've created a new `PolicyStatement` inline with the `addToPolicy` (Python: `add_to_policy`) call. You can also pass in an existing policy statement or one you've modified. The `PolicyStatement` object has numerous methods for adding principals, resources, conditions, and actions.

If you're using a construct that requires a role to function correctly, you can do one of the following:

- Pass in an existing role when instantiating the construct object.
- Let the construct create a new role for you, trusting the appropriate service principal. The following example uses such a construct: a CodeBuild project.

Example

TypeScript

```
import * as codebuild from 'aws-cdk-lib/aws-codebuild';

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole: iam.IRole | undefined = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole,
});
```

JavaScript

```
const codebuild = require('aws-cdk-lib/aws-codebuild');

// imagine roleOrUndefined is a function that might return a Role object
// under some conditions, and undefined under other conditions
const someRole = roleOrUndefined();

const project = new codebuild.Project(this, 'Project', {
  // if someRole is undefined, the Project creates a new default role,
  // trusting the codebuild.amazonaws.com service principal
  role: someRole
});
```

Python

```
import aws_cdk.aws_codebuild as codebuild

# imagine role_or_none is a function that might return a Role object
# under some conditions, and None under other conditions
some_role = role_or_none();

project = codebuild.Project(self, "Project",
# if role is None, the Project creates a new default role,
# trusting the codebuild.amazonaws.com service principal
role=some_role)
```

Java

```
import software.amazon.awscdk.services.iam.Role;
import software.amazon.awscdk.services.codebuild.Project;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
Role someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
Project project = Project.Builder.create(this, "Project")
        .role(someRole).build();
```

C#

```
using Amazon.CDK.AWS.CodeBuild;

// imagine roleOrNull is a function that might return a Role object
// under some conditions, and null under other conditions
var someRole = roleOrNull();

// if someRole is null, the Project creates a new default role,
// trusting the codebuild.amazonaws.com service principal
var project = new Project(this, "Project", new ProjectProps
{
    Role = someRole
});
```

Once the object is created, the role (whether the role passed in or the default one created by the construct) is available as the property `role`. However, this property is not available on external resources. Therefore, these constructs have an `addToRolePolicy` (Python: `add_to_role_policy`) method.

The method does nothing if the construct is an external resource, and it calls the `addToPolicy` (Python: `add_to_policy`) method of the `role` property otherwise. This saves you the trouble of handling the undefined case explicitly.

The following example demonstrates:

Example

TypeScript

```
// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,   // ... and so on defining the policy
}));
```

JavaScript

```
// project is imported into the CDK application
const project = codebuild.Project.fromProjectName(this, 'Project', 'ProjectName');

// project is imported, so project.role is undefined, and this call has no effect
project.addToRolePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW   // ... and so on defining the policy
}));
```

Python

```
# project is imported into the CDK application
project = codebuild.Project.from_project_name(self, 'Project', 'ProjectName')

# project is imported, so project.role is undefined, and this call has no effect
project.add_to_role_policy(iam.PolicyStatement(
  effect=iam.Effect.ALLOW,   # ... and so on defining the policy
  )
)
```

Java

```
// project is imported into the CDK application
Project project = Project.fromProjectName(this, "Project", "ProjectName");

// project is imported, so project.getRole() is null, and this call has no effect
project.addToRolePolicy(PolicyStatement.Builder.create()
        .effect(Effect.ALLOW)   // .. and so on defining the policy
        .build();
```

```
)
```

C#

```
// project is imported into the CDK application
var project = Project.FromProjectName(this, "Project", "ProjectName");

// project is imported, so project.role is null, and this call has no effect
project.AddToRolePolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.ALLOW, // ... and so on defining the policy
}));
```

Resource policies

A few resources in AWS, such as Amazon S3 buckets and IAM roles, also have a resource policy. These constructs have an `addToResourcePolicy` method (Python: `add_to_resource_policy`), which takes a [`PolicyStatement`] as its argument. Every policy statement added to a resource policy must specify at least one principal.

In the following example, the [Amazon S3 bucket] `bucket` grants a role with the `s3:SomeAction` permission to itself.

Example

TypeScript

```
bucket.addToResourcePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['s3:SomeAction'],
  resources: [bucket.bucketArn],
  principals: [role]
}));
```

JavaScript

```
bucket.addToResourcePolicy(new iam.PolicyStatement({
  effect: iam.Effect.ALLOW,
  actions: ['s3:SomeAction'],
  resources: [bucket.bucketArn],
```

```
  principals: [role]
}));
```

Python

```
bucket.add_to_resource_policy(iam.PolicyStatement(
    effect=iam.Effect.ALLOW,
    actions=["s3:SomeAction"],
    resources=[bucket.bucket_arn],
    principals=role))
```

Java

```
bucket.addToResourcePolicy(PolicyStatement.Builder.create()
        .effect(Effect.ALLOW)
        .actions(Arrays.asList("s3:SomeAction"))
        .resources(Arrays.asList(bucket.getBucketArn()))
        .principals(Arrays.asList(role))
        .build());
```

C#

```
bucket.AddToResourcePolicy(new PolicyStatement(new PolicyStatementProps
{
    Effect = Effect.ALLOW,
    Actions = new string[] { "s3:SomeAction" },
    Resources = new string[] { bucket.BucketArn },
    Principals = new IPrincipal[] { role }
}));
```

Using external IAM objects

If you have defined an IAM user, principal, group, or role outside your AWS CDK app, you can use that IAM object in your AWS CDK app. To do so, create a reference to it using its ARN or its name. (Use the name for users, groups, and roles.) The returned reference can then be used to grant permissions or to construct policy statements as explained previously.

- For users, call `User.fromUserArn()` or `User.fromUserName()`. `User.fromUserAttributes()` is also available, but currently provides the same functionality as `User.fromUserArn()`.

- For principals, instantiate an `ArnPrincipal` object.
- For groups, call `Group.fromGroupArn()` or `Group.fromGroupName()`.
- For roles, call `Role.fromRoleArn()` or `Role.fromRoleName()`.

Policies (including managed policies) can be used in similar fashion using the following methods. You can use references to these objects anywhere an IAM policy is required.

- `Policy.fromPolicyName`
- `ManagedPolicy.fromManagedPolicyArn`
- `ManagedPolicy.fromManagedPolicyName`
- `ManagedPolicy.fromAwsManagedPolicyName`

Note

As with all references to external AWS resources, you cannot modify external IAM objects in your CDK app.

Context values and the AWS CDK

Context values are key-value pairs that can be associated with an app, stack, or construct. They may be supplied to your app from a file (usually either `cdk.json` or `cdk.context.json` in your project directory) or on the command line.

The CDK Toolkit uses context to cache values retrieved from your AWS account during synthesis. Values include the Availability Zones in your account or the Amazon Machine Image (AMI) IDs currently available for Amazon EC2 instances. Because these values are provided by your AWS account, they can change between runs of your CDK application. This makes them a potential source of unintended change. The CDK Toolkit's caching behavior "freezes" these values for your CDK app until you decide to accept the new values.

Imagine the following scenario without context caching. Let's say you specified "latest Amazon Linux" as the AMI for your Amazon EC2 instances, and a new version of this AMI was released. Then, the next time you deployed your CDK stack, your already-deployed instances would be using the outdated ("wrong") AMI and would need to be upgraded. Upgrading would result in replacing all your existing instances with new ones, which would probably be unexpected and undesired.

Instead, the CDK records your account's available AMIs in your project's `cdk.context.json` file, and uses the stored value for future synthesis operations. This way, the list of AMIs is no longer a potential source of change. You can also be sure that your stacks will always synthesize to the same AWS CloudFormation templates.

Not all context values are cached values from your AWS environment. AWS CDK feature flags are also context values. You can also create your own context values for use by your apps or constructs.

Context keys are strings. Values may be any type supported by JSON: numbers, strings, arrays, or objects.

> **Tip**
>
> If your constructs create their own context values, incorporate your library's package name in its keys so they won't conflict with other packages' context values.

Many context values are associated with a particular AWS environment, and a given CDK app can be deployed in more than one environment. The key for such values includes the AWS account and Region so that values from different environments do not conflict.

The following context key illustrates the format used by the AWS CDK, including the account and Region.

```
availability-zones:account=123456789012:region=eu-central-1
```

> **Important**
>
> Cached context values are managed by the AWS CDK and its constructs, including constructs you may write. Do not add or change cached context values by manually editing files. It can be useful, however, to review `cdk.context.json` occasionally to see what values are being cached. Context values that don't represent cached values should be stored under the `context` key of `cdk.json`. This way, they won't be cleared when cached values are cleared.

Sources of context values

Context values can be provided to your AWS CDK app in six different ways:

- Automatically from the current AWS account.
- Through the `--context` option to the `cdk` command. (These values are always strings.)
- In the project's `cdk.context.json` file.
- In the `context` key of the project's `cdk.json` file.
- In the `context` key of your `~/.cdk.json` file.
- In your AWS CDK app using the `construct.node.setContext()` method.

The project file `cdk.context.json` is where the AWS CDK caches context values retrieved from your AWS account. This practice avoids unexpected changes to your deployments when, for example, a new Availability Zone is introduced. The AWS CDK does not write context data to any of the other files listed.

> **Important**
>
> Because they're part of your application's state, `cdk.json` and `cdk.context.json` must be committed to source control along with the rest of your app's source code. Otherwise, deployments in other environments (for example, a CI pipeline) might produce inconsistent results.

Context values are scoped to the construct that created them; they are visible to child constructs, but not to parents or siblings. Context values that are set by the AWS CDK Toolkit (the `cdk` command) can be set automatically, from a file, or from the `--context` option. Context values from these sources are implicitly set on the `App` construct. Therefore, they're visible to every construct in every stack in the app.

Your app can read a context value using the `construct.node.tryGetContext` method. If the requested entry isn't found on the current construct or any of its parents, the result is `undefined`. (Alternatively, the result could be your language's equivalent, such as `None` in Python.)

Context methods

The AWS CDK supports several context methods that enable AWS CDK apps to obtain contextual information from the AWS environment. For example, you can get a list of Availability Zones that are available in a given AWS account and Region, using the `stack.availabilityZones` method.

The following are the context methods:

[HostedZone.fromLookup]

Gets the hosted zones in your account.

[stack.availabilityZones]

Gets the supported Availability Zones.

[StringParameter.valueFromLookup]

Gets a value from the current Region’s Amazon EC2 Systems Manager Parameter Store.

[Vpc.fromLookup]

Gets the existing Amazon Virtual Private Clouds in your accounts.

[LookupMachineImage]

Looks up a machine image for use with a NAT instance in an Amazon Virtual Private Cloud.

If a required context value isn’t available, the AWS CDK app notifies the CDK Toolkit that the context information is missing. Next, the CLI queries the current AWS account for the information and stores the resulting context information in the `cdk.context.json` file. Then, it executes the AWS CDK app again with the context values.

Viewing and managing context

Use the `cdk context` command to view and manage the information in your `cdk.context.json` file. To see this information, use the `cdk context` command without any options. The output should be something like the following.

```
Context found in cdk.json:

##################################################################################################
# # # Key                                                                # Value
                                       #
##################################################################################################
# 1 # availability-zones:account=123456789012:region=eu-central-1 # [ "eu-central-1a",
 "eu-central-1b", "eu-central-1c" ]   #
##################################################################################################
# 2 # availability-zones:account=123456789012:region=eu-west-1    # [ "eu-west-1a",
 "eu-west-1b", "eu-west-1c" ]            #
##################################################################################################
```

```
Run

  cdk context --reset KEY_OR_NUMBER

 to remove a context key. If it is a cached value, it will be refreshed on the next

  cdk synth

.
```

To remove a context value, run `cdk context --reset`, specifying the value's corresponding key or number. The following example removes the value that corresponds to the second key in the preceding example. This value represents the list of Availability Zones in the Europe (Ireland) Region.

```
cdk context --reset 2
```

```
Context value
availability-zones:account=123456789012:region=eu-west-1
reset. It will be refreshed on the next SDK synthesis run.
```

Therefore, if you want to update to the latest version of the Amazon Linux AMI, use the preceding example to do a controlled update of the context value and reset it. Then, synthesize and deploy your app again.

```
$ cdk synth
```

To clear all of the stored context values for your app, run `cdk context --clear`, as follows.

```
$ cdk context --clear
```

Only context values stored in `cdk.context.json` can be reset or cleared. The AWS CDK does not touch other context values. Therefore, to protect a context value from being reset using these commands, you might copy the value to `cdk.json`.

AWS CDK Toolkit `--context` flag

Use the `--context` (`-c` for short) option to pass runtime context values to your CDK app during synthesis or deployment.

```
$ cdk synth --context key=value MyStack
```

To specify multiple context values, repeat the `--context` option any number of times, providing one key-value pair each time.

```
$ cdk synth --context key1=value1 --context key2=value2 MyStack
```

When synthesizing multiple stacks, the specified context values are passed to all stacks. To provide different context values to individual stacks, either use different keys for the values, or use multiple `cdk synth` or `cdk deploy` commands.

Context values passed from the command line are always strings. If a value is usually of some other type, your code must be prepared to convert or parse the value. You might have non-string context values provided in other ways (for example, in `cdk.context.json`). To make sure this kind of value works as expected, confirm that the value is a string before converting it.

Example

Following is an example of using an existing Amazon VPC using AWS CDK context.

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

export class ExistsVpcStack extends cdk.Stack {

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {

    super(scope, id, props);

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid,
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});
```

```
    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString(),
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const ec2 = require('aws-cdk-lib/aws-ec2');

class ExistsVpcStack extends cdk.Stack {

  constructor(scope, id, props) {

    super(scope, id, props);

    const vpcid = this.node.tryGetContext('vpcid');
    const vpc = ec2.Vpc.fromLookup(this, 'VPC', {
      vpcId: vpcid
    });

    const pubsubnets = vpc.selectSubnets({subnetType: ec2.SubnetType.PUBLIC});

    new cdk.CfnOutput(this, 'publicsubnets', {
      value: pubsubnets.subnetIds.toString()
    });
  }
}

module.exports = { ExistsVpcStack }
```

Python

```
import aws_cdk as cdk
import aws_cdk.aws_ec2 as ec2
from constructs import Construct

class ExistsVpcStack(cdk.Stack):

    def __init__(scope: Construct, id: str, **kwargs):

        super().__init__(scope, id, **kwargs)
```

```
        vpcid = self.node.try_get_context("vpcid")
        vpc = ec2.Vpc.from_lookup(self, "VPC", vpc_id=vpcid)

        pubsubnets = vpc.select_subnets(subnetType=ec2.SubnetType.PUBLIC)

        cdk.CfnOutput(self, "publicsubnets",
            value=pubsubnets.subnet_ids.to_string())
```

Java

```
import software.amazon.awscdk.CfnOutput;

import software.amazon.awscdk.services.ec2.Vpc;
import software.amazon.awscdk.services.ec2.VpcLookupOptions;
import software.amazon.awscdk.services.ec2.SelectedSubnets;
import software.amazon.awscdk.services.ec2.SubnetSelection;
import software.amazon.awscdk.services.ec2.SubnetType;
import software.constructs.Construct;

public class ExistsVpcStack extends Stack {
    public ExistsVpcStack(Construct context, String id) {
        this(context, id, null);
    }

    public ExistsVpcStack(Construct context, String id, StackProps props) {
        super(context, id, props);

        String vpcId = (String)this.getNode().tryGetContext("vpcid");
        Vpc vpc = (Vpc)Vpc.fromLookup(this, "VPC", VpcLookupOptions.builder()
                .vpcId(vpcId).build());

        SelectedSubnets pubSubNets = vpc.selectSubnets(SubnetSelection.builder()
                .subnetType(SubnetType.PUBLIC).build());

        CfnOutput.Builder.create(this, "publicsubnets")
                .value(pubSubNets.getSubnetIds().toString()).build();
    }
}
```

C#

```
using Amazon.CDK;
```

```
using Amazon.CDK.AWS.EC2;
using Constructs;

class ExistsVpcStack : Stack
{
    public ExistsVpcStack(Construct scope, string id, StackProps props) :
 base(scope, id, props)
    {
        var vpcId = (string)this.Node.TryGetContext("vpcid");
        var vpc = Vpc.FromLookup(this, "VPC", new VpcLookupOptions
        {
            VpcId = vpcId
        });

        SelectedSubnets pubSubNets = vpc.SelectSubnets([new SubnetSelection
        {
            SubnetType = SubnetType.PUBLIC
        }]);

        new CfnOutput(this, "publicsubnets", new CfnOutputProps {
            Value = pubSubNets.SubnetIds.ToString()
        });
    }
}
```

You can use `cdk diff` to see the effects of passing in a context value on the command line:

```
$ cdk diff -c vpcid=vpc-0cb9c31031d0d3e22
```

```
Stack ExistsvpcStack
Outputs
[+] Output publicsubnets publicsubnets:
 {"Value":"subnet-06e0ea7dd302d3e8f,subnet-01fc0acfb58f3128f"}
```

The resulting context values can be viewed as shown here.

```
$ cdk context -j
```

```
{
  "vpc-provider:account=123456789012:filter.vpc-id=vpc-0cb9c31031d0d3e22:region=us-
east-1": {
```

```
    "vpcId": "vpc-0cb9c31031d0d3e22",
    "availabilityZones": [
      "us-east-1a",
      "us-east-1b"
    ],
    "privateSubnetIds": [
      "subnet-03ecfc033225be285",
      "subnet-0cded5da53180ebfa"
    ],
    "privateSubnetNames": [
      "Private"
    ],
    "privateSubnetRouteTableIds": [
      "rtb-0e955393ced0ada04",
      "rtb-05602e7b9f310e5b0"
    ],
    "publicSubnetIds": [
      "subnet-06e0ea7dd302d3e8f",
      "subnet-01fc0acfb58f3128f"
    ],
    "publicSubnetNames": [
      "Public"
    ],
    "publicSubnetRouteTableIds": [
      "rtb-00d1fdfd823c82289",
      "rtb-04bb1969b42969bcb"
    ]
  }
}
```

AWS CDK feature flags

The AWS CDK uses *feature flags* to enable potentially breaking behaviors in a release. Flags are stored as Context values and the AWS CDK values in `cdk.json` (or `~/.cdk.json`). They are not removed by the `cdk context --reset` or `cdk context --clear` commands.

Feature flags are disabled by default. Existing projects that do not specify the flag will continue to work as before with later AWS CDK releases. New projects created using `cdk init` include flags enabling all features available in the release that created the project. Edit `cdk.json` to disable any flags for which you prefer the earlier behavior. You can also add flags to enable new behaviors after upgrading the AWS CDK.

A list of all current feature flags can be found on the AWS CDK GitHub repository in FEATURE_FLAGS.md. See the `CHANGELOG` in a given release for a description of any new feature flags added in that release.

Reverting to v1 behavior

In CDK v2, the defaults for some feature flags have been changed with respect to v1. You can set these back to `false` to revert to specific AWS CDK v1 behavior. Use the `cdk diff` command to inspect the changes to your synthesized template to see if any of these flags are needed.

`@aws-cdk/core:newStyleStackSynthesis`

Use the new stack synthesis method, which assumes bootstrap resources with well-known names. Requires modern bootstrapping, but in turn allows CI/CD via CDK Pipelines and cross-account deployments out of the box.

`@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId`

If your application uses multiple Amazon API Gateway API keys and associates them to usage plans.

`@aws-cdk/aws-rds:lowercaseDbIdentifier`

If your application uses Amazon RDS database instance or database clusters, and explicitly specifies the identifier for these.

`@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021`

If your application uses the TLS_V1_2_2019 security policy with Amazon CloudFront distributions. CDK v2 uses security policy TLSv1.2_2021 by default.

`@aws-cdk/core:stackRelativeExports`

If your application uses multiple stacks and you refer to resources from one stack in another, this determines whether absolute or relative path is used to construct AWS CloudFormation exports.

`@aws-cdk/aws-lambda:recognizeVersionProps`

If set to `false`, the CDK includes metadata when detecting whether a Lambda function has changed. This can cause deployment failures when only the metadata has changed, since duplicate versions are not allowed. There is no need to revert this flag if you've made at least one change to all Lambda Functions in your application.

The syntax for reverting these flags in `cdk.json` is shown here.

```
{
  "context": {
    "@aws-cdk/core:newStyleStackSynthesis": false,
    "@aws-cdk/aws-apigateway:usagePlanKeyOrderInsensitiveId": false,
    "@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021": false,
    "@aws-cdk/aws-rds:lowercaseDbIdentifier": false,
    "@aws-cdk/core:stackRelativeExports": false,
    "@aws-cdk/aws-lambda:recognizeVersionProps": false
  }
}
```

Aspects and the AWS CDK

Aspects are a way to apply an operation to all constructs in a given scope. The aspect could modify the constructs, such as by adding tags. Or it could verify something about the state of the constructs, such as making sure that all buckets are encrypted.

To apply an aspect to a construct and all constructs in the same scope, call `Aspects.of(<SCOPE>).add()` with a new aspect, as shown in the following example.

Example

TypeScript

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

JavaScript

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

Python

```
Aspects.of(my_construct).add(SomeAspect(...))
```

Java

```
Aspects.of(myConstruct).add(new SomeAspect(...));
```

C#

```
Aspects.Of(myConstruct).add(new SomeAspect(...));
```

Go

```
awscdk.Aspects_Of(stack).Add(awscdk.NewTag(...))
```

The AWS CDK uses aspects to tag resources, but the framework can also be used for other purposes. For example, you can use it to validate or change the AWS CloudFormation resources that are defined for you by higher-level constructs.

Aspects in detail

Aspects employ the visitor pattern. An aspect is a class that implements the following interface.

Example

TypeScript

```
interface IAspect {
   visit(node: IConstruct): void;}
```

JavaScript

JavaScript doesn't have interfaces as a language feature. Therefore, an aspect is simply an instance of a class having a `visit` method that accepts the node to be operated on.

Python

Python doesn't have interfaces as a language feature. Therefore, an aspect is simply an instance of a class having a `visit` method that accepts the node to be operated on.

Java

```
public interface IAspect {
    public void visit(Construct node);
}
```

C#

```
public interface IAspect
```

```
{
    void Visit(IConstruct node);
}
```

Go

```
type IAspect interface {
  Visit(node constructs.IConstruct)
}
```

When you call `Aspects.of(<SCOPE>).add(…)` , the construct adds the aspect to an internal list of aspects. You can obtain the list with `Aspects.of(<SCOPE>)`.

During the [prepare phase](#), the AWS CDK calls the `visit` method of the object for the construct and each of its children in top-down order.

The `visit` method is free to change anything in the construct. In strongly typed languages, cast the received construct to a more specific type before accessing construct-specific properties or methods.

Aspects don't propagate across `Stage` construct boundaries, because `Stages` are self-contained and immutable after definition. Apply aspects on the `Stage` construct itself (or lower) if you want them to visit constructs inside the `Stage`.

Example

The following example validates that all buckets created in the stack have versioning enabled. The aspect adds an error annotation to the constructs that fail the validation. This results in the `synth` operation failing and prevents deploying the resulting cloud assembly.

Example

TypeScript

```
class BucketVersioningChecker implements IAspect {
  public visit(node: IConstruct): void {
    // See that we're dealing with a CfnBucket
    if (node instanceof s3.CfnBucket) {

      // Check for versioning property, exclude the case where the property
```

```
      // can be a token (IResolvable).
      if (!node.versioningConfiguration
        || (!Tokenization.isResolvable(node.versioningConfiguration)
            && node.versioningConfiguration.status !== 'Enabled')) {
        Annotations.of(node).addError('Bucket versioning is not enabled');
      }
    }
  }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

JavaScript

```
class BucketVersioningChecker {
   visit(node) {
    // See that we're dealing with a CfnBucket
    if ( node instanceof s3.CfnBucket) {

      // Check for versioning property, exclude the case where the property
      // can be a token (IResolvable).
      if (!node.versioningConfiguration
        || !Tokenization.isResolvable(node.versioningConfiguration)
            && node.versioningConfiguration.status !== 'Enabled')) {
        Annotations.of(node).addError('Bucket versioning is not enabled');
      }
    }
  }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

Python

```
@jsii.implements(cdk.IAspect)
class BucketVersioningChecker:

  def visit(self, node):
    # See that we're dealing with a CfnBucket
    if isinstance(node, s3.CfnBucket):
```

```
        # Check for versioning property, exclude the case where the property
        # can be a token (IResolvable).
        if (not node.versioning_configuration or
                not Tokenization.is_resolvable(node.versioning_configuration)
                    and node.versioning_configuration.status != "Enabled"):
            Annotations.of(node).add_error('Bucket versioning is not enabled')

# Later, apply to the stack
Aspects.of(stack).add(BucketVersioningChecker())
```

Java

```
public class BucketVersioningChecker implements IAspect
{
    @Override
    public void visit(Construct node)
    {
        // See that we're dealing with a CfnBucket
        if (node instanceof CfnBucket)
        {
            CfnBucket bucket = (CfnBucket)node;
            Object versioningConfiguration = bucket.getVersioningConfiguration();
            if (versioningConfiguration == null ||
                    !Tokenization.isResolvable(versioningConfiguration.toString())
 &&
                    !versioningConfiguration.toString().contains("Enabled"))
                Annotations.of(bucket.getNode()).addError("Bucket versioning is not
 enabled");
        }
    }
}

// Later, apply to the stack
Aspects.of(stack).add(new BucketVersioningChecker());
```

C#

```
class BucketVersioningChecker : Amazon.Jsii.Runtime.Deputy.DeputyBase, IAspect
{
    public void Visit(IConstruct node)
    {
        // See that we're dealing with a CfnBucket
        if (node is CfnBucket)
```

```
        {
            var bucket = (CfnBucket)node;
            if (bucket.VersioningConfiguration is null ||
                    !Tokenization.IsResolvable(bucket.VersioningConfiguration) &&
                    !bucket.VersioningConfiguration.ToString().Contains("Enabled"))
                Annotations.Of(bucket.Node).AddError("Bucket versioning is not
 enabled");
        }
    }
}

// Later, apply to the stack
Aspects.Of(stack).add(new BucketVersioningChecker());
```

AWS CDK prerequisites

Complete all prerequisites before getting started with the AWS Cloud Development Kit (AWS CDK).

Set up your AWS account

If you or your organization are new to AWS, you must set up your AWS account. This includes signing up for an AWS account, securing your root user, determining your method of managing users, and creating an administrative user. To manage users, you can use AWS Identity and Access Management (IAM) or AWS IAM Identity Center. We recommend that you use IAM Identity Center. For more information, see the following:

- What is IAM? in the *IAM User Guide*.
- What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

After setting up an AWS account, you should have an administrative user and the ability to create and manage additional users using IAM or IAM Identity Center.

Before moving forward, we recommend that you take time to learn the recommended best practices in AWS Identity and Access Management. For more information, see Security best practices and use cases in AWS Identity and Access Management in the *IAM User Guide*.

Install and configure the AWS CLI

When you develop AWS CDK applications on your local machine, you will use the AWS Cloud Development Kit (AWS CDK) Command Line Interface (CLI) to interact with AWS, such as deploying applications to provision your AWS resources. To interact with AWS outside of the AWS Management Console, you must configure security credentials on your local machine. To do this, we recommend that you install and use the AWS Command Line Interface (AWS CLI).

For instructions on installing the AWS CLI, see Install or update to the latest version of the AWS CLI in the *AWS Command Line Interface User Guide*.

How you configure security credentials will depend on how you or your organization manages users. For instructions, see Authentication and access credentials in the *AWS Command Line Interface User Guide*.

After installing and configuring the AWS CLI, you should have the following:

- The AWS CLI installed on your local machine.
- Credentials configured in a `config` on your local machine using the AWS CLI.

Install Node.js and programming language prerequisites

All AWS CDK developers, regardless of the supported programming language that you will use, require Node.js 22.x or later. All supported programming languages use the same backend, which runs on Node.js. We recommend a version in active long-term support.

For more information on supported Node.js versions, see Supported Node versions.

Other programming language prerequisites depend on the language that you will use to develop AWS CDK applications:

Example

TypeScript

- TypeScript 3.8 or later (`npm -g install typescript`)

JavaScript

- No additional requirements

Python

- Python 3.7 or later including `pip` and `virtualenv`

Java

- Java Development Kit (JDK) 8 (a.k.a. 1.8) or later
- Apache Maven 3.5 or later

  Java IDE recommended (we use Eclipse` in some examples in this guide). IDE must be able to import Maven projects. Check to make sure that your project is set to use Java 1.8. Set the JAVA_HOME environment variable to the path where you have installed the JDK.

C#

.NET Core 3.1 or later, or .NET 6.0 or later.

Visual Studio 2019 (any edition) or Visual Studio Code recommended.

Go

Go 1.1.8 or later.

Third-party language deprecation

Each language version is only supported until it is EOL (End Of Life) and is subject to change with prior notice.

Next steps

To get started with the AWS CDK, see Getting started with the AWS CDK.

Getting started with the AWS CDK

Get started with the AWS Cloud Development Kit (AWS CDK) by installing and configuring the AWS CDK Command Line Interface (AWS CDK CLI). Then, use the CDK CLI to create your first CDK app, bootstrap your AWS environment, and deploy your application.

Prerequisites

Before getting started with the AWS CDK, complete all prerequisites. These prerequisites are required for those that are new to AWS or new to programming. For instructions, see AWS CDK prerequisites.

We recommend that you have a basic understanding of what the AWS CDK is. For more information, see What is the AWS CDK? and Learn AWS CDK core concepts.

Install the AWS CDK CLI

Use the Node Package Manager to install the CDK CLI. We recommend that you install it globally using the following command:

```
$ npm install -g aws-cdk
```

To install a specific version of the CDK CLI, use the following command structure:

```
$ npm install -g aws-cdk@X.YY.Z
```

If you want to use multiple versions of the AWS CDK, consider installing a matching version of the CDK CLI in individual CDK projects. To do this, remove the `-g` option from the `npm install` command. Then, use `npx aws-cdk` to invoke the CDK CLI. This will run a local version if it exists. Otherwise, the globally installed version will be used.

Troubleshoot a CDK CLI installation

If you get a permission error, and have administrator access on your system, run the following:

```
$ sudo npm install -g aws-cdk
```

If you receive an error message, try uninstalling the CDK CLI by running the following:

```
$ npm uninstall -g aws-cdk
```

Then, repeat steps to reinstall the CDK CLI.

Verify a successful CDK CLI installation

Run the following command to verify a successful installation. The AWS CDK CLI should output the version number:

```
$ cdk --version
```

Configure the AWS CDK CLI

After installing the CDK CLI, you can start using it to develop applications on your local machine. To interact with AWS, such as deploying applications, you must have security credentials configured on your local machine with permissions to perform any actions that you initiate.

To configure security credentials on your local machine, you use the AWS CLI. How you configure security credentials depends on how you manage users. For instructions, see [Authentication and access credentials]() in the *AWS Command Line Interface User Guide*.

The CDK CLI will automatically use the security credentials that you configure with the AWS CLI. For example, if you are an IAM Identity Center user, you can use the `aws configure sso` command to configure security credentials. If you are an IAM user, you can use the `aws configure` command. The AWS CLI will guide you through configuring security credentials on your local machine and save the necessary information in your `config` and `credentials` files. Then, when you use the CDK CLI, such as deploying an application with `cdk deploy`, the CDK CLI will use your configured security credentials.

Just like the AWS CLI, the CDK CLI will use your `default` profile by default. You can specify a profile using the CDK CLI `--profile` option. For more information on using security credentials with the CDK CLI, see Configure security credentials for the AWS CDK CLI.

(Optional) Install additional AWS CDK tools

The AWS Toolkit for Visual Studio Code is an open source plug-in for Visual Studio Code that helps you create, debug, and deploy applications on AWS. The toolkit provides an integrated experience for developing AWS CDK applications. It includes the AWS CDK Explorer feature to list your AWS CDK projects and browse the various components of the CDK application. For instructions, see the following:

- Installing the AWS Toolkit for Visual Studio Code.
- AWS CDK for VS Code.

Create your first CDK app

You're now ready to get started with using the AWS CDK by creating your first CDK app. For instructions, see Tutorial: Create your first AWS CDK app.

Tutorial: Create your first AWS CDK app

Get started with using the AWS Cloud Development Kit (AWS CDK) by using the AWS CDK Command Line Interface (AWS CDK CLI) to develop your first CDK app, bootstrap your AWS environment, and deploy your application on AWS.

Prerequisites

Before starting this tutorial, complete all set up steps in Getting started with the AWS CDK.

About this tutorial

In this tutorial, you will create and deploy a simple application on AWS using the AWS CDK. The application consists of an AWS Lambda function that returns a `Hello World!` message when invoked. The function will be invoked through a Lambda function URL that serves as a dedicated HTTP(S) endpoint for your Lambda function.

Through this tutorial, you will perform the following:

- **Create your project** – Create a CDK project using the CDK CLI `cdk init` command.
- **Configure your AWS environment** – Configure the AWS environment that you will deploy your application into.
- **Bootstrap your AWS environment** – Prepare your AWS environment for deployment by bootstrapping it using the CDK CLI `cdk bootstrap` command.
- **Develop your app** – Use constructs from the AWS Construct Library to define your Lambda function and Lambda function URL resources.

- **Prepare your app for deployment –** Use the CDK CLI to build your app and synthesize an AWS CloudFormation template.
- **Deploy your app –** Use the CDK CLI `cdk deploy` command to deploy your application and provision your AWS resources.
- **Interact with your application –** Interact with your deployed Lambda function on AWS by invoking it and receiving a response.
- **Modify your app –** Modify your Lambda function and deploy to implement your changes.
- **Delete your app –** Delete all resources that you created by using the CDK CLI `cdk destroy` command.

Step 1: Create your CDK project

In this step, you create a new CDK project. A CDK project should be in its own directory, with its own local module dependencies.

To create a CDK project

1. From a starting directory of your choice, create and navigate to a directory named `hello-cdk`:

   ```
   $ mkdir hello-cdk && cd hello-cdk
   ```

   > **Important**
   >
   > Be sure to name your project directory `hello-cdk`, *exactly as shown here*. The CDK CLI uses this directory name to name things within your CDK code. If you use a different directory name, you will run into issues during this tutorial.

2. From the `hello-cdk` directory, initialize a new CDK project using the CDK CLI`cdk init` command. Specify the `app` template and your preferred programming language with the `--language` option:

   Example

   TypeScript

   ```
   $ cdk init app --language typescript
   ```

JavaScript

```
$ cdk init app --language javascript
```

Python

```
$ cdk init app --language python
```

After the app has been created, also enter the following two commands. These activate the app's Python virtual environment and installs the AWS CDK core dependencies.

```
$ source .venv/bin/activate # On Windows, run `.\venv\Scripts\activate` instead
$ python -m pip install -r requirements.txt
```

Java

```
$ cdk init app --language java
```

If you are using an IDE, you can now open or import the project. In Eclipse, for example, choose **File** > **Import** > **Maven** > **Existing Maven Projects**. Make sure that the project settings are set to use Java 8 (1.8).

C#

```
$ cdk init app --language csharp
```

If you are using Visual Studio, open the solution file in the `src` directory.

Go

```
$ cdk init app --language go
```

After the app has been created, also enter the following command to install the AWS Construct Library modules that the app requires.

```
$ go get
```

The `cdk init` command creates a structure of files and folders within the `hello-cdk` directory to help organize the source code for your CDK app. This structure of files and folders is called your CDK *project*. Take a moment to explore your CDK project.

If you have Git installed, each project you create using `cdk init` is also initialized as a Git repository.

During project initialization, the CDK CLI creates a CDK app containing a single CDK stack. The CDK app instance is created using the App construct. The following is a portion of this code from your CDK application file:

Example

TypeScript

Located in `bin/hello-cdk.ts`:

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { HelloCdkStack } from '../lib/hello-cdk-stack';

const app = new cdk.App();
new HelloCdkStack(app, 'HelloCdkStack', {
});
```

JavaScript

Located in `bin/hello-cdk.js`:

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { HelloCdkStack } = require('../lib/hello-cdk-stack');

const app = new cdk.App();
new HelloCdkStack(app, 'HelloCdkStack', {
});
```

Python

Located in `app.py`:

```
#!/usr/bin/env python3
import os

import aws_cdk as cdk

from hello_cdk.hello_cdk_stack import HelloCdkStack

app = cdk.App()
HelloCdkStack(app, "HelloCdkStack",)

app.synth()
```

Java

Located in `src/main/java/.../HelloCdkApp.java` :

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class HelloCdkApp {
  public static void main(final String[] args) {
    App app = new App();

    new HelloCdkStack(app, "HelloCdkStack", StackProps.builder()
      .build());

    app.synth();
  }
}
```

C#

Located in `src/HelloCdk/Program.cs`:

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace HelloCdk
{
  sealed class Program
  {
    public static void Main(string[] args)
    {
      var app = new App();
      new HelloCdkStack(app, "HelloCdkStack", new StackProps
      {});
      app.Synth();
    }
  }
}
```

Go

Located in `hello-cdk.go`:

```
package main

import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
)

// ...

func main() {
  defer jsii.Close()

  app := awscdk.NewApp(nil)

  NewHelloCdkStack(app, "HelloCdkStack", &HelloCdkStackProps{
    awscdk.StackProps{
      Env: env(),
    },
  })

  app.Synth(nil)
}
```

```
// ...
```

The CDK stack is created using the `Stack` construct. The following is a portion of this code from your CDK stack file:

Example

TypeScript

Located in `lib/hello-cdk-stack.ts`:

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

export class HelloCdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define your constructs here

  }
}
```

JavaScript

Located in `lib/hello-cdk-stack.js`:

```
const { Stack } = require('aws-cdk-lib');

class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define your constructs here

  }
}

module.exports = { HelloCdkStack }
```

Python

Located in `hello_cdk/hello_cdk_stack.py`:

```
from aws_cdk import (
  Stack,
)
from constructs import Construct

class HelloCdkStack(Stack):

  def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
    super().__init__(scope, construct_id, **kwargs)

    # Define your constructs here
```

Java

Located in `src/main/java/.../HelloCdkStack.java` :

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class HelloCdkStack extends Stack {
  public HelloCdkStack(final Construct scope, final String id) {
    this(scope, id, null);
  }

  public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
    super(scope, id, props);

    // Define your constructs here
  }
}
```

C#

Located in `src/HelloCdk/HelloCdkStack.cs`:

```
using Amazon.CDK;
using Constructs;

namespace HelloCdk
{
  public class HelloCdkStack : Stack
  {
    internal HelloCdkStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
    {
      // Define your constructs here
    }
  }
}
```

Go

Located in `hello-cdk.go`:

```
package main

import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
)

type HelloCdkStackProps struct {
  awscdk.StackProps
}

func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  return stack
}

// ...
```

Step 2: Configure your AWS environment

In this step, you configure the AWS environment for your CDK stack. By doing this, you specify which environment your CDK stack will be deployed to.

First, determine the AWS environment that you want to use. An AWS environment consists of an AWS account and AWS Region.

When you use the AWS CLI to configure security credentials on your local machine, you can then use the AWS CLI to obtain AWS environment information for a specific profile.

To use the AWS CLI to obtain your AWS account ID

1. Run the following AWS CLI command to get the AWS account ID for your `default` profile:

```
$ aws sts get-caller-identity --query "Account" --output text
```

2. If you prefer to use a named profile, provide the name of your profile using the `--profile` option:

```
$ aws sts get-caller-identity --profile your-profile-name --query "Account" --
output text
```

To use the AWS CLI to obtain your AWS Region

1. Run the following AWS CLI command to get the Region that you configured for your `default` profile:

```
$ aws configure get region
```

2. If you prefer to use a named profile, provide the name of your profile using the `--profile` option:

```
$ aws configure get region --profile your-profile-name
```

Next, you will configure the AWS environment for your CDK stack by modifying the `HelloCdkStack` instance in your *application file*. For this tutorial, you will hard code your AWS environment information. This is recommended for production environments. For information on other ways to configure environments, see Configure environments to use with the AWS CDK.

To configure the environment for your CDK stack

1. In your *application file*, use the `env` property of the `Stack` construct to configure your environment. The following is an example:

   Example

   TypeScript

   Located in `bin/hello-cdk.ts`:

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { HelloCdkStack } from '../lib/hello-cdk-stack';

const app = new cdk.App();
new HelloCdkStack(app, 'HelloCdkStack', {
  env: { account: '123456789012', region: 'us-east-1' },
});
```

   JavaScript

   Located in `bin/hello-cdk.js`:

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { HelloCdkStack } = require('../lib/hello-cdk-stack');

const app = new cdk.App();
new HelloCdkStack(app, 'HelloCdkStack', {
  env: { account: '123456789012', region: 'us-east-1' },
});
```

   Python

   Located in `app.py`:

```
#!/usr/bin/env python3
import os

import aws_cdk as cdk
```

```
from hello_cdk.hello_cdk_stack import HelloCdkStack

app = cdk.App()
HelloCdkStack(app, "HelloCdkStack",
  env=cdk.Environment(account='123456789012', region='us-east-1'),
  )

app.synth()
```

Java

Located in `src/main/java/.../HelloCdkApp.java` :

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class HelloCdkApp {
    public static void main(final String[] args) {
        App app = new App();

        new HelloCdkStack(app, "HelloCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .account("123456789012")
                        .region("us-east-1")
                        .build())

                .build());

        app.synth();
    }
}
```

C#

Located in `src/HelloCdk/Program.cs`:

```
using Amazon.CDK;
```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloCdk
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new HelloCdkStack(app, "HelloCdkStack", new StackProps
            {
                Env = new Amazon.CDK.Environment
                {
                    Account = "123456789012",
                    Region = "us-east-1",
                }
            });
            app.Synth();
        }
    }
}
```

Go

Located in `hello-cdk.go`:

```
package main

import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
)

// ...

func main() {
  defer jsii.Close()

  app := awscdk.NewApp(nil)
```

```
  NewHelloCdkStack(app, "HelloCdkStack", &HelloCdkStackProps{
    awscdk.StackProps{
      Env: env(),
    },
  })

  app.Synth(nil)
}

func env() *awscdk.Environment {
 return &awscdk.Environment{
  Account: jsii.String("123456789012"),
  Region:  jsii.String("us-east-1"),
 }
}
```

Step 3: Bootstrap your AWS environment

In this step, you bootstrap the AWS environment that you configured in the previous step. This prepares your environment for CDK deployments.

To bootstrap your environment, run the following from the root of your CDK project:

```
$ cdk bootstrap
```

By bootstrapping from the root of your CDK project, you don't have to provide any additional information. The CDK CLI obtains environment information from your project. When you bootstrap outside of a CDK project, you must provide environment information with the `cdk bootstrap` command. For more information, see Bootstrap your environment for use with the AWS CDK.

Step 4: Build your CDK app

In most programming environments, you build or compile code after making changes. This isn't necessary with the AWS CDK since the CDK CLI will automatically perform this step. However, you can still build manually when you want to catch syntax and type errors. The following is an example:

Example

TypeScript

```
$ npm run build

> hello-cdk@0.1.0 build
> tsc
```

JavaScript

No build step is necessary.

Python

No build step is necessary.

Java

```
$ mvn compile -q
```

Or press `Control-B` in Eclipse (other Java IDEs may vary)

C#

```
$ dotnet build src
```

Or press F6 in Visual Studio

Go

```
$ go build
```

Step 5: List the CDK stacks in your app

At this point, you should have a CDK app containing a single CDK stack. To verify, use the CDK CLI `cdk list` command to display your stacks. The output should display a single stack named `HelloCdkStack`:

```
$ cdk list
```

```
HelloCdkStack
```

If you don't see this output, verify that you are in the correct working directory of your project and try again. If you still don't see your stack, repeat Step 1: Create your CDK project and try again.

Step 6: Define your Lambda function

In this step, you import the `aws_lambda` module from the AWS Construct Library and use the `Function` L2 construct.

Modify your CDK stack file as follows:

Example

TypeScript

Located in `lib/hello-cdk-stack.ts`:

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// Import the Lambda module
import * as lambda from 'aws-cdk-lib/aws-lambda';

export class HelloCdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define the Lambda function resource
    const myFunction = new lambda.Function(this, "HelloWorldFunction", {
      runtime: lambda.Runtime.NODEJS_20_X, // Provide any supported Node.js runtime
      handler: "index.handler",
      code: lambda.Code.fromInline(`
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello World!'),
          };
        };
      `),
    });
  }
}
```

JavaScript

Located in `lib/hello-cdk-stack.js`:

```
const { Stack } = require('aws-cdk-lib');
// Import the Lambda module
const lambda = require('aws-cdk-lib/aws-lambda');

class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    const myFunction = new lambda.Function(this, "HelloWorldFunction", {
      runtime: lambda.Runtime.NODEJS_20_X, // Provide any supported Node.js runtime
      handler: "index.handler",
      code: lambda.Code.fromInline(`
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello World!'),
          };
        };
      `),
    });

  }
}

module.exports = { HelloCdkStack }
```

Python

Located in `hello_cdk/hello_cdk_stack.py`:

```
from aws_cdk import (
  Stack,
  aws_lambda as _lambda, # Import the Lambda module
)
from constructs import Construct

class HelloCdkStack(Stack):

```

```
  def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
    super().__init__(scope, construct_id, **kwargs)

    # Define the Lambda function resource
    my_function = _lambda.Function(
      self, "HelloWorldFunction",
      runtime = _lambda.Runtime.NODEJS_20_X, # Provide any supported Node.js runtime
      handler = "index.handler",
      code = _lambda.Code.from_inline(
        """
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello World!'),
          };
        };
        """
      ),
    )
```

Java

Located in `src/main/java/…/HelloCdkStack.java` :

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
// Import Lambda function
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class HelloCdkStack extends Stack {
  public HelloCdkStack(final Construct scope, final String id) {
    this(scope, id, null);
  }

  public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
    super(scope, id, props);

    // Define the Lambda function resource
```

```
    Function myFunction = Function.Builder.create(this, "HelloWorldFunction")
      .runtime(Runtime.NODEJS_20_X) // Provide any supported Node.js runtime
      .handler("index.handler")
      .code(Code.fromInline(
        "exports.handler = async function(event) {" +
        " return {" +
        " statusCode: 200," +
        " body: JSON.stringify('Hello World!')" +
        " };" +
        "};"))
      .build();

  }
}
```

C#

Located in `src/main/java/.../HelloCdkStack.java` :

```
using Amazon.CDK;
using Constructs;
// Import the Lambda module
using Amazon.CDK.AWS.Lambda;

namespace HelloCdk
{
  public class HelloCdkStack : Stack
  {
    internal HelloCdkStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
    {
      // Define the Lambda function resource
      var myFunction = new Function(this, "HelloWorldFunction", new FunctionProps
      {
        Runtime = Runtime.NODEJS_20_X, // Provide any supported Node.js runtime
        Handler = "index.handler",
        Code = Code.FromInline(@"
          exports.handler = async function(event) {
            return {
              statusCode: 200,
              body: JSON.stringify('Hello World!'),
            };
          };
        "),
```

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

Go

Located in `hello-cdk.go`:

```
package main

import (
  "github.com/aws/aws-cdk-go/awscdk/v2"
  "github.com/aws/constructs-go/constructs/v10"
  "github.com/aws/jsii-runtime-go"
  // Import the Lambda module
  "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
)

type HelloCdkStackProps struct {
  awscdk.StackProps
}

func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  // Define the Lambda function resource
  myFunction := awslambda.NewFunction(stack, jsii.String("HelloWorldFunction"),
 &awslambda.FunctionProps{
    Runtime: awslambda.Runtime_NODEJS_20_X(), // Provide any supported Node.js
 runtime
    Handler: jsii.String("index.handler"),
    Code: awslambda.Code_FromInline(jsii.String(`
      exports.handler = async function(event) {
        return {
          statusCode: 200,
          body: JSON.stringify('Hello World!'),
        };
      };
```

```
    `)),
  })

  return stack
}

// ...
```

Let's take a closer look at the `Function` construct. Like all constructs, the `Function` class takes three parameters:

- **scope –** Defines your `Stack` instance as the parent of the `Function` construct. All constructs that define AWS resources are created within the scope of a stack. You can define constructs inside of constructs, creating a hierarchy (tree). Here, and in most cases, the scope is `this` (`self` in Python).
- **Id –** The construct ID of the `Function` within your AWS CDK app. This ID, plus a hash based on the function's location within the stack, uniquely identifies the function during deployment. The AWS CDK also references this ID when you update the construct in your app and re-deploy to update the deployed resource. Here, your construct ID is `HelloWorldFunction`. Functions can also have a name, specified with the `functionName` property. This is different from the construct ID.
- **props –** A bundle of values that define properties of the function. Here you define the `runtime`, `handler`, and `code` properties.

  Props are represented differently in the languages supported by the AWS CDK.

  - In TypeScript and JavaScript, `props` is a single argument and you pass in an object containing the desired properties.
  - In Python, props are passed as keyword arguments.
  - In Java, a Builder is provided to pass the props. There are two: one for `FunctionProps`, and a second for `Function` to let you build the construct and its props object in one step. This code uses the latter.
  - In C#, you instantiate a `FunctionProps` object using an object initializer and pass it as the third parameter.

    If a construct's props are optional, you can omit the `props` parameter entirely.

All constructs take these same three arguments, so it's easy to stay oriented as you learn about new ones. And as you might expect, you can subclass any construct to extend it to suit your needs, or if you want to change its defaults.

Step 7: Define your Lambda function URL

In this step, you use the `addFunctionUrl` helper method of the `Function` construct to define a Lambda function URL. To output the value of this URL at deployment, you will create an AWS CloudFormation output using the `CfnOutput` construct.

Add the following to your CDK stack file:

Example

TypeScript

Located in `lib/hello-cdk-stack.ts`:

```
// ...

export class HelloCdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define the Lambda function resource
    // ...

    // Define the Lambda function URL resource
    const myFunctionUrl = myFunction.addFunctionUrl({
      authType: lambda.FunctionUrlAuthType.NONE,
    });

    // Define a CloudFormation output for your URL
    new cdk.CfnOutput(this, "myFunctionUrlOutput", {
      value: myFunctionUrl.url,
    })

  }
}
```

JavaScript

Located in `lib/hello-cdk-stack.js`:

```
const { Stack, CfnOutput } = require('aws-cdk-lib');  // Import CfnOutput

class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    // ...

    // Define the Lambda function URL resource
    const myFunctionUrl = myFunction.addFunctionUrl({
      authType: lambda.FunctionUrlAuthType.NONE,
    });

    // Define a CloudFormation output for your URL
    new CfnOutput(this, "myFunctionUrlOutput", {
      value: myFunctionUrl.url,
    })

  }
}

module.exports = { HelloCdkStack }
```

Python

Located in `hello_cdk/hello_cdk_stack.py`:

```
from aws_cdk import (
  # ...
  CfnOutput # Import CfnOutput
)
from constructs import Construct

class HelloCdkStack(Stack):

  def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
    super().__init__(scope, construct_id, **kwargs)

    # Define the Lambda function resource
    # ...

    # Define the Lambda function URL resource
```

```
    my_function_url = my_function.add_function_url(
      auth_type = _lambda.FunctionUrlAuthType.NONE,
    )

    # Define a CloudFormation output for your URL
    CfnOutput(self, "myFunctionUrlOutput", value=my_function_url.url)
```

Java

Located in `src/main/java/.../HelloCdkStack.java` :

```
package com.myorg;

// ...
// Import Lambda function URL
import software.amazon.awscdk.services.lambda.FunctionUrl;
import software.amazon.awscdk.services.lambda.FunctionUrlAuthType;
import software.amazon.awscdk.services.lambda.FunctionUrlOptions;
// Import CfnOutput
import software.amazon.awscdk.CfnOutput;

public class HelloCdkStack extends Stack {
  public HelloCdkStack(final Construct scope, final String id) {
    this(scope, id, null);
  }

  public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
    super(scope, id, props);

    // Define the Lambda function resource
    // ...

    // Define the Lambda function URL resource
    FunctionUrl myFunctionUrl =
 myFunction.addFunctionUrl(FunctionUrlOptions.builder()
      .authType(FunctionUrlAuthType.NONE)
      .build());

    // Define a CloudFormation output for your URL
    CfnOutput.Builder.create(this, "myFunctionUrlOutput")
      .value(myFunctionUrl.getUrl())
      .build();
  }
```

```
}
```

C#

Located in `src/main/java/…/HelloCdkStack.java` :

```
// ...

namespace HelloCdk
{
  public class HelloCdkStack : Stack
  {
    internal HelloCdkStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
    {
      // Define the Lambda function resource
      // ...

      // Define the Lambda function URL resource
      var myFunctionUrl = myFunction.AddFunctionUrl(new FunctionUrlOptions
      {
        AuthType = FunctionUrlAuthType.NONE
      });

      // Define a CloudFormation output for your URL
      new CfnOutput(this, "myFunctionUrlOutput", new CfnOutputProps
      {
        Value = myFunctionUrl.Url
      });
    }
  }
}
```

Go

Located in `hello-cdk.go`:

```
// ...

func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
```

```
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  // Define the Lambda function resource
  // ...

  // Define the Lambda function URL resource
  myFunctionUrl := myFunction.AddFunctionUrl(&awslambda.FunctionUrlOptions{
    AuthType: awslambda.FunctionUrlAuthType_NONE,
  })

  // Define a CloudFormation output for your URL
  awscdk.NewCfnOutput(stack, jsii.String("myFunctionUrlOutput"),
 &awscdk.CfnOutputProps{
    Value: myFunctionUrl.Url(),
  })

  return stack
}

// ...
```

Warning

To keep this tutorial simple, your Lambda function URL is defined without authentication. When deployed, this creates a publicly accessible endpoint that can be used to invoke your function. When you are done with this tutorial, follow Step 12: Delete your application to delete these resources.

Step 8: Synthesize a CloudFormation template

In this step, you prepare for deployment by synthesizing a CloudFormation template with the CDK CLI `cdk synth` command. This command performs basic validation of your CDK code, runs your CDK app, and generates a CloudFormation template from your CDK stack.

If your app contains more than one stack, you must specify which stacks to synthesize. Since your app contains a single stack, the CDK CLI automatically detects the stack to synthesize.

If you don't synthesize a template, the CDK CLI will automatically perform this step when you deploy. However, we recommend that you run this step before each deployment to check for synthesis errors.

Before synthesizing a template, you can optionally build your application to catch syntax and type errors. For instructions, see Step 4: Build your CDK app.

To synthesize a CloudFormation template, run the following from the root of your project:

```
$ cdk synth
```

> **Note**
>
> If you receive an error like the following, verify that you are in the `hello-cdk` directory and try again:
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

If successful, the CDK CLI will output a `YAML`–formatted CloudFormation template to `stdout` and save a `JSON`–formatted template in the `cdk.out` directory of your project.

The following is an example output of the CloudFormation template:

AWS CloudFormation template

```
Resources:
  HelloWorldFunctionServiceRole<unique-identifier>:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
```

```
          - Ref: AWS::Partition
          - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: HelloCdkStack/HelloWorldFunction/ServiceRole/Resource
  HelloWorldFunction<unique-identifier>:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        ZipFile: "

          \        exports.handler = async function(event) {

          \          return {

          \            statusCode: 200,

          \            body: JSON.stringify('Hello World!'),

          \          };

          \        };

          \      "
      Handler: index.handler
      Role:
        Fn::GetAtt:
          - HelloWorldFunctionServiceRole<unique-identifier>
          - Arn
      Runtime: nodejs20.x
    DependsOn:
      - HelloWorldFunctionServiceRole<unique-identifier>
    Metadata:
      aws:cdk:path: HelloCdkStack/HelloWorldFunction/Resource
  HelloWorldFunctionFunctionUrl<unique-identifier>:
    Type: AWS::Lambda::Url
    Properties:
      AuthType: NONE
      TargetFunctionArn:
        Fn::GetAtt:
          - HelloWorldFunction<unique-identifier>
          - Arn
    Metadata:
      aws:cdk:path: HelloCdkStack/HelloWorldFunction/FunctionUrl/Resource
  HelloWorldFunctioninvokefunctionurl<unique-identifier>:
```

```
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunctionUrl
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunction<unique-identifier>
          - Arn
      FunctionUrlAuthType: NONE
      Principal: "*"
    Metadata:
      aws:cdk:path: HelloCdkStack/HelloWorldFunction/invoke-function-url
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Analytics: v2:deflate64:<unique-identifier>
    Metadata:
      aws:cdk:path: HelloCdkStack/CDKMetadata/Default
    Condition: CDKMetadataAvailable
Outputs:
  myFunctionUrlOutput:
    Value:
      Fn::GetAtt:
        - HelloWorldFunctionFunctionUrl<unique-identifier>
        - FunctionUrl
Parameters:
  BootstrapVersion:
    Type: AWS::SSM::Parameter::Value<String>
    Default: /cdk-bootstrap/<unique-identifier>/version
    Description: Version of the CDK Bootstrap resources in this environment,
 automatically retrieved from SSM Parameter Store. [cdk:skip]
Rules:
  CheckBootstrapVersion:
    Assertions:
      - Assert:
          Fn::Not:
            - Fn::Contains:
                - - "1"
                  - "2"
                  - "3"
                  - "4"
                  - "5"
                - Ref: BootstrapVersion
```

```
        AssertDescription: CDK bootstrap stack version 6 required. Please run 'cdk
 bootstrap' with a recent version of the CDK CLI.
```

Note

Every generated template contains an `AWS::CDK::Metadata` resource by default. The AWS CDK team uses this metadata to gain insight into AWS CDK usage and find ways to improve it. For details, including how to opt out of version reporting, see Version reporting.

By defining a single L2 construct, the AWS CDK creates an extensive CloudFormation template containing your Lambda resources, along with the permissions and glue logic required for your resources to interact within your application.

Step 9: Deploy your CDK stack

In this step, you use the CDK CLI `cdk deploy` command to deploy your CDK stack. This command retrieves your generated CloudFormation template and deploys it through AWS CloudFormation, which provisions your resources as part of a CloudFormation stack.

From the root of your project, run the following. Confirm changes if prompted:

```
$ cdk deploy

#  Synthesis time: 2.69s

HelloCdkStack:  start: Building <unique-identifier>:current_account-current_region
HelloCdkStack:  success: Built <unique-identifier>:current_account-current_region
HelloCdkStack:  start: Publishing <unique-identifier>:current_account-current_region
HelloCdkStack:  success: Published <unique-identifier>:current_account-current_region
This deployment will make potentially sensitive changes according to your current
 security approval level (--require-approval broadening).
Please confirm you intend to make the following modifications:

IAM Statement Changes
##############################################################################################
#   # Resource                              # Effect # Action                  #
 Principal                    # Condition #
##############################################################################################
# + # ${HelloWorldFunction.Arn}             # Allow  # lambda:InvokeFunctionUrl # *
                        #           #
```

```
###############################################################################################
# + # ${HelloWorldFunction/ServiceRole.Arn} # Allow  # sts:AssumeRole          #
 Service:lambda.amazonaws.com #           #
###############################################################################################
IAM Policy Changes
###############################################################################################
#   # Resource                          # Managed Policy ARN
                                  #
###############################################################################################
# + # ${HelloWorldFunction/ServiceRole} # arn:${AWS::Partition}:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole #
###############################################################################################
(NOTE: There may be security-related changes not in this list. See https://github.com/
aws/aws-cdk/issues/1299)

Do you wish to deploy these changes (y/n)? y
```

Similar to `cdk synth`, you don't have to specify the AWS CDK stack since the app contains a single stack.

During deployment, the CDK CLI displays progress information as your stack is deployed. When complete, you can go to the AWS CloudFormation console to view your `HelloCdkStack` stack. You can also go to the Lambda console to view your `HelloWorldFunction` resource.

When deployment completes, the CDK CLI will output your endpoint URL. Copy this URL for the next step. The following is an example:

```
...
HelloCdkStack: deploying... [1/1]
HelloCdkStack: creating CloudFormation changeset...

 #  HelloCdkStack

#  Deployment time: 41.65s

Outputs:
HelloCdkStack.myFunctionUrlOutput = https://<api-id>.lambda-url.<Region>.on.aws/
Stack ARN:
arn:aws:cloudformation:<Region:account-id>:stack/HelloCdkStack/<unique-identifier>

#  Total time: 44.34s
```

Step 10: Interact with your application on AWS

In this step, you interact with your application on AWS by invoking your Lambda function through the function URL. When you access the URL, your Lambda function returns the `Hello World!` message.

To invoke your function, access the function URL through your browser or from the command line. The following is an example:

```
$ curl https://<api-id>.lambda-url.<Region>.on.aws/
"Hello World!"%
```

Step 11: Modify your application

In this step, you modify the message that the Lambda function returns when invoked. You perform a diff using the CDK CLI `cdk diff` command to preview your changes and deploy to update your application. You then interact with your application on AWS to see your new message.

Modify the `myFunction` instance in your CDK stack file as follows:

Example

TypeScript

Located in `lib/hello-cdk-stack.ts`:

```
// ...

export class HelloCdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Modify the Lambda function resource
    const myFunction = new lambda.Function(this, "HelloWorldFunction", {
      runtime: lambda.Runtime.NODEJS_20_X, // Provide any supported Node.js runtime
      handler: "index.handler",
      code: lambda.Code.fromInline(`
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello CDK!'),
```

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

    // ...
  }
}
```

JavaScript

Located in `lib/hello-cdk-stack.js`:

```
// ...

class HelloCdkStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Modify the Lambda function resource
    const myFunction = new lambda.Function(this, "HelloWorldFunction", {
      runtime: lambda.Runtime.NODEJS_20_X, // Provide any supported Node.js runtime
      handler: "index.handler",
      code: lambda.Code.fromInline(`
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello CDK!'),
          };
        };
      `),
    });

    // ...

  }
}

module.exports = { HelloCdkStack }
```

Python

Located in `hello_cdk/hello_cdk_stack.py`:

```
# ...

class HelloCdkStack(Stack):

  def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
    super().__init__(scope, construct_id, **kwargs)

    # Modify the Lambda function resource
    my_function = _lambda.Function(
      self, "HelloWorldFunction",
      runtime = _lambda.Runtime.NODEJS_20_X, # Provide any supported Node.js runtime
      handler = "index.handler",
      code = _lambda.Code.from_inline(
        """
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello CDK!'),
          };
        };
        """
      ),
    )

    # ...
```

Java

Located in `src/main/java/.../HelloCdkStack.java` :

```
// ...

public class HelloCdkStack extends Stack {
  public HelloCdkStack(final Construct scope, final String id) {
    this(scope, id, null);
  }

  public HelloCdkStack(final Construct scope, final String id, final StackProps
 props) {
    super(scope, id, props);

    // Modify the Lambda function resource
    Function myFunction = Function.Builder.create(this, "HelloWorldFunction")
```

```
      .runtime(Runtime.NODEJS_20_X) // Provide any supported Node.js runtime
      .handler("index.handler")
      .code(Code.fromInline(
        "exports.handler = async function(event) {" +
        " return {" +
        " statusCode: 200," +
        " body: JSON.stringify('Hello CDK!')" +
        " };" +
        "};"))
      .build();

    // ...
  }
}
```

C#

```
// ...

namespace HelloCdk
{
  public class HelloCdkStack : Stack
  {
    internal HelloCdkStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
    {
      // Modify the Lambda function resource
      var myFunction = new Function(this, "HelloWorldFunction", new FunctionProps
      {
        Runtime = Runtime.NODEJS_20_X, // Provide any supported Node.js runtime
        Handler = "index.handler",
        Code = Code.FromInline(@"
          exports.handler = async function(event) {
            return {
              statusCode: 200,
              body: JSON.stringify('Hello CDK!'),
            };
          };
        "),
      });

      // ...
    }
```

```
  }
}
```

Go

```
// ...

type HelloCdkStackProps struct {
  awscdk.StackProps
}

func NewHelloCdkStack(scope constructs.Construct, id string, props
 *HelloCdkStackProps) awscdk.Stack {
  var sprops awscdk.StackProps
  if props != nil {
    sprops = props.StackProps
  }
  stack := awscdk.NewStack(scope, &id, &sprops)

  // Modify the Lambda function resource
  myFunction := awslambda.NewFunction(stack, jsii.String("HelloWorldFunction"),
 &awslambda.FunctionProps{
    Runtime: awslambda.Runtime_NODEJS_20_X(), // Provide any supported Node.js
 runtime
    Handler: jsii.String("index.handler"),
    Code: awslambda.Code_FromInline(jsii.String(`
      exports.handler = async function(event) {
        return {
          statusCode: 200,
          body: JSON.stringify('Hello CDK!'),
        };
      };
    `)),
  })

// ...

}
```

Currently, your code changes have not made any direct updates to your deployed Lambda resource. Your code defines the desired state of your resource. To modify your deployed resource, you will use the CDK CLI to synthesize the desired state into a new AWS CloudFormation template. Then, you will deploy your new CloudFormation template as a change set. Change sets make only the necessary changes to reach your new desired state.

To preview your changes, run the `cdk diff` command. The following is an example:

```
$ cdk diff
Stack HelloCdkStack
Hold on while we create a read-only change set to get a diff with accurate replacement
 information (use --no-change-set to use a less accurate but faster template-only diff)
Resources
[~] AWS::Lambda::Function HelloWorldFunction HelloWorldFunction<unique-identifier>
 ## [~] Code
     ## [~] .ZipFile:
         ## [-]
                exports.handler = async function(event) {
                    return {
                      statusCode: 200,
                      body: JSON.stringify('Hello World!'),
                    };
                };

         ## [+]
                exports.handler = async function(event) {
                    return {
                      statusCode: 200,
                      body: JSON.stringify('Hello CDK!'),
                    };
                };


#  Number of stacks with differences: 1
```

To create this diff, the CDK CLI queries your AWS account account for the latest AWS CloudFormation template for the `HelloCdkStack` stack. Then, it compares the latest template with the template it just synthesized from your app.

To implement your changes, run the `cdk deploy` command. The following is an example:

```
$ cdk deploy

#  Synthesis time: 2.12s

```

```
HelloCdkStack:  start: Building <unique-identifier>:current_account-current_region
HelloCdkStack:  success: Built <unique-identifier>:current_account-current_region
HelloCdkStack:  start: Publishing <unique-identifier>:current_account-current_region
HelloCdkStack:  success: Published <unique-identifier>:current_account-current_region
HelloCdkStack: deploying... [1/1]
HelloCdkStack: creating CloudFormation changeset...

 #  HelloCdkStack

#  Deployment time: 26.96s

Outputs:
HelloCdkStack.myFunctionUrlOutput = https://<unique-identifier>.lambda-
url.<Region>.on.aws/
Stack ARN:
arn:aws:cloudformation:<Region:account-id>:stack/HelloCdkStack/<unique-identifier>

#  Total time: 29.07s
```

To interact with your application, repeat Step 10: Interact with your application on AWS. The following is an example:

```
$ curl https://<api-id>.lambda-url.<Region>.on.aws/
"Hello CDK!"%
```

Step 12: Delete your application

In this step, you use the CDK CLI `cdk destroy` command to delete your application. This command deletes the CloudFormation stack associated with your CDK stack, which includes the resources you created.

To delete your application, run the `cdk destroy` command and confirm your request to delete the application. The following is an example:

```
$ cdk destroy
Are you sure you want to delete: HelloCdkStack (y/n)? y
HelloCdkStack: destroying... [1/1]

 #  HelloCdkStack: destroyed
```

Next steps

Congratulations! You've completed this tutorial and have used the AWS CDK to successfully create, modify, and delete resources in the AWS Cloud. You're now ready to begin using the AWS CDK.

To learn more about using the AWS CDK in your preferred programming language, see Work with the AWS CDK library.

For additional resources, see the following:

- Try the CDK Workshop for a more in-depth tour involving a more complex project.
- See the API reference to begin exploring the CDK constructs available for your favorite AWS services.
- Visit Construct Hub to discover constructs created by AWS and others.
- Explore Examples of using the AWS CDK.

The AWS CDK is an open-source project. To contribute, see to Contributing to the AWS Cloud Development Kit (AWS CDK).

Work with the AWS CDK library

Import and use the AWS Cloud Development Kit (AWS CDK) library to define your AWS Cloud infrastructure with a supported programming language.

Import the AWS CDK Library

The AWS CDK Library is often referred to by its TypeScript package name of `aws-cdk-lib`. The actual package name varies by language. The following is an example of how to install and import the CDK Library:

Example

TypeScript

Install	`npm install aws-cdk-lib`
Import	`import * as cdk from 'aws-cdk-lib';`

JavaScript

Install	`npm install aws-cdk-lib`
Import	`const cdk = require('aws-cdk-lib');`

Python

Install	`python -m pip install aws-cdk-lib`
Import	`import aws_cdk as cdk`

Java

In `pom.xml`, add	`Group software.amazon.awscdk; artifact aws-cdk-lib`
Import	`import software.amazon.awscdk.App;`

C#

Install	`dotnet add package Amazon.CDK.Lib`
Import	`using Amazon.CDK;`

Go

Install	`go get github.com/aws/aws-cdk-go/awscdk/v2`
Import	`import ( "github.com/aws/aws-cdk-go/awscdk/v2" )`

The `construct` base class and supporting code is in the `constructs` library. Experimental constructs, where the API is still undergoing refinement, are distributed as separate modules.

Using the AWS CDK API Reference

Use the AWS CDK API reference as you develop with the AWS CDK.

Each module's reference material is broken into the following sections.

- *Overview*: Introductory material you'll need to know to work with the service in the AWS CDK, including concepts and examples.

- *Constructs*: Library classes that represent one or more concrete AWS resources. These are the "curated" (L2) resources or patterns (L3 resources) that provide a high-level interface with sane defaults.
- *Classes*: Non-construct classes that provide functionality used by constructs in the module.
- *Structs*: Data structures (attribute bundles) that define the structure of composite values such as properties (the `props` argument of constructs) and options.
- *Interfaces*: Interfaces, whose names all begin with "I", define the absolute minimum functionality for the corresponding construct or other class. The CDK uses construct interfaces to represent AWS resources that are defined outside your AWS CDK app and referenced by methods such as `Bucket.fromBucketArn()`.
- *Enums*: Collections of named values for use in specifying certain construct parameters. Using an enumerated value allows the CDK to check these values for validity during synthesis.
- *CloudFormation Resources*: These L1 constructs, whose names begin with "Cfn", represent exactly the resources defined in the CloudFormation specification. They are automatically generated from that specification with each CDK release. Each L2 or L3 construct encapsulates one or more CloudFormation resources.
- *CloudFormation Property Types*: The collection of named values that define the properties for each CloudFormation Resource.

Interfaces compared with construct classes

The AWS CDK uses interfaces in a specific way that may not be obvious even if you are familiar with interfaces as a programming concept.

The AWS CDK supports using resources defined outside CDK applications using methods such as `Bucket.fromBucketArn()`. External resources cannot be modified and may not have all the functionality available with resources defined in your CDK app using e.g. the `Bucket` class. Interfaces, then, represent the bare minimum functionality available in the CDK for a given AWS resource type, *including external resources*.

When instantiating resources in your CDK app, then, you should always use concrete classes such as `Bucket`. When specifying the type of an argument you are accepting in one of your own constructs, use the interface type such as `IBucket` if you are prepared to deal with external resources (that is, you won't need to change them). If you require a CDK-defined construct, specify the most general type you can use.

Some interfaces are minimum versions of properties or options bundles associated with specific classes, rather than constructs. Such interfaces can be useful when subclassing to accept arguments that you'll pass on to your parent class. If you require one or more additional properties, you'll want to implement or derive from this interface, or from a more specific type.

Note

Some programming languages supported by the AWS CDK don't have an interface feature. In these languages, interfaces are just ordinary classes. You can identify them by their names, which follow the pattern of an initial "I" followed by the name of some other construct (e.g. `IBucket`). The same rules apply.

Managing dependencies

Dependencies for your AWS CDK app or library are managed using package management tools. These tools are commonly used with the programming languages.

Typically, the AWS CDK supports the language's standard or official package management tool if there is one. Otherwise, the AWS CDK will support the language's most popular or widely supported one. You may also be able to use other tools, especially if they work with the supported tools. However, official support for other tools is limited.

The AWS CDK supports the following package managers:

Language	Supported package management tool
TypeScript/JavaScript	NPM (Node Package Manager) or Yarn
Python	PIP (Package Installer for Python)
Java	Maven
C#	NuGet
Go	Go modules

When you create a new project using the AWS CDK CLI `cdk init` command, dependencies for the CDK core libraries and stable constructs are automatically specified.

For more information on managing dependencies for supported programming languages, see the following:

- Managing dependencies in TypeScript.
- Managing dependencies in JavaScript.
- Managing dependencies in Python.
- Managing dependencies in Java.
- Managing dependencies in C#.
- Managing dependencies in Go.

Comparing AWS CDK in TypeScript with other languages

TypeScript was the first language supported for developing AWS CDK applications. Therefore, a substantial amount of example CDK code is written in TypeScript. If you are developing in another language, it might be useful to compare how AWS CDK code is implemented in TypeScript compared to your language of choice. This can help you use the examples throughout documentation.

Importing a module

Example

TypeScript/JavaScript

TypeScript supports importing either an entire namespace, or individual objects from a namespace. Each namespace includes constructs and other classes for use with a given AWS service.

```
// Import main CDK library as cdk
import * as cdk from 'aws-cdk-lib';   // ES6 import preferred in TS
const cdk = require('aws-cdk-lib');   // Node.js require() preferred in JS

// Import specific core CDK classes
import { Stack, App } from 'aws-cdk-lib';
const { Stack, App } = require('aws-cdk-lib');

// Import AWS S3 namespace as s3 into current namespace
import { aws_s3 as s3 } from 'aws-cdk-lib';   // TypeScript
```

```
const s3 = require('aws-cdk-lib/aws-s3');     // JavaScript

// Having imported cdk already as above, this is also valid
const s3 = cdk.aws_s3;

// Now use s3 to access the S3 types
const bucket = s3.Bucket(...);

// Selective import of s3.Bucket
import { Bucket } from 'aws-cdk-lib/aws-s3';        // TypeScript
const { Bucket } = require('aws-cdk-lib/aws-s3');   // JavaScript

// Now use Bucket to instantiate an S3 bucket
const bucket = Bucket(...);
```

Python

Like TypeScript, Python supports namespaced module imports and selective imports. Namespaces in Python look like **aws_cdk.** *xxx*, where *xxx* represents an AWS service name, such as **s3** for Amazon S3. (Amazon S3 is used in these examples).

```
# Import main CDK library as cdk
import aws_cdk as cdk

# Selective import of specific core classes
from aws_cdk import Stack, App

# Import entire module as s3 into current namespace
import aws_cdk.aws_s3 as s3

# s3 can now be used to access classes it contains
bucket = s3.Bucket(...)

# Selective import of s3.Bucket into current namespace
from aws_cdk.s3 import Bucket

# Bucket can now be used to instantiate a bucket
bucket = Bucket(...)
```

Java

Java's imports work differently from TypeScript's. Each import statement imports either a single class name from a given package, or all classes defined in that package (using *). Classes may be accessed using either the class name by itself if it has been imported, or the *qualified* class name including its package.

Libraries are named like `software.amazon.awscdk.services.xxx` for the AWS Construct Library (the main library is `software.amazon.awscdk`). The Maven group ID for AWS CDK packages is `software.amazon.awscdk`.

```
// Make certain core classes available
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.App;

// Make all Amazon S3 construct library classes available
import software.amazon.awscdk.services.s3.*;

// Make only Bucket and EventType classes available
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.EventType;

// An imported class may now be accessed using the simple class name (assuming that
 name
// does not conflict with another class)
Bucket bucket = Bucket.Builder.create(...).build();

// We can always use the qualified name of a class (including its package) even
 without an
// import directive
software.amazon.awscdk.services.s3.Bucket bucket =
    software.amazon.awscdk.services.s3.Bucket.Builder.create(...)
        .build();

// Java 10 or later can use var keyword to avoid typing the type twice
var bucket =
    software.amazon.awscdk.services.s3.Bucket.Builder.create(...)
        .build();
```

C#

In C#, you import types with the `using` directive. There are two styles. One gives you access to all the types in the specified namespace by using their plain names. With the other, you can refer to the namespace itself by using an alias.

Packages are named like `Amazon.CDK.AWS.xxx` for AWS Construct Library packages. (The core module is `Amazon.CDK`.)

```
// Make CDK base classes available under cdk
using cdk = Amazon.CDK;

// Make all Amazon S3 construct library classes available
using Amazon.CDK.AWS.S3;

// Now we can access any S3 type using its name
var bucket = new Bucket(...);

// Import the S3 namespace under an alias
using s3 = Amazon.CDK.AWS.S3;

// Now we can access an S3 type through the namespace alias
var bucket = new s3.Bucket(...);

// We can always use the qualified name of a type (including its namespace) even
 without a
// using directive
var bucket = new Amazon.CDK.AWS.S3.Bucket(...);
```

Go

Each AWS Construct Library module is provided as a Go package.

```
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"           // CDK core package
    "github.com/aws/aws-cdk-go/awscdk/v2/awss3"     // AWS S3 construct library
 module
)

// now instantiate a bucket
bucket := awss3.NewBucket(...)

// use aliases for brevity/clarity
import (
    cdk "github.com/aws/aws-cdk-go/awscdk/v2"           // CDK core package
    s3  "github.com/aws/aws-cdk-go/awscdk/v2/awss3"     // AWS S3 construct library
 module
)
```

```
bucket := s3.NewBucket(...)
```

Instantiating a construct

AWS CDK construct classes have the same name in all supported languages. Most languages use the `new` keyword to instantiate a class (Python and Go do not). Also, in most languages, the keyword `this` refers to the current instance. (Python uses `self` by convention.) You should pass a reference to the current instance as the `scope` parameter to every construct you create.

The third argument to an AWS CDK construct is `props`, an object containing attributes needed to build the construct. This argument may be optional, but when it is required, the supported languages handle it in idiomatic ways. The names of the attributes are also adapted to the language's standard naming patterns.

Example

TypeScript/JavaScript

```
// Instantiate default Bucket
const bucket = new s3.Bucket(this, 'amzn-s3-demo-bucket');

// Instantiate Bucket with bucketName and versioned properties
const bucket = new s3.Bucket(this, 'amzn-s3-demo-bucket', {
  bucketName: 'amzn-s3-demo-bucket',
   versioned: true,
});

// Instantiate Bucket with websiteRedirect, which has its own sub-properties
const bucket = new s3.Bucket(this, 'amzn-s3-demo-bucket', {
  websiteRedirect: {host: 'aws.amazon.com'}});
```

Python

Python doesn't use a `new` keyword when instantiating a class. The properties argument is represented using keyword arguments, and the arguments are named using `snake_case`.

If a props value is itself a bundle of attributes, it is represented by a class named after the property, which accepts keyword arguments for the subproperties.

In Python, the current instance is passed to methods as the first argument, which is named `self` by convention.

```
# Instantiate default Bucket
bucket = s3.Bucket(self, "amzn-s3-demo-bucket")

# Instantiate Bucket with bucket_name and versioned properties
bucket = s3.Bucket(self, "amzn-s3-demo-bucket", bucket_name="amzn-s3-demo-bucket",
 versioned=true)

# Instantiate Bucket with website_redirect, which has its own sub-properties
bucket = s3.Bucket(self, "amzn-s3-demo-bucket", website_redirect=s3.WebsiteRedirect(
            host_name="aws.amazon.com"))
```

Java

In Java, the props argument is represented by a class named `XxxxProps` (for example, `BucketProps` for the `Bucket` construct's props). You build the props argument using a builder pattern.

Each `XxxxProps` class has a builder. There is also a convenient builder for each construct that builds the props and the construct in one step, as shown in the following example.

Props are named the same as in TypeScript, using `camelCase`.

```
// Instantiate default Bucket
Bucket bucket = Bucket(self, "amzn-s3-demo-bucket");

// Instantiate Bucket with bucketName and versioned properties
Bucket bucket = Bucket.Builder.create(self, "amzn-s3-demo-bucket")
                      .bucketName("amzn-s3-demo-bucket").versioned(true)
                      .build();

# Instantiate Bucket with websiteRedirect, which has its own sub-properties
Bucket bucket = Bucket.Builder.create(self, "amzn-s3-demo-bucket")
                      .websiteRedirect(new websiteRedirect.Builder()
                          .hostName("aws.amazon.com").build())
                      .build();
```

C#

In C#, props are specified using an object initializer to a class named `XxxxProps` (for example, `BucketProps` for the `Bucket` construct's props).

Props are named similarly to TypeScript, except using `PascalCase`.

It is convenient to use the `var` keyword when instantiating a construct, so you don't need to type the class name twice. However, your local code style guide may vary.

```
// Instantiate default Bucket
var bucket = Bucket(self, "amzn-s3-demo-bucket");

// Instantiate Bucket with BucketName and Versioned properties
var bucket =  Bucket(self, "amzn-s3-demo-bucket", new BucketProps {
                      BucketName = "amzn-s3-demo-bucket",
                      Versioned  = true});

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
var bucket = Bucket(self, "amzn-s3-demo-bucket", new BucketProps {
                      WebsiteRedirect = new WebsiteRedirect {
                              HostName = "aws.amazon.com"
                      }});
```

Go

To create a construct in Go, call the function `NewXxxxxx` where `Xxxxxxx` is the name of the construct. The constructs' properties are defined as a struct.

In Go, all construct parameters are pointers, including values like numbers, Booleans, and strings. Use the convenience functions like `jsii.String` to create these pointers.

```
// Instantiate default Bucket
bucket := awss3.NewBucket(stack, jsii.String("amzn-s3-demo-bucket"), nil)

// Instantiate Bucket with BucketName and Versioned properties
bucket1 := awss3.NewBucket(stack, jsii.String("amzn-s3-demo-bucket"),
&awss3.BucketProps{
 BucketName: jsii.String("amzn-s3-demo-bucket"),
 Versioned:  jsii.Bool(true),
})

// Instantiate Bucket with WebsiteRedirect, which has its own sub-properties
bucket2 := awss3.NewBucket(stack, jsii.String("amzn-s3-demo-bucket"),
&awss3.BucketProps{
 WebsiteRedirect: &awss3.RedirectTarget{
  HostName: jsii.String("aws.amazon.com"),
 }})
```

Accessing members

It is common to refer to attributes or properties of constructs and other AWS CDK classes and use these values as, for example, inputs to build other constructs. The naming differences described previously for methods apply here also. Furthermore, in Java, it is not possible to access members directly. Instead, a getter method is provided.

Example

TypeScript/JavaScript

Names are `camelCase`.

```
bucket.bucketArn
```

Python

Names are `snake_case`.

```
bucket.bucket_arn
```

Java

A getter method is provided for each property; these names are `camelCase`.

```
bucket.getBucketArn()
```

C#

Names are `PascalCase`.

```
bucket.BucketArn
```

Go

Names are `PascalCase`.

```
bucket.BucketArn
```

Enum constants

Enum constants are scoped to a class, and have uppercase names with underscores in all languages (sometimes referred to as `SCREAMING_SNAKE_CASE`). Since class names also use the same casing in all supported languages except Go, qualified enum names are also the same in these languages.

```
s3.BucketEncryption.KMS_MANAGED
```

In Go, enum constants are attributes of the module namespace and are written as follows.

```
awss3.BucketEncryption_KMS_MANAGED
```

Object interfaces

The AWS CDK uses TypeScript object interfaces to indicate that a class implements an expected set of methods and properties. You can recognize an object interface because its name starts with `I`. A concrete class indicates the interfaces that it implements by using the `implements` keyword.

Example

TypeScript/JavaScript

> **Note**
>
> JavaScript doesn't have an interface feature. You can ignore the `implements` keyword and the class names following it.

```
import { IAspect, IConstruct } from 'aws-cdk-lib';

class MyAspect implements IAspect {
  public visit(node: IConstruct) {
    console.log('Visited', node.node.path);
  }
}
```

Python

Python doesn't have an interface feature. However, for the AWS CDK you can indicate interface implementation by decorating your class with `@jsii.implements(interface)`.

```
from aws_cdk import IAspect, IConstruct
import jsii

@jsii.implements(IAspect)
class MyAspect():
  def visit(self, node: IConstruct) -> None:
    print("Visited", node.node.path)
```

Java

```
import software.amazon.awscdk.IAspect;
import software.amazon.awscdk.IConstruct;

public class MyAspect implements IAspect {
    public void visit(IConstruct node) {
        System.out.format("Visited %s", node.getNode().getPath());
    }
}
```

C#

```
using Amazon.CDK;

public class MyAspect : IAspect
{
    public void Visit(IConstruct node)
    {
        System.Console.WriteLine($"Visited ${node.Node.Path}");
    }
}
```

Go

Go structs do not need to explicitly declare which interfaces they implement. The Go compiler determines implementation based on the methods and properties available on the structure. For example, in the following code, `MyAspect` implements the `IAspect` interface because it provides a `Visit` method that takes a construct.

```
type MyAspect struct {
}

func (a MyAspect) Visit(node constructs.IConstruct) {
 fmt.Println("Visited", *node.Node().Path())
}
```

Working with the AWS CDK in TypeScript

TypeScript is a fully-supported client language for the AWS Cloud Development Kit (AWS CDK) and is considered stable. Working with the AWS CDK in TypeScript uses familiar tools, including Microsoft's TypeScript compiler (`tsc`), Node.js and the Node Package Manager (`npm`). You may also use Yarn if you prefer, though the examples in this Guide use NPM. The modules comprising the AWS Construct Library are distributed via the NPM repository, npmjs.org.

You can use any editor or IDE. Many AWS CDK developers use Visual Studio Code (or its open-source equivalent VSCodium), which has excellent support for TypeScript.

Get started with TypeScript

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See Getting started with the AWS CDK.

You also need TypeScript itself (version 3.8 or later). If you don't already have it, you can install it using `npm`.

```
$ npm install -g typescript
```

> **Note**
>
> If you get a permission error, and have administrator access on your system, try `sudo npm install -g typescript`.

Keep TypeScript up to date with a regular `npm update -g typescript`.

> **Note**
>
> Third-party language deprecation: language version is only supported until its EOL (End Of Life) shared by the vendor or community and is subject to change with prior notice.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory. Use the `--language` option and specify `typescript`:

```
$ mkdir my-project
$ cd my-project
$ cdk init app --language typescript
```

Creating a project also installs the [`aws-cdk-lib`] module and its dependencies.

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Hyphens in the folder name are converted to underscores. However, the name should otherwise follow the form of a TypeScript identifier; for example, it should not start with a number or contain spaces.

Using local `tsc` and `cdk`

For the most part, this guide assumes you install TypeScript and the CDK Toolkit globally (`npm install -g typescript aws-cdk`), and the provided command examples (such as `cdk synth`) follow this assumption. This approach makes it easy to keep both components up to date, and since both take a strict approach to backward compatibility, there is generally little risk in always using the latest versions.

Some teams prefer to specify all dependencies within each project, including tools like the TypeScript compiler and the CDK Toolkit. This practice lets you pin these components to specific versions and ensure that all developers on your team (and your CI/CD environment) use exactly those versions. This eliminates a possible source of change, helping to make builds and deployments more consistent and repeatable.

The CDK includes dependencies for both TypeScript and the CDK Toolkit in the TypeScript project template's `package.json`, so if you want to use this approach, you don't need to make any changes to your project. All you need to do is use slightly different commands for building your app and for issuing `cdk` commands.

Operation	Use global tools	Use local tools
Initialize project	`cdk init --language typescript`	`npx aws-cdk init --language typescript`
Build	`tsc`	`npm run build`
Run CDK Toolkit command	`cdk …`	`npm run cdk …` or `npx aws-cdk …`

`npx aws-cdk` runs the version of the CDK Toolkit installed locally in the current project, if one exists, falling back to the global installation, if any. If no global installation exists, `npx` downloads a temporary copy of the CDK Toolkit and runs that. You may specify an arbitrary version of the CDK Toolkit using the `@` syntax: `npx aws-cdk@2.0 --version` prints `2.0.0`.

Tip

Set up an alias so you can use the `cdk` command with a local CDK Toolkit installation.

macOS/Linux

```
$ alias cdk="npx aws-cdk"
```

Windows

```
doskey cdk=npx aws-cdk $*
```

Managing AWS Construct Library modules

Use the Node Package Manager (`npm`) to install and update AWS Construct Library modules for use by your apps, as well as other packages you need. (You may use `yarn` instead of `npm` if you prefer.) `npm` also installs the dependencies for those modules automatically.

Most AWS CDK constructs are in the main CDK package, named `aws-cdk-lib`, which is a default dependency in new projects created by `cdk init`. "Experimental" AWS Construct Library modules, where higher-level constructs are still under development, are named like `@aws-cdk/<SERVICE-NAME>-alpha`. The service name has an *aws-* prefix. If you're unsure of a module's name, [search for it on NPM](https://www.npmjs.com).

> **Note**
>
> The [CDK API Reference](https://docs.aws.amazon.com/cdk/api/v2/) also shows the package names.

For example, the command below installs the experimental module for AWS CodeStar.

```
$ npm install @aws-cdk/aws-codestar-alpha
```

Some services' Construct Library support is in more than one namespace. For example, besides `aws-route53`, there are three additional Amazon Route 53 namespaces, `aws-route53-targets`, `aws-route53-patterns`, and `aws-route53resolver`.

Your project's dependencies are maintained in `package.json`. You can edit this file to lock some or all of your dependencies to a specific version or to allow them to be updated to newer versions under certain criteria. To update your project's NPM dependencies to the latest permitted version according to the rules you specified in `package.json`:

```
$ npm update
```

In TypeScript, you import modules into your code under the same name you use to install them using NPM. We recommend the following practices when importing AWS CDK classes and AWS Construct Library modules in your applications. Following these guidelines will help make your code consistent with other AWS CDK applications as well as easier to understand.

- Use ES6-style `import` directives, not `require()`.
- Generally, import individual classes from `aws-cdk-lib`.

  ```
  import { App, Stack } from 'aws-cdk-lib';
  ```

- If you need many classes from `aws-cdk-lib`, you may use a namespace alias of `cdk` instead of importing the individual classes. Avoid doing both.

```
import * as cdk from 'aws-cdk-lib';
```

- Generally, import AWS service constructs using short namespace aliases.

```
import { aws_s3 as s3 } from 'aws-cdk-lib';
```

Managing dependencies in TypeScript

In TypeScript CDK projects, dependencies are specified in the `package.json` file in the project's main directory. The core AWS CDK modules are in a single `NPM` package called `aws-cdk-lib`.

When you install a package using `npm install`, NPM records the package in `package.json` for you.

If you prefer, you may use Yarn in place of NPM. However, the CDK does not support Yarn's plug-and-play mode, which is default mode in Yarn 2. Add the following to your project's `.yarnrc.yml` file to turn off this feature.

```
nodeLinker: node-modules
```

CDK applications

The following is an example `package.json` file generated by the `cdk init --language typescript` command:

```
{
  "name": "my-package",
  "version": "0.1.0",
  "bin": {
    "my-package": "bin/my-package.js"
  },
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  },
  "devDependencies": {
    "@types/jest": "^26.0.10",
```

```
    "@types/node": "10.17.27",
    "jest": "^26.4.2",
    "ts-jest": "^26.2.0",
    "aws-cdk": "2.16.0",
    "ts-node": "^9.0.0",
    "typescript": "~3.9.7"
  },
  "dependencies": {
    "aws-cdk-lib": "2.16.0",
    "constructs": "^10.0.0",
    "source-map-support": "^0.5.16"
  }
}
```

For deployable CDK apps, `aws-cdk-lib` must be specified in the `dependencies` section of `package.json`. You can use a caret (^) version number specifier to indicate that you will accept later versions than the one specified as long as they are within the same major version.

For experimental constructs, specify exact versions for the alpha construct library modules, which have APIs that may change. Do not use ^ or ~ since later versions of these modules may bring API changes that can break your app.

Specify versions of libraries and tools needed to test your app (for example, the `jest` testing framework) in the `devDependencies` section of `package.json`. Optionally, use ^ to specify that later compatible versions are acceptable.

Third-party construct libraries

If you're developing a construct library, specify its dependencies using a combination of the `peerDependencies` and `devDependencies` sections, as shown in the following example `package.json` file.

```
{
  "name": "my-package",
  "version": "0.0.1",
  "peerDependencies": {
    "aws-cdk-lib": "^2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.14.0",
```

```
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "10.0.0",
    "jsii": "^1.50.0",
    "aws-cdk": "^2.14.0"
  }
}
```

In `peerDependencies`, use a caret (^) to specify the lowest version of `aws-cdk-lib` that your library works with. This maximizes the compatibility of your library with a range of CDK versions. Specify exact versions for alpha construct library modules, which have APIs that may change. Using `peerDependencies` makes sure that there is only one copy of all CDK libraries in the `node_modules` tree.

In `devDependencies`, specify the tools and libraries you need for testing, optionally with ^ to indicate that later compatible versions are acceptable. Specify exactly (without ^ or ~) the lowest versions of `aws-cdk-lib` and other CDK packages that you advertise your library be compatible with. This practice makes sure that your tests run against those versions. This way, if you inadvertently use a feature found only in newer versions, your tests can catch it.

Warning

`peerDependencies` are installed automatically only by NPM 7 and later. If you are using NPM 6 or earlier, or if you are using Yarn, you must include the dependencies of your dependencies in `devDependencies`. Otherwise, they won't be installed, and you will receive a warning about unresolved peer dependencies.

Installing and updating dependencies

Run the following command to install your project's dependencies.

Example

NPM

```
# Install the latest version of everything that matches the ranges in 'package.json'
npm install

# Install the same exact dependency versions as recorded in 'package-lock.json'
npm ci
```

Yarn

```
# Install the latest version of everything that matches the ranges in 'package.json'
$ yarn upgrade

# Install the same exact dependency versions as recorded in 'yarn.lock'
$ yarn install --frozen-lockfile
```

To update the installed modules, the preceding `npm install` and `yarn upgrade` commands can be used. Either command updates the packages in `node_modules` to the latest versions that satisfy the rules in `package.json`. However, they do not update `package.json` itself, which you might want to do to set a new minimum version. If you host your package on GitHub, you can configure Dependabot version updates to automatically update `package.json`. Alternatively, use npm-check-updates.

> **Important**
>
> By design, when you install or update dependencies, NPM and Yarn choose the latest version of every package that satisfies the requirements specified in `package.json`. There is always a risk that these versions may be broken (either accidentally or intentionally). Test thoroughly after updating your project's dependencies.

AWS CDK idioms in TypeScript

Props

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*. Argument *props* is a bundle of key/value pairs that the construct uses to configure the AWS resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

In TypeScript, the shape of `props` is defined using an interface that tells you the required and optional arguments and their types. Such an interface is defined for each kind of `props` argument, usually specific to a single construct or method. For example, the `Bucket` construct (in the `aws-cdk-lib/aws-s3` module) specifies a `props` argument conforming to the `BucketProps` interface.

If a property is itself an object, for example the `websiteRedirect` property of `BucketProps`, that object will have its own interface to which its shape must conform, in this case `RedirectTarget`.

If you are subclassing an AWS Construct Library class (or overriding a method that takes a props-like argument), you can inherit from the existing interface to create a new one that specifies any new props your code requires. When calling the parent class or base method, generally you can pass the entire props argument you received, since any attributes provided in the object but not specified in the interface will be ignored.

A future release of the AWS CDK could coincidentally add a new property with a name you used for your own property. Passing the value you receive up the inheritance chain can then cause unexpected behavior. It's safer to pass a shallow copy of the props you received with your property removed or set to `undefined`. For example:

```
super(scope, name, {...props, encryptionKeys: undefined});
```

Alternatively, name your properties so that it is clear that they belong to your construct. This way, it is unlikely they will collide with properties in future AWS CDK releases. If there are many of them, use a single appropriately-named object to hold them.

Missing values

Missing values in an object (such as props) have the value `undefined` in TypeScript. Version 3.7 of the language introduced operators that simplify working with these values, making it easier to specify defaults and "short-circuit" chaining when an undefined value is reached. For more information about these features, see the TypeScript 3.7 Release Notes, specifically the first two features, Optional Chaining and Nullish Coalescing.

Build and run CDK apps

Generally, you should be in the project's root directory when building and running your application.

Node.js cannot run TypeScript directly; instead, your application is converted to JavaScript using the TypeScript compiler, `tsc`. The resulting JavaScript code is then executed.

The AWS CDK automatically does this whenever it needs to run your app. However, it can be useful to compile manually to check for errors and to run tests. To compile your TypeScript app manually, issue `npm run build`. You may also issue `npm run watch` to enter watch mode, in which the TypeScript compiler automatically rebuilds your app whenever you save changes to a source file.

Working with the AWS CDK in JavaScript

JavaScript is a fully-supported client language for the AWS CDK and is considered stable. Working with the AWS Cloud Development Kit (AWS CDK) in JavaScript uses familiar tools, including Node.js and the Node Package Manager (`npm`). You may also use Yarn if you prefer, though the examples in this Guide use NPM. The modules comprising the AWS Construct Library are distributed via the NPM repository, npmjs.org.

You can use any editor or IDE. Many AWS CDK developers use Visual Studio Code (or its open-source equivalent VSCodium), which has good support for JavaScript.

Get started with JavaScript

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See Getting started with the AWS CDK.

JavaScript AWS CDK applications require no additional prerequisites beyond these.

Note

Third-party language deprecation: language version is only supported until its EOL (End Of Life) shared by the vendor or community and is subject to change with prior notice.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory. Use the `--language` option and specify `javascript`:

```
$ mkdir my-project
$ cd my-project
$ cdk init app --language javascript
```

Creating a project also installs the aws-cdk-lib module and its dependencies.

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Hyphens in the folder name are converted to underscores. However, the name should otherwise follow the form of a JavaScript identifier; for example, it should not start with a number or contain spaces.

Using local cdk

For the most part, this guide assumes you install the CDK Toolkit globally (`npm install -g aws-cdk`), and the provided command examples (such as `cdk synth`) follow this assumption. This approach makes it easy to keep the CDK Toolkit up to date, and since the CDK takes a strict approach to backward compatibility, there is generally little risk in always using the latest version.

Some teams prefer to specify all dependencies within each project, including tools like the CDK Toolkit. This practice lets you pin such components to specific versions and ensure that all developers on your team (and your CI/CD environment) use exactly those versions. This eliminates a possible source of change, helping to make builds and deployments more consistent and repeatable.

The CDK includes a dependency for the CDK Toolkit in the JavaScript project template's `package.json`, so if you want to use this approach, you don't need to make any changes to your project. All you need to do is use slightly different commands for building your app and for issuing `cdk` commands.

Operation	Use global tools	Use local tools
Initialize project	`cdk init --language javascript`	`npx aws-cdk init --language javascript`
Run CDK Toolkit command	`cdk …`	`npm run cdk …` or `npx aws-cdk …`

`npx aws-cdk` runs the version of the CDK Toolkit installed locally in the current project, if one exists, falling back to the global installation, if any. If no global installation exists, `npx` downloads a temporary copy of the CDK Toolkit and runs that. You may specify an arbitrary version of the CDK Toolkit using the `@` syntax: `npx aws-cdk@1.120 --version` prints `1.120.0`.

Tip

Set up an alias so you can use the `cdk` command with a local CDK Toolkit installation.

macOS/Linux

```
$ alias cdk="npx aws-cdk"
```

Windows

```
doskey cdk=npx aws-cdk $*
```

Managing AWS Construct Library modules

Use the Node Package Manager (npm) to install and update AWS Construct Library modules for use by your apps, as well as other packages you need. (You may use `yarn` instead of `npm` if you prefer.) `npm` also installs the dependencies for those modules automatically.

Most AWS CDK constructs are in the main CDK package, named `aws-cdk-lib`, which is a default dependency in new projects created by `cdk init`. "Experimental" AWS Construct Library modules, where higher-level constructs are still under development, are named like `aws-cdk-lib/<SERVICE-NAME>-alpha`. The service name has an *aws-* prefix. If you're unsure of a module's name, search for it on NPM.

Note

The CDK API Reference also shows the package names.

For example, the command below installs the experimental module for AWS CodeStar.

```
npm install @aws-cdk/aws-codestar-alpha
```

Some services' Construct Library support is in more than one namespace. For example, besides `aws-route53`, there are three additional Amazon Route 53 namespaces, `aws-route53-targets`, `aws-route53-patterns`, and `aws-route53resolver`.

Your project's dependencies are maintained in `package.json`. You can edit this file to lock some or all of your dependencies to a specific version or to allow them to be updated to newer versions under certain criteria. To update your project's NPM dependencies to the latest permitted version according to the rules you specified in `package.json`:

```
npm update
```

In JavaScript, you import modules into your code under the same name you use to install them using NPM. We recommend the following practices when importing AWS CDK classes and AWS Construct Library modules in your applications. Following these guidelines will help make your code consistent with other AWS CDK applications as well as easier to understand.

- Use `require()`, not ES6-style `import` directives. Older versions of Node.js do not support ES6 imports, so using the older syntax is more widely compatible. (If you really want to use ES6 imports, use esm to ensure your project is compatible with all supported versions of Node.js.)
- Generally, import individual classes from `aws-cdk-lib`.

```
const { App, Stack } = require('aws-cdk-lib');
```

- If you need many classes from `aws-cdk-lib`, you may use a namespace alias of `cdk` instead of importing the individual classes. Avoid doing both.

```
const cdk = require('aws-cdk-lib');
```

- Generally, import AWS Construct Libraries using short namespace aliases.

```
const { s3 } = require('aws-cdk-lib/aws-s3');
```

Managing dependencies in JavaScript

In JavaScript CDK projects, dependencies are specified in the `package.json` file in the project's main directory. The core AWS CDK modules are in a single `NPM` package called `aws-cdk-lib`.

When you install a package using `npm install`, NPM records the package in `package.json` for you.

If you prefer, you may use Yarn in place of NPM. However, the CDK does not support Yarn's plug-and-play mode, which is default mode in Yarn 2. Add the following to your project's `.yarnrc.yml` file to turn off this feature.

```
nodeLinker: node-modules
```

CDK applications

The following is an example `package.json` file generated by the `cdk init --language typescript` command. The file generated for JavaScript is similar, only without the TypeScript-related entries.

```
{
  "name": "my-package",
  "version": "0.1.0",
  "bin": {
    "my-package": "bin/my-package.js"
  },
  "scripts": {
    "build": "tsc",
    "watch": "tsc -w",
    "test": "jest",
    "cdk": "cdk"
  },
  "devDependencies": {
    "@types/jest": "^26.0.10",
    "@types/node": "10.17.27",
    "jest": "^26.4.2",
    "ts-jest": "^26.2.0",
    "aws-cdk": "2.16.0",
    "ts-node": "^9.0.0",
    "typescript": "~3.9.7"
  },
  "dependencies": {
    "aws-cdk-lib": "2.16.0",
    "constructs": "^10.0.0",
    "source-map-support": "^0.5.16"
  }
}
```

For deployable CDK apps, `aws-cdk-lib` must be specified in the `dependencies` section of `package.json`. You can use a caret (^) version number specifier to indicate that you will accept later versions than the one specified as long as they are within the same major version.

For experimental constructs, specify exact versions for the alpha construct library modules, which have APIs that may change. Do not use ^ or ~ since later versions of these modules may bring API changes that can break your app.

Specify versions of libraries and tools needed to test your app (for example, the `jest` testing framework) in the `devDependencies` section of `package.json`. Optionally, use ^ to specify that later compatible versions are acceptable.

Third-party construct libraries

If you're developing a construct library, specify its dependencies using a combination of the `peerDependencies` and `devDependencies` sections, as shown in the following example `package.json` file.

```
{
  "name": "my-package",
  "version": "0.0.1",
  "peerDependencies": {
    "aws-cdk-lib": "^2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "2.14.0",
    "@aws-cdk/aws-appsync-alpha": "2.10.0-alpha",
    "constructs": "10.0.0",
    "jsii": "^1.50.0",
    "aws-cdk": "^2.14.0"
  }
}
```

In `peerDependencies`, use a caret (^) to specify the lowest version of `aws-cdk-lib` that your library works with. This maximizes the compatibility of your library with a range of CDK versions. Specify exact versions for alpha construct library modules, which have APIs that may change. Using `peerDependencies` makes sure that there is only one copy of all CDK libraries in the `node_modules` tree.

In `devDependencies`, specify the tools and libraries you need for testing, optionally with ^ to indicate that later compatible versions are acceptable. Specify exactly (without ^ or ~) the lowest versions of `aws-cdk-lib` and other CDK packages that you advertise your library be compatible with. This practice makes sure that your tests run against those versions. This way, if you inadvertently use a feature found only in newer versions, your tests can catch it.

> **Warning**
>
> `peerDependencies` are installed automatically only by NPM 7 and later. If you are using NPM 6 or earlier, or if you are using Yarn, you must include the dependencies of your dependencies in `devDependencies`. Otherwise, they won't be installed, and you will receive a warning about unresolved peer dependencies.

Installing and updating dependencies

Run the following command to install your project's dependencies.

Example

NPM

```
# Install the latest version of everything that matches the ranges in 'package.json'
npm install

# Install the same exact dependency versions as recorded in 'package-lock.json'
npm ci
```

Yarn

```
# Install the latest version of everything that matches the ranges in 'package.json'
yarn upgrade

# Install the same exact dependency versions as recorded in 'yarn.lock'
yarn install --frozen-lockfile
```

To update the installed modules, the preceding `npm install` and `yarn upgrade` commands can be used. Either command updates the packages in `node_modules` to the latest versions that satisfy the rules in `package.json`. However, they do not update `package.json` itself, which you might want to do to set a new minimum version. If you host your package on GitHub, you can configure [Dependabot version updates](#) to automatically update `package.json`. Alternatively, use [npm-check-updates](#).

Important

By design, when you install or update dependencies, NPM and Yarn choose the latest version of every package that satisfies the requirements specified in `package.json`. There is always a risk that these versions may be broken (either accidentally or intentionally). Test thoroughly after updating your project's dependencies.

AWS CDK idioms in JavaScript

Props

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the AWS resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

Using an IDE or editor that has good JavaScript autocomplete will help avoid misspelling property names. If a construct is expecting an `encryptionKeys` property, and you spell it `encryptionkeys`, when instantiating the construct, you haven't passed the value you intended. This can cause an error at synthesis time if the property is required, or cause the property to be silently ignored if it is optional. In the latter case, you may get a default behavior you intended to override. Take special care here.

When subclassing an AWS Construct Library class (or overriding a method that takes a props-like argument), you may want to accept additional properties for your own use. These values will be ignored by the parent class or overridden method, because they are never accessed in that code, so you can generally pass on all the props you received.

A future release of the AWS CDK could coincidentally add a new property with a name you used for your own property. Passing the value you receive up the inheritance chain can then cause unexpected behavior. It's safer to pass a shallow copy of the props you received with your property removed or set to `undefined`. For example:

```
super(scope, name, {...props, encryptionKeys: undefined});
```

Alternatively, name your properties so that it is clear that they belong to your construct. This way, it is unlikely they will collide with properties in future AWS CDK releases. If there are many of them, use a single appropriately-named object to hold them.

Missing values

Missing values in an object (such as `props`) have the value `undefined` in JavaScript. The usual techniques apply for dealing with these. For example, a common idiom for accessing a property of a value that may be undefined is as follows:

```
// a may be undefined, but if it is not, it may have an attribute b
// c is undefined if a is undefined, OR if a doesn't have an attribute b
let c = a && a.b;
```

However, if a could have some other "falsy" value besides `undefined`, it is better to make the test more explicit. Here, we'll take advantage of the fact that `null` and `undefined` are equal to test for them both at once:

```
let c = a == null ? a : a.b;
```

Tip

Node.js 14.0 and later support new operators that can simplify the handling of undefined values. For more information, see the optional chaining and nullish coalescing proposals.

Using TypeScript examples with JavaScript

TypeScript is the language we use to develop the AWS CDK, and it was the first language supported for developing applications, so many available AWS CDK code examples are written in TypeScript. These code examples can be a good resource for JavaScript developers; you just need to remove the TypeScript-specific parts of the code.

TypeScript snippets often use the newer ECMAScript `import` and `export` keywords to import objects from other modules and to declare the objects to be made available outside the current module. Node.js has just begun supporting these keywords in its latest releases. Depending on the version of Node.js you're using (or wish to support), you might rewrite imports and exports to use the older syntax.

Imports can be replaced with calls to the `require()` function.

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Bucket, BucketPolicy } from 'aws-cdk-lib/aws-s3';
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const { Bucket, BucketPolicy } = require('aws-cdk-lib/aws-s3');
```

Exports can be assigned to the `module.exports` object.

Example

TypeScript

```
export class Stack1 extends cdk.Stack {
  // ...
}

export class Stack2 extends cdk.Stack {
  // ...
}
```

JavaScript

```
class Stack1 extends cdk.Stack {
  // ...
}

class Stack2 extends cdk.Stack {
  // ...
}

module.exports = { Stack1, Stack2 }
```

> **Note**
>
> An alternative to using the old-style imports and exports is to use the esm module.

Once you've got the imports and exports sorted, you can dig into the actual code. You may run into these commonly-used TypeScript features:

- Type annotations
- Interface definitions
- Type conversions/casts
- Access modifiers

Type annotations may be provided for variables, class members, function parameters, and function return types. For variables, parameters, and members, types are specified by following the identifier with a colon and the type. Function return values follow the function signature and consist of a colon and the type.

To convert type-annotated code to JavaScript, remove the colon and the type. Class members must have some value in JavaScript; set them to `undefined` if they only have a type annotation in TypeScript.

Example

TypeScript

```
var encrypted: boolean = true;

class myStack extends cdk.Stack {
    bucket: s3.Bucket;
    // ...
}

function makeEnv(account: string, region: string) : object {
    // ...
}
```

JavaScript

```
var encrypted = true;

class myStack extends cdk.Stack {
    bucket = undefined;
    // ...
}

function makeEnv(account, region) {
    // ...
}
```

In TypeScript, interfaces are used to give bundles of required and optional properties, and their types, a name. You can then use the interface name as a type annotation. TypeScript will make sure that the object you use as, for example, an argument to a function has the required properties of the right types.

```
interface myFuncProps {
    code: lambda.Code,
    handler?: string
}
```

JavaScript does not have an interface feature, so once you've removed the type annotations, delete the interface declarations entirely.

When a function or method returns a general-purpose type (such as `object`), but you want to treat that value as a more specific child type to access properties or methods that are not part of the more general type's interface, TypeScript lets you *cast* the value using `as` followed by a type or interface name. JavaScript doesn't support (or need) this, so simply remove `as` and the following identifier. A less-common cast syntax is to use a type name in brackets, `<LikeThis>`; these casts, too, must be removed.

Finally, TypeScript supports the access modifiers `public`, `protected`, and `private` for members of classes. All class members in JavaScript are public. Simply remove these modifiers wherever you see them.

Knowing how to identify and remove these TypeScript features goes a long way toward adapting short TypeScript snippets to JavaScript. But it may be impractical to convert longer TypeScript examples in this fashion, since they are more likely to use other TypeScript features. For these situations, we recommend [Sucrase](). Sucrase won't complain if code uses an undefined variable, for example, as `tsc` would. If it is syntactically valid, then with few exceptions, Sucrase can translate it to JavaScript. This makes it particularly valuable for converting snippets that may not be runnable on their own.

Migrating to TypeScript

Many JavaScript developers move to [TypeScript]() as their projects get larger and more complex. TypeScript is a superset of JavaScript—all JavaScript code is valid TypeScript code, so no changes to your code are required—and it is also a supported AWS CDK language. Type annotations and other TypeScript features are optional and can be added to your AWS CDK app as you find value in them. TypeScript also gives you early access to new JavaScript features, such as optional chaining and nullish coalescing, before they're finalized—and without requiring that you upgrade Node.js.

TypeScript's "shape-based" interfaces, which define bundles of required and optional properties (and their types) within an object, allow common mistakes to be caught while you're writing the code, and make it easier for your IDE to provide robust autocomplete and other real-time coding advice.

Coding in TypeScript does involve an additional step: compiling your app with the TypeScript compiler, `tsc`. For typical AWS CDK apps, compilation requires a few seconds at most.

The easiest way to migrate an existing JavaScript AWS CDK app to TypeScript is to create a new TypeScript project using `cdk init app --language typescript`, then copy your source files (and any other necessary files, such as assets like AWS Lambda function source code) to the new project. Rename your JavaScript files to end in `.ts` and begin developing in TypeScript.

Working with the AWS CDK in Python

Python is a fully-supported client language for the AWS Cloud Development Kit (AWS CDK) and is considered stable. Working with the AWS CDK in Python uses familiar tools, including the standard Python implementation (CPython), virtual environments with `virtualenv`, and the Python package installer `pip`. The modules comprising the AWS Construct Library are distributed via [pypi.org](). The Python version of the AWS CDK even uses Python-style identifiers (for example, `snake_case` method names).

You can use any editor or IDE. Many AWS CDK developers use [Visual Studio Code]() (or its open-source equivalent [VSCodium]()), which has good support for Python via an [official extension](). The

IDLE editor included with Python will suffice to get started. The Python modules for the AWS CDK do have type hints, which are useful for a linting tool or an IDE that supports type validation.

Get started with Python

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See Getting started with the AWS CDK.

Python AWS CDK applications require Python 3.6 or later. If you don't already have it installed, download a compatible version for your operating system at python.org. If you run Linux, your system may have come with a compatible version, or you may install it using your distro's package manager (`yum`, `apt`, etc.). Mac users may be interested in Homebrew, a Linux-style package manager for macOS.

> **Note**
>
> Third-party language deprecation: language version is only supported until its EOL (End Of Life) shared by the vendor or community and is subject to change with prior notice.

The Python package installer, `pip`, and virtual environment manager, `virtualenv`, are also required. Windows installations of compatible Python versions include these tools. On Linux, `pip` and `virtualenv` may be provided as separate packages in your package manager. Alternatively, you may install them with the following commands:

```
python -m ensurepip --upgrade
python -m pip install --upgrade pip
python -m pip install --upgrade virtualenv
```

If you encounter a permission error, run the above commands with the `--user` flag so that the modules are installed in your user directory, or use `sudo` to obtain the permissions to install the modules system-wide.

> **Note**
>
> It is common for Linux distros to use the executable name `python3` for Python 3.x, and have `python` refer to a Python 2.x installation. Some distros have an optional package you can install that makes the `python` command refer to Python 3. Failing that, you can

adjust the command used to run your application by editing `cdk.json` in the project's main directory.

> **Note**
>
> On Windows, you may want to invoke Python (and `pip`) using the `py` executable, the [Python launcher for Windows](). Among other things, the launcher allows you to easily specify which installed version of Python you want to use.
> If typing `python` at the command line results in a message about installing Python from the Windows Store, even after installing a Windows version of Python, open Windows' Manage App Execution Aliases settings panel and turn off the two App Installer entries for Python.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory. Use the `--language` option and specify `python`:

```
$ mkdir my-project
$ cd my-project
$ cdk init app --language python
```

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Hyphens in the folder name are converted to underscores. However, the name should otherwise follow the form of a Python identifier; for example, it should not start with a number or contain spaces.

To work with the new project, activate its virtual environment. This allows the project's dependencies to be installed locally in the project folder, instead of globally.

```
$ source .venv/bin/activate
```

> **Note**
>
> You may recognize this as the Mac/Linux command to activate a virtual environment. The Python templates include a batch file, `source.bat`, that allows the same command to

be used on Windows. The traditional Windows command, `.\venv\Scripts\activate`, works, too.

If you initialized your AWS CDK project using CDK Toolkit v1.70.0 or earlier, your virtual environment is in the `.env` directory instead of `.venv`.

Important

Activate the project's virtual environment whenever you start working on it. Otherwise, you won't have access to the modules installed there, and modules you install will go in the Python global module directory (or will result in a permission error).

After activating your virtual environment for the first time, install the app's standard dependencies:

```
$ python -m pip install -r requirements.txt
```

Managing AWS Construct Library modules

Use the Python package installer, `pip`, to install and update AWS Construct Library modules for use by your apps, as well as other packages you need. `pip` also installs the dependencies for those modules automatically. If your system does not recognize `pip` as a standalone command, invoke `pip` as a Python module, like this:

```
$ python -m pip <PIP-COMMAND>
```

Most AWS CDK constructs are in `aws-cdk-lib`. Experimental modules are in separate modules named like `aws-cdk.<SERVICE-NAME>.alpha`. The service name includes an *aws* prefix. If you're unsure of a module's name, [search for it at PyPI](). For example, the command below installs the AWS CodeStar library.

```
$ python -m pip install aws-cdk.aws-codestar-alpha
```

Some services' constructs are in more than one namespace. For example, besides `aws-cdk.aws-route53`, there are three additional Amazon Route 53 namespaces, named `aws-route53-targets`, `aws-route53-patterns`, and `aws-route53resolver`.

> **Note**
>
> The Python edition of the CDK API Reference also shows the package names.

The names used for importing AWS Construct Library modules into your Python code look like the following.

```
import aws_cdk.aws_s3 as s3
import aws_cdk.aws_lambda as lambda_
```

We recommend the following practices when importing AWS CDK classes and AWS Construct Library modules in your applications. Following these guidelines will help make your code consistent with other AWS CDK applications as well as easier to understand.

- Generally, import individual classes from top-level `aws_cdk`.

  ```
  from aws_cdk import App, Construct
  ```

- If you need many classes from the `aws_cdk`, you may use a namespace alias of `cdk` instead of importing individual classes. Avoid doing both.

  ```
  import aws_cdk as cdk
  ```

- Generally, import AWS Construct Libraries using short namespace aliases.

  ```
  import aws_cdk.aws_s3 as s3
  ```

After installing a module, update your project's `requirements.txt` file, which lists your project's dependencies. It is best to do this manually rather than using `pip freeze`. `pip freeze` captures the current versions of all modules installed in your Python virtual environment, which can be useful when bundling up a project to be run elsewhere.

Usually, though, your `requirements.txt` should list only top-level dependencies (modules that your app depends on directly) and not the dependencies of those libraries. This strategy makes updating your dependencies simpler.

You can edit `requirements.txt` to allow upgrades; simply replace the == preceding a version number with ~= to allow upgrades to a higher compatible version, or remove the version requirement entirely to specify the latest available version of the module.

With `requirements.txt` edited appropriately to allow upgrades, issue this command to upgrade your project's installed modules at any time:

```
$ pip install --upgrade -r requirements.txt
```

Managing dependencies in Python

In Python, you specify dependencies by putting them in `requirements.txt` for applications or `setup.py` for construct libraries. Dependencies are then managed with the PIP tool. PIP is invoked in one of the following ways:

```
pip <command options>
python -m pip <command options>
```

The `python -m pip` invocation works on most systems; `pip` requires that PIP's executable be on the system path. If `pip` doesn't work, try replacing it with `python -m pip`.

The `cdk init --language python` command creates a virtual environment for your new project. This lets each project have its own versions of dependencies, and also a basic `requirements.txt` file. You must activate this virtual environment by running `source .venv/bin/activate` each time you begin working with the project. On Windows, run `.\venv\Scripts\activate` instead

CDK applications

The following is an example `requirements.txt` file. Because PIP does not have a dependency-locking feature, we recommend that you use the == operator to specify exact versions for all dependencies, as shown here.

```
aws-cdk-lib==2.14.0
aws-cdk.aws-appsync-alpha==2.10.0a0
```

Installing a module with `pip install` does not automatically add it to `requirements.txt`. You must do that yourself. If you want to upgrade to a later version of a dependency, edit its version number in `requirements.txt`.

To install or update your project's dependencies after creating or editing `requirements.txt`, run the following:

```
python -m pip install -r requirements.txt
```

> **Tip**
>
> The `pip freeze` command outputs the versions of all installed dependencies in a format that can be written to a text file. This can be used as a requirements file with `pip install -r`. This file is convenient for pinning all dependencies (including transitive ones) to the exact versions that you tested with. To avoid problems when upgrading packages later, use a separate file for this, such as `freeze.txt` (not `requirements.txt`). Then, regenerate it when you upgrade your project's dependencies.

Third-party construct libraries

In libraries, dependencies are specified in `setup.py`, so that transitive dependencies are automatically downloaded when the package is consumed by an application. Otherwise, every application that wants to use your package needs to copy your dependencies into their `requirements.txt`. An example `setup.py` is shown here.

```
from setuptools import setup

setup(
  name='my-package',
  version='0.0.1',
  install_requires=[
    'aws-cdk-lib==2.14.0',
  ],
  ...
)
```

To work on the package for development, create or activate a virtual environment, then run the following command.

```
python -m pip install -e .
```

Although PIP automatically installs transitive dependencies, there can only be one installed copy of any one package. The version that is specified highest in the dependency tree is selected; applications always have the last word in what version of packages get installed.

AWS CDK idioms in Python

Language conflicts

In Python, `lambda` is a language keyword, so you cannot use it as a name for the AWS Lambda construct library module or Lambda functions. The Python convention for such conflicts is to use a trailing underscore, as in `lambda_`, in the variable name.

By convention, the second argument to AWS CDK constructs is named `id`. When writing your own stacks and constructs, calling a parameter `id` "shadows" the Python built-in function `id()`, which returns an object's unique identifier. This function isn't used very often, but if you should happen to need it in your construct, rename the argument, for example `construct_id`.

Arguments and properties

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

scope and *id* should always be passed as positional arguments, not keyword arguments, because their names change if the construct accepts a property named *scope* or *id*.

In Python, props are expressed as keyword arguments. If an argument contains nested data structures, these are expressed using a class which takes its own keyword arguments at instantiation. The same pattern is applied to other method calls that take a structured argument.

For example, in a Amazon S3 bucket's `add_lifecycle_rule` method, the `transitions` property is a list of `Transition` instances.

```
bucket.add_lifecycle_rule(
  transitions=[
    Transition(
      storage_class=StorageClass.GLACIER,
      transition_after=Duration.days(10)
    )
```

```
  ]
)
```

When extending a class or overriding a method, you may want to accept additional arguments for your own purposes that are not understood by the parent class. In this case you should accept the arguments you don't care about using the **kwargs idiom, and use keyword-only arguments to accept the arguments you're interested in. When calling the parent's constructor or the overridden method, pass only the arguments it is expecting (often just **kwargs). Passing arguments that the parent class or method doesn't expect results in an error.

```
class MyConstruct(Construct):
    def __init__(self, id, *, MyProperty=42, **kwargs):
        super().__init__(self, id, **kwargs)
        # ...
```

A future release of the AWS CDK could coincidentally add a new property with a name you used for your own property. This won't cause any technical issues for users of your construct or method (since your property isn't passed "up the chain," the parent class or overridden method will simply use a default value) but it may cause confusion. You can avoid this potential problem by naming your properties so they clearly belong to your construct. If there are many new properties, bundle them into an appropriately-named class and pass it as a single keyword argument.

Missing values

The AWS CDK uses `None` to represent missing or undefined values. When working with **kwargs, use the dictionary's `get()` method to provide a default value if a property is not provided. Avoid using `kwargs[...]` , as this raises `KeyError` for missing values.

```
encrypted = kwargs.get("encrypted")         # None if no property "encrypted" exists
encrypted = kwargs.get("encrypted", False)  # specify default of False if property is
 missing
```

Some AWS CDK methods (such as `tryGetContext()` to get a runtime context value) may return `None`, which you will need to check explicitly.

Using interfaces

Python doesn't have an interface feature as some other languages do, though it does have abstract base classes, which are similar. (If you're not familiar with interfaces, Wikipedia has a good introduction.) TypeScript, the language in which the AWS CDK is implemented, does provide interfaces, and constructs and other AWS CDK objects often require an object that adheres to a particular interface, rather than inheriting from a particular class. So the AWS CDK provides its own interface feature as part of the JSII layer.

To indicate that a class implements a particular interface, you can use the `@jsii.implements` decorator:

```
from aws_cdk import IAspect, IConstruct
import jsii

@jsii.implements(IAspect)
class MyAspect():
    def visit(self, node: IConstruct) -> None:
        print("Visited", node.node.path)
```

Type pitfalls

Python uses dynamic typing, where all variables may refer to a value of any type. Parameters and return values may be annotated with types, but these are "hints" and are not enforced. This means that in Python, it is easy to pass the incorrect type of value to a AWS CDK construct. Instead of getting a type error during build, as you would from a statically-typed language, you may instead get a runtime error when the JSII layer (which translates between Python and the AWS CDK's TypeScript core) is unable to deal with the unexpected type.

In our experience, the type errors Python programmers make tend to fall into these categories.

- Passing a single value where a construct expects a container (Python list or dictionary) or vice versa.
- Passing a value of a type associated with a layer 1 (`CfnXxxxxx`) construct to a L2 or L3 construct, or vice versa.

The AWS CDK Python modules do include type annotations, so you can use tools that support them to help with types. If you are not using an IDE that supports these, such as PyCharm, you might want to call the MyPy type validator as a step in your build process. There are also runtime type checkers that can improve error messages for type-related errors.

Working with the AWS CDK in Java

Java is a fully-supported client language for the AWS CDK and is considered stable. You can develop AWS CDK applications in Java using familiar tools, including the JDK (Oracle's, or an OpenJDK distribution such as Amazon Corretto) and Apache Maven.

The AWS CDK supports Java 8 and later. We recommend using the latest version you can, however, because later versions of the language include improvements that are particularly convenient for developing AWS CDK applications. For example, Java 9 introduces the `Map.of()` method (a convenient way to declare hashmaps that would be written as object literals in TypeScript). Java 10 introduces local type inference using the `var` keyword.

> **Note**
>
> Most code examples in this Developer Guide work with Java 8. A few examples use `Map.of()`; these examples include comments noting that they require Java 9.

You can use any text editor, or a Java IDE that can read Maven projects, to work on your AWS CDK apps. We provide Eclipse hints in this Guide, but IntelliJ IDEA, NetBeans, and other IDEs can import Maven projects and can be used for developing AWS CDK applications in Java.

It is possible to write AWS CDK applications in JVM-hosted languages other than Java (for example, Kotlin, Groovy, Clojure, or Scala), but the experience may not be particularly idiomatic, and we are unable to provide any support for these languages.

Get started with Java

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See Getting started with the AWS CDK.

Java AWS CDK applications require Java 8 (v1.8) or later. We recommend Amazon Corretto, but you can use any OpenJDK distribution or Oracle's JDK. You will also need Apache Maven 3.5 or later. You can also use tools such as Gradle, but the application skeletons generated by the AWS CDK Toolkit are Maven projects.

Note

Third-party language deprecation: language version is only supported until its EOL (End Of Life) shared by the vendor or community and is subject to change with prior notice.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory. Use the `--language` option and specify `java`:

```
$ mkdir my-project
$ cd my-project
$ cdk init app --language java
```

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Hyphens in the folder name are converted to underscores. However, the name should otherwise follow the form of a Java identifier; for example, it should not start with a number or contain spaces.

The resulting project includes a reference to the `software.amazon.awscdk` Maven package. It and its dependencies are automatically installed by Maven.

If you are using an IDE, you can now open or import the project. In Eclipse, for example, choose **File** > **Import** > **Maven** > **Existing Maven Projects**. Make sure that the project settings are set to use Java 8 (1.8).

Managing AWS Construct Library modules

Use Maven to install AWS Construct Library packages, which are in the group `software.amazon.awscdk`. Most constructs are in the artifact `aws-cdk-lib`, which is added to new Java projects by default. Modules for services whose higher-level CDK support is still being developed are in separate "experimental" packages, named with a short version (no AWS or Amazon prefix) of their service's name. Search the Maven Central Repository to find the names of all AWS CDK and AWS Construct Module libraries.

Note

The Java edition of the CDK API Reference also shows the package names.

Some services' AWS Construct Library support is in more than one namespace. For example, Amazon Route 53 has its functionality divided into `software.amazon.awscdk.route53`, `route53-patterns`, `route53resolver`, and `route53-targets`.

The main AWS CDK package is imported in Java code as `software.amazon.awscdk`. Modules for the various services in the AWS Construct Library live under `software.amazon.awscdk.services` and are named similarly to their Maven package name. For example, the Amazon S3 module's namespace is `software.amazon.awscdk.services.s3`.

We recommend writing a separate Java `import` statement for each AWS Construct Library class you use in each of your Java source files, and avoiding wildcard imports. You can always use a type's fully-qualified name (including its namespace) without an `import` statement.

If your application depends on an experimental package, edit your project's `pom.xml` and add a new `<dependency>` element in the `<dependencies>` container. For example, the following `<dependency>` element specifies the CodeStar experimental construct library module:

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>codestar-alpha</artifactId>
    <version>2.0.0-alpha.10</version>
</dependency>
```

Tip

If you use a Java IDE, it probably has features for managing Maven dependencies. We recommend editing `pom.xml` directly, however, unless you are absolutely sure the IDE's functionality matches what you'd do by hand.

Managing dependencies in Java

In Java, dependencies are specified in `pom.xml` and installed using Maven. The `<dependencies>` container includes a `<dependency>` element for each package. Following is a section of `pom.xml` for a typical CDK Java app.

```
<dependencies>
    <dependency>
        <groupId>software.amazon.awscdk</groupId>
        <artifactId>aws-cdk-lib</artifactId>
```

```
        <version>2.14.0</version>
    </dependency>
    <dependency>
        <groupId>software.amazon.awscdk</groupId>
        <artifactId>appsync-alpha</artifactId>
        <version>2.10.0-alpha.0</version>
    </dependency>
</dependencies>
```

Tip

Many Java IDEs have integrated Maven support and visual `pom.xml` editors, which you may find convenient for managing dependencies.

Maven does not support dependency locking. Although it's possible to specify version ranges in `pom.xml`, we recommend you always use exact versions to keep your builds repeatable.

Maven automatically installs transitive dependencies, but there can only be one installed copy of each package. The version that is specified highest in the POM tree is selected; applications always have the last word in what version of packages get installed.

Maven automatically installs or updates your dependencies whenever you build (`mvn compile`) or package (`mvn package`) your project. The CDK Toolkit does this automatically every time you run it, so generally there is no need to manually invoke Maven.

AWS CDK idioms in Java

Props

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

In Java, props are expressed using the Builder pattern. Each construct type has a corresponding props type; for example, the `Bucket` construct (which represents an Amazon S3 bucket) takes as its props an instance of `BucketProps`.

The `BucketProps` class (like every AWS Construct Library props class) has an inner class called `Builder`. The `BucketProps.Builder` type offers methods to set the various properties of a

BucketProps instance. Each method returns the Builder instance, so the method calls can be chained to set multiple properties. At the end of the chain, you call build() to actually produce the BucketProps object.

```
Bucket bucket = new Bucket(this, "amzn-s3-demo-bucket", new BucketProps.Builder()
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build());
```

Constructs, and other classes that take a props-like object as their final argument, offer a shortcut. The class has a Builder of its own that instantiates it and its props object in one step. This way, you don't need to explicitly instantiate (for example) both BucketProps and a Bucket--and you don't need an import for the props type.

```
Bucket bucket = Bucket.Builder.create(this, "amzn-s3-demo-bucket")
                           .versioned(true)
                           .encryption(BucketEncryption.KMS_MANAGED)
                           .build();
```

When deriving your own construct from an existing construct, you may want to accept additional properties. We recommend that you follow these builder patterns. However, this isn't as simple as subclassing a construct class. You must provide the moving parts of the two new Builder classes yourself. You may prefer to simply have your construct accept one or more additional arguments. You should provide additional constructors when an argument is optional.

Generic structures

In some APIs, the AWS CDK uses JavaScript arrays or untyped objects as input to a method. (See, for example, AWS CodeBuild's BuildSpec.fromObject() method.) In Java, these objects are represented as java.util.Map<String, Object>. In cases where the values are all strings, you can use Map<String, String>.

Java does not provide a way to write literals for such containers like some other languages do. In Java 9 and later, you can use java.util.Map.of() to conveniently define maps of up to ten entries inline with one of these calls.

```
java.util.Map.of(
    "base-directory", "dist",
    "files", "LambdaStack.template.json"
```

```
)
```

To create maps with more than ten entries, use `java.util.Map.ofEntries()`.

If you are using Java 8, you could provide your own methods similar to to these.

JavaScript arrays are represented as `List<Object>` or `List<String>` in Java. The method `java.util.Arrays.asList` is convenient for defining short `List`s.

```
List<String> cmds = Arrays.asList("cd lambda", "npm install", "npm install typescript")
```

Missing values

In Java, missing values in AWS CDK objects such as props are represented by `null`. You must explicitly test any value that could be `null` to make sure it contains a value before doing anything with it. Java does not have "syntactic sugar" to help handle null values as some other languages do. You may find Apache ObjectUtil's `defaultIfNull` and `firstNonNull` useful in some situations. Alternatively, write your own static helper methods to make it easier to handle potentially null values and make your code more readable.

Build and run CDK applications

The AWS CDK automatically compiles your app before running it. However, it can be useful to build your app manually to check for errors and to run tests. You can do this in your IDE (for example, press Control-B in Eclipse) or by issuing `mvn compile` at a command prompt while in your project's root directory.

Run any tests you've written by running `mvn test` at a command prompt.

Working with the AWS CDK in C#

.NET is a fully-supported client language for the AWS CDK and is considered stable. C# is the main .NET language for which we provide examples and support. You can choose to write AWS CDK applications in other .NET languages, such as Visual Basic or F#, but AWS offers limited support for using these languages with the CDK.

You can develop AWS CDK applications in C# using familiar tools including Visual Studio, Visual Studio Code, the `dotnet` command, and the NuGet package manager. The modules comprising the AWS Construct Library are distributed via nuget.org.

We suggest using Visual Studio 2019 (any edition) on Windows to develop AWS CDK apps in C#.

Get started with C#

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See Getting started with the AWS CDK.

C# AWS CDK applications require .NET Core v3.1 or later, available here.

The .NET toolchain includes `dotnet`, a command-line tool for building and running .NET applications and managing NuGet packages. Even if you work mainly in Visual Studio, this command can be useful for batch operations and for installing AWS Construct Library packages.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory. Use the `--language` option and specify `csharp`:

```
mkdir my-project
cd my-project
cdk init app --language csharp
```

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Hyphens in the folder name are converted to underscores. However, the name should otherwise follow the form of a C# identifier; for example, it should not start with a number or contain spaces.

The resulting project includes a reference to the `Amazon.CDK.Lib` NuGet package. It and its dependencies are installed automatically by NuGet.

Managing AWS Construct Library modules

The .NET ecosystem uses the NuGet package manager. The main CDK package, which contains the core classes and all stable service constructs, is `Amazon.CDK.Lib`. Experimental modules, where new functionality is under active development, are named like `Amazon.CDK.AWS.<SERVICE-NAME>.Alpha`, where the service name is a short name without an AWS or Amazon prefix. For example, the NuGet package name for the AWS IoT module is `Amazon.CDK.AWS.IoT.Alpha`. If you can't find a package you want, search Nuget.org.

> **Note**
>
> The .NET edition of the CDK API Reference also shows the package names.

Some services' AWS Construct Library support is in more than one module. For example, AWS IoT has a second module named `Amazon.CDK.AWS.IoT.Actions.Alpha`.

The AWS CDK's main module, which you'll need in most AWS CDK apps, is imported in C# code as `Amazon.CDK`. Modules for the various services in the AWS Construct Library live under `Amazon.CDK.AWS` . For example, the Amazon S3 module's namespace is `Amazon.CDK.AWS.S3`.

We recommend writing C# `using` directives for the CDK core constructs and for each AWS service you use in each of your C# source files. You may find it convenient to use an alias for a namespace or type to help resolve name conflicts. You can always use a type's fully-qualfiied name (including its namespace) without a `using` statement.

Managing dependencies in C#

In C# AWS CDK apps, you manage dependencies using NuGet. NuGet has four standard, mostly equivalent interfaces. Use the one that suits your needs and working style. You can also use compatible tools, such as Paket or MyGet or even edit the `.csproj` file directly.

NuGet does not let you specify version ranges for dependencies. Every dependency is pinned to a specific version.

After updating your dependencies, Visual Studio will use NuGet to retrieve the specified versions of each package the next time you build. If you are not using Visual Studio, use the `dotnet restore` command to update your dependencies.

Editing the project file directly

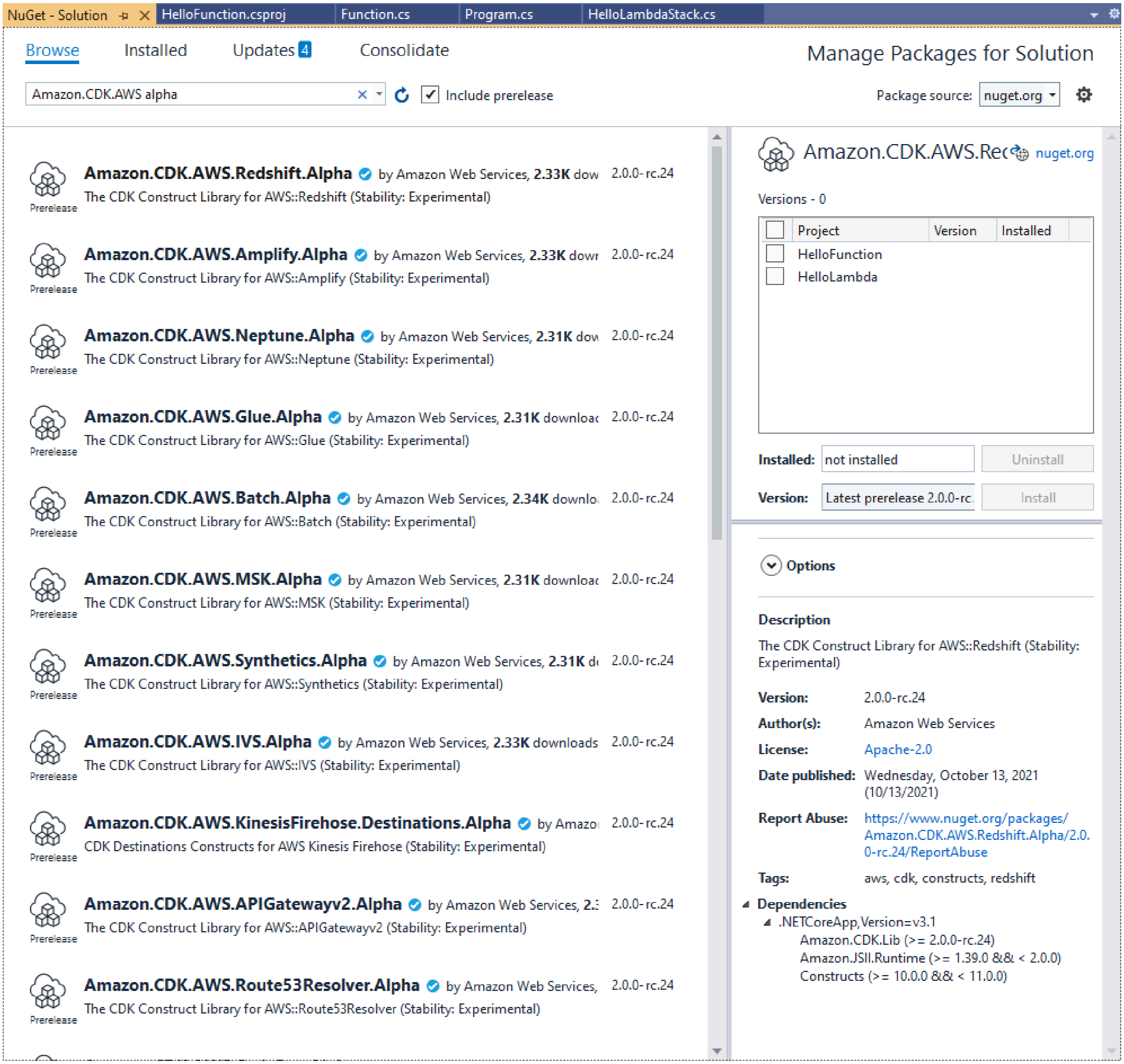
Your project's `.csproj` file contains an `<ItemGroup>` container that lists your dependencies as `<PackageReference` elements.

```
<ItemGroup>
    <PackageReference Include="Amazon.CDK.Lib" Version="2.14.0" />
    <PackageReference Include="Constructs" Version="%constructs-version%" />
</ItemGroup>
```

The Visual Studio NuGet GUI

Visual Studio’s NuGet tools are accessible from **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**. Use the **Browse** tab to find the AWS Construct Library packages you want to install. You can choose the desired version, including prerelease versions of your modules, and add them to any of the open projects.

> **Note**
>
> All AWS Construct Library modules deemed "experimental" (see AWS CDK versioning) are flagged as prerelease in NuGet and have an `alpha` name suffix.



Look on the **Updates** page to install new versions of your packages.

The NuGet console

The NuGet console is a PowerShell-based interface to NuGet that works in the context of a Visual Studio project. You can open it in Visual Studio by choosing **Tools** > **NuGet Package Manager** > **Package Manager Console**. For more information about using this tool, see Install and Manage Packages with the Package Manager Console in Visual Studio.

The `dotnet` command

The `dotnet` command is the primary command line tool for working with Visual Studio C# projects. You can invoke it from any Windows command prompt. Among its many capabilities, `dotnet` can add NuGet dependencies to a Visual Studio project.

Assuming you're in the same directory as the Visual Studio project (`.csproj`) file, issue a command like the following to install a package. Because the main CDK library is included when you create a project, you only need to explicitly install experimental modules. Experimental modules require you to specify an explicit version number.

```
dotnet add package Amazon.CDK.AWS.IoT.Alpha -v <VERSION-NUMBER>
```

You can issue the command from another directory. To do so, include the path to the project file, or to the directory that contains it, after the `add` keyword. The following example assumes that you are in your AWS CDK project's main directory.

```
dotnet add src/<PROJECT-DIR> package Amazon.CDK.AWS.IoT.Alpha -v <VERSION-NUMBER>
```

To install a specific version of a package, include the `-v` flag and the desired version.

To update a package, issue the same `dotnet add` command you used to install it. For experimental modules, again, you must specify an explicit version number.

For more information about managing packages using the `dotnet` command, see [Install and Manage Packages Using the dotnet CLI].

The `nuget` command

The `nuget` command line tool can install and update NuGet packages. However, it requires your Visual Studio project to be set up differently from the way `cdk init` sets up projects. (Technical details: `nuget` works with `Packages.config` projects, while `cdk init` creates a newer-style `PackageReference` project.)

We do not recommend the use of the `nuget` tool with AWS CDK projects created by `cdk init`. If you are using another type of project, and want to use `nuget`, see the [NuGet CLI Reference].

AWS CDK idioms in C#

Props

All AWS Construct Library classes are instantiated using three arguments: the *scope* in which the construct is being defined (its parent in the construct tree), an *id*, and *props*, a bundle of key/value pairs that the construct uses to configure the resources it creates. Other classes and methods also use the "bundle of attributes" pattern for arguments.

In C#, props are expressed using a props type. In idiomatic C# fashion, we can use an object initializer to set the various properties. Here we're creating an Amazon S3 bucket using the `Bucket` construct; its corresponding props type is `BucketProps`.

```
var bucket = new Bucket(this, "amzn-s3-demo-bucket", new BucketProps {
    Versioned = true
});
```

> **Tip**
>
> Add the package `Amazon.JSII.Analyzers` to your project to get required-values checking in your props definitions inside Visual Studio.

When extending a class or overriding a method, you may want to accept additional props for your own purposes that are not understood by the parent class. To do this, subclass the appropriate props type and add the new attributes.

```
// extend BucketProps for use with MimeBucket
class MimeBucketProps : BucketProps {
    public string MimeType { get; set; }
}

// hypothetical bucket that enforces MIME type of objects inside it
class MimeBucket : Bucket {
     public MimeBucket( readonly Construct scope, readonly string id, readonly
 MimeBucketProps props=null) : base(scope, id, props) {
         // ...
     }
}
```

```
// instantiate our MimeBucket class
var bucket = new MimeBucket(this, "amzn-s3-demo-bucket", new MimeBucketProps {
    Versioned = true,
    MimeType = "image/jpeg"
});
```

When calling the parent class's initializer or overridden method, you can generally pass the props you received. The new type is compatible with its parent, and extra props you added are ignored.

A future release of the AWS CDK could coincidentally add a new property with a name you used for your own property. This won't cause any technical issues using your construct or method (since your property isn't passed "up the chain," the parent class or overridden method will simply use a default value) but it may cause confusion for your construct's users. You can avoid this potential problem by naming your properties so they clearly belong to your construct. If there are many new properties, bundle them into an appropriately-named class and pass them as a single property.

Generic structures

In some APIs, the AWS CDK uses JavaScript arrays or untyped objects as input to a method. (See, for example, AWS CodeBuild's `BuildSpec.fromObject()` method.) In C#, these objects are represented as `System.Collections.Generic.Dictionary<String, Object>`. In cases where the values are all strings, you can use `Dictionary<String, String>`. JavaScript arrays are represented as `object[]` or `string[]` array types in C#.

> **Tip**
>
> You might define short aliases to make it easier to work with these specific dictionary types.
>
> ```
> using StringDict = System.Collections.Generic.Dictionary<string, string>;
> using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
> ```

Missing values

In C#, missing values in AWS CDK objects such as props are represented by `null`. The null-conditional member access operator `?.` and the null coalescing operator `??` are convenient for working with these values.

```
// mimeType is null if props is null or if props.MimeType is null
string mimeType = props?.MimeType;

// mimeType defaults to text/plain. either props or props.MimeType can be null
string MimeType = props?.MimeType ?? "text/plain";
```

Build and run CDK applications

The AWS CDK automatically compiles your app before running it. However, it can be useful to build your app manually to check for errors and run tests. You can do this by pressing F6 in Visual Studio or by issuing `dotnet build src` from the command line, where `src` is the directory in your project directory that contains the Visual Studio Solution (`.sln`) file.

Working with the AWS CDK in Go

Go is a fully-supported client language for the AWS Cloud Development Kit (AWS CDK) and is considered stable. Working with the AWS CDK in Go uses familiar tools. The Go version of the AWS CDK even uses Go-style identifiers.

Unlike the other languages the CDK supports, Go is not a traditional object-oriented programming language. Go uses composition where other languages often leverage inheritance. We have tried to employ idiomatic Go approaches as much as possible, but there are places where the CDK may differ.

This topic provides guidance when working with the AWS CDK in Go. See the announcement blog post for a walkthrough of a simple Go project for the AWS CDK.

Get started with Go

To work with the AWS CDK, you must have an AWS account and credentials and have installed Node.js and the AWS CDK Toolkit. See Getting started with the AWS CDK.

The Go bindings for the AWS CDK use the standard Go toolchain, v1.18 or later. You can use the editor of your choice.

> **Note**
>
> Third-party language deprecation: language version is only supported until its EOL (End Of Life) shared by the vendor or community and is subject to change with prior notice.

Creating a project

You create a new AWS CDK project by invoking `cdk init` in an empty directory. Use the `--language` option and specify go:

```
mkdir my-project
cd my-project
cdk init app --language go
```

`cdk init` uses the name of the project folder to name various elements of the project, including classes, subfolders, and files. Hyphens in the folder name are converted to underscores. However, the name should otherwise follow the form of a Go identifier; for example, it should not start with a number or contain spaces.

The resulting project includes a reference to the core AWS CDK Go module, `github.com/aws/aws-cdk-go/awscdk/v2`, in `go.mod`. Issue `go get` to install this and other required modules.

Managing AWS Construct Library modules

In most AWS CDK documentation and examples, the word "module" is often used to refer to AWS Construct Library modules, one or more per AWS service, which differs from idiomatic Go usage of the term. The CDK Construct Library is provided in one Go module with the individual Construct Library modules, which support the various AWS services, provided as Go packages within that module.

Some services' AWS Construct Library support is in more than one Construct Library module (Go package). For example, Amazon Route 53 has three Construct Library modules in addition to the main `awsroute53` package, named `awsroute53patterns`, `awsroute53resolver`, and `awsroute53targets`.

The AWS CDK's core package, which you'll need in most AWS CDK apps, is imported in Go code as `github.com/aws/aws-cdk-go/awscdk/v2`. Packages for the various services in the AWS Construct Library live under `github.com/aws/aws-cdk-go/awscdk/v2`. For example, the Amazon S3 module's namespace is `github.com/aws/aws-cdk-go/awscdk/v2/awss3`.

```
import (
        "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
        // ...
)
```

Once you have imported the Construct Library modules (Go packages) for the services you want to use in your app, you access constructs in that module using, for example, `awss3.Bucket`.

Managing dependencies in Go

In Go, dependencies versions are defined in `go.mod`. The default `go.mod` is similar to the one shown here.

```
module my-package

go 1.16

require (
  github.com/aws/aws-cdk-go/awscdk/v2 v2.16.0
  github.com/aws/constructs-go/constructs/v10 v10.0.5
  github.com/aws/jsii-runtime-go v1.29.0
)
```

Package names (modules, in Go parlance) are specified by URL with the required version number appended. Go's module system does not support version ranges.

Issue the `go get` command to install all required modules and update `go.mod`. To see a list of available updates for your dependencies, issue `go list -m -u all`.

AWS CDK idioms in Go

Field and method names

Field and method names use camel casing (`likeThis`) in TypeScript, the CDK's language of origin. In Go, these follow Go conventions, so are Pascal-cased (`LikeThis`).

Cleaning up

In your `main` method, use `defer jsii.Close()` to make sure your CDK app cleans up after itself.

Missing values and pointer conversion

In Go, missing values in AWS CDK objects such as property bundles are represented by `nil`. Go doesn't have nullable types; the only type that can contain `nil` is a pointer. To allow values to be optional, then, all CDK properties, arguments, and return values are pointers, even for primitive types. This applies to required values as well as optional ones, so if a required value later becomes optional, no breaking change in type is needed.

When passing literal values or expressions, use the following helper functions to create pointers to the values.

- `jsii.String`
- `jsii.Number`
- `jsii.Bool`
- `jsii.Time`

For consistency, we recommend that you use pointers similarly when defining your own constructs, even though it may seem more convenient to, for example, receive your construct's `id` as a string rather than a pointer to a string.

When dealing with optional AWS CDK values, including primitive values as well as complex types, you should explicitly test pointers to make sure they are not `nil` before doing anything with them. Go does not have "syntactic sugar" to help handle empty or missing values as some other languages do. However, required values in property bundles and similar structures are guaranteed to exist (construction fails otherwise), so these values need not be `nil`-checked.

Constructs and Props

Constructs, which represent one or more AWS resources and their associated attributes, are represented in Go as interfaces. For example, `awss3.Bucket` is an interface. Every construct has a factory function, such as `awss3.NewBucket`, to return a struct that implements the corresponding interface.

All factory functions take three arguments: the `scope` in which the construct is being defined (its parent in the construct tree), an `id`, and `props`, a bundle of key/value pairs that the construct uses to configure the resources it creates. The "bundle of attributes" pattern is also used elsewhere in the AWS CDK.

In Go, props are represented by a specific struct type for each construct. For example, an `awss3.Bucket` takes a props argument of type `awss3.BucketProps`. Use a struct literal to write props arguments.

```
var bucket = awss3.NewBucket(stack, jsii.String("amzn-s3-demo-bucket"),
 &awss3.BucketProps{
```

```
    Versioned: jsii.Bool(true),
})
```

Generic structures

In some places, the AWS CDK uses JavaScript arrays or untyped objects as input to a method. (See, for example, AWS CodeBuild's `BuildSpec.fromObject()` method.) In Go, these objects are represented as slices and an empty interface, respectively.

The CDK provides variadic helper functions such as `jsii.Strings` for building slices containing primitive types.

```
jsii.Strings("One", "Two", "Three")
```

Developing custom constructs

In Go, it is usually more straightforward to write a new construct than to extend an existing one. First, define a new struct type, anonymously embedding one or more existing types if extension-like semantics are desired. Write methods for any new functionality you're adding and the fields necessary to hold the data they need. Define a props interface if your construct needs one. Finally, write a factory function `NewMyConstruct()` to return an instance of your construct.

If you are simply changing some default values on an existing construct or adding a simple behavior at instantiation, you don't need all that plumbing. Instead, write a factory function that calls the factory function of the construct you're "extending." In other CDK languages, for example, you might create a `TypedBucket` construct that enforces the type of objects in an Amazon S3 bucket by overriding the `s3.Bucket` type and, in your new type's initializer, adding a bucket policy that allows only specified filename extensions to be added to the bucket. In Go, it is easier to simply write a `NewTypedBucket` that returns an `s3.Bucket` (instantiated using `s3.NewBucket`) to which you have added an appropriate bucket policy. No new construct type is necessary because the functionality is already available in the standard bucket construct; the new "construct" just provides a simpler way to configure it.

Building, synthesizing, and deploying

The AWS CDK automatically compiles your app before running it. However, it can be useful to build your app manually to check for errors and to run tests. You can do this by issuing `go build` at a command prompt while in your project's root directory.

Run any tests you've written by running `go test` at a command prompt.

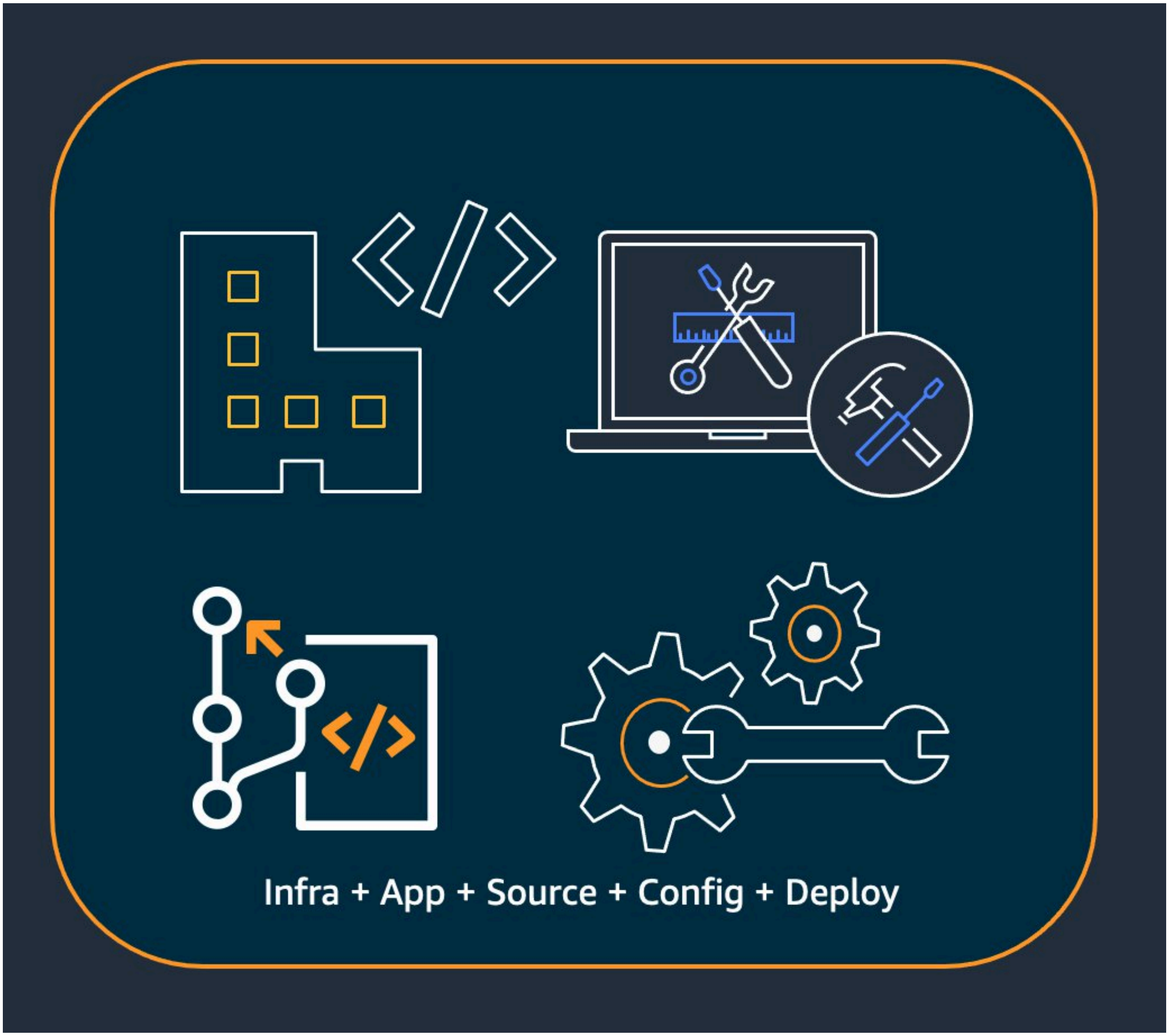
Best practices for developing and deploying cloud infrastructure with the AWS CDK

With the AWS CDK, developers or administrators can define their cloud infrastructure by using a supported programming language. CDK applications should be organized into logical units, such as API, database, and monitoring resources, and optionally have a pipeline for automated deployments. The logical units should be implemented as constructs including the following:

- Infrastructure (such as Amazon S3 buckets, Amazon RDS databases, or an Amazon VPC network)
- Runtime code (such as AWS Lambda functions)
- Configuration code

Stacks define the deployment model of these logical units. For a more detailed introduction to the concepts behind the CDK, see Getting started with the AWS CDK.

The AWS CDK reflects careful consideration of the needs of our customers and internal teams and of the failure patterns that often arise during the deployment and ongoing maintenance of complex cloud applications. We discovered that failures are often related to "out-of-band" changes to an application that aren't fully tested, such as configuration changes. Therefore, we developed the AWS CDK around a model in which your entire application is defined in code, not only business logic but also infrastructure and configuration. That way, proposed changes can be carefully reviewed, comprehensively tested in environments resembling production to varying degrees, and fully rolled back if something goes wrong.



At deployment time, the AWS CDK synthesizes a cloud assembly that contains the following:

- AWS CloudFormation templates that describe your infrastructure in all target environments
- File assets that contain your runtime code and their supporting files

With the CDK, every commit in your application's main version control branch can represent a complete, consistent, deployable version of your application. Your application can then be deployed automatically whenever a change is made.

The philosophy behind the AWS CDK leads to our recommended best practices, which we have divided into four broad categories.

- Organization best practices
- Coding best practices
- Construct best practices
- Application best practices

> **Tip**
>
> Also consider best practices for AWS CloudFormation and the individual AWS services that you use, where applicable to CDK-defined infrastructure.

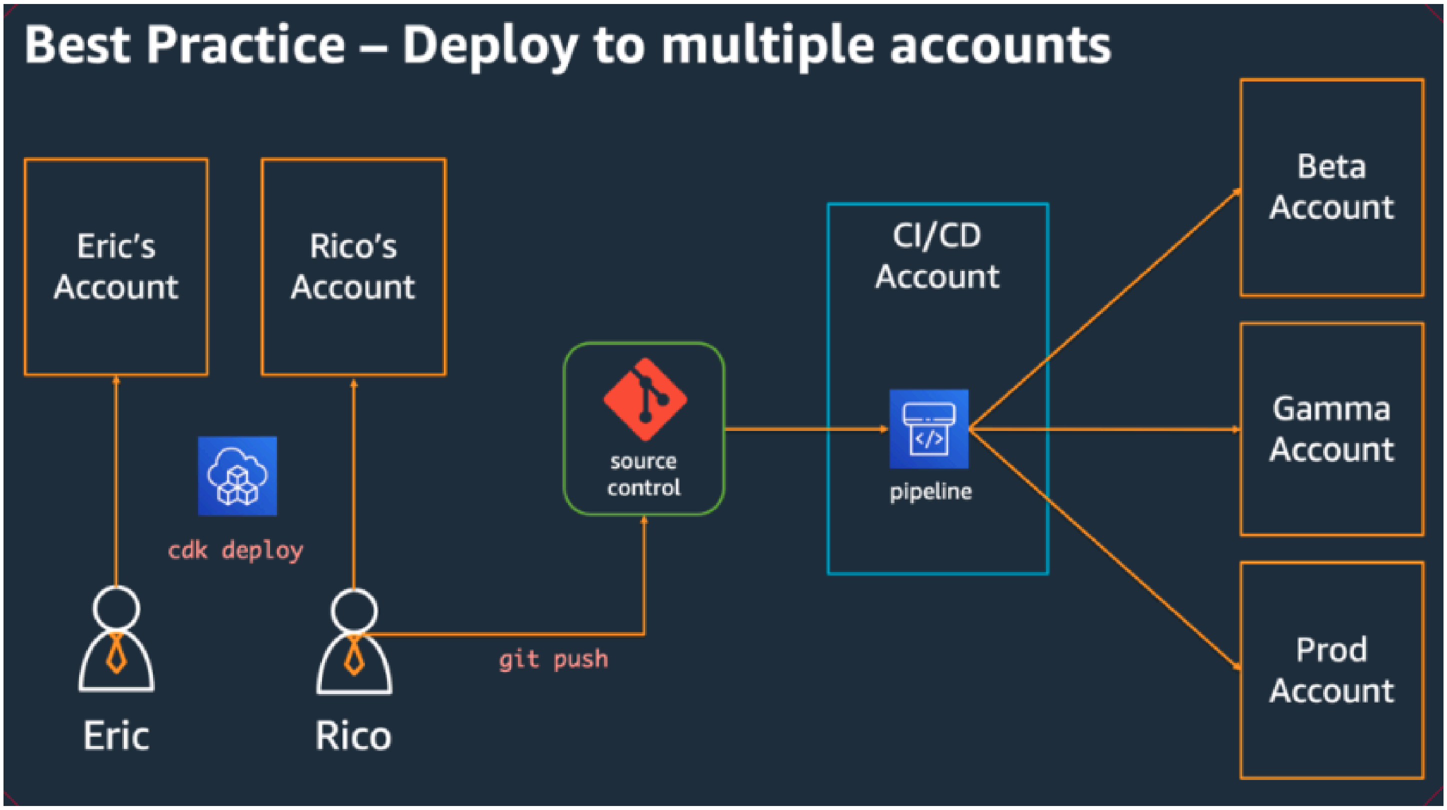
Organization best practices

In the beginning stages of AWS CDK adoption, it's important to consider how to set up your organization for success. It's a best practice to have a team of experts responsible for training and guiding the rest of the company as they adopt the CDK. The size of this team might vary, from one or two people at a small company to a full-fledged Cloud Center of Excellence (CCoE) at a larger company. This team is responsible for setting standards and policies for cloud infrastructure at your company, and also for training and mentoring developers.

The CCoE might provide guidance on what programming languages should be used for cloud infrastructure. Details will vary from one organization to the next, but a good policy helps make sure that developers can understand and maintain the company's cloud infrastructure.

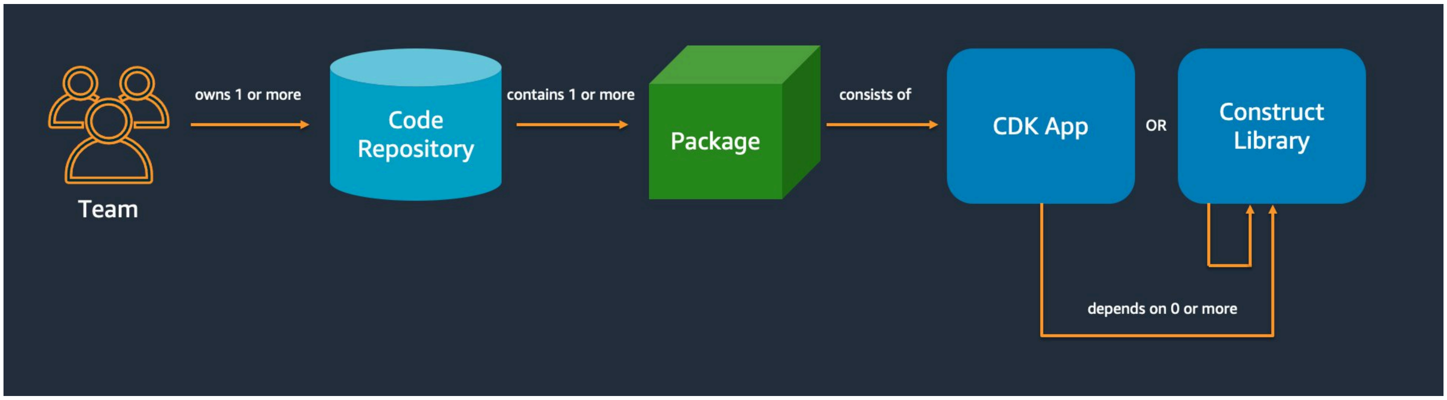
The CCoE also creates a "landing zone" that defines your organizational units within AWS. A landing zone is a pre-configured, secure, scalable, multi-account AWS environment based on best practice blueprints. To tie together the services that make up your landing zone, you can use AWS Control Tower, which configures and manages your entire multi-account system from a single user interface.

Development teams should be able to use their own accounts for testing and deploy new resources in these accounts as needed. Individual developers can treat these resources as extensions of their own development workstation. Using CDK Pipelines, the AWS CDK applications can then be deployed via a CI/CD account to testing, integration, and production environments (each isolated in its own AWS Region or account). This is done by merging the developers' code into your organization's canonical repository.



Coding best practices

This section presents best practices for organizing your AWS CDK code. The following diagram shows the relationship between a team and that team's code repositories, packages, applications, and construct libraries.



Start simple and add complexity only when you need it

The guiding principle for most of our best practices is to keep things simple as possible—but no simpler. Add complexity only when your requirements dictate a more complicated solution. With the AWS CDK, you can refactor your code as necessary to support new requirements. You don’t have to architect for all possible scenarios upfront.

Align with the AWS Well-Architected Framework

The AWS Well-Architected Framework defines a *component* as the code, configuration, and AWS resources that together deliver against a requirement. A component is often the unit of technical ownership, and is decoupled from other components. The term *workload* is used to identify a set of components that together deliver business value. A workload is usually the level of detail that business and technology leaders communicate about.

An AWS CDK application maps to a component as defined by the AWS Well-Architected Framework. AWS CDK apps are a mechanism to codify and deliver Well-Architected cloud application best practices. You can also create and share components as reusable code libraries through artifact repositories, such as AWS CodeArtifact.

Every application starts with a single package in a single repository

A single package is the entry point of your AWS CDK app. Here, you define how and where to deploy the different logical units of your application. You also define the CI/CD pipeline to deploy the application. The app’s constructs define the logical units of your solution.

Use additional packages for constructs that you use in more than one application. (Shared constructs should also have their own lifecycle and testing strategy.) Dependencies between packages in the same repository are managed by your repo’s build tooling.

Although it’s possible, we don’t recommend putting multiple applications in the same repository, especially when using automated deployment pipelines. Doing this increases the "blast radius" of changes during deployment. When there are multiple applications in a repository, changes to one application trigger deployment of the others (even if the others haven’t changed). Furthermore, a break in one application prevents the other applications from being deployed.

Move code into repositories based on code lifecycle or team ownership

When packages begin to be used in multiple applications, move them to their own repository. This way, the packages can be referenced by application build systems that use them, and they can also be updated on cadences independent of the application lifecycles. However, at first it might make sense to put all shared constructs in one repository.

Also, move packages to their own repository when different teams are working on them. This helps enforce access control.

To consume packages across repository boundaries, you need a private package repository —similar to NPM, PyPi, or Maven Central, but internal to your organization. You also need a release process that builds, tests, and publishes the package to the private package repository. CodeArtifact can host packages for most popular programming languages.

Dependencies on packages in the package repository are managed by your language's package manager, such as NPM for TypeScript or JavaScript applications. Your package manager helps to make sure that builds are repeatable. It does this by recording the specific versions of every package that your application depends on. It also lets you upgrade those dependencies in a controlled manner.

Shared packages need a different testing strategy. For a single application, it might be good enough to deploy the application to a testing environment and confirm that it still works. But shared packages must be tested independently of the consuming application, as if they were being released to the public. (Your organization might choose to actually release some shared packages to the public.)

Keep in mind that a construct can be arbitrarily simple or complex. A `Bucket` is a construct, but `CameraShopWebsite` could be a construct, too.

Infrastructure and runtime code live in the same package

In addition to generating AWS CloudFormation templates for deploying infrastructure, the AWS CDK also bundles runtime assets like Lambda functions and Docker images and deploys them alongside your infrastructure. This makes it 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 to do this. These two kinds of code don't need to live in separate repositories or even in separate packages.

To evolve the two kinds of code together, you can use a self-contained construct that completely describes a piece of functionality, including its infrastructure and logic. With a self-contained construct, you can test the two kinds of code in isolation, share and reuse the code across projects, and version all the code in sync.

Construct best practices

This section contains best practices for developing constructs. Constructs are reusable, composable modules that encapsulate resources. They're the building blocks of AWS CDK apps.

Model with constructs, deploy with stacks

Stacks are the unit of deployment: everything in a stack is deployed together. So when building your application's higher-level logical units from multiple AWS resources, represent each logical unit as a Construct, not as a Stack. Use stacks only to describe how your constructs should be composed and connected for your various deployment scenarios.

For example, if one of your logical units is a website, the constructs that make it up (such as an Amazon S3 bucket, API Gateway, Lambda functions, or Amazon RDS tables) should be composed into a single high-level construct. Then that construct should be instantiated in one or more stacks for deployment.

By using constructs for building and stacks for deploying, you improve reuse potential of your infrastructure and give yourself more flexibility in how it's deployed.

Configure with properties and methods, not environment variables

Environment variable lookups inside constructs and stacks are a common anti-pattern. Both constructs and stacks should accept a properties object to allow for full configurability completely in code. Doing otherwise introduces a dependency on the machine that the code will run on, which creates yet more configuration information that you have to track and manage.

In general, environment variable lookups should be limited to the top level of an AWS CDK app. They should also be used to pass in information that's needed for running in a development environment. For more information, see Environments for the AWS CDK.

Unit test your infrastructure

To consistently run a full suite of unit tests at build time in all environments, avoid network lookups during synthesis and model all your production stages in code. (These best practices are covered later.) If any single commit always results in the same generated template, you can trust the unit tests that you write to confirm that the generated templates look the way you expect. For more information, see Test AWS CDK applications.

Don’t change the logical ID of stateful resources

Changing the logical ID of a resource results in the resource being replaced with a new one at the next deployment. For stateful resources like databases and S3 buckets, or persistent infrastructure like an Amazon VPC, this is seldom what you want. Be careful about any refactoring of your AWS CDK code that could cause the ID to change. Write unit tests that assert that the logical IDs of your stateful resources remain static. The logical ID is derived from the `id` you specify when you instantiate the construct, and the construct’s position in the construct tree. For more information, see Logical IDs.

Constructs aren’t enough for compliance

Many enterprise customers write their own wrappers for L2 constructs (the "curated" constructs that represent individual AWS resources with built-in sane defaults and best practices). These wrappers enforce security best practices such as static encryption and specific IAM policies. For example, you might create a `MyCompanyBucket` that you then use in your applications in place of the usual Amazon S3 `Bucket` construct. This pattern is useful for surfacing security guidance early in the software development lifecycle, but don’t rely on it as the sole means of enforcement.

Instead, use AWS features such as service control policies and permission boundaries to enforce your security guardrails at the organization level. Use Aspects and the AWS CDK or tools like CloudFormation Guard to make assertions about the security properties of infrastructure elements before deployment. Use AWS CDK for what it does best.

Finally, keep in mind that writing your own "L2+" constructs might prevent your developers from taking advantage of AWS CDK packages such as AWS Solutions Constructs or third-party constructs from Construct Hub. These packages are typically built on standard AWS CDK constructs and won’t be able to use your wrapper constructs.

Application best practices

In this section we discuss how to write your AWS CDK applications, combining constructs to define how your AWS resources are connected.

Make decisions at synthesis time

Although AWS CloudFormation lets you make decisions at deployment time (using `Conditions`, `{ Fn::If }`, and `Parameters`), and the AWS CDK gives you some access to these mechanisms, we recommend against using them. The types of values that you can use and the types of operations you can perform on them are limited compared to what's available in a general-purpose programming language.

Instead, try to make all decisions, such as which construct to instantiate, in your AWS CDK application by using your programming language's `if` statements and other features. For example, a common CDK idiom, iterating over a list and instantiating a construct with values from each item in the list, simply isn't possible using AWS CloudFormation expressions.

Treat AWS CloudFormation as an implementation detail that the AWS CDK uses for robust cloud deployments, not as a language target. You're not writing AWS CloudFormation templates in TypeScript or Python, you're writing CDK code that happens to use CloudFormation for deployment.

Use generated resource names, not physical names

Names are a precious resource. Each name can only be used once. Therefore, if you hardcode a table name or bucket name into your infrastructure and application, you can't deploy that piece of infrastructure twice in the same account. (The name we're talking about here is the name specified by, for example, the `bucketName` property on an Amazon S3 bucket construct.)

What's worse, you can't make changes to the resource that require it to be replaced. If a property can only be set at resource creation, such as the `KeySchema` of an Amazon DynamoDB table, then that property is immutable. Changing this property requires a new resource. However, the new resource must have the same name in order to be a true replacement. But it can't have the same name while the existing resource is still using that name.

A better approach is to specify as few names as possible. If you omit resource names, the AWS CDK will generate them for you in a way that won't cause problems. Suppose you have a table as a resource. You can then pass the generated table name as an environment variable into your AWS Lambda function. In your AWS CDK application, you can reference the table name as `table.tableName`. Alternatively, you can generate a configuration file on your Amazon EC2 instance on startup, or write the actual table name to the AWS Systems Manager Parameter Store so your application can read it from there.

If the place you need it is another AWS CDK stack, that's even more straightforward. Supposing that one stack defines the resource and another stack needs to use it, the following applies:

- If the two stacks are in the same AWS CDK app, pass a reference between the two stacks. For example, save a reference to the resource's construct as an attribute of the defining stack (`this.stack.uploadBucket = amzn-s3-demo-bucket`). Then, pass that attribute to the constructor of the stack that needs the resource.
- When the two stacks are in different AWS CDK apps, use a static `from` method to use an externally defined resource based on its ARN, name, or other attributes. (For example, use `Table.fromArn()` for a DynamoDB table). Use the `CfnOutput` construct to print the ARN or other required value in the output of `cdk deploy`, or look in the AWS Management Console. Alternatively, the second app can read the CloudFormation template generated by the first app and retrieve that value from the `Outputs` section.

Define removal policies and log retention

The AWS CDK attempts to keep you from losing data by defaulting to policies that retain everything you create. For example, the default removal policy on resources that contain data (such as Amazon S3 buckets and database tables) is not to delete the resource when it is removed from the stack. Instead, the resource is orphaned from the stack. Similarly, the CDK's default is to retain all logs forever. In production environments, these defaults can quickly result in the storage of large amounts of data that you don't actually need, and a corresponding AWS bill.

Consider carefully what you want these policies to be for each production resource and specify them accordingly. Use [Aspects and the AWS CDK](#) to validate the removal and logging policies in your stack.

Separate your application into multiple stacks as dictated by deployment requirements

There is no hard and fast rule to how many stacks your application needs. You'll usually end up basing the decision on your deployment patterns. Keep in mind the following guidelines:

- It's typically more straightforward to keep as many resources in the same stack as possible, so keep them together unless you know you want them separated.
- Consider keeping stateful resources (like databases) in a separate stack from stateless resources. You can then turn on termination protection on the stateful stack. This way, you can freely destroy or create multiple copies of the stateless stack without risk of data loss.
- Stateful resources are more sensitive to construct renaming—renaming leads to resource replacement. Therefore, don't nest stateful resources inside constructs that are likely to be moved around or renamed (unless the state can be rebuilt if lost, like a cache). This is another good reason to put stateful resources in their own stack.

Commit `cdk.context.json` to avoid non-deterministic behavior

Determinism is key to successful AWS CDK deployments. An AWS CDK app should have essentially the same result whenever it is deployed to a given environment.

Since your AWS CDK app is written in a general-purpose programming language, it can execute arbitrary code, use arbitrary libraries, and make arbitrary network calls. For example, you could use an AWS SDK to retrieve some information from your AWS account while synthesizing your app. Recognize that doing so will result in additional credential setup requirements, increased latency, and a chance, however small, of failure every time you run `cdk synth`.

Never modify your AWS account or resources during synthesis. Synthesizing an app should not have side effects. Changes to your infrastructure should happen only in the deployment phase, after the AWS CloudFormation template has been generated. This way, if there's a problem, AWS CloudFormation can automatically roll back the change. To make changes that can't be easily made within the AWS CDK framework, use custom resources to execute arbitrary code at deployment time.

Even strictly read-only calls are not necessarily safe. Consider what happens if the value returned by a network call changes. What part of your infrastructure will that impact? What will happen to already-deployed resources? Following are two example situations in which a sudden change in values might cause a problem.

- If you provision an Amazon VPC to all available Availability Zones in a specified Region, and the number of AZs is two on deployment day, then your IP space gets split in half. If AWS launches a new Availability Zone the next day, the next deployment after that tries to split your IP space into thirds, requiring all subnets to be recreated. This probably won't be possible because your Amazon EC2 instances are still running, and you'll have to clean this up manually.

- If you query for the latest Amazon Linux machine image and deploy an Amazon EC2 instance, and the next day a new image is released, a subsequent deployment picks up the new AMI and replaces all your instances. This might not be what you expected to happen.

These situations can be pernicious because the AWS-side change might occur after months or years of successful deployments. Suddenly your deployments are failing "for no reason" and you long ago forgot what you did and why.

Fortunately, the AWS CDK includes a mechanism called *context providers* to record a snapshot of non-deterministic values. This allows future synthesis operations to produce exactly the same template as they did when first deployed. The only changes in the new template are the changes that *you* made in your code. When you use a construct's `.fromLookup()` method, the result of the call is cached in `cdk.context.json`. You should commit this to version control along with the rest of your code to make sure that future executions of your CDK app use the same value. The CDK Toolkit includes commands to manage the context cache, so you can refresh specific entries when you need to. For more information, see Context values and the AWS CDK.

If you need some value (from AWS or elsewhere) for which there is no native CDK context provider, we recommend writing a separate script. The script should retrieve the value and write it to a file, then read that file in your CDK app. Run the script only when you want to refresh the stored value, not as part of your regular build process.

Let the AWS CDK manage roles and security groups

With the AWS CDK construct library's `grant()` convenience methods, you can create AWS Identity and Access Management roles that grant access to one resource by another using minimally scoped permissions. For example, consider a line like the following:

```
amzn-s3-demo-bucket.grantRead(myLambda)
```

This single line adds a policy to the Lambda function's role (which is also created for you). That role and its policies are more than a dozen lines of CloudFormation that you don't have to write. The AWS CDK grants only the minimal permissions required for the function to read from the bucket.

If you require developers to always use predefined roles that were created by a security team, AWS CDK coding becomes much more complicated. Your teams could lose a lot of flexibility in how they design their applications. A better alternative is to use service control policies and permission boundaries to make sure that developers stay within the guardrails.

Model all production stages in code

In traditional AWS CloudFormation scenarios, your goal is to produce a single artifact that is parameterized so that it can be deployed to various target environments after applying configuration values specific to those environments. In the CDK, you can, and should, build that configuration into your source code. Create a stack for your production environment, and create a separate stack for each of your other stages. Then, put the configuration values for each stack in the code. Use services like Secrets Manager and Systems Manager Parameter Store for sensitive values that you don't want to check in to source control, using the names or ARNs of those resources.

When you synthesize your application, the cloud assembly created in the `cdk.out` folder contains a separate template for each environment. Your entire build is deterministic. There are no out-of-band changes to your application, and any given commit always yields the exact same AWS CloudFormation template and accompanying assets. This makes unit testing much more reliable.

Measure everything

Achieving the goal of full continuous deployment, with no human intervention, requires a high level of automation. That automation is only possible with extensive amounts of monitoring. To measure all aspects of your deployed resources, create metrics, alarms, and dashboards. Don't stop at measuring things like CPU usage and disk space. Also record your business metrics, and use those measurements to automate deployment decisions like rollbacks. Most of the L2 constructs in AWS CDK have convenience methods to help you create metrics, such as the `metricUserErrors()` method on the `dynamodb.Table` class.

AWS CDK security best practices

The AWS Cloud Development Kit (AWS CDK) is a powerful tool that developers can use to configure AWS services and provision infrastructure on AWS. With any tool that provides such control and capabilities, organizations will need to establish policies and practices to ensure that the tool is being used in safe and secure ways. For example, organizations may want to restrict developer access to specific services to ensure that they can't tamper with compliance or cost control measures that are configured in the account.

Often, there can be a tension between security and productivity, and each organization needs to establish the proper balance for themselves. This topic provides security best practices for the AWS CDK that you can consider as you create and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

Follow IAM security best practices

AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources. Organizations, individuals, and the AWS CDK use IAM to manage permissions that determine the actions that can be performed on AWS resources. When using IAM, follow the IAM security best practices. For more information, see Security best practices and use cases in AWS Identity and Access Management in the *IAM User Guide*.

Manage permissions for the AWS CDK

When you use the AWS CDK across your organization to develop and manage your infrastructure, you'll want to consider the following scenarios where managing permissions will be important:

- **Permissions for AWS CDK deployments –** These permissions determine who can make changes to your AWS resources and what changes they can make.
- **Permissions between resources –** These are the permissions that allow interactions between the AWS resources that you create and manage with the AWS CDK.

Manage permissions for AWS CDK deployments

Developers use the AWS CDK to define infrastructure locally on their development machines. This infrastructure is implemented in AWS environments through deployments that typically involve using the AWS CDK Command Line Interface (AWS CDK CLI). With deployments, you may want to control what changes developers can make in your environments. For example, you might have an Amazon Virtual Private Cloud (Amazon VPC) resource that you don't want developers to modify.

By default, the CDK CLI uses a combination of the actor's security credentials and IAM roles that are created during bootstrapping to receive permissions for deployments. The actor's security credentials are first used for authentication and IAM roles are then assumed to perform various actions during deployment, such as using the AWS CloudFormation service to create resources. For more information on how CDK deployments work, including the IAM roles that are used, see Deploy AWS CDK applications.

To restrict who can perform deployments and the actions that can be performed during deployment, consider the following:

- The actor's security credentials are the first set of credentials used to authenticate to AWS. From here, the permissions used to perform actions during deployment are granted to the IAM roles that are assumed during the deployment workflow. You can restrict who can perform deployments by limiting who can assume these roles. You can also restrict the actions that can be performed during deployment by replacing these IAM roles with your own.
- Permissions for performing deployments are given to the `DeploymentActionRole`. You can control permissions for who can perform deployments by limiting who can assume this role. By using a role for deployments, you can perform cross-account deployments since the role can be assumed by AWS identities in a different account. By default, all identities in the same AWS account with the appropriate `AssumeRole` policy statement can assume this role.
- Permissions for creating and modifying resources through AWS CloudFormation are given to the `CloudFormationExecutionRole`. This role also requires permission to read from the bootstrap resources. You control the permissions that CDK deployments have by using a managed policy for the `CloudFormationExecutionRole` and optionally by configuring a permissions boundary. By default, this role has `AdministratorAccess` permissions with no permission boundary.
- Permissions for interacting with bootstrap resources are given to the `FilePublishingRole` and `ImagePublishingRole`. The actor performing deployments must have permission to assume these roles. By default, all identities in the same AWS account with the appropriate `AssumeRole` policy statement can assume this role.
- Permissions for accessing bootstrap resources to perform lookups are given to the `LookupRole`. The actor performing deployments must have permission to assume this role. By default, this role has `readOnly` access to the bootstrap resources. By default, all identities in the same AWS account with the appropriate `AssumeRole` policy statement can assume this role.

To configure the IAM identities in your AWS account with permission to assume these roles, add a policy with the following policy statement to the identities:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "AssumeCDKRoles",
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "*",
    "Condition": {
      "StringEquals": {
        "iam:ResourceTag/aws-cdk:bootstrap-role": [
          "image-publishing",
          "file-publishing",
          "deploy",
          "lookup"
        ]
      }
    }
  }]
}
```

Modify the permissions for the roles assumed during deployment

By modifying permissions for the roles assumed during deployment, you can manage the actions that can be performed during deployment. To modify permissions, you create your own IAM roles and specify them when bootstrapping your environment. When you customize bootstrapping, you will have to customize synthesis. For general instructions, see Customize AWS CDK bootstrapping.

Modify the security credentials and roles used during deployment

The roles and bootstrap resources that are used during deployments are determined by the CDK stack synthesizer that you use. To modify this behavior, you can customize synthesis. For more information, see Configure and perform CDK stack synthesis.

Considerations for granting least privilege access

Granting least privilege access is a security best practice that we recommend that you consider as you develop your security strategy. For more information, see SEC03-BP02 Grant least privilege access in the *AWS Well-Architected Framework Guide*.

Often, granting least privilege access involves restricting IAM policies to the minimum access necessary to perform a given task. Attempting to grant least privilege access through fine-grained permissions with the CDK using this approach can impact CDK deployments and cause you to have to create wider-scoped permissions than you'd like. The following are a few things to consider when using this approach:

- Determining an exhaustive list of permissions that allow developers to use the AWS CDK to provision infrastructure through CloudFormation is difficult and complex.
- If you want to be fine-grained, permissions may become too long to fit within the maximum length of IAM policy documents.
- Providing an incomplete set of permissions can severely impact developer productivity and deployments.

With the CDK, deployments are performed using CloudFormation. CloudFormation initiates a set of AWS API calls in order using the permissions that are provided. The permissions necessary at any point in time depends on many factors:

- The AWS services that are being modified. Specifically, the resources and properties that are being used and changed.
- The current state of the CloudFormation stack.
- Issues that may occur during deployments and if rollbacks are needed, which will require `Delete` permissions in addition to `Create`.

When the provided permissions are incomplete, manual intervention will be required. The following are a few examples:

- If you discover incomplete permissions during roll forward, you'll need to pause deployment, and take time to discuss and provision new permissions before continuing.
- If deployment rolls back and the permissions to apply the roll back are missing, it may leave your CloudFormation stack in a state that will require a lot of manual work to recover from.

Since this approach can result in complications and severely limit developer productivity, we don't recommend it. Instead, we recommend implementing guardrails and preventing bypass.

Implementing guardrails and preventing bypass

You can implement guardrails, compliance rules, auditing, and monitoring by using services such as AWS Control Tower, AWS Config, AWS CloudTrail, AWS Security Hub, and others. With this approach, you grant developers permission to do everything, except tampering with the existing validation mechanisms. Developers have the freedom to implement changes quickly, as long as they stay within policy. This is the approach we recommend when using the AWS CDK. For more information on guardrails, see Controls in the *Management and Governance Cloud Environment Guide*.

We also recommend using *permissions boundaries* or *service control policies (SCPs)* as a way of implementing guardrails. For more information on implementing permissions boundaries with the AWS CDK, see Create and apply permissions boundaries for the AWS CDK.

If you are using any compliance control mechanisms, set them up during the bootstrapping phase. Make sure that the `CloudFormationExecutionRole` or developer-accessible identities have policies or permissions boundaries attached that prevent bypass of the mechanisms that you put in place. The appropriate policies depends on the specific mechanisms that you use.

Manage permissions between resources provisioned by the AWS CDK

How you manage permissions between resources that are provisioned by the AWS CDK depends on whether you allow the CDK to create roles and policies.

When you use L2 constructs from the AWS Construct Library to define your infrastructure, you can use the provided `grant` methods to provision permissions between resources. With `grant` methods, you specify the type of access you want between resources and the AWS CDK provisions least privilege IAM roles to accomplish your intent. This approach meets security requirements for most organizations while being efficient for developers. For more information, see Define permissions for L2 constructs with the AWS CDK.

If you want to work around this feature by replacing the automatically generated roles with manually created ones, consider the following:

- Your IAM roles will need to be manually created, slowing down application development.
- When IAM roles need to be manually created and managed, people will often combine multiple logical roles into a single role to make them easier to manage. This runs counter to the least privilege principle.
- Since these roles will need to be created before deployment, the resources that need to be referenced will not yet exist. Therefore, you'll need to use wildcards, which runs counter to the least privilege principle.
- A common workaround to using wildcards is to mandate that all resources be given a predictable name. However, this interferes with CloudFormation's ability to replace resources when

necessary and may slow down or block development. Because of this, we recommend that you allow CloudFormation to create unique resource names for you.

- It will be impossible to perform continuous delivery since manual actions must be performed prior to every deployment.

When organizations want to prevent the CDK from creating roles, it is usually to prevent developers from being able to create IAM roles. The concern is that by giving developers permission to create IAM roles using the AWS CDK, they could possibly elevate their own privileges. To mitigate against this, we recommend using *permission boundaries* or *service control policies (SCPs)*. With permission boundaries, you can set limits for what developers and the CDK are allowed to do. For more information on using permission boundaries with the CDK, see Create and apply permissions boundaries for the AWS CDK.

Migrating from AWS CDK v1 to AWS CDK v2

Version 2 of the AWS Cloud Development Kit (AWS CDK) is designed to make writing infrastructure as code in your preferred programming language easier. This topic describes the changes between v1 and v2 of the AWS CDK.

> **Tip**
>
> To identify stacks deployed with AWS CDK v1, use the awscdk-v1-stack-finder utility.

The main changes from AWS CDK v1 to CDK v2 are as follows.

- AWS CDK v2 consolidates the stable parts of the AWS Construct Library, including the core library, into a single package, `aws-cdk-lib`. Developers no longer need to install additional packages for the individual AWS services they use. This single-package approach also means that you don't have to synchronize the versions of the various CDK library packages.

  L1 (CfnXXXX) constructs, which represent the exact resources available in AWS CloudFormation, are always considered stable and so are included in `aws-cdk-lib`.
- Experimental modules, where we're still working with the community to develop new L2 or L3 constructs, are not included in `aws-cdk-lib`. Instead, they're distributed as individual packages. Experimental packages are named with an `alpha` suffix and a semantic version number. The semantic version number matches the first version of the AWS Construct Library that they're compatible with, also with an `alpha` suffix. Constructs move into `aws-cdk-lib` after being designated stable, permitting the main Construct Library to adhere to strict semantic versioning.

  Stability is specified at the service level. For example, if we begin creating one or more L2 constructs for Amazon AppFlow, which at this writing has only L1 constructs, they first appear in a module named `@aws-cdk/aws-appflow-alpha`. Then, they move to `aws-cdk-lib` when we feel that the new constructs meet the fundamental needs of customers.

  Once a module has been designated stable and incorporated into `aws-cdk-lib`, new APIs are added using the "BetaN" convention described in the next bullet.

  A new version of each experimental module is released with every release of the AWS CDK. For the most part, however, they needn't be kept in sync. You can upgrade `aws-cdk-lib` or the experimental module whenever you want. The exception is that when two or more related experimental modules depend on each other, they must be the same version.

- For stable modules to which new functionality is being added, new APIs (whether entirely new constructs or new methods or properties on an existing construct) receive a `Beta1` suffix while work is in progress. (Followed by `Beta2`, `Beta3`, and so on when breaking changes are needed.) A version of the API without the suffix is added when the API is designated stable. All methods except the latest (whether beta or final) are then deprecated.

  For example, if we add a new method `grantPower()` to a construct, it initially appears as `grantPowerBeta1()`. If breaking changes are needed (for example, a new required parameter or property), the next version of the method would be named `grantPowerBeta2()`, and so on. When work is complete and the API is finalized, the method `grantPower()` (with no suffix) is added, and the BetaN methods are deprecated.

  All the beta APIs remain in the Construct Library until the next major version (3.0) release, and their signatures will not change. You'll see deprecation warnings if you use them, so you should move to the final version of the API at your earliest convenience. However, no future AWS CDK 2.x releases will break your application.

- The `Construct` class has been extracted from the AWS CDK into a separate library, along with related types. This is done to support efforts to apply the Construct Programming Model to other domains. If you are writing your own constructs or using related APIs, you must declare the `constructs` module as a dependency and make minor changes to your imports. If you are using advanced features, such as hooking into the CDK app lifecycle, more changes may be needed. For full details, see the RFC.

- Deprecated properties, methods, and types in AWS CDK v1.x and its Construct Library have been removed completely from the CDK v2 API. In most supported languages, these APIs produce warnings under v1.x, so you may have already migrated to the replacement APIs. A complete list of deprecated APIs in CDK v1.x is available on GitHub.

- Behavior that was gated by feature flags in AWS CDK v1.x is enabled by default in CDK v2. The earlier feature flags are no longer needed, and in most cases they're not supported. A few are still available to let you revert to CDK v1 behavior in very specific circumstances. For more information, see Updating feature flags.

- With CDK v2, the environments you deploy into must be bootstrapped using the modern bootstrap stack. The legacy bootstrap stack (the default under v1) is no longer supported. CDK v2 furthermore requires a new version of the modern stack. To upgrade your existing environments, re-bootstrap them. It is no longer necessary to set any feature flags or environment variables to use the modern bootstrap stack.

Important

The modern bootstrap template effectively grants the permissions implied by the `--cloudformation-execution-policies` to any AWS account in the `--trust` list. By default, this extends permissions to read and write to any resource in the bootstrapped account. Make sure to configure the bootstrapping stack with policies and trusted accounts that you are comfortable with.

New prerequisites

Most requirements for AWS CDK v2 are the same as for AWS CDK v1.x. Additional requirements are listed here.

- For TypeScript developers, TypeScript 3.8 or later is required.
- A new version of the CDK Toolkit is required for use with CDK v2. Now that CDK v2 is generally available, v2 is the default version when installing the CDK Toolkit. It is backward-compatible with CDK v1 projects, so you don't need to keep the earlier version installed unless you want to create CDK v1 projects. To upgrade, issue `npm install -g aws-cdk`.

Upgrading from AWS CDK v2 Developer Preview

If you're using the CDK v2 Developer Preview, you have dependencies in your project on a Release Candidate version of the AWS CDK, such as `2.0.0-rc1`. Update these to `2.0.0`, then update the modules installed in your project.

Example

TypeScript

`npm install` or `yarn install`

JavaScript

`npm install` or `yarn install`

Python

```
python -m pip install -r requirements.txt
```

Java

```
mvn package
```

C#

```
dotnet restore
```

Go

```
go get
```

After updating your dependencies, issue `npm update -g aws-cdk` to update the CDK Toolkit to the release version.

Migrating from AWS CDK v1 to CDK v2

To migrate your app to AWS CDK v2, first update the feature flags in `cdk.json`. Then update your app's dependencies and imports as necessary for the programming language that it's written in.

Updating to a recent v1

We are seeing a number of customers upgrading from an old version of AWS CDK v1 to the most recent version of v2 in one step. While it is certainly possible to do that, you would be both upgrading across multiple years of changes (that unfortunately may not all have had the same amount of evolution testing we have today), as well as upgrading across versions with new defaults and a different code organization.

For the safest upgrade experience and to more easily diagnose the sources of any unexpected changes, we recommend you separate those two steps: first upgrade to the latest v1 version, then make the switch to v2 afterwards.

Updating feature flags

Remove the following v1 feature flags from `cdk.json` if they exist, as these are all active by default in AWS CDK v2. If their old effect is important for your infrastructure, you will need to make source code changes. See the list of flags on GitHub for more information.

- `@aws-cdk/core:enableStackNameDuplicates`
- `aws-cdk:enableDiffNoFail`
- `@aws-cdk/aws-ecr-assets:dockerIgnoreSupport`
- `@aws-cdk/aws-secretsmanager:parseOwnedSecretName`
- `@aws-cdk/aws-kms:defaultKeyPolicies`
- `@aws-cdk/aws-s3:grantWriteWithoutAcl`
- `@aws-cdk/aws-efs:defaultEncryptionAtRest`

A handful of v1 feature flags can be set to `false` in order to revert to specific AWS CDK v1 behaviors; see Reverting to v1 behavior or the list on GitHub for a complete reference.

For both types of flags, use the `cdk diff` command to inspect the changes to your synthesized template to see if the changes to any of these flags affect your infrastructure.

CDK Toolkit compatibility

CDK v2 requires v2 or later of the CDK Toolkit. This version is backward-compatible with CDK v1 apps. Therefore, you can use a single globally installed version of CDK Toolkit with all your AWS CDK projects, whether they use v1 or v2. An exception is that CDK Toolkit v2 only creates CDK v2 projects.

If you need to create both v1 and v2 CDK projects, **do not install CDK Toolkit v2 globally.** (Remove it if you already have it installed: `npm remove -g aws-cdk`.) To invoke the CDK Toolkit, use `npx` to run v1 or v2 of the CDK Toolkit as desired.

```
npx aws-cdk@1.x init app --language typescript
npx aws-cdk@2.x init app --language typescript
```

> **Tip**
>
> Set up command line aliases so you can use the cdk and cdk1 commands to invoke the desired version of the CDK Toolkit.
>
> macOS/Linux
>
> ```
> alias cdk1="npx aws-cdk@1.x"
> alias cdk="npx aws-cdk@2.x"
> ```
>
> Windows
>
> ```
> doskey cdk1=npx aws-cdk@1.x $*
> doskey cdk=npx aws-cdk@2.x $*
> ```

Updating dependencies and imports

Update your app's dependencies, then install the new packages. Finally, update the imports in your code.

Example

TypeScript

Applications

For CDK apps, update `package.json` as follows. Remove dependencies on v1-style individual stable modules and establish the lowest version of `aws-cdk-lib` you require for your application (2.0.0 here).

Experimental constructs are provided in separate, independently versioned packages with names that end in `alpha` and an alpha version number. The alpha version number corresponds to the first release of `aws-cdk-lib` with which they're compatible. Here, we have pinned `aws-codestar` to v2.0.0-alpha.1.

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
```

```
    "@aws-cdk/aws-codestar-alpha": "2.0.0-alpha.1",
    "constructs": "^10.0.0"
  }
}
```

Construct libraries

For construct libraries, establish the lowest version of `aws-cdk-lib` you require for your application (2.0.0 here) and update `package.json` as follows.

Note that `aws-cdk-lib` appears both as a peer dependency and a dev dependency.

```
{
  "peerDependencies": {
    "aws-cdk-lib": "^2.0.0",
    "constructs": "^10.0.0"
  },
  "devDependencies": {
    "aws-cdk-lib": "^2.0.0",
    "constructs": "^10.0.0",
    "typescript": "~3.9.0"
  }
}
```

Note

You should perform a major version bump on your library's version number when releasing a v2-compatible library, because this is a breaking change for library consumers. It is not possible to support both CDK v1 and v2 with a single library. To continue to support customers who are still using v1, you could maintain the earlier release in parallel, or create a new package for v2.
It's up to you how long you want to continue supporting AWS CDK v1 customers. You might take your cue from the lifecycle of CDK v1 itself, which entered maintenance on June 1, 2022 and will reach end-of-life on June 1, 2023. For full details, see [AWS CDK Maintenance Policy](https://github.com/aws/aws-cdk-rfcs/blob/master/text/0079-cdk-2.0.md).

Both libraries and apps

Install the new dependencies by running `npm install` or `yarn install`.

Change your imports to import `Construct` from the new `constructs` module, core types such as `App` and `Stack` from the top level of `aws-cdk-lib`, and stable Construct Library modules for the services you use from namespaces under `aws-cdk-lib`.

```
import { Construct } from 'constructs';
import { App, Stack } from 'aws-cdk-lib';                 // core constructs
import { aws_s3 as s3 } from 'aws-cdk-lib';               // stable module
import * as codestar from '@aws-cdk/aws-codestar-alpha';  // experimental module
```

JavaScript

Update `package.json` as follows. Remove dependencies on v1-style individual stable modules and establish the lowest version of `aws-cdk-lib` you require for your application (2.0.0 here).

Experimental constructs are provided in separate, independently versioned packages with names that end in `alpha` and an alpha version number. The alpha version number corresponds to the first release of `aws-cdk-lib` with which they're compatible. Here, we have pinned `aws-codestar` to v2.0.0-alpha.1.

```
{
  "dependencies": {
    "aws-cdk-lib": "^2.0.0",
    "@aws-cdk/aws-codestar-alpha": "2.0.0-alpha.1",
    "constructs": "^10.0.0"
  }
}
```

Install the new dependencies by running `npm install` or `yarn install`.

Change your app's imports to do the following:

- Import `Construct` from the new `constructs` module
- Import core types, such as `App` and `Stack`, from the top level of `aws-cdk-lib`
- Import AWS Construct Library modules from namespaces under `aws-cdk-lib`

```
const { Construct } = require('constructs');
const { App, Stack } = require('aws-cdk-lib');              // core constructs
const s3 = require('aws-cdk-lib').aws_s3;                   // stable module
```

```
const codestar = require('@aws-cdk/aws-codestar-alpha');    // experimental module
```

Python

Update `requirements.txt` or the `install_requires` definition in `setup.py` as follows. Remove dependencies on v1-style individual stable modules.

Experimental constructs are provided in separate, independently versioned packages with names that end in `alpha` and an alpha version number. The alpha version number corresponds to the first release of `aws-cdk-lib` with which they're compatible. Here, we have pinned `aws-codestar` to v2.0.0alpha1.

```
install_requires=[
     "aws-cdk-lib>=2.0.0",
     "constructs>=10.0.0",
     "aws-cdk.aws-codestar-alpha>=2.0.0alpha1",
     # ...
],
```

Tip

Uninstall any other versions of AWS CDK modules already installed in your app's virtual environment using `pip uninstall`. Then Install the new dependencies with `python -m pip install -r requirements.txt`.

Change your app's imports to do the following:

- Import `Construct` from the new `constructs` module
- Import core types, such as `App` and `Stack`, from the top level of `aws_cdk`
- Import AWS Construct Library modules from namespaces under `aws_cdk`

```
from constructs import Construct
from aws_cdk import App, Stack                    # core constructs
from aws_cdk import aws_s3 as s3                  # stable module
import aws_cdk.aws_codestar_alpha as codestar     # experimental module

# ...
```

```
class MyConstruct(Construct):
    # ...

class MyStack(Stack):
    # ...

s3.Bucket(...)
```

Java

In `pom.xml`, remove all `software.amazon.awscdk` dependencies for stable modules and replace them with dependencies on `software.constructs` (for `Construct`) and `software.amazon.awscdk`.

Experimental constructs are provided in separate, independently versioned packages with names that end in `alpha` and an alpha version number. The alpha version number corresponds to the first release of `aws-cdk-lib` with which they're compatible. Here, we have pinned `aws-codestar` to v2.0.0-alpha.1.

```
<dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>aws-cdk-lib</artifactId>
    <version>2.0.0</version>
</dependency><dependency>
    <groupId>software.amazon.awscdk</groupId>
    <artifactId>code-star-alpha</artifactId>
    <version>2.0.0-alpha.1</version>
</dependency>
<dependency>
    <groupId>software.constructs</groupId>
    <artifactId>constructs</artifactId>
    <version>10.0.0</version>
</dependency>
```

Install the new dependencies by running `mvn package`.

Change your code to do the following:

- Import `Construct` from the new `software.constructs` library
- Import core classes, like `Stack` and `App`, from `software.amazon.awscdk`
- Import service constructs from `software.amazon.awscdk.services`

```
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.App;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.codestar.alpha.GitHubRepository;
```

C#

The most straightforward way to upgrade the dependencies of a C# CDK application is to edit the `.csproj` file manually. Remove all stable `Amazon.CDK.*` package references and replace them with references to the `Amazon.CDK.Lib` and `Constructs` packages.

Experimental constructs are provided in separate, independently versioned packages with names that end in `alpha` and an alpha version number. The alpha version number corresponds to the first release of `aws-cdk-lib` with which they're compatible. Here, we have pinned `aws-codestar` to v2.0.0-alpha.1.

```
<PackageReference Include="Amazon.CDK.Lib" Version="2.0.0" />
<PackageReference Include="Amazon.CDK.AWS.Codestar.Alpha" Version="2.0.0-alpha.1" />
<PackageReference Include="Constructs" Version="10.0.0" />
```

Install the new dependencies by running `dotnet restore`.

Change the imports in your source files as follows.

```
using Constructs;                   // for Construct class
using Amazon.CDK;                   // for core classes like App and Stack
using Amazon.CDK.AWS.S3;            // for stable constructs like Bucket
using Amazon.CDK.Codestar.Alpha;    // for experimental constructs
```

Go

Issue `go get` to update your dependencies to the latest version and update your project's `.mod` file.

Testing your migrated app before deploying

Before deploying your stacks, use `cdk diff` to check for unexpected changes to the resources. Changes to logical IDs (causing replacement of resources) are **not** expected.

Expected changes include but are not limited to:

- Changes to the `CDKMetadata` resource.
- Updated asset hashes.
- Changes related to the new-style stack synthesis. Applies if your app used the legacy stack synthesizer in v1.
- The addition of a `CheckBootstrapVersion` rule.

Unexpected changes are typically not caused by upgrading to AWS CDK v2 in itself. Usually, they're the result of deprecated behavior that was previously changed by feature flags. This is a symptom of upgrading from a version of CDK earlier than about 1.85.x. You would see the same changes upgrading to the latest v1.x release. You can usually resolve this by doing the following:

1. Upgrade your app to the latest v1.x release
2. Remove feature flags
3. Revise your code as necessary
4. Deploy
5. Upgrade to v2

> **Note**
>
> If your upgraded app is undeployable after the two-stage upgrade, report the issue.

When you are ready to deploy the stacks in your app, consider deploying a copy first so you can test it. The easiest way to do this is to deploy it into a different Region. However, you can also change the IDs of your stacks. After testing, be sure to destroy the testing copy with `cdk destroy`.

Troubleshooting

TypeScript `'from' expected` or `';' expected` error in imports

Upgrade to TypeScript 3.8 or later.

Run 'cdk bootstrap'

If you see an error like the following:

```
#  MyStack failed: Error: MyStack: SSM parameter /cdk-bootstrap/hnb659fds/version
 not found. Has the environment been bootstrapped? Please run 'cdk bootstrap' (see
 https://docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html)
    at CloudFormationDeployments.validateBootstrapStackVersion (.../aws-cdk/lib/api/
cloudformation-deployments.ts:323:13)
    at processTicksAndRejections (internal/process/task_queues.js:97:5)
MyStack: SSM parameter /cdk-bootstrap/hnb659fds/version not found. Has the
 environment been bootstrapped? Please run 'cdk bootstrap' (see https://
docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html)
```

AWS CDK v2 requires an updated bootstrap stack, and furthermore, all v2 deployments require bootstrap resources. (With v1, you could deploy simple stacks without bootstrapping.) For complete details, see AWS CDK bootstrapping.

Finding v1 stacks

When migrating your CDK application from v1 to v2, you might want to identify the deployed AWS CloudFormation stacks that were created using v1. To do this, run the following command:

```
npx awscdk-v1-stack-finder
```

For usage details, see the awscdk-v1-stack-finder README.

Migrate existing resources and AWS CloudFormation templates to the AWS CDK

The CDK Migrate feature is in preview release for AWS CDK and is subject to change.

Use the AWS Cloud Development Kit (AWS CDK) Command Line Interface (AWS CDK CLI) to migrate deployed AWS resources, deployed AWS CloudFormation stacks, and local AWS CloudFormation templates to AWS CDK.

How migration works

Use the AWS CDK CLI `cdk migrate` command to migrate from the following sources:

- Deployed AWS resources.
- Deployed AWS CloudFormation stacks.
- Local AWS CloudFormation templates.

Deployed AWS resources

You can migrate deployed AWS resources from a specific environment (AWS account and AWS Region) that are not associated with an AWS CloudFormation stack.

The AWS CDK CLI utilizes the *IaC generator* service to scan for resources in your AWS environment to gather resource details. To learn more about IaC generator, see Generating templates for existing resources in the *AWS CloudFormation User Guide*.

After gathering resource details, the AWS CDK CLI creates a new CDK app that includes a single stack containing your migrated resources.

Deployed AWS CloudFormation stacks

You can migrate a single AWS CloudFormation stack into a new AWS CDK app. The AWS CDK CLI will retrieve the AWS CloudFormation template of your stack and create a new CDK app. The CDK app will consist of a single stack that contains your migrated AWS CloudFormation stack.

Local AWS CloudFormation templates

You can migrate from a local AWS CloudFormation template. Local templates may or may not contain deployed resources. The AWS CDK CLI will create a new CDK app that contains a single stack with your resources.

After migrating, you can manage, modify, and deploy your CDK app to AWS CloudFormation to provision or update your resources.

Benefits of CDK Migrate

Migrating resources into AWS CDK has historically been a manual process that requires time and expertise with AWS CloudFormation and AWS CDK to even begin. With CDK Migrate, the AWS CDK CLI facilitates a majority of the migration effort for you in a fraction of the time. CDK Migrate will quickly get you started with using the AWS CDK to develop and manage new and existing applications on AWS.

Considerations

General considerations

CDK Migrate vs. CDK Import

The `cdk import` command can import deployed resources into a new or existing CDK app. When importing, each resource will have to manually be defined as an L1 construct in your app. We recommend using `cdk import` to import one or more resources at a time into a new or existing CDK app. To learn more, see [Import existing resources into a stack](#).

The `cdk migrate` command migrates from deployed resources, deployed AWS CloudFormation stacks, or local AWS CloudFormation templates into a new CDK app. During migration, the AWS CDK CLI uses `cdk import` to import your resources into the new CDK app. The AWS CDK CLI also generates L1 constructs for each resource for you. We recommend using `cdk migrate` when importing from a supported migration source into a new AWS CDK app.

CDK Migrate creates L1 constructs only

The newly created CDK app will include L1 constructs only. You can add higher-level constructs to your app after migration.

CDK Migrate creates CDK apps that contain a single stack

The newly created CDK app will contain a single stack.

When migrating deployed resources, all migrated resources will be contained within a single stack in the new CDK app.

When migrating AWS CloudFormation stacks, you can only migrate a single AWS CloudFormation stack into a single stack in the new CDK app.

Migrating assets

Project assets, such as AWS Lambda code, will not directly migrate into the new CDK app. After migration, you can specify asset values to include them in the CDK app.

Migrating stateful resources

When migrating stateful resources, such as databases and Amazon Simple Storage Service (Amazon S3) buckets, you'd most often want to migrate the existing resource instead of creating a new resource.

To migrate and preserve stateful resources, do the following:

- Verify that your stateful resource supports import. For more information, see Resource type support in the *AWS CloudFormation User Guide*.
- After migration, verify that the migrated resource's logical ID in the new CDK app matches the logical ID of the deployed resource.
- If migrating from an AWS CloudFormation stack, verify that the stack name in the new CDK app matches the AWS CloudFormation stack.
- Deploy the CDK app using the same AWS account and AWS Region of the migrated resource.

Considerations when migrating from an AWS CloudFormation template

CDK Migrate supports single template migration

When migrating AWS CloudFormation templates, you can select a single template to migrate. Nested templates are not supported.

Migrating templates with intrinsic functions

When migrating from an AWS CloudFormation template that uses intrinsic functions, the AWS CDK CLI will attempt to migrate your logic into the CDK app with the `Fn` class. To learn more, see class Fn in the *AWS Cloud Development Kit (AWS CDK) API Reference*.

Considerations when migrating from deployed resources

Scan limitations

When scanning your environment for resources, IaC generator has specific limitations on the data it can retrieve and quota limitations when scanning. To learn more, see Considerations in the *AWS CloudFormation User Guide*.

Prerequisites

Before using the `cdk migrate` command, complete all set up steps in Getting started with the AWS CDK.

Get started with CDK Migrate

To begin, run the AWS CDK CLI `cdk migrate` command from a directory of your choice. Provide required and optional options, depending on the type of migration you are performing.

For a full list and description of options that you can use with `cdk migrate`, see cdk migrate.

The following are some important options that you may want to provide.

Stack name

The only required option is `--stack-name`. Use this option to specify a name for the stack that will be created within the AWS CDK app after migration. The stack name will also be used as the name of your AWS CloudFormation stack at deployment.

Language

Use `--language` to specify the programming language of the new CDK app.

AWS account and AWS Region

The AWS CDK CLI retrieves AWS account and AWS Region information from default sources. For more information, see Environments for the AWS CDK. You can use `--account` and `--region` options with `cdk migrate` to provide other values.

Output directory of your new CDK project

By default, the AWS CDK CLI will create a new CDK project in your working directory and use the value you provide with `--stack-name` to name the project folder. If a folder with the same name currently exists, the AWS CDK CLI will overwrite that folder.

You can specify a different output path for the new CDK project folder with the `--output-path` option.

Migration source

Provide an option to specify the source you are migrating from.

- `--from-path` – Migrate from a local AWS CloudFormation template.
- `--from-scan` – Migrate from deployed resources in an AWS account and AWS Region.
- `--from-stack` – Migrate from an AWS CloudFormation stack.

Depending on your migration source, you can provide additional options to customize the `cdk migrate` command.

Migrate from an AWS CloudFormation stack

To migrate from a deployed AWS CloudFormation stack, provide the `--from-stack` option. Provide the name of your deployed AWS CloudFormation stack with `--stack-name`. The following is an example:

```
$ cdk migrate --from-stack --stack-name "myCloudFormationStack"
```

The AWS CDK CLI will do the following:

1. Retrieve the AWS CloudFormation template of your deployed stack.
2. Run `cdk init` to initialize a new CDK app.
3. Create a stack within the CDK app that contains your migrated AWS CloudFormation stack.

When you migrate from a deployed AWS CloudFormation stack, the AWS CDK CLI attempts to match deployed resource logical IDs and the deployed AWS CloudFormation stack name to the migrated resources and stack in the new CDK app.

After migration, you can manage and modify your CDK app normally. When you deploy, AWS CloudFormation will identify the deployment as an AWS CloudFormation stack update due to the matching AWS CloudFormation stack name. Resources with matching logical IDs will be updated. For more information on deploying, see Manage and deploy your CDK app.

Migrate from an AWS CloudFormation template

CDK Migrate supports migrating from AWS CloudFormation templates formatted in `JSON` or `YAML`.

To migrate from a local AWS CloudFormation template, use the `--from-path` option and provide a path to the local template. You must also provide the required `--stack-name` option. The following is an example:

```
$ cdk migrate --from-path "./template.json" --stack-name "myCloudFormationStack"
```

The AWS CDK CLI will do the following:

1. Retrieve your local AWS CloudFormation template.
2. Run `cdk init` to initialize a new CDK app.
3. Create a stack within the CDK app that contains your migrated AWS CloudFormation template.

After migration, you can manage and modify your CDK app normally. At deployment, you have the following options:

- **Update an AWS CloudFormation stack** – If the local AWS CloudFormation template was previously deployed, you can update the deployed AWS CloudFormation stack.
- **Deploy a new AWS CloudFormation stack** – If the local AWS CloudFormation template was never deployed, or if you want to create a new stack from a previously deployed template, you can deploy a new AWS CloudFormation stack.

Migrate from an AWS SAM template

To migrate from an AWS Serverless Application Model (AWS SAM) template, you must first convert it to an AWS CloudFormation template or deploy to create an AWS CloudFormation stack.

To convert an AWS SAM template to AWS CloudFormation, you can use the AWS SAM CLI `sam validate --debug` command. You may have to set `lint` to `false` in your `samconfig.toml` file before running this command.

To convert to an AWS CloudFormation stack, deploy the AWS SAM template using the AWS SAM CLI. Then migrate from the deployed stack.

Migrate from deployed resources

To migrate from deployed AWS resources, provide the `--from-scan` option. You must also provide the required `--stack-name` option. The following is an example:

```
$ cdk migrate --from-scan --stack-name "myCloudFormationStack"
```

The AWS CDK CLI will do the following:

1. **Scan your account for resource and property details** – The AWS CDK CLI utilizes IaC generator to scan your account and gather details.
2. **Generate an AWS CloudFormation template** – After scanning, the AWS CDK CLI utilizes IaC generator to create an AWS CloudFormation template.
3. **Initialize a new CDK app and migrate your template** – The AWS CDK CLI runs `cdk init` to initialize a new AWS CDK app and migrates your AWS CloudFormation template into the CDK app as a single stack.

Use filters

By default, the AWS CDK CLI will scan the entire AWS environment and migrate resources up to the maximum quota limit of IaC generator. You can provide filters with the AWS CDK CLI to specify a criteria for which resources get migrated from your account to the new CDK app. To learn more, see `--filter` .

Scanning for resources with IaC generator

Depending on the number of resources in your account, scanning may take a few minutes. A progress bar will display during the scanning process.

Supported resource types

The AWS CDK CLI will migrate resources supported by the IaC generator. For a full list, see [Resource type support](#) in the *AWS CloudFormation User Guide*.

Resolve write-only properties

Some supported resources contain write-only properties. These properties can be written to, to configure the property, but can't be read by IaC generator or AWS CloudFormation to obtain the value. For example, a property used to specify a database password may be write-only for security reasons.

When scanning resources during migration, IaC generator will detect resources that may contain write-only properties and will categorize them into any of the following types:

- `MUTUALLY_EXCLUSIVE_PROPERTIES` – These are write-only properties for a specific resource that are interchangeable and serve a similar purpose. One of the mutually exclusive properties are required to configure your resource. For example, the `S3Bucket`, `ImageUri`, and `ZipFile` properties for an `AWS::Lambda::Function` resource are mutually exclusive write-only properties. Any one of them can be used to specify your function assets, but you must use one.
- `MUTUALLY_EXCLUSIVE_TYPES` – These are required write-only properties that accept multiple configuration types. For example, the `Body` property of an `AWS::ApiGateway::RestApi` resource accepts an object or string type.
- `UNSUPPORTED_PROPERTIES` – These are write-only properties that don't fall under the other two categories. They are either optional properties or required properties that accept an array of objects.

For more information on write-only properties and how IaC generator manages them when scanning for deployed resources and creating AWS CloudFormation templates, see [IaC generator and write-only properties](#) in the *AWS CloudFormation User Guide*.

After migration, you must specify write-only property values in the new CDK app. The AWS CDK CLI will append a **Warnings** section to the CDK project's `ReadMe` file to document any write-only properties that were identified by IaC generator. The following is an example:

```
# Welcome to your CDK TypeScript project
...
## Warnings
### Write-only properties
Write-only properties are resource property values that can be written to but can't be
 read by AWS CloudFormation or CDK Migrate. For more information, see [IaC generator
 and write-only properties](https://docs.aws.amazon.com/AWSCloudFormation/latest/
UserGuide/generate-IaC-write-only-properties.html).

Write-only properties discovered during migration are organized here by resource ID and
 categorized by write-only property type. Resolve write-only properties by providing
 property values in your CDK app. For guidance, see [Resolve write-only properties]
(https://docs.aws.amazon.com/cdk/v2/guide/migrate.html#migrate-resources-writeonly).
### MyLambdaFunction
- **UNSUPPORTED_PROPERTIES**:
  - SnapStart/ApplyOn: Applying SnapStart setting on function resource type.Possible
 values: [PublishedVersions, None]
This property can be replaced with other types
  - Code/S3ObjectVersion: For versioned objects, the version of the deployment package
 object to use.
This property can be replaced with other exclusive properties
- **MUTUALLY_EXCLUSIVE_PROPERTIES**:
  - Code/S3Bucket: An Amazon S3 bucket in the same AWS Region as your function. The
 bucket can be in a different AWS account.
This property can be replaced with other exclusive properties
  - Code/S3Key: The Amazon S3 key of the deployment package.
This property can be replaced with other exclusive properties
```

- Warnings are organized under headings that identify the resource's logical ID that they are associated with.
- Warnings are categorized by type. These types come directly from IaC generator.

To resolve write-only properties

1. Identify write-only properties to resolve from the **Warnings** section of your CDK project's `ReadMe` file. Here, you can take note of the resources in your CDK app that may contain write-only properties and identify the write-only property types that were discovered.

   a. For `MUTUALLY_EXCLUSIVE_PROPERTIES`, determine which mutually exclusive property to configure in your AWS CDK app.

   b. For `MUTUALLY_EXCLUSIVE_TYPES`, determine which accepted type that you will use to configure the property.

   c. For `UNSUPPORTED_PROPERTIES`, determine if the property is optional or required. Then, configure as necessary.

2. Use guidance from [IaC generator and write-only properties] to reference what the warning types mean.

3. In your CDK app, write-only property values to resolve will also be specified in the `Props` section of your app. Provide the correct values here. For property descriptions and guidance, you can reference the [AWS CDK API Reference].

   The following is an example of the `Props` section within a migrated CDK app with two write-only properties to resolve:

```
export interface MyTestAppStackProps extends cdk.StackProps {
  /**
   * The Amazon S3 key of the deployment package.
   */
  readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Keym8P82: string;
  /**
   * An Amazon S3 bucket in the same AWS Region as your function. The bucket can
 be in a different AWS account.
   */
  readonly lambdaFunction00asdfasdfsadf008grk1CodeS3Bucketzidw8: string;
}
```

   Once you resolve all write-only property values, you're ready to prepare for deployment.

The migrate.json file

The AWS CDK CLI creates a `migrate.json` file in your AWS CDK project during migration. This file contains reference information on your deployed resources. When you deploy your CDK app for the first time, the AWS CDK CLI uses this file to reference your deployed resources, associates your resources with the new AWS CloudFormation stack, and deletes the file.

Manage and deploy your CDK app

When migrating to AWS CDK, the new CDK app may not be deployment-ready immediately. This topic describes action items to consider while managing and deploying your new CDK app.

Prepare for deployment

Before deploying, you must prepare your CDK app.

Synthesize your app

Use the `cdk synth` command to synthesize the stack in your CDK app into an AWS CloudFormation template.

If you migrated from a deployed AWS CloudFormation stack or template, you can compare the synthesized template to the migrated template to verify resource and property values.

To learn more about `cdk synth`, see [Synthesize stacks](#).

Perform a diff

If you migrated from a deployed AWS CloudFormation stack, you can use the cdk diff command to compare with the stack in your new CDK app.

To learn more about cdk diff, see [Compare stacks](#).

Bootstrap your environment

If you are deploying from an AWS environment for the first time, use `cdk bootstrap` to prepare your environment. To learn more, see [AWS CDK bootstrapping](#).

Deploy your CDK app

When you deploy a CDK app, the AWS CDK CLI utilizes the AWS CloudFormation service to provision your resources. Resources are bundled into a single stack in the CDK app and are deployed as a single AWS CloudFormation stack.

Depending on where you migrated from, you can deploy to create a new AWS CloudFormation stack or update an existing AWS CloudFormation stack.

Deploy to create a new AWS CloudFormation stack

If you migrated from deployed resources, the AWS CDK CLI will automatically create a new AWS CloudFormation stack at deployment. Your deployed resources will be included in the new AWS CloudFormation stack.

If you migrated from a local AWS CloudFormation template that was never deployed, the AWS CDK CLI will automatically create a new AWS CloudFormation stack at deployment.

If you migrated from a deployed AWS CloudFormation stack or local AWS CloudFormation template that was previously deployed, you can deploy to create a new AWS CloudFormation stack. To create a new stack, do the following:

- Deploy to a new AWS environment. This consists of using a different AWS account or deploying to a different AWS Region.
- If you want to deploy a new stack to the same AWS environment of the migrated stack or template, you must modify the stack name in your CDK app to a new value. You must also modify all logical IDs of resources in your CDK app. Then, you can deploy to the same environment to create a new stack and new resources.

Deploy to update an existing AWS CloudFormation stack

If you migrated from a deployed AWS CloudFormation stack or local AWS CloudFormation template that was previously deployed, you can deploy to update the existing AWS CloudFormation stack.

Verify that the stack name in your CDK app matches the stack name of the deployed AWS CloudFormation stack and deploy to the same AWS environment.

Configure security credentials for the AWS CDK CLI

When you use the AWS Cloud Development Kit (AWS CDK) to develop applications in your local environment, you will primarily use the AWS CDK Command Line Interface (AWS CDK CLI) to interact with AWS. For example, you can use the CDK CLI to deploy your application or to delete your resources from your AWS environment.

To use the CDK CLI to interact with AWS, you must configure security credentials on your local machine. This lets AWS know who you are and what permissions you have.

To learn more about security credentials, see AWS security credentials in the *IAM User Guide*.

Prerequisites

Configuring security credentials is part of the *getting started* process. Complete all prerequisites and previous steps at Getting started with the AWS CDK.

How to configure security credentials

How you configure security credentials depends on how you or your organization manages users. Whether you use AWS Identity and Access Management (IAM) or AWS IAM Identity Center, we recommend that you use the AWS Command Line Interface (AWS CLI) to configure and manage security credentials for the CDK CLI. This includes using AWS CLI commands like `aws configure` to configure security credentials on your local machine. However, you can use alternative methods such as manually updating your `config` and `credentials` files, or setting environment variables.

For guidance on configuring security credentials using the AWS CLI, along with information on configuration and credential precedence when using different methods, see Authentication and access credentials in the *AWS Command Line Interface User Guide*. The CDK CLI adheres to the same configuration and credential precedence of the AWS CLI. The `--profile` command line option takes precedence over environment variables. If you have both the `AWS_PROFILE` and `CDK_DEFAULT_PROFILE` environment variables configured, the `AWS_PROFILE` environment variable takes precedence.

If you configure multiple profiles, you can use the CDK CLI `--profile` option with any command to specify the profile from your `credentials` and `config` files to use for authentication. If you don't provide `--profile`, the `default` profile will be used.

If you prefer to quickly configure basic settings, including security credentials, see Set up the AWS CLI in the *AWS Command Line Interface User Guide*.

Once you've configured security credentials on your local machine, you can use the CDK CLI to interact with AWS.

Configure and manage security credentials for IAM Identity Center users

IAM Identity Center users can authenticate with IAM Identity Center or manually by using short-term credentials.

Authenticate with IAM Identity Center to generate short-term credentials

You can configure the AWS CLI to authenticate with IAM Identity Center. This is the recommended approach of configuring security credentials for IAM Identity Center users. IAM Identity Center users can use the AWS CLI `aws configure sso` wizard to configure an IAM Identity Center profile and `sso-session`, which gets stored in the `config` file on your local machine. For instructions, see Configure the AWS CLI to use AWS IAM Identity Center in the *AWS Command Line Interface User Guide*.

Next, you can use the AWS CLI `aws sso login` command to request refreshed credentials. You can also use this command to switch profiles. For instructions, see Use an IAM Identity Center named profile in the *AWS Command Line Interface User Guide*.

Once authenticated, you can use the CDK CLI to interact with AWS for the duration of your session. For an example, see Example: Authenticate with IAM Identity Center automatic token refresh for use with the AWS CDK CLI.

Manually configure short-term credentials

As an alternative to using the AWS CLI and authenticating with IAM Identity Center, IAM Identity Center users can obtain short-term credentials from the AWS Management Console and manually configure the `credentials` and `config` files on their local machine. Once configured, you can use the CDK CLI to interact with AWS until your credentials expire. For instructions, see Authenticate with short-term credentials in the *AWS Command Line Interface User Guide*.

Configure and manage security credentials for IAM users

IAM users can use an IAM role or IAM user credentials with the CDK CLI.

Use an IAM role to configure short-term credentials

IAM users can assume IAM roles to gain additional (or different) permissions. For IAM users, this is the recommended approach since it provides short-term credentials.

First, the IAM role and user's permission to assume the role must be configured. This is typically performed by an administrator using the AWS Management Console or AWS CLI. Then, the IAM user can use the AWS CLI to assume the role and configure short-term credentials on their local machine. For instructions, see Use an IAM role in the AWS CLI in the *AWS Command Line Interface User Guide*.

Use IAM user credentials

> **Warning**
>
> To avoid security risks, we don't recommend using IAM user credentials since they provide long-term access. If you must use long-term credentials, we recommend that you update access keys as an IAM security best practice.

IAM users can obtain access keys from the AWS Management Console. You can then use the AWS CLI to configure long-term credentials on your local machine. For instructions, see Authenticate with IAM user credentials in the *AWS Command Line Interface User Guide*.

Additional information

To learn about the different ways that you can sign in to AWS, depending on the type of user you are, see What is AWS Sign-In? in the *AWS Sign-In User Guide*.

For reference information when using AWS SDKs and tools, including the AWS CLI, see the AWS SDKs and Tools Reference Guide.

Example: Authenticate with IAM Identity Center automatic token refresh for use with the AWS CDK CLI

In this example, we configure the AWS Command Line Interface (AWS CLI) to authenticate our user with the AWS IAM Identity Center token provider configuration. The SSO token provider configuration lets the AWS CLI automatically retrieve refreshed authentication tokens to generate short-term credentials that we can use with the AWS Cloud Development Kit (AWS CDK) Command Line Interface (AWS CDK CLI).

Prerequisites

This example assumes that the following prerequisites have been completed:

- Prerequisites required to get set up with AWS and install our starting CLI tools. For more information, see Prerequisites.
- IAM Identity Center has been set up by our organization as the method of managing users.
- At least one user has been created in IAM Identity Center.

Step 1: Configure the AWS CLI

For detailed instructions on this step, see Configure the AWS CLI to use IAM Identity Center token provider credentials with automatic authentication refresh in the *AWS Command Line Interface User Guide*.

We sign in to the AWS access portal provided by our organization to gather our IAM Identity Center information. This includes the **SSO start URL** and **SSO Region**.

Next, we use the AWS CLI `aws configure sso` command to configure an IAM Identity Center profile and `sso-session` on our local machine:

```
$ aws configure sso
SSO session name (Recommended): <my-sso>
SSO start URL [None]: <https://my-sso-portal.awsapps.com/start>
SSO region [None]: <us-east-1>
SSO registration scopes [sso:account:access]: <ENTER>
```

The AWS CLI attempts to open our default browser to begin the login process for our IAM Identity Center account. If the AWS CLI is unable to open our browser, instructions are provided to manually start the login process. This process associates the IAM Identity Center session with our current AWS CLI session.

After establishing our session, the AWS CLI displays the AWS accounts available to us:

```
There are 2 AWS accounts available to you.
> DeveloperAccount, developer-account-admin@example.com (<123456789011>)
  ProductionAccount, production-account-admin@example.com (<123456789022>)
```

We use the arrow keys to select our **DeveloperAccount**.

Next, the AWS CLI displays the IAM roles available to us from our selected account:

```
Using the account ID
There are 2 roles available to you.
> ReadOnly
  FullAccess
```

We use the arrow keys to select **FullAccess**.

Next, the AWS CLI prompts us to complete configuration by specifying a default output format, default AWS Region, and name for our profile:

```
CLI default client Region [None]: <us-west-2> <ENTER>
CLI default output format [None]: <json> <ENTER>
CLI profile name [123456789011_FullAccess]: <my-dev-profile> <ENTER>
```

The AWS CLI displays a final message, showing how to use the named profile with the AWS CLI:

```
To use this profile, specify the profile name using --profile, as shown:

aws s3 ls --profile <my-dev-profile>
```

After completing this step, our `config` file will look like the following:

```
[profile <my-dev-profile>]
sso_session = <my-sso>
sso_account_id = <123456789011>
sso_role_name = <fullAccess>
region = <us-west-2>
```

```
output = <json>

[sso-session <my-sso>]
sso_region = <us-east-1>
sso_start_url = <https://my-sso-portal.awsapps.com/start>
sso_registration_scopes = <sso:account:access>
```

We can now use this `sso-session` and named profile to request security credentials.

Step 2: Use the AWS CLI to generate security credentials

For detailed instructions on this step, see Use an IAM Identity Center named profile in the *AWS Command Line Interface User Guide*.

We use the AWS CLI `aws sso login` command to request security credentials for our profile:

```
$ aws sso login --profile <my-dev-profile>
```

The AWS CLI attempts to open our default browser and verifies our IAM log in. If we are not currently signed into IAM Identity Center, we will be prompted to complete the sign in process. If the AWS CLI is unable to open our browser, instructions are provided to manually start the authorization process.

After successfully logging in, the AWS CLI caches our IAM Identity Center session credentials. These credentials include an expiration timestamp. When they expire, the AWS CLI will request that we sign in to IAM Identity Center again.

Using valid IAM Identity Center credentials, the AWS CLI securely retrieves AWS credentials for the IAM role specified in our profile. From here, we can use the AWS CDK CLI with our credentials.

Step 3: Use the CDK CLI

With any CDK CLI command, we use the `--profile` option to specify the named profile that we generated credentials for. If our credentials are valid, the CDK CLI will successfully perform the command. The following is an example:

```
$ cdk diff --profile <my-dev-profile>
Stack CdkAppStack
Hold on while we create a read-only change set to get a diff with accurate replacement
 information (use --no-change-set to use a less accurate but faster template-only diff)
```

```
Resources
[-] AWS::S3::Bucket amzn-s3-demo-bucket amzn-s3-demo-bucket5AF9C99B destroy

Outputs
[-] Output BucketRegion: {"Value":{"Ref":"AWS::Region"}}


#  Number of stacks with differences: 1
```

When our credentials expire, an error message like the following will display:

```
$ cdk diff --profile <my-dev-profile>
Stack CdkAppStack

Unable to resolve AWS account to use. It must be either configured when you define your
 CDK Stack, or through the environment
```

To refresh our credentials, we use the AWS CLI `aws sso login` command:

```
$ aws sso login --profile <my-dev-profile>
```

Configure environments to use with the AWS CDK

You can configure AWS environments in multiple ways to use with the AWS Cloud Development Kit (AWS CDK). The best method of managing AWS environments will vary, based on your specific needs.

Each CDK stack in your application must eventually be associated with an environment to determine where the stack gets deployed to.

For an introduction to AWS environments, see Environments for the AWS CDK.

Where you can specify environments from

You can specify environments in credentials and configuration files, or by using the `env` property of the `Stack` construct from the AWS Construct Library.

Credentials and configuration files

You can use the AWS Command Line Interface (AWS CLI) to create `credentials` and `config` files that store, organize, and manage your AWS environment information. To learn more about these files, see Configuration and credential file settings in the *AWS Command Line Interface User Guide*.

Values stored in these files are organized by *profiles*. How you name your profiles and the key-value pairs in these files will vary based on your method of configuring programmatic access. To learn more about the different methods, see Configure security credentials for the AWS CDK CLI.

In general, the AWS CDK resolves AWS account information from your `credentials` file and AWS Region information from your `config` file.

Once you have your `credentials` and `config` files configured, you can specify the environment to use with the AWS CDK CLI and through environment variables.

env property of the Stack construct

You can specify the environment for each stack by using the `env` property of the `Stack` construct. This property defines an account and Region to use. You can pass hard-coded values to this property or pass environment variables that are offered by the CDK.

To pass environment variables, use the `AWS_DEFAULT_ACCOUNT` and `AWS_DEFAULT_REGION` environment variables. These environment variables can pass values from your `credentials` and `config` files. You can also use logic within your CDK code to determine the values of these environment variables.

Environment precedence with the AWS CDK

If you use multiple methods of specifying environments, the AWS CDK adheres to the following precedence:

1. Hard-coded values specified with the `env` property of the `Stack` construct.
2. `AWS_DEFAULT_ACCOUNT` and `AWS_DEFAULT_REGION` environment variables specified with the `env` property of the `Stack` construct.
3. Environment information associated with the profile from your `credentials` and `config` files and passed to the CDK CLI using the `--profile` option.
4. The `default` profile from your `credentials` and `config` files.

When to specify environments

When you develop with the CDK, you start by defining CDK stacks, which contain constructs that represent AWS resources. Next, you synthesize each CDK stack into an AWS CloudFormation template. You then deploy the CloudFormation template to your environment. How you specify environments determines when your environment information gets applied and can affect CDK behavior and outcomes.

Specify environments at template synthesis

When you specify environment information using the `env` property of the `Stack` construct, your environment information is applied at template synthesis. Running `cdk synth` or `cdk deploy` produces an environment-specific CloudFormation template.

If you use environment variables within the `env` property, you must use the `--profile` option with CDK CLI commands to pass in the profile containing your environment information from your credentials and configuration files. This information will then be applied at template synthesis to produce an environment-specific template.

Environment information within the CloudFormation template takes precedence over other methods. For example, if you provide a different environment with `cdk deploy --profile <profile>`, the profile will be ignored.

When you provide environment information in this way, you can use environment-dependent code and logic within your CDK app. This also means that the synthesized template could be different, based on the machine, user, or session that it's synthesized under. This approach is often acceptable or desirable during development, but is not recommended for production use.

Specify environments at stack deployment

If you don't specify an environment using the `env` property of the `Stack` construct, the CDK CLI will produce an environment-agnostic CloudFormation template at synthesis. You can then specify the environment to deploy to by using `cdk deploy --profile <profile>`.

If you don't specify a profile when deploying an environment-agnostic template, the CDK CLI will attempt to use environment values from the `default` profile of your `credentials` and `config` files at deployment.

If environment information is not available at deployment, AWS CloudFormation will attempt to resolve environment information at deployment through environment-related attributes such as `stack.account`, `stack.region`, and `stack.availabilityZones`.

For environment-agnostic stacks, constructs within the stack cannot use environment information and you cannot use logic that requires environment information. For example, you cannot write code like `if (stack.region ==== 'us-east-1')` or use construct methods that require environment information such as `Vpc.fromLookup`. To use these features, you must specify an environment with the `env` property.

For environment-agnostic stacks, any construct that uses Availability Zones will see two Availability Zones, allowing the stack to be deployed to any Region.

How to specify environments with the AWS CDK

Specify hard-coded environments for each stack

Use the `env` property of the `Stack` construct to specify AWS environment values for your stack. The following is an example:

Example

TypeScript

```
const envEU  = { account: '2383838383', region: 'eu-west-1' };
```

```
const envUSA = { account: '8373873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

JavaScript

```
const envEU  = { account: '2383838383', region: 'eu-west-1' };
const envUSA = { account: '8373873873', region: 'us-west-2' };

new MyFirstStack(app, 'first-stack-us', { env: envUSA });
new MyFirstStack(app, 'first-stack-eu', { env: envEU });
```

Python

```
env_EU = cdk.Environment(account="8373873873", region="eu-west-1")
env_USA = cdk.Environment(account="2383838383", region="us-west-2")

MyFirstStack(app, "first-stack-us", env=env_USA)
MyFirstStack(app, "first-stack-eu", env=env_EU)
```

Java

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv("8373873873", "eu-west-1");
        Environment envUSA = makeEnv("2383838383", "us-west-2");

        new MyFirstStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyFirstStack(app, "first-stack-eu", StackProps.builder()
```

```
                .env(envEU).build());

        app.synth();
    }
}
```

C#

```
Amazon.CDK.Environment makeEnv(string account, string region)
{
    return new Amazon.CDK.Environment
    {
        Account = account,
        Region = region
    };
}

var envEU = makeEnv(account: "8373873873", region: "eu-west-1");
var envUSA = makeEnv(account: "2383838383", region: "us-west-2");

new MyFirstStack(app, "first-stack-us", new StackProps { Env=envUSA });
new MyFirstStack(app, "first-stack-eu", new StackProps { Env=envEU });
```

Go

```
env_EU := awscdk.Environment{
 Account: jsii.String("8373873873"),
 Region:  jsii.String("eu-west-1"),
}

env_USA := awscdk.Environment{
 Account: jsii.String("2383838383"),
 Region:  jsii.String("us-west-2"),
}

MyFirstStack(app, "first-stack-us", &awscdk.StackProps{
 Env: &env_USA,
})

MyFirstStack(app, "first-stack-eu", &awscdk.StackProps{
 Env: &env_EU,
})
```

We recommend this approach for production environments. By explicitly specifying the environment in this way, you can ensure that the stack is always deployed to the specific environment.

Specify environments using environment variables

The AWS CDK provides two environment variables that you can use within your CDK code: `CDK_DEFAULT_ACCOUNT` and `CDK_DEFAULT_REGION`. When you use these environment variables within the `env` property of your stack instance, you can pass environment information from your credentials and configuration files using the CDK CLI `--profile` option.

The following is an example of how to specify these environment variables:

Example

TypeScript

Access environment variables via Node's `process` object.

> **Note**
>
> You need the `DefinitelyTyped` module to use `process` in TypeScript. `cdk init` installs this module for you. However, you should install this module manually if you are working with a project created before it was added, or if you didn't set up your project using `cdk init`.
>
> ```
> npm install @types/node
> ```

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

JavaScript

Access environment variables via Node's `process` object.

```
new MyDevStack(app, 'dev', {
```

```
  env: {
    account: process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEFAULT_REGION
}});
```

Python

Use the `os` module's `environ` dictionary to access environment variables.

```
import os
MyDevStack(app, "dev", env=cdk.Environment(
    account=os.environ["CDK_DEFAULT_ACCOUNT"],
    region=os.environ["CDK_DEFAULT_REGION"]))
```

Java

Use `System.getenv()` to get the value of an environment variable.

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") :
 account;
        region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv(null, null);
        Environment envUSA = makeEnv(null, null);

        new MyDevStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyDevStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());
```

```
        app.synth();
    }
}
```

C#

Use `System.Environment.GetEnvironmentVariable()` to get the value of an environment variable.

```
Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
 System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
 System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}

new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

Go

```
import "os"

MyDevStack(app, "dev", &awscdk.StackProps{
 Env: &awscdk.Environment{
  Account: jsii.String(os.Getenv("CDK_DEFAULT_ACCOUNT")),
  Region:  jsii.String(os.Getenv("CDK_DEFAULT_REGION")),
    },
})
```

By specifying environments using environment variables, you can have the same CDK stack synthesize to AWS CloudFormation templates for different environments. This means that you can deploy the same CDK stack to different AWS environments without having to modify your CDK code. You only have to specify the profile to use when running `cdk synth`.

This approach is great for development environments when deploying the same stack to different environments. However, we do not recommend this approach for production environments since the same CDK code can synthesize different templates, depending on the machine, user, or session that it's synthesized under.

Specify environments from your credentials and configuration files with the CDK CLI

When deploying an environment-agnostic template, use the `--profile` option with any CDK CLI command to specify the profile to use. The following is an example that deploys a CDK stack named `myStack` using the `prod` profile that is defined in the `credentials` and `config` files:

```
$ cdk deploy <myStack> --profile <prod>
```

For more information on the `--profile` option, along with other CDK CLI commands and options, see AWS CDK CLI command reference.

Considerations when configuring environments with the AWS CDK

Services that you define by using constructs within your stacks must support the Region that you are deploying to. For a list of supported AWS services per region, see AWS Services by Region.

You must have valid AWS Identity and Access Management (IAM) credentials to perform stack deployments with the AWS CDK into your specified environments.

Examples

Synthesize an environment–agnostic CloudFormation template from a CDK stack

In this example, we create an environment-agnostic CloudFormation template from our CDK stack. We can then deploy this template to any environment.

The following is our example CDK stack. This stack defines an Amazon S3 bucket and a CloudFormation stack output for the bucket's Region. For this example, `env` is not defined:

Example

TypeScript

```
export class CdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the S3 bucket
    const bucket = new s3.Bucket(this, 'amzn-s3-demo-bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });

    // Create an output for the bucket's Region
    new cdk.CfnOutput(this, 'BucketRegion', {
      value: bucket.env.region,
    });
  }
}
```

JavaScript

```
class CdkAppStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Create the S3 bucket
    const bucket = new s3.Bucket(this, 'amzn-s3-demo-bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });

    // Create an output for the bucket's Region
    new cdk.CfnOutput(this, 'BucketRegion', {
      value: bucket.env.region,
    });
  }
}
```

Python

```
class CdkAppStack(cdk.Stack):

    def __init__(self, scope: cdk.Construct, id: str, **kwargs) -> None:
```

```
        super().__init__(scope, id, **kwargs)

        # Create the S3 bucket
        bucket = s3.Bucket(self, 'amzn-s3-demo-bucket',
            removal_policy=cdk.RemovalPolicy.DESTROY
        )

        # Create an output for the bucket's Region
        cdk.CfnOutput(self, 'BucketRegion',
            value=bucket.env.region
        )
```

Java

```
public class CdkAppStack extends Stack {

    public CdkAppStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        // Create the S3 bucket
        Bucket bucket = Bucket.Builder.create(this, "amzn-s3-demo-bucket")
            .removalPolicy(RemovalPolicy.DESTROY)
            .build();

        // Create an output for the bucket's Region
        CfnOutput.Builder.create(this, "BucketRegion")
            .value(this.getRegion())
            .build();
    }
}
```

C#

```
namespace MyCdkApp
{
    public class CdkAppStack : Stack
    {
        public CdkAppStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
        {
            // Create the S3 bucket
            var bucket = new Bucket(this, "amzn-s3-demo-bucket", new BucketProps
```

```
            {
                RemovalPolicy = RemovalPolicy.DESTROY
            });

            // Create an output for the bucket's Region
            new CfnOutput(this, "BucketRegion", new CfnOutputProps
            {
                Value = this.Region
            });
        }
    }
}
```

Go

```
func NewCdkAppStack(scope constructs.Construct, id string, props *CdkAppStackProps)
 awscdk.Stack {
 stack := awscdk.NewStack(scope, &id, &props.StackProps)

 // Create the S3 bucket
 bucket := awss3.NewBucket(stack, jsii.String("amzn-s3-demo-bucket"),
 &awss3.BucketProps{
  RemovalPolicy: awscdk.RemovalPolicy_DESTROY,
 })

 // Create an output for the bucket's Region
 awscdk.NewCfnOutput(stack, jsii.String("BucketRegion"), &awscdk.CfnOutputProps{
  Value: stack.Region(),
 })

 return stack
}
```

When we run `cdk synth`, the CDK CLI produces a CloudFormation template with the pseudo parameter `AWS::Region` as the output value for the bucket's Region. This parameter will be resolved at deployment:

```
Outputs:
  BucketRegion:
    Value:
      Ref: AWS::Region
```

To deploy this stack to an environment that is specified in the `dev` profile of our credentials and configuration files, we run the following:

```
$ cdk deploy CdkAppStack --profile dev
```

If we don't specify a profile, the CDK CLI will attempt to use environment information from the `default` profile in our credentials and configuration files.

Use logic to determine environment information at template synthesis

In this example, we configure the `env` property of our `stack` instance to use a valid expression. We specify two additional environment variables, `CDK_DEPLOY_ACCOUNT` and `CDK_DEPLOY_REGION`. These environment variables can override defaults at synthesis time if they exist:

Example

TypeScript

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEPLOY_ACCOUNT || process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEPLOY_REGION || process.env.CDK_DEFAULT_REGION
}});
```

JavaScript

```
new MyDevStack(app, 'dev', {
  env: {
    account: process.env.CDK_DEPLOY_ACCOUNT || process.env.CDK_DEFAULT_ACCOUNT,
    region: process.env.CDK_DEPLOY_REGION || process.env.CDK_DEFAULT_REGION
}});
```

Python

```
MyDevStack(app, "dev", env=cdk.Environment(
    account=os.environ.get("CDK_DEPLOY_ACCOUNT", os.environ["CDK_DEFAULT_ACCOUNT"]),
    region=os.environ.get("CDK_DEPLOY_REGION", os.environ["CDK_DEFAULT_REGION"])
    )
)
```

Java

```
public class MyApp {

    // Helper method to build an environment
    static Environment makeEnv(String account, String region) {
        account = (account == null) ? System.getenv("CDK_DEPLOY_ACCOUNT") : account;
        region = (region == null) ? System.getenv("CDK_DEPLOY_REGION") : region;
        account = (account == null) ? System.getenv("CDK_DEFAULT_ACCOUNT") :
 account;
        region = (region == null) ? System.getenv("CDK_DEFAULT_REGION") : region;

        return Environment.builder()
                .account(account)
                .region(region)
                .build();
    }

    public static void main(final String argv[]) {
        App app = new App();

        Environment envEU = makeEnv(null, null);
        Environment envUSA = makeEnv(null, null);

        new MyDevStack(app, "first-stack-us", StackProps.builder()
                .env(envUSA).build());
        new MyDevStack(app, "first-stack-eu", StackProps.builder()
                .env(envEU).build());

        app.synth();
    }
}
```

C#

```
Amazon.CDK.Environment makeEnv(string account=null, string region=null)
{
    return new Amazon.CDK.Environment
    {
        Account = account ??
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_ACCOUNT") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_ACCOUNT"),
        Region = region ??
```

```
            System.Environment.GetEnvironmentVariable("CDK_DEPLOY_REGION") ??
            System.Environment.GetEnvironmentVariable("CDK_DEFAULT_REGION")
    };
}

new MyDevStack(app, "dev", new StackProps { Env = makeEnv() });
```

Go

```
var account, region string
var b bool

if account, b = os.LookupEnv("CDK_DEPLOY_ACCOUNT"); !b || len(account) == 0 {
 account = os.Getenv("CDK_DEFAULT_ACCOUNT")
}
if region, b = os.LookupEnv("CDK_DEPLOY_REGION"); !b || len(region) == 0 {
 region = os.Getenv("CDK_DEFAULT_REGION")
}

MyDevStack(app, "dev", &awscdk.StackProps{
 Env: &awscdk.Environment{
  Account: &account,
  Region:  &region,
 },
})
```

With our stack's environment declared this way, we can then write a short script or batch file and set variables from command line arguments, then call `cdk deploy`. The following is an example. Any arguments beyond the first two are passed through to `cdk deploy` to specify command line options or arguments:

Example

macOS/Linux

```
#!/usr/bin/env bash
if [[ $# -ge 2 ]]; then
    export CDK_DEPLOY_ACCOUNT=$1
    export CDK_DEPLOY_REGION=$2
    shift; shift
    npx cdk deploy "$@"
```

```
    exit $?
else
    echo 1>&2 "Provide account and region as first two args."
    echo 1>&2 "Additional args are passed through to cdk deploy."
    exit 1
fi
```

Save the script as `cdk-deploy-to.sh`, then execute `chmod +x cdk-deploy-to.sh` to make it executable.

Windows

```
@findstr /B /V @ %~dpnx0 > %~dpn0.ps1 && powershell -ExecutionPolicy Bypass
 %~dpn0.ps1 %*
@exit /B %ERRORLEVEL%
if ($args.length -ge 2) {
    $env:CDK_DEPLOY_ACCOUNT, $args = $args
    $env:CDK_DEPLOY_REGION,  $args = $args
    npx cdk deploy $args
    exit $lastExitCode
} else {
    [console]::error.writeline("Provide account and region as first two args.")
    [console]::error.writeline("Additional args are passed through to cdk deploy.")
    exit 1
}
```

The Windows version of the script uses PowerShell to provide the same functionality as the macOS/Linux version. It also contains instructions to allow it to be run as a batch file so it can be easily invoked from a command line. It should be saved as `cdk-deploy-to.bat`. The file `cdk-deploy-to.ps1` will be created when the batch file is invoked.

We can then write additional scripts that use the `cdk-deploy-to` script to deploy to specific environments. The following is an example:

Example

macOS/Linux

```
#!/usr/bin/env bash
# cdk-deploy-to-test.sh
./cdk-deploy-to.sh 123457689 us-east-1 "$@"
```

Windows

```
@echo off
rem cdk-deploy-to-test.bat
cdk-deploy-to 135792469 us-east-1 %*
```

The following is an example that uses the `cdk-deploy-to` script to deploy to multiple environments. If the first deployment fails, the process stops:

Example

macOS/Linux

```
#!/usr/bin/env bash
# cdk-deploy-to-prod.sh
./cdk-deploy-to.sh 135792468 us-west-1 "$@" || exit
./cdk-deploy-to.sh 246813579 eu-west-1 "$@"
```

Windows

```
@echo off
rem cdk-deploy-to-prod.bat
cdk-deploy-to 135792469 us-west-1 %* || exit /B
cdk-deploy-to 245813579 eu-west-1 %*
```

AWS CDK telemetry

AWS Cloud Development Kit (AWS CDK) applications are configured to collect and report on usage data to gain insight into how the AWS CDK is being used. To understand the data that is collected, how it's used, and how to opt in or out of data reporting, see Configure AWS CDK Library usage data reporting.

We are not currently collecting or reporting AWS CDK CLI usage data. However, when AWS CDK CLI telemetry is released, customers will be opted in to CDK CLI usage data collection and reporting by default. To learn how to opt out of CDK CLI telemetry in advance of its release, see Configure AWS CDK CLI telemetry.

Configure AWS CDK Library usage data reporting

What is CDK library usage data reporting?

AWS Cloud Development Kit (AWS CDK) applications are configured to collect and report on usage data to gain insight into how the AWS CDK is being used. The CDK team uses this data to do the following:

- **Communicate with customers** – Identify stacks using a construct with known security or reliability issues and send out communication on customer-impacting topics.
- **Inform CDK development** – Gain insight on CDK usage to better inform CDK development.
- **Investigate CDK issues** – When bugs are reported, usage data provides valuable insight to the CDK team when troubleshooting.

What usage data is collected?

There are two categories of application usage data collected by the CDK:

- General usage data
- Additional usage data

General usage data collection

The CDK collects the following types of general usage data from your CDK applications:

- Versions of CDK libraries used.
- Names of constructs used from the following NPM modules:
  - AWS CDK core modules
  - AWS Construct Library modules
  - AWS Solutions Constructs module
  - AWS Render Farm Deployment Kit module

> **Note**
>
> Before version 1.93.0, the AWS CDK reported the names and versions of the modules loaded during synthesis, instead of the constructs used in the stack.

Additional usage data collection

Starting with CDK version 2.178.0, usage data collection expanded to include the following additional usage data:

- **CDK–defined property keys –** When you use a built-in property of an L2 construct, the property key will be collected. This includes built-in property keys nested in dictionary objects.
- **Boolean and enum property values from CDK–defined property keys –** For CDK–defined property keys, property values of only Boolean and enum types will be collected. All other types, such as string values or construct references will be redacted.
- **Method name, keys, and property values of Boolean and enum types –** When you use an L2 construct method, we will collect the method name, property keys, and property values of Boolean and enum types.

For property keys and values that you uniquely create, the entire object will be redacted. For example, if you use `InlineApiDefinition` to define an OpenAPI specification and pass it into a `RestApi` construct, the entire `InlineApiDefinition` object will be redacted.

For more information on additional usage data collection, including its benefits and potential concerns, see the CDK Collecting Additional Metadata (under feature flag) #33198 discussion in the *aws-cdk repository*.

How the CDK collects application usage data

At synthesis, the CDK collects usage data from your application and stores it within the `AWS::CDK::Metadata` resource. The following is an example of this on a synthesized AWS CloudFormation template:

```
CDKMetadata:
  Type: "AWS::CDK::Metadata"
  Properties:
    Analytics:
 "v2:deflate64:H4sIAND9SGAAAzXKSw5AMBAA0L1b2PdzBYnEAdio3RglglY60zQi7u6TWL/
XKmNUlxeQSOKwaPTBqrNhwEWU3hGHiCzK0dWWfAxoL/Fd8mvk+QkS/0X6BdjnCdgmOOQKWz
+AqqLDt2Y3YMnLYWwAAAA="
```

The `Analytics` property is a gzipped, base64-encoded, prefix-encoded list of the constructs in the stack.

How to opt out or opt in to usage data reporting

Your options to opt out or opt in to general usage data reporting and additional usage data reporting depends on the CDK version you used to originally create your app.

By default, CDK applications are configured to automatically opt in to usage data reporting as follows:

- **All CDK apps –** Opted in to general usage data reporting.
- **CDK apps created with a version older than v2.178.0 –** Not automatically opted in to additional usage data reporting.
- **CDK apps created with v2.178.0 or newer –** Opted in to additional usage data reporting.

> **Warning**
>
> If you choose to opt out, the CDK will not be able to identify if you've been impacted by security issues and will not send you notifications for them.

Opt out of all usage data reporting

To opt out of all usage data reporting, you can use the AWS Cloud Development Kit (AWS CDK) Command Line Interface (CLI) or configure your project's `cdk.json` file.

To opt out of all usage data reporting using the CDK CLI

- Use the `--no-version-reporting` option with any CDK CLI command to opt out for a single command. The following is an example of opting out during template synthesis:

  ```
  $ cdk synth --no-version-reporting
  ```

  Since the CDK synthesizes templates automatically when you run `cdk deploy`, you should also use `--no-version-reporting` with the `cdk deploy` command.

To opt out of all usage data reporting by configuring the cdk.json file

1. Set `versionReporting` to `false` in `./cdk.json` or `~/.cdk.json`. This opts you out by default. The following is an example:

   ```
   {
     "app": "...",
     "versionReporting": false
   }
   ```

2. After configuring, you can override this behavior and opt in by specifying `--version-reporting` on an individual command.

Opt out of only the additional usage data reporting

If your CDK app was created with a CDK version older than 2.178.0, you are automatically opted out of additional usage data reporting, even if you are opted in to general usage data reporting. You don't have to do anything to opt out of additional usage data reporting.

If your CDK app was created with CDK version 2.178.0 or newer, you must opt out of all usage data reporting. You can't opt out of only the additional usage data reporting.

Opt in to usage data reporting

If your CDK app was created with CDK version 2.178.0 or newer, you can opt in to all usage data reporting by setting `versionReporting` to `true`. This is the default behavior of CDK apps.

If your CDK app was created with a CDK version older than 2.178.0, you can opt in to general usage data reporting by setting `versionReporting` to `true`. To opt in to additional usage data reporting, you must enable a feature flag.

> **Note**
>
> These steps are for CDK apps originally created with a version older than 2.178.0

1. Verify that you are now using CDK version 2.178.0 or newer.
2. In your CDK configuration file, specify `@aws-cdk/core:enableAdditionalMetadataCollection` as `true`. The following is an example:

```
{
  "context": {
    "@aws-cdk/core:enableAdditionalMetadataCollection": "true"
  }
}
```

3. You can also use the `ENABLE_ADDITIONAL_METADATA_COLLECTION` value with the [`FeatureFlags`](#) class. The following is an example:

```
import * as cdk from 'aws-cdk-lib';
import { FeatureFlags } from 'aws-cdk-lib';
import * as cx_api from 'aws-cdk-lib/cx-api';
import { Construct } from 'constructs';

export class MyStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Set the feature flag ENABLE_ADDITIONAL_METADATA_COLLECTION to true
    FeatureFlags.of(this).add(cx_api.ENABLE_ADDITIONAL_METADATA_COLLECTION, true);

    // Your stack resources go here
    new cdk.aws_s3.Bucket(this, 'MyBucket');
  }
}

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

Examples

General and additional usage data collected from a CDK application

The following is an example of a CDK app:

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

class MyStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Create an S3 bucket (L2 construct)
    const myBucket = new s3.Bucket(this, 'MyBucket', {
      bucketName: 'my-cdk-example-bucket', // String type
      versioned: true,                     // Boolean type
      removalPolicy: cdk.RemovalPolicy.DESTROY, // ENUM type
      lifecycleRules: [{                   // Array of object type
        expirationDate: new Date('2019-10-01'),
        objectSizeLessThan: 600,
        objectSizeGreaterThan: 500,
      }],
    });

    // Use a method of the L2 construct to define additional properties
    myBucket.addLifecycleRule({
      id: 'ExpireOldObjects',
      enabled: true, // Boolean
      expiration: cdk.Duration.days(90), // Expire objects after 90 days
    });
  }
}

// Define the CDK app and stack
const app = new cdk.App();
new MyStack(app, 'MyStack');
app.synth();
```

At synthesis, usage data is collected, compressed, and stored in the `AWS::CDK::Metadata` resource.

The following is an example of general usage data collected with a CDK version older than 2.178.0:

```
{
    "fqn": "aws-cdk-lib.aws-s3.Bucket",
    "version": "v2.170.0"
}
```

The following is an example of usage data collected, including the additional usage data introduced in CDK version 2.178.0:

```
{
    "fqn": "aws-cdk-lib.aws_s3.Bucket",
    "version": "2.170.0",
    "metadata": [
        {
            "type": "aws:cdk:analytics:construct",
            "data": {
                "bucketName": "*",
                "versioned": true,
                "removalPolicy": "cdk.RemovalPolicy.DESTROY",
                "lifecycleRules": [
                    {
                        "expirationDate": "*",
                        "objectSizeLessThan": "*",
                        "objectSizeGreaterThan": "*"
                    }
                ]
            }
        },
        {
            "type": "aws:cdk:analytics:method",
            "data": {
                "name": "addLifecycleRule",
                "prop": {
                    "id": "*",
                    "enabled": true,
                    "expiration": "*",
                }
            }
        }
    ]
}
```

Configure AWS CDK CLI telemetry

What is CDK CLI telemetry?

Important

AWS CDK CLI telemetry has not been released. We are not collecting or reporting CDK CLI usage data. When the feature is released, this documentation will be updated with information about the data that is being collected.

When AWS CDK CLI telemetry is released, customers are opted in to CLI usage data collection and reporting by default. You can opt out of CDK CLI telemetry in advance of its release using the CDK CLI, context values, or an environment variable.

How to opt out of CDK CLI telemetry

To view your current telemetry status based on context values and environment variables, run the following command:

```
cdk cli-telemetry --status
```

Opt out using context values

If there are conflicts in the context values from various sources, the CDK uses the following order of precedence.

1. Context values specified using a CDK CLI command with the `--context` option
2. Context values in cdk.json
3. Context values in cdk.context.json
4. Context values in ~/.cdk.json

To opt out using a CDK CLI command

Run the following CDK CLI command to disable telemetry:

```
cdk cli-telemetry --disable
```

Running this command records the action to your local `cdk.context.json` file and applies to the current CDK application only.

To opt out using the application configuration file

Set `cli-telemetry: false` in an application's `cdk.json` file. This disables telemetry for the application.

To opt out using the global configuration file

Set `cli-telemetry: false` in the `~/.cdk.json` file to disable telemetry for all CDK applications on the same computer.

Opt out using an environment variable

Set the `CDK_DISABLE_CLI_TELEMETRY` environment variable to `true` to disable CLI telemetry in the that environment.

```
CDK_DISABLE_CLI_TELEMETRY=true
```

The value of this environment variable takes precendence over any context values that have been set.

> **Tip**
>
> Add this environment variable to your shell startup file, such as `.bashrc`, to disable telemetry every time you open a terminal window.

Opt out of all usage data reporting

To opt out of all usage data reporting for a single command, use the `--no-version-reporting` global CLI option. This option turns off both application usage data collection and CDK CLI telemetry.

If you are currently opting out of application usage data collection using the `--no-version-reporting` option, you are automatically opted out of CDK CLI telemetry for that specific command execution, regardless of context values.

> **Note**
>
> If you set `analyticsReporting` at the stack level, it takes precedence over the `--no-version-reporting` option for application usage data collection only.

Send telemetry data to a local file

If you choose to stay opted in to CDK CLI telemetry, you can choose to send your telemetry data to a local file. You can then review the file to audit the telemetry data we are collecting.

Use the following option to send telemetry data to a local file:

```
--telemetry-file='path/to/local/file'
```

> **Note**
>
> The `--telemetry-file` option is being released prior to the launch of the telemetry feature. It is considered production-ready and safe to use. However, you must opt in by providing the `--unstable` option. The `--unstable` flag will be ignored after launch.

The following command sends telemetry data to a local file:

```
cdk deploy --unstable=telemetry --telemetry-file='path/to/local/file'
```

Bootstrap your environment for use with the AWS CDK

Bootstrap your AWS environment to prepare it for AWS Cloud Development Kit (AWS CDK) stack deployments.

- For an introduction to environments, see Environments for the AWS CDK.
- For an introduction to bootstrapping, see AWS CDK bootstrapping.

How to bootstrap your environment

You can use the AWS CDK Command Line Interface (AWS CDK CLI) or your preferred AWS CloudFormation deployment tool to bootstrap your environment.

Use the CDK CLI

You can use the CDK CLI `cdk bootstrap` command to bootstrap your environment. This is the method that we recommend if you don't require significant modifications to bootstrapping.

Bootstrap from any working directory

To bootstrap from any working directory, provide the environment to bootstrap as a command line argument. The following is an example:

```
$ cdk bootstrap <aws://123456789012/us-east-1>
```

> **Tip**
>
> If you don't have your AWS account number, you can get it from the AWS Management Console. You can also use the following AWS CLI command to display your default account information, including your account number:
>
> ```
> $ aws sts get-caller-identity
> ```
>
> If you have named profiles in your AWS `config` and `credentials` files, use the `--profile` option to retrieve account information for a specific profile. The following is an example:
>
> ```
> $ aws sts get-caller-identity --profile <prod>
> ```

To display the default Region, use the `aws configure get` command:

```
$ aws configure get region
$ aws configure get region --profile <prod>
```

When providing an argument, the `aws://` prefix is optional. The following is valid:

```
$ cdk bootstrap <123456789012/us-east-1>
```

To bootstrap multiple environments at the same time, provide multiple arguments:

```
$ cdk bootstrap <aws://123456789012/us-east-1> <aws://123456789012/us-east-2>
```

Bootstrap from the parent directory of a CDK project

You can run `cdk bootstrap` from the parent directory of a CDK project containing a `cdk.json` file. If you don't provide an environment as an argument, the CDK CLI will obtain environment information from default sources, such as your `config` and `credentials` files or any environment information specified for your CDK stack.

When you bootstrap from the parent directory of a CDK project, environments provided from command line arguments take precedence over other sources.

To bootstrap an environment that is specified in your `config` and `credentials` files, use the `--profile` option:

```
$ cdk bootstrap --profile <prod>
```

For more information on the `cdk bootstrap` command and supported options, see [cdk bootstrap](#).

Use any AWS CloudFormation tool

You can copy the [bootstrap template](#) from the *aws-cdk-cli GitHub repository* or obtain the template with the `cdk bootstrap --show-template` command. Then, use any AWS CloudFormation tool to deploy the template into your environment.

With this method, you can use AWS CloudFormation StackSets or AWS Control Tower. You can also use the AWS CloudFormation console or the AWS Command Line Interface (AWS CLI). You can make modifications to your template before you deploy it. This method may be more flexible and suitable for large-scale deployments.

The following is an example of using the `--show-template` option to retrieve and save the bootstrap template to your local machine:

Example

macOS/Linux

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

Windows

On Windows, PowerShell must be used to preserve the encoding of the template.

```
powershell "cdk bootstrap --show-template | Out-File -encoding utf8 bootstrap-template.yaml"
```

Note

If CDK notices are appearing in your AWS CloudFormation template output, provide the `--no-notices` option with your command.

To deploy this template using the CDK CLI, you can run the following:

```
$ cdk bootstrap --template <bootstrap-template.yaml>
```

The following is an example of using the AWS CLI to deploy the template:

Example

macOS/Linux

```
aws cloudformation create-stack \
  --stack-name CDKToolkit \
  --template-body file://<path/to/>bootstrap-template.yaml \
  --capabilities CAPABILITY_NAMED_IAM \
```

```
  --region <us-west-1>
```

Windows

```
aws cloudformation create-stack ^
  --stack-name CDKToolkit ^
  --template-body file://<path/to/>bootstrap-template.yaml ^
  --capabilities CAPABILITY_NAMED_IAM ^
  --region <us-west-1>
```

For information on using CloudFormation StackSets to bootstrap multiple environments, see Bootstrapping multiple AWS accounts for AWS CDK using CloudFormation StackSets in the *AWS Cloud Operations & Migrations Blog*.

When to bootstrap your environment

You must bootstrap each AWS environment before you deploy into the environment. We recommend that you proactively bootstrap each environment that you plan to use. You can do this before you plan on actually deploying CDK apps into the environment. By proactively bootstrapping your environments, you prevent potential future issues such as Amazon S3 bucket name conflicts or deploying CDK apps into environments that haven't been bootstrapped.

It's okay to bootstrap an environment more than once. If an environment has already been bootstrapped, the bootstrap stack will be upgraded if necessary. Otherwise, nothing will happen.

If you attempt to deploy a CDK stack into an environment that hasn't been bootstrapped, you will see an error like the following:

```
$ cdk deploy

#  Synthesis time: 2.02s

 # Deployment failed: Error: BootstrapExampleStack: SSM parameter /cdk-bootstrap/
hnb659fds/version not found. Has the environment been bootstrapped? Please run 'cdk
 bootstrap' (see https://docs.aws.amazon.com/cdk/latest/guide/bootstrapping.html)
```

Update your bootstrap stack

Periodically, the CDK team will update the bootstrap template to a new version. When this happens, we recommend that you update your bootstrap stack. If you haven't customized the bootstrapping process, you can update your bootstrap stack by following the same steps that you took to originally bootstrap your environment. For more information, see Bootstrap template version history.

Default resources created during bootstrapping

IAM roles created during bootstrapping

By default, bootstrapping provisions the following AWS Identity and Access Management (IAM) roles in your environment:

- `CloudFormationExecutionRole`
- `DeploymentActionRole`
- `FilePublishingRole`
- `ImagePublishingRole`
- `LookupRole`

`CloudFormationExecutionRole`

This IAM role is a CloudFormation service role that grants CloudFormation permission to perform stack deployments on your behalf. This role gives CloudFormation permission to perform AWS API calls in your account, including deploying stacks.

By using a service role, the permissions provisioned for the service role determine what actions can be performed on your CloudFormation resources. Without this service role, the security credentials you provide with the CDK CLI would determine what CloudFormation is allowed to do.

`DeploymentActionRole`

This IAM role grants permission to perform deployments into your environment. It is assumed by the CDK CLI during deployments.

By using a role for deployments, you can perform cross-account deployments since the role can be assumed by AWS identities in a different account.

`FilePublishingRole`

This IAM role grants permission to perform actions against the bootstrapped Amazon Simple Storage Service (Amazon S3) bucket, including uploading and deleting assets. It is assumed by the CDK CLI during deployments.

`ImagePublishingRole`

This IAM role grants permission to perform actions against the bootstrapped Amazon Elastic Container Registry (Amazon ECR) repository. It is assumed by the CDK CLI during deployments.

`LookupRole`

This IAM role grants `readOnly` permission to look up context values from the AWS environment. It is assumed by the CDK CLI when performing tasks such as template synthesis and deployments.

Resource IDs created during bootstrapping

When you deploy the default bootstrap template, physical IDs for bootstrap resources are created using the following structure: `cdk-<qualifier>-<description>-<account-ID>-<Region>`.

- **Qualifier** – A nine character unique string value of `hnb659fds`. The actual value has no significance.
- **Description** – A short description of the resource. For example, `container-assets`.
- **Account ID** – The AWS account ID of the environment.
- **Region** – The AWS Region of the environment.

The following is an example physical ID of the Amazon S3 staging bucket created during bootstrapping: `cdk-hnb659fds-assets-012345678910-us-west-1`.

Permissions to use when bootstrapping your environment

When bootstrapping an AWS environment, the IAM identity performing the bootstrapping must have at least the following permissions:

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

```
                "ecr:*",
                "ssm:*",
                "s3:*",
                "iam:*"
            ],
            "Resource": "*"
        }
    ]
}
```

Over time, the bootstrap stack, including the resources that are created and permissions they require, may change. With future changes, you may need to modify the permissions required to bootstrap an environment.

Customize bootstrapping

If the default bootstrap template doesn't suit your needs, you can customize the bootstrapping of resources into your environment in the following ways:

- Use command line options with the `cdk bootstrap` command – This method is best for making small, specific changes that are supported through command line options.
- Modify the default bootstrap template and deploy it – This method is best for making complex changes or if you want complete control over the configuration of resources provisioned during bootstrapping.

For more information on customizing bootstrapping, see Customize AWS CDK bootstrapping.

Bootstrapping with CDK Pipelines

If you are using CDK Pipelines to deploy into another account's environment, and you receive a message like the following:

```
Policy contains a statement with one or more invalid principals
```

This error message means that the appropriate IAM roles do not exist in the other environment. The most likely cause is that the environment has not been bootstrapped. Bootstrap the environment and try again.

Protecting your bootstrap stack from deletion

If a bootstrap stack is deleted, the AWS resources that were originally provisioned in the environment to support CDK deployments will also be deleted. This will cause the pipeline to stop working. If this happens, there is no general solution for recovery.

After your environment is bootstrapped, do not delete and recreate the environment's bootstrap stack. Instead, try to update the bootstrap stack to a new version by running the `cdk bootstrap` command again.

To protect against accidental deletion of your bootstrap stack, we recommend that you provide the `--termination-protection` option with the `cdk bootstrap` command to enable termination protection. You can enable termination protection on new or existing bootstrap stacks. For instructions on enabling termination protection, see Enable termination protection for the bootstrap stack.

Bootstrap template version history

The bootstrap template is versioned and evolves over time with the AWS CDK itself. If you provide your own bootstrap template, keep it up to date with the canonical default template. You want to make sure that your template continues to work with all CDK features.

Note

Earlier versions of the bootstrap template created an AWS KMS key in each bootstrapped environment by default. To avoid charges for the KMS key, re-bootstrap these environments using `--no-bootstrap-customer-key`. The current default is no KMS key, which helps avoid these charges.

This section contains a list of the changes made in each version.

Template version	AWS CDK version	Changes
1	1.40.0	Initial version of template with Bucket, Key, Repository, and Roles.

Template version	AWS CDK version	Changes
2	1.45.0	Split asset publishing role into separate file and image publishing roles.
3	1.46.0	Add `FileAssetKeyArn` export to be able to add decrypt permissions to asset consumers.
4	1.61.0	AWS KMS permissions are now implicit via Amazon S3 and no longer require `FileAssetKeyArn` . Add `CdkBootstrapVersion` SSM parameter so the bootstrap stack version can be verified without knowing the stack name.
5	1.87.0	Deployment role can read SSM parameter.
6	1.108.0	Add lookup role separate from deployment role.
6	1.109.0	Attach `aws-cdk:bootstrap-role` tag to deployment, file publishing, and image publishing roles.

Template version	AWS CDK version	Changes
7	1.110.0	Deployment role can no longer read Buckets in the target account directly. (However, this role is effectively an administrator, and could always use its AWS CloudFormation permissions to make the bucket readable anyway).
8	1.114.0	The lookup role has full read-only permissions to the target environment, and has a `aws-cdk:bootstrap-role` tag as well.
9	2.1.0	Fixes Amazon S3 asset uploads from being rejected by commonly referenced encryption SCP.
10	2.4.0	Amazon ECR ScanOnPush is now enabled by default.
11	2.18.0	Adds policy allowing Lambda to pull from Amazon ECR repos so it survives re-bootstrapping.
12	2.20.0	Adds support for experimental `cdk import`.
13	2.25.0	Makes container images in bootstrap-created Amazon ECR repositories immutable.

Template version	AWS CDK version	Changes
14	2.34.0	Turns off Amazon ECR image scanning at the repository level by default to allow bootstrapping Regions that do not support image scanning.
15	2.60.0	KMS keys cannot be tagged.
16	2.69.0	Addresses Security Hub finding KMS.2.
17	2.72.0	Addresses Security Hub finding ECR.3.
18	2.80.0	Reverted changes made for version 16 as they don't work in all partitions and are are not recommended.
19	2.106.1	Reverted changes made to version 18 where AccessControl property was removed from the template. (#27964)
20	2.119.0	Add `ssm:GetParameters` action to the AWS CloudFormation deploy IAM role. For more information, see #28336.
21	2.149.0	Add condition to the file publishing role.

Template version	AWS CDK version	Changes
22	2.160.0	Add `sts:TagSession` permissions to the trust policy of bootstrap IAM roles.
23	2.161.0	Add `cloudformation:RollbackStack` and `cloudformation:ContinueUpdateRollback` permissions to the trust policy of the deploy IAM role. This provides permissions for the `cdk rollback` command.
24	2.165.0	Change the duration of days that noncurrent objects in the bootstrap bucket will be retained, from 365 to 30 days. Since the new `cdk gc` command introduces the ability to delete objects in the bootstrap bucket, this new behavior ensures that deleted objects remain in the bootstrap bucket for 30 days instead of 365 days. For more information on this change, see `aws-cdk` PR [#31949]().

Template version	AWS CDK version	Changes
25	2.165.0	Add support to the bootstrap bucket for the removal of incomplete multipart uploads. Incomplete multipart uploads will be deleted after 1 day. For more information on this change, see `aws-cdk` PR [#31956](#).
26	2.1002.0	Add two deletion-related policies (`UpdateReplacePolicy` and `DeletionPolicy` to the `FileAssetsBucketEncryptionKey`) resource. These policies ensure that the old AWS KMS key resource will be properly deleted when the bootstrap stack is updated or deleted. For more information on this change, see `aws-cdk-cli` PR [#100](#).
27	2.1003.0	Add new Amazon ECR resource policy to grant Amazon EMR Serverless specific permissions for retrieving container images. For more information on this change, see `aws-cdk-cli` PR [#112](#).

Upgrade from legacy to modern bootstrap template

The AWS CDK v1 supported two bootstrapping templates, legacy and modern. CDK v2 supports only the modern template. For reference, here are the high-level differences between these two templates.

Feature	Legacy (v1 only)	Modern (v1 and v2)
Cross-account deployments	Not allowed	Allowed
AWS CloudFormation Permissions	Deploys using current user's permissions (determined by AWS profile, environment variables, etc.)	Deploys using the permissions specified when the bootstrap stack was provisioned (for example, by using `--trust`)
Versioning	Only one version of bootstrap stack is available	Bootstrap stack is versioned; new resources can be added in future versions, and AWS CDK apps can require a minimum version
Resources*	Amazon S3 bucket	• Amazon S3 bucket • AWS KMS key • IAM roles • Amazon ECR repository • SSM parameter for versioning
AWS KMS key	IAM roles	Amazon ECR repository
Resource naming	Automatically generated	Deterministic
Bucket encryption	Default key	AWS managed key by default. You can customize to use a customer managed key.

- *We will add additional resources to the bootstrap template as needed.*

An environment that was bootstrapped using the legacy template must be upgraded to use the modern template for CDK v2 by re-bootstrapping. Re-deploy all AWS CDK applications in the environment at least once before deleting the legacy bucket.

Address Security Hub Findings

If you are using AWS Security Hub, you may see findings reported on some of the resources created by the AWS CDK bootstrapping process. Security Hub findings help you find resource configurations you should double-check for accuracy and safety. We have reviewed these specific resource configurations with AWS Security and are confident they do not constitute a security problem.

[KMS.2] IAM principals should not have IAM inline policies that allow decryption actions on all KMS keys

The *deploy role* (`DeploymentActionRole`) grants permission to read encrypted data, which is necessary for cross-account deployments with CDK Pipelines. Policies in this role do not grant permission to all data. It only grants permission to read encrypted data from Amazon S3 and AWS KMS, and only when those resources allow it through their bucket or key policy.

The following is a snippet of these two statements in the *deploy role* from the bootstrap template:

```
DeploymentActionRole:
    Type: AWS::IAM::Role
    Properties:
      ...
      Policies:
        - PolicyDocument:
            Statement:
              ...
              - Sid: PipelineCrossAccountArtifactsBucket
                Effect: Allow
                Action:
                  - s3:GetObject*
                  - s3:GetBucket*
                  - s3:List*
                  - s3:Abort*
                  - s3:DeleteObject*
                  - s3:PutObject*
```

```
        Resource: "*"
        Condition:
          StringNotEquals:
            s3:ResourceAccount:
              Ref: AWS::AccountId
      - Sid: PipelineCrossAccountArtifactsKey
        Effect: Allow
        Action:
          - kms:Decrypt
          - kms:DescribeKey
          - kms:Encrypt
          - kms:ReEncrypt*
          - kms:GenerateDataKey*
        Resource: "*"
        Condition:
          StringEquals:
            kms:ViaService:
              Fn::Sub: s3.${AWS::Region}.amazonaws.com
      ...
```

Why does Security Hub flag this?

The policies contain a `Resource: *` combined with a `Condition` clause. Security Hub flags the `*` wildcard. This wildcard is used because at the time the account is bootstrapped, the AWS KMS key created by CDK Pipelines for the CodePipeline artifact bucket does not exist yet, and therefore, can't be referenced on the bootstrap template by ARN. In addition, Security Hub does not consider the `Condition` clause when raising this flag. This `Condition` restricts `Resource: *` to requests made from the same AWS account of the AWS KMS key. These requests must come from Amazon S3 in the same AWS Region as the AWS KMS key.

Do I need to fix this finding?

As long as you have not modified the AWS KMS key on your bootstrap template to be overly permissive, the *deploy role* does not allow more access than it needs. Therefore, it is not necessary to fix this finding.

What if I want to fix this finding?

How you fix this finding depends on whether or not you will be using CDK Pipelines for cross-account deployments.

To fix the Security Hub finding and use CDK Pipelines for cross-account deployments

1. If you have not done so, deploy the CDK bootstrap stack using the `cdk bootstrap` command.
2. If you have not done so, create and deploy your CDK Pipeline. For instructions, see Continuous integration and delivery (CI/CD) using CDK Pipelines.
3. Obtain the AWS KMS key ARN of the CodePipeline artifact bucket. This resource is created during pipeline creation.
4. Obtain a copy of the CDK bootstrap template to modify it. The following is an example, using the AWS CDK CLI:

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

5. Modify the template by replacing `Resource: *` of the `PipelineCrossAccountArtifactsKey` statement with your ARN value.
6. Deploy the template to update your bootstrap stack. The following is an example, using the CDK CLI:

```
$ cdk bootstrap aws://<account-id>/<region> --template bootstrap-
template.yaml
```

To fix the Security Hub finding if you're not using CDK Pipelines for cross-account deployments

1. Obtain a copy of the CDK bootstrap template to modify it. The following is an example, using the CDK CLI:

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

2. Delete the `PipelineCrossAccountArtifactsBucket` and `PipelineCrossAccountArtifactsKey` statements from the template.
3. Deploy the template to update your bootstrap stack. The following is an example, using the CDK CLI:

```
$ cdk bootstrap aws://<account-id>/<region> --template bootstrap-
template.yaml
```

Considerations

Since bootstrapping provisions resources in your environment, you may incur AWS charges when those resources are used with the AWS CDK.

Customize AWS CDK bootstrapping

You can customize AWS Cloud Development Kit (AWS CDK) bootstrapping by using the AWS CDK Command Line Interface (AWS CDK CLI) or by modifying and deploying the AWS CloudFormation bootstrap template.

For an introduction to bootstrapping, see [AWS CDK bootstrapping](#).

Use the CDK CLI to customize bootstrapping

The following are a few examples of how you can customize bootstrapping by using the CDK CLI. For a list of all `cdk bootstrap` options, see [cdk bootstrap](#).

Override the name of the Amazon S3 bucket

Use the `--bootstrap-bucket-name` option to override the default Amazon S3 bucket name. This may require that you modify template synthesis. For more information, see [Customize CDK stack synthesis](#).

Modify server-side encryption keys for the Amazon S3 bucket

By default, the Amazon S3 bucket in the bootstrap stack is configure to use AWS managed keys for server-side encryption. To use an existing customer managed key, use the `--bootstrap-kms-key-id` option and provide a value for the AWS Key Management Service (AWS KMS) key to use. If you want more control over the encryption key, provide `--bootstrap-customer-key` to use a customer managed key.

Attach managed policies to the deployment role assumed by AWS CloudFormation

By default, stacks are deployed with full administrator permissions using the `AdministratorAccess` policy. To use your own managed policies, use the `--cloudformation-execution-policies` option and provide the ARNs of the managed policies to attach to the deployment role.

To provide multiple policies, pass them a single string, separated by commas:

```
$ cdk bootstrap --cloudformation-execution-policies "arn:aws:iam::aws:policy/
AWSLambda_FullAccess,arn:aws:iam::aws:policy/AWSCodeDeployFullAccess"
```

To avoid deployment failures, be sure that the policies you specify are sufficient for any deployments that you will perform into the environment being bootstrapped.

Change the qualifier that is added to the names of resources in your bootstrap stack

By default, the `hnb659fds` qualifier is added to the physical ID of resources in your bootstrap stack. To change this value, use the `--qualifier` option.

This modification is useful when provisioning multiple bootstrap stacks in the same environment to avoid name clashes.

Changing the qualifier is intended for name isolation between automated tests of the CDK itself. Unless you can very precisely scope down the IAM permissions given to the CloudFormation execution role, there are no permission isolation benefits to having two different bootstrap stacks in a single account. Therefore, there's usually no need to change this value.

When you change the qualifier, your CDK app must pass the changed value to the stack synthesizer. For more information, see Customize CDK stack synthesis.

Add tags to the bootstrap stack

Use the `--tags` option in the format of `KEY=VALUE` to add CloudFormation tags to your bootstrap stack.

Specify additional AWS accounts that can deploy into the environment being bootstrapped

Use the `--trust` option to provide additional AWS accounts that are allowed to deploy into the environment being bootstrapped. By default, the account performing the bootstrapping will always be trusted.

This option is useful when you are bootstrapping an environment that a CDK Pipeline from another environment will deploy into.

When you use this option, you must also provide `--cloudformation-execution-policies`.

To add trusted accounts to an existing bootstrap stack, you must specify all of the accounts to trust, including those that you may have previously provided. If you only provide new accounts to trust, the previously trusted accounts will be removed.

The following is an example that trusts two accounts:

```
$ cdk bootstrap aws://123456789012/us-west-2 --trust 234567890123 --trust
 987654321098 --cloudformation-execution-policies arn:aws:iam::aws:policy/
AdministratorAccess
 #  Bootstrapping environment aws://123456789012/us-west-2...
Trusted accounts for deployment: 234567890123, 987654321098
Trusted accounts for lookup: (none)
Execution policies: arn:aws:iam::aws:policy/AdministratorAccess
CDKToolkit: creating CloudFormation changeset...
 #  Environment aws://123456789012/us-west-2 bootstrapped.
```

Specify additional AWS accounts that can look up information in the environment being bootstrapped

Use the `--trust-for-lookup` option to specify AWS accounts that are allowed to look up context information from the environment being bootstrapped. This option is useful to give accounts permission to synthesize stacks that will be deployed into the environment, without actually giving them permission to deploy those stacks directly.

Enable termination protection for the bootstrap stack

If a bootstrap stack is deleted, the AWS resources that were originally provisioned in the environment will also be deleted. After your environment is bootstrapped, we recommend that you don’t delete and recreate the environment’s bootstrap stack, unless you are intentionally doing so. Instead, try to update the bootstrap stack to a new version by running the `cdk bootstrap` command again.

Use the `--termination-protection` option to manage termination protection settings for the bootstrap stack. By enabling termination protection, you prevent the bootstrap stack and its resources from being accidentally deleted. This is especially important if you use CDK Pipelines since there is no general recovery option if you accidentally delete the bootstrap stack.

After enabling termination protection, you can use the AWS CLI or AWS CloudFormation console to verify.

To enable termination protection

1. Run the following command to enable termination protection on a new or existing bootstrap stack:

```
$ cdk bootstrap --termination-protection
```

2. Use the AWS CLI or CloudFormation console to verify. The following is an example, using the AWS CLI. If you modified your bootstrap stack name, replace `CDKToolkit` with your stack name:

```
$ aws cloudformation describe-stacks --stack-name <CDKToolkit> --query
 "Stacks[0].EnableTerminationProtection"
true
```

Modify the default bootstrap template

When you need more customization than the CDK CLI can provide, you can modify the bootstrap template as needed. Then, deploy the template to bootstrap your environment.

To modify and deploy the default bootstrap template

1. Obtain the default bootstrap template using the `--show-template` option. By default, the CDK CLI will output the template in your terminal window. You can modify the CDK CLI command to save the template to your local machine. The following is an example:

```
$ cdk bootstrap --show-template > <my-bootstrap-template.yaml>
```

2. Modify the bootstrap template as needed. Any changes that you make should adhere to the bootstrapping template contract. For more information on the bootstrapping template contract, see Follow the bootstrap contract.

   To ensure that your customizations are not accidentally overwritten later by someone running `cdk bootstrap` using the default template, change the default value of the `BootstrapVariant` template parameter. The CDK CLI will only allow overwriting the bootstrap stack with templates that have the same `BootstrapVariant` and an equal or higher version than the template that is currently deployed.

3. Deploy your modified template using your preferred AWS CloudFormation deployment method. The following is an example that uses the CDK CLI:

```
$ cdk bootstrap --template <my-bootstrap-template.yaml>
```

Follow the bootstrap contract

For your CDK apps to properly deploy, the CloudFormation templates produced during synthesis must correctly specify the resources created during bootstrapping. These resources are commonly referred to as *bootstrap resources*. Bootstrapping creates resources in your AWS environment that are used by the AWS CDK to perform deployments and manage application assets. Synthesis produces CloudFormation templates from each CDK stack in your application. These templates don't just define the AWS resources that will be provisioned from your application. They also specify the bootstrap resources to use during deployment.

During synthesis, the CDK CLI doesn't know specifically how your AWS environment has been bootstrapped. Instead, the CDK CLI produces CloudFormation templates based on the synthesizer that you configure. Therefore, when you customize bootstrapping, you may need to customize synthesis. For instructions on customizing synthesis, see Customize CDK stack synthesis. The purpose is to ensure that your synthesized CloudFormation templates are compatible with your bootstrapped environment. This compatibility is referred to as the *bootstrap contract*.

The simplest method to customize stack synthesis is by modifying the `DefaultStackSynthesizer` class in your `Stack` instance. If you require customization beyond what this class can offer, you can write your own synthesizer as a class that implements `IStackSynthesizer` (perhaps deriving from `DefaultStackSynthesizer`).

When you customize bootstrapping, follow the bootstrap template contract to remain compatible with `DefaultStackSynthesizer`. If you modify bootstrapping beyond the bootstrap template contract, you will need to write your own synthesizer.

Versioning

The bootstrap template should contain a resource to create an Amazon EC2 Systems Manager (SSM) parameter with a well-known name and an output to reflect the template's version:

```
Resources:
  CdkBootstrapVersion:
    Type: AWS::SSM::Parameter
    Properties:
```

```
      Type: String
      Name:
        Fn::Sub: '/cdk-bootstrap/${Qualifier}/version'
      Value: 4
Outputs:
  BootstrapVersion:
    Value:
      Fn::GetAtt: [CdkBootstrapVersion, Value]
```

Roles

The `DefaultStackSynthesizer` requires five IAM roles for five different purposes. If you are not using the default roles, you must specify your IAM role ARNs within your `DefaultStackSynthesizer` object. The roles are as follows:

- The *deployment role* is assumed by the CDK CLI and by AWS CodePipeline to deploy into an environment. Its `AssumeRolePolicy` controls who can deploy into the environment. In the template, you can see the permissions that this role needs.
- The *lookup role* is assumed by the CDK CLI to perform context lookups in an environment. Its `AssumeRolePolicy` controls who can deploy into the environment. The permissions this role needs can be seen in the template.
- The *file publishing role* and the *image publishing role* are assumed by the CDK CLI and by AWS CodeBuild projects to publish assets into an environment. They're used to write to the Amazon S3 bucket and the Amazon ECR repository, respectively. These roles require write access to these resources.
- *The AWS CloudFormation execution role* is passed to AWS CloudFormation to perform the actual deployment. Its permissions are the permissions that the deployment will execute under. The permissions are passed to the stack as a parameter that lists managed policy ARNs.

Outputs

The CDK CLI requires that the following CloudFormation outputs exist on the bootstrap stack:

- `BucketName` – The name of the file asset bucket.
- `BucketDomainName` – The file asset bucket in domain name format.
- `BootstrapVersion` – The current version of the bootstrap stack.

Template history

The bootstrap template is versioned and evolves over time with the AWS CDK itself. If you provide your own bootstrap template, keep it up to date with the canonical default template. You want to make sure that your template continues to work with all CDK features. For more information, see Bootstrap template version history.

Create and apply permissions boundaries for the AWS CDK

A *permissions boundary* is an AWS Identity and Access Management (IAM) advanced feature that you can use to set the maximum permissions that an IAM entity, such as a user or role, can have. You can use permissions boundaries to restrict the actions that IAM entities can perform when using the AWS Cloud Development Kit (AWS CDK).

To learn more about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.

When to use permissions boundaries with the AWS CDK

Consider applying permissions boundaries when you need to restrict developers in your organization from performing certain actions with the AWS CDK. For example, if there are specific resources in your AWS environment that you don't want developers to modify, you can create and apply a permissions boundary.

How to apply permissions boundaries with the AWS CDK

Create the permissions boundary

First, you create the permissions boundary, using an AWS managed policy or a customer managed policy to set the boundary for an IAM entity (user or role). This policy limits the maximum permissions for the user or role. For instructions on creating permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.

Permissions boundaries set the maximum permissions that an IAM entity can have, but don't grant permissions on their own. You must use permissions boundaries with IAM policies to effectively limit and grant the proper permissions for your organization. You must also prevent IAM entities from being able to escape the boundaries that you set. For an example, see Delegating responsibility to others using permissions boundaries in the *IAM User Guide*.

Apply the permissions boundary during bootstrapping

After creating the permissions boundary, you can enforce it for the AWS CDK by applying it during bootstrapping.

Use the `--custom-permissions-boundary` option and specify the name of the permissions boundary to apply. The following is an example that applies a permissions boundary named `cdk-permissions-boundary`:

```
$ cdk bootstrap --custom-permissions-boundary <cdk-permissions-boundary>
```

By default, the CDK uses the `CloudFormationExecutionRole` IAM role, defined in the bootstrap template, to receive permissions for performing deployments. By applying the custom permissions boundary during bootstrapping, the permissions boundary gets attached to this role. The permissions boundary will then set the maximum permissions that can be performed by developers in your organization when using the AWS CDK. To learn more about this role, see IAM roles created during bootstrapping.

When you apply permissions boundaries in this way, they are applied to the specific environment that you bootstrap. To use the same permissions boundary across multiple environments, you must apply the permissions boundary for each environment during bootstrapping. You can also apply different permissions boundaries for different environments.

Learn more

For more information on permissions boundaries, see When and where to use IAM permissions boundaries in the *AWS Security Blog*.

Troubleshoot AWS CDK bootstrapping issues

Troubleshoot common issues when bootstrapping your environment with the AWS Cloud Development Kit (AWS CDK).

- For an introduction to bootstrapping, see AWS CDK bootstrapping.
- For instructions on bootstrapping, see Bootstrap your environment for use with the AWS CDK.

When bootstrapping using the default template, you get a 'CREATE_FAILED' error for the Amazon S3 bucket

When bootstrapping using the AWS CDK Command Line Interface (CDK CLI) `cdk bootstrap` command with the default bootstrap template, you receive the following error:

```
CREATE_FAILED | AWS::S3::Bucket | <BucketName> already exists
```

Before troubleshooting, ensure that you are using the latest version of the CDK CLI.

- To check your version, run `cdk --version`.
- To install the latest version, run `npm install -g aws-cdk`.

After installing the latest version, try bootstrapping your environment again. If you still receive the same error, continue with troubleshooting.

Common causes

When you bootstrap your environment, the AWS CDK generates physical IDs for your bootstrap resources. For more information, see [Resource IDs created during bootstrapping](#).

Unlike the other bootstrap resources, Amazon S3 bucket names are global. This means that each bucket name must be unique across all AWS accounts in all AWS Regions within a partition. For more information, see [Buckets overview](#) in the *Amazon S3 User Guide*. Therefore, the most common cause of this error is that the physical ID generated as your bucket name already exists somewhere within the partition. This could be within your account or another account.

The following is an example bucket name: `cdk-hnb659fds-assets-012345678910-us-west-1`. While unlikely, due to the qualifier and account ID being a part of the name, it is possible that this name for an Amazon S3 bucket is used by another AWS account. Since bucket names are globally scoped, it can't be used by you if its used by a different account in the same partition. Most likely, a bucket with the same name exists somewhere in your account. This could be in the Region you are attempting to bootstrap, or another Region.

Generally, the resolution is to locate this bucket in your account and determine what to do with it, or customize bootstrapping to create bootstrap resources of a different name.

Resolution

First, determine if a bucket with the same name exists within your AWS account. Using an AWS identity with valid permissions to lookup Amazon S3 buckets in your account, you can do this in the following ways:

- Use the AWS Command Line Interface (AWS CLI) `aws s3 ls` command to view a list of all your buckets.
- Look up bucket names for each Region in your account using the [Amazon S3 console](https://console.aws.amazon.com/s3/).

If a bucket with the same name exists, determine if it's being used. If it's not being used, consider deleting the bucket and attempting to bootstrap your environment again.

If a bucket with the same name exists and you don't want to delete it, determine if it's already associated with a bootstrap stack in your account. You may have to check multiple Regions. The Region in the Amazon S3 bucket name doesn't necessarily mean that the bucket is in that Region. To check if it's associated with the `CDKToolkit` bootstrap stack, you can do either of the following for each Region:

- Use the AWS CLI `aws cloudformation describe-stack-resources --stack-name <CDKToolkit> --region <Region>` command to view the resources in your bootstrap stack and check if the bucket is listed.
- In the [AWS CloudFormation console](https://console.aws.amazon.com/cloudformation/), locate the `CDKToolkit` stack. Then, on the **Resources** tab, check if the bucket exists.

If the bucket is associated with a bootstrap stack, determine if the bootstrap stack is in the same Region that you are attempting to bootstrap. If it is, your environment is already bootstrapped and you should be able to start using the CDK to deploy applications into your environment. If the Amazon S3 bucket is associated with a bootstrap stack in a different Region, you'll need to determine what to do. Possible resolutions include renaming the existing Amazon S3 bucket, deleting the current Amazon S3 bucket if its not being used, or using a new name for the Amazon S3 bucket you are attempting to create.

If you are unable to locate an Amazon S3 bucket with the same name in your account, it may exist in a different account. To resolve this, you'll need to customize bootstrapping to create new names for all of your bootstrap resources or for just your Amazon S3 bucket. To create new names for all bootstrap resources, you can modify the qualifier. To create a new name for only your Amazon S3 bucket, you can provide a new bucket name.

To customize bootstrapping, you can use options with the CDK CLI `cdk bootstrap` command or by modifying the bootstrap template. For instructions, see Customize AWS CDK bootstrapping.

If you customize bootstrapping, you will need to apply the same changes to synthesis before you can properly deploy an application. For instructions, see Customize CDK stack synthesis.

For example, you can provide a new qualifier with `cdk bootstrap`:

```
$ cdk bootstrap --qualifier <abcde0123>
```

The following is an example Amazon S3 bucket name that will be created with this modification: `cdk-abcde0123-assets-012345678910-us-west-1`. All bootstrap resources created during bootstrapping will use this qualifier.

When developing your CDK app, you must specify your custom qualifier in your synthesizer. This helps the CDK with identifying your bootstrap resources during synthesis and deployment. The following is an example of customizing the default synthesizer for your stack instance:

Example

TypeScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'abcde0123',
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'abcde0123',
  }),
})
```

Python

```
MyStack(self, "MyStack",
```

```
    synthesizer=DefaultStackSynthesizer(
        qualifier="abcde0123"
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    .qualifier("abcde0123")
    .build())
  .build();
)
```

C#

```
new MyStack(app, "MyStack", new StackProps
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        Qualifier = "abcde0123"
    })
});
```

Go

```
func NewMyStack(scope constructs.Construct, id string, props *MyStackProps)
 awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 synth := awscdk.NewDefaultStackSynthesizer(&awscdk.DefaultStackSynthesizerProps{
  Qualifier: jsii.String("abcde0123"),
 })

 stack.SetSynthesizer(synth)

 return stack
}
```

You can also specify the new qualifier in the `cdk.json` file of your CDK project:

```
{
  "app": "...",
  "context": {
    "@aws-cdk/core:bootstrapQualifier": "abcde0123"
  }
}
```

To modify only the Amazon S3 bucket name, you can use the [`--bootstrap-bucket-name`](#) option. The following is an example:

```
$ cdk bootstrap --bootstrap-bucket-name '<my-new-bucket-name>'
```

When developing your CDK app, you must specify your new bucket name in your synthesizer. The following is an example of customizing the default synthesizer for your stack instance:

Example

TypeScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    fileAssetsBucketName: 'my-new-bucket-name',
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    fileAssetsBucketName: 'my-new-bucket-name',
  }),
})
```

Python

```
MyStack(self, "MyStack",
    synthesizer=DefaultStackSynthesizer(
        file_assets_bucket_name='my-new-bucket-name'
```

```
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    .fileAssetsBucketName("my-new-bucket-name")
    .build())
  .build();
)
```

C#

```
new MyStack(app, "MyStack", new StackProps
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        FileAssetsBucketName = "my-new-bucket-name"
    })
});
```

Go

```
func NewMyStack(scope constructs.Construct, id string, props *MyStackProps)
 awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 synth := awscdk.NewDefaultStackSynthesizer(&awscdk.DefaultStackSynthesizerProps{
  FileAssetsBucketName: jsii.String("my-new-bucket-name"),
 })

 stack.SetSynthesizer(synth)

 return stack
}
```

Prevention

We recommend that you proactively bootstrap each AWS environment that you plan to use. For more information, see When to bootstrap your environment. Specifically for the Amazon S3 bucket naming issue, this will create Amazon S3 buckets in each AWS environment and prevent others from using your Amazon S3 bucket name.

Develop AWS CDK applications

Manage your infrastructure on AWS by developing applications using the AWS Cloud Development Kit (AWS CDK).

Prerequisites

Before you can start developing applications, complete all set up steps in Getting started with the AWS CDK.

Developing AWS CDK applications overview

At a high-level, developing CDK applications involves the following steps:

1. **Create a CDK project** – A CDK project consists of the files and folders that store and organize your CDK code.
2. **Create a CDK app** – Within a CDK project, you use the App construct to define a CDK application.
3. **Create a CDK stack** – Within the scope of your CDK app, you define one or more CDK stacks.
4. **Define your infrastructure** – Within the scope of a CDK stack, you use constructs from the AWS Construct Library to define the AWS resources and properties that will become your infrastructure. Using a general-purpose programming language of your choice, you can use logic, such as conditional statements and loops, to define your infrastructure based on certain conditions.

Get started with developing CDK applications

To get started, you can use the AWS CDK Command Line Interface (AWS CDK CLI) `cdk init` command. Provide the `--language` option to specify the programming language to use. This command creates a starting CDK project and imports the main AWS Construct Library and core modules.

Import and use the AWS CDK Library

After you create a CDK project, import and use constructs from the AWS CDK Library to begin defining your infrastructure. For instructions, see Work with the AWS CDK library.

Next steps

When ready to deploy your application, use the CDK CLI `cdk deploy` command. For instructions, see Deploy AWS CDK applications.

Customize constructs from the AWS Construct Library

Customize constructs from the AWS Construct Library through escape hatches, raw overrides, and custom resources.

Use escape hatches

The AWS Construct Library provides constructs of varying levels of abstraction.

At the highest level, your AWS CDK application and the stacks in it are themselves abstractions of your entire cloud infrastructure, or significant chunks of it. They can be parameterized to deploy them in different environments or for different needs.

Abstractions are powerful tools for designing and implementing cloud applications. The AWS CDK gives you the power not only to build with its abstractions, but also to create new abstractions. Using the existing open-source L2 and L3 constructs as guidance, you can build your own L2 and L3 constructs to reflect your own organization's best practices and opinions.

No abstraction is perfect, and even good abstractions cannot cover every possible use case. During development, you may find a construct that almost fits your needs, requiring a small or large customization.

For this reason, the AWS CDK provides ways to *break out* of the construct model. This includes moving to a lower-level abstraction or to a different model entirely. Escape hatches let you *escape* the AWS CDK paradigm and customize it in ways that suit your needs. Then, you can wrap your changes in a new construct to abstract away the underlying complexity and provide a clean API for other developers.

The following are examples of situations where you can use escape hatches:

- An AWS service feature is available through AWS CloudFormation, but there are no L2 constructs for it.
- An AWS service feature is available through AWS CloudFormation, and there are L2 constructs for the service, but these don't yet expose the feature. Because L2 constructs are curated by the CDK team, they may not be immediately available for new features.

- The feature is not yet available through AWS CloudFormation at all.

  To determine whether a feature is available through AWS CloudFormation, see AWS Resource and Property Types Reference.

Develop escape hatches for L1 constructs

If L2 constructs are not available for the service, you can use the automatically generated L1 constructs. These resources can be recognized by their name starting with `Cfn`, such as `CfnBucket` or `CfnRole`. You instantiate them exactly as you would use the equivalent AWS CloudFormation resource.

For example, to instantiate a low-level Amazon S3 bucket L1 with analytics enabled, you would write something like the following.

Example

TypeScript

```
new s3.CfnBucket(this, 'amzn-s3-demo-bucket', {
  analyticsConfigurations: [
    {
      id: 'Config',
      // ...
    }
  ]
});
```

JavaScript

```
new s3.CfnBucket(this, 'amzn-s3-demo-bucket', {
  analyticsConfigurations: [
    {
      id: 'Config'
      // ...
    }
  ]
});
```

Python

```
s3.CfnBucket(self, "amzn-s3-demo-bucket",
    analytics_configurations: [
        dict(id="Config",
             # ...
             )
    ]
)
```

Java

```
CfnBucket.Builder.create(this, "amzn-s3-demo-bucket")
    .analyticsConfigurations(Arrays.asList(java.util.Map.of(    // Java 9 or later
        "id", "Config", // ...
    ))).build();
```

C#

```
new CfnBucket(this, 'amzn-s3-demo-bucket', new CfnBucketProps {
    AnalyticsConfigurations = new Dictionary<string, string>
    {
        ["id"] = "Config",
        // ...
    }
});
```

There might be rare cases where you want to define a resource that doesn't have a corresponding `CfnXxx` class. This could be a new resource type that hasn't yet been published in the AWS CloudFormation resource specification. In cases like this, you can instantiate the `cdk.CfnResource` directly and specify the resource type and properties. This is shown in the following example.

Example

TypeScript

```
new cdk.CfnResource(this, 'amzn-s3-demo-bucket', {
  type: 'AWS::S3::Bucket',
  properties: {
```

```
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config',
        // ...
      }
    ]
  }
});
```

JavaScript

```
new cdk.CfnResource(this, 'amzn-s3-demo-bucket', {
  type: 'AWS::S3::Bucket',
  properties: {
    // Note the PascalCase here! These are CloudFormation identifiers.
    AnalyticsConfigurations: [
      {
        Id: 'Config'
        // ...
      }
    ]
  }
});
```

Python

```
cdk.CfnResource(self, 'amzn-s3-demo-bucket',
  type="AWS::S3::Bucket",
  properties=dict(
    # Note the PascalCase here! These are CloudFormation identifiers.
    "AnalyticsConfigurations": [
      {
        "Id": "Config",
        # ...
      }
    ]
  )
)
```

Java

```
CfnResource.Builder.create(this, "amzn-s3-demo-bucket")
```

```
        .type("AWS::S3::Bucket")
        .properties(java.util.Map.of(    // Map.of requires Java 9 or later
            // Note the PascalCase here! These are CloudFormation identifiers
            "AnalyticsConfigurations", Arrays.asList(
                    java.util.Map.of("Id", "Config", // ...
                    ))))
        .build();
```

C#

```
new CfnResource(this, "amzn-s3-demo-bucket", new CfnResourceProps
{
    Type = "AWS::S3::Bucket",
    Properties = new Dictionary<string, object>
    {   // Note the PascalCase here! These are CloudFormation identifiers
        ["AnalyticsConfigurations"] = new Dictionary<string, string>[]
        {
            new Dictionary<string, string> {
                ["Id"] = "Config"
            }
        }
    }
});
```

Develop escape hatches for L2 constructs

If an L2 construct is missing a feature or you're trying to work around an issue, you can modify the L1 construct that's encapsulated by the L2 construct.

All L2 constructs contain within them the corresponding L1 construct. For example, the high-level `Bucket` construct wraps the low-level `CfnBucket` construct. Because the `CfnBucket` corresponds directly to the AWS CloudFormation resource, it exposes all features that are available through AWS CloudFormation.

The basic approach to get access to the L1 construct is to use `construct.node.defaultChild` (Python: `default_child`), cast it to the right type (if necessary), and modify its properties. Again, let's take the example of a `Bucket`.

Example

TypeScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config',
    // ...
  }
];
```

JavaScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild;

// Change its properties
cfnBucket.analyticsConfiguration = [
  {
    id: 'Config'
    // ...
  }
];
```

Python

```
# Get the CloudFormation resource
cfn_bucket = bucket.node.default_child

# Change its properties
cfn_bucket.analytics_configuration = [
    {
        "id": "Config",
        # ...
    }
]
```

Java

```
// Get the CloudFormation resource
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

cfnBucket.setAnalyticsConfigurations(
        Arrays.asList(java.util.Map.of(    // Java 9 or later
            "Id", "Config", // ...
        ));
)
```

C#

```
// Get the CloudFormation resource
var cfnBucket = (CfnBucket)bucket.Node.DefaultChild;

cfnBucket.AnalyticsConfigurations = new List<object> {
    new Dictionary<string, string>
    {
        ["Id"] = "Config",
        // ...
    }
};
```

You can also use this object to change AWS CloudFormation options such as `Metadata` and `UpdatePolicy`.

Example

TypeScript

```
cfnBucket.cfnOptions.metadata = {
  MetadataKey: 'MetadataValue'
};
```

JavaScript

```
cfnBucket.cfnOptions.metadata = {
  MetadataKey: 'MetadataValue'
};
```

Python

```
cfn_bucket.cfn_options.metadata = {
    "MetadataKey": "MetadataValue"
}
```

Java

```
cfnBucket.getCfnOptions().setMetadata(java.util.Map.of(    // Java 9+
    "MetadataKey", "Metadatavalue"));
```

C#

```
cfnBucket.CfnOptions.Metadata = new Dictionary<string, object>
{
    ["MetadataKey"] = "Metadatavalue"
};
```

Use un-escape hatches

The AWS CDK also provides the capability to go *up* an abstraction level, which we might refer to as an "un-escape" hatch. If you have an L1 construct, such as `CfnBucket`, you can create a new L2 construct (`Bucket` in this case) to wrap the L1 construct.

This is convenient when you create an L1 resource but want to use it with a construct that requires an L2 resource. It's also helpful when you want to use convenience methods like `.grantXxxxx()` that aren't available on the L1 construct.

You move to the higher abstraction level using a static method on the L2 class called `.fromCfnXxxxx()`--for example, `Bucket.fromCfnBucket()` for Amazon S3 buckets. The L1 resource is the only parameter.

Example

TypeScript

```
b1 = new s3.CfnBucket(this, "buck09", { ... });
b2 = s3.Bucket.fromCfnBucket(b1);
```

JavaScript

```
b1 = new s3.CfnBucket(this, "buck09", { ...} );
b2 = s3.Bucket.fromCfnBucket(b1);
```

Python

```
b1 = s3.CfnBucket(self, "buck09", ...)
 b2 = s3.from_cfn_bucket(b1)
```

Java

```
CfnBucket b1 = CfnBucket.Builder.create(this, "buck09")
        // ....
                          .build();
IBucket b2 = Bucket.fromCfnBucket(b1);
```

C#

```
var b1 = new CfnBucket(this, "buck09", new CfnBucketProps { ... });
var v2 = Bucket.FromCfnBucket(b1);
```

L2 constructs created from L1 constructs are proxy objects that refer to the L1 resource, similar to those created from resource names, ARNs, or lookups. Modifications to these constructs do not affect the final synthesized AWS CloudFormation template (since you have the L1 resource, however, you can modify that instead). For more information on proxy objects, see Referencing resources in your AWS account.

To avoid confusion, do not create multiple L2 constructs that refer to the same L1 construct. For example, if you extract the `CfnBucket` from a `Bucket` using the technique in the previous section, you shouldn't create a second `Bucket` instance by calling `Bucket.fromCfnBucket()` with that `CfnBucket`. It actually works as you'd expect (only one `AWS::S3::Bucket` is synthesized) but it makes your code more difficult to maintain.

Use raw overrides

If there are properties that are missing from the L1 construct, you can bypass all typing using raw overrides. This also makes it possible to delete synthesized properties.

Use one of the `addOverride` methods (Python: `add_override`) methods, as shown in the following example.

Example

TypeScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild as s3.CfnBucket;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// use index (0 here) to address an element of a list
cfnBucket.addOverride('Properties.Tags.0.Value', 'NewValue');
cfnBucket.addDeletionOverride('Properties.Tags.0');

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
cfnBucket.addPropertyOverride('Tags.0.Value', 'NewValue');
cfnBucket.addPropertyDeletionOverride('Tags.0');
```

JavaScript

```
// Get the CloudFormation resource
const cfnBucket = bucket.node.defaultChild ;

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride('Properties.VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addDeletionOverride('Properties.VersioningConfiguration.Status');

// use index (0 here) to address an element of a list
cfnBucket.addOverride('Properties.Tags.0.Value', 'NewValue');
cfnBucket.addDeletionOverride('Properties.Tags.0');

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride('VersioningConfiguration.Status', 'NewStatus');
cfnBucket.addPropertyDeletionOverride('VersioningConfiguration.Status');
cfnBucket.addPropertyOverride('Tags.0.Value', 'NewValue');
```

```
cfnBucket.addPropertyDeletionOverride('Tags.0');
```

Python

```
# Get the CloudFormation resource
cfn_bucket = bucket.node.default_child

# Use dot notation to address inside the resource template fragment
cfn_bucket.add_override("Properties.VersioningConfiguration.Status", "NewStatus")
cfn_bucket.add_deletion_override("Properties.VersioningConfiguration.Status")

# use index (0 here) to address an element of a list
cfn_bucket.add_override("Properties.Tags.0.Value", "NewValue")
cfn_bucket.add_deletion_override("Properties.Tags.0")

# addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfn_bucket.add_property_override("VersioningConfiguration.Status", "NewStatus")
cfn_bucket.add_property_deletion_override("VersioningConfiguration.Status")
cfn_bucket.add_property_override("Tags.0.Value", "NewValue")
cfn_bucket.add_property_deletion_override("Tags.0")
```

Java

```
// Get the CloudFormation resource
CfnBucket cfnBucket = (CfnBucket)bucket.getNode().getDefaultChild();

// Use dot notation to address inside the resource template fragment
cfnBucket.addOverride("Properties.VersioningConfiguration.Status", "NewStatus");
cfnBucket.addDeletionOverride("Properties.VersioningConfiguration.Status");

// use index (0 here) to address an element of a list
cfnBucket.addOverride("Properties.Tags.0.Value", "NewValue");
cfnBucket.addDeletionOverride("Properties.Tags.0");

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.addPropertyOverride("VersioningConfiguration.Status", "NewStatus");
cfnBucket.addPropertyDeletionOverride("VersioningConfiguration.Status");
cfnBucket.addPropertyOverride("Tags.0.Value", "NewValue");
cfnBucket.addPropertyDeletionOverride("Tags.0");
```

C#

```
// Get the CloudFormation resource
var cfnBucket = (CfnBucket)bucket.node.defaultChild;

// Use dot notation to address inside the resource template fragment
cfnBucket.AddOverride("Properties.VersioningConfiguration.Status", "NewStatus");
cfnBucket.AddDeletionOverride("Properties.VersioningConfiguration.Status");

// use index (0 here) to address an element of a list
cfnBucket.AddOverride("Properties.Tags.0.Value", "NewValue");
cfnBucket.AddDeletionOverride("Properties.Tags.0");

// addPropertyOverride is a convenience function for paths starting with
 "Properties."
cfnBucket.AddPropertyOverride("VersioningConfiguration.Status", "NewStatus");
cfnBucket.AddPropertyDeletionOverride("VersioningConfiguration.Status");
cfnBucket.AddPropertyOverride("Tags.0.Value", "NewValue");
cfnBucket.AddPropertyDeletionOverride("Tags.0");
```

Use custom resources

If the feature isn't available through AWS CloudFormation, but only through a direct API call, you must write an AWS CloudFormation Custom Resource to make the API call you need. You can use the AWS CDK to write custom resources and wrap them into a regular construct interface. From the perspective of a consumer of your construct, the experience will feel native.

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 needs to make only a single API call, consider using the AwsCustomResource. This makes it possible to perform arbitrary SDK calls during an AWS CloudFormation deployment. Otherwise, you should write your own Lambda function to perform the work you need to get done.

The subject is too broad to cover completely here, but the following links should get you started:

- Custom Resources
- Custom-Resource Example
- For a more fully fledged example, see the DnsValidatedCertificate class in the CDK standard library. This is implemented as a custom resource.

Get a value from an environment variable

To get the value of an environment variable, use code like the following. This code gets the value of the environment variable `amzn-s3-demo-bucket`.

Example

TypeScript

```
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.amzn-s3-demo-bucket;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.amzn-s3-demo-bucket || "DefaultName";
```

JavaScript

```
// Sets bucket_name to undefined if environment variable not set
var bucket_name = process.env.amzn-s3-demo-bucket;

// Sets bucket_name to a default if env var doesn't exist
var bucket_name = process.env.amzn-s3-demo-bucket || "DefaultName";
```

Python

```
import os

# Raises KeyError if environment variable doesn't exist
bucket_name = os.environ["amzn-s3-demo-bucket"]

# Sets bucket_name to None if environment variable doesn't exist
bucket_name = os.getenv("amzn-s3-demo-bucket")

# Sets bucket_name to a default if env var doesn't exist
bucket_name = os.getenv("amzn-s3-demo-bucket", "DefaultName")
```

Java

```
// Sets bucketName to null if environment variable doesn't exist
String bucketName = System.getenv("amzn-s3-demo-bucket");

// Sets bucketName to a default if env var doesn't exist
```

```
String bucketName = System.getenv("amzn-s3-demo-bucket");
if (bucketName == null) bucketName = "DefaultName";
```

C#

```
using System;

// Sets bucket name to null if environment variable doesn't exist
string bucketName = Environment.GetEnvironmentVariable("amzn-s3-demo-bucket");

// Sets bucket_name to a default if env var doesn't exist
string bucketName = Environment.GetEnvironmentVariable("amzn-s3-demo-bucket") ??
 "DefaultName";
```

Use CloudFormation parameters to get a CloudFormation value

Use AWS CloudFormation parameters within AWS Cloud Development Kit (AWS CDK) applications to input custom values into your synthesized CloudFormation templates at deployment.

For an introduction, see Parameters and the AWS CDK.

Define parameters in your CDK app

Use the `CfnParameter` class to define a parameter. You'll want to specify at least a type and a description for most parameters, though both are technically optional. The description appears when the user is prompted to enter the parameter's value in the AWS CloudFormation console. For more information on the available types, see Types.

Note

You can define parameters in any scope. However, we recommend defining parameters at the stack level so that their logical ID doesn't change when you refactor your code.

Example

TypeScript

```
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
```

```
  type: "String",
  description: "The name of the Amazon S3 bucket where uploaded files will be
 stored."});
```

JavaScript

```
const uploadBucketName = new CfnParameter(this, "uploadBucketName", {
  type: "String",
  description: "The name of the Amazon S3 bucket where uploaded files will be
 stored."});
```

Python

```
upload_bucket_name = CfnParameter(self, "uploadBucketName", type="String",
    description="The name of the Amazon S3 bucket where uploaded files will be
 stored.")
```

Java

```
CfnParameter uploadBucketName = CfnParameter.Builder.create(this,
 "uploadBucketName")
        .type("String")
        .description("The name of the Amazon S3 bucket where uploaded files will be
 stored")
        .build();
```

C#

```
var uploadBucketName = new CfnParameter(this, "uploadBucketName", new
 CfnParameterProps
{
    Type = "String",
    Description = "The name of the Amazon S3 bucket where uploaded files will be
 stored"
});
```

Use parameters

A `CfnParameter` instance exposes its value to your CDK app via a token. Like all tokens, the parameter's token is resolved at synthesis time. But it resolves to a reference to the parameter defined in the AWS CloudFormation template (which will be resolved at deploy time), rather than to a concrete value.

You can retrieve the token as an instance of the `Token` class, or in string, string list, or numeric encoding. Your choice depends on the kind of value required by the class or method that you want to use the parameter with.

Example

TypeScript

Property	kind of value
`value`	`Token` class instance
`valueAsList`	The token represented as a string list
`valueAsNumber`	The token represented as a number
`valueAsString`	The token represented as a string

JavaScript

Property	kind of value
`value`	`Token` class instance
`valueAsList`	The token represented as a string list
`valueAsNumber`	The token represented as a number
`valueAsString`	The token represented as a string

Python

Property	kind of value
`value`	`Token` class instance

Property	kind of value
`value_as_list`	The token represented as a string list
`value_as_number`	The token represented as a number
`value_as_string`	The token represented as a string

Java

Property	kind of value
`getValue()`	`Token` class instance
`getValueAsList()`	The token represented as a string list
`getValueAsNumber()`	The token represented as a number
`getValueAsString()`	The token represented as a string

C#

Property	kind of value
`Value`	`Token` class instance
`ValueAsList`	The token represented as a string list
`ValueAsNumber`	The token represented as a number
`ValueAsString`	The token represented as a string

For example, to use a parameter in a `Bucket` definition:

Example

TypeScript

```
const bucket = new Bucket(this, "amzn-s3-demo-bucket",
  { bucketName: uploadBucketName.valueAsString});
```

JavaScript

```
const bucket = new Bucket(this, "amzn-s3-demo-bucket",
  { bucketName: uploadBucketName.valueAsString});
```

Python

```
bucket = Bucket(self, "amzn-s3-demo-bucket",
    bucket_name=upload_bucket_name.value_as_string)
```

Java

```
Bucket bucket = Bucket.Builder.create(this, "amzn-s3-demo-bucket")
        .bucketName(uploadBucketName.getValueAsString())
        .build();
```

C#

```
var bucket = new Bucket(this, "amzn-s3-demo-bucket")
{
    BucketName = uploadBucketName.ValueAsString
};
```

Deploy CDK apps containing parameters

When you deploy a generated AWS CloudFormation template through the AWS CloudFormation console, you will be prompted to provide the values for each parameter.

You can also provide parameter values using the CDK CLI `cdk deploy` command, or by specifying parameter values in your CDK project's stack file.

Provide parameter values with cdk deploy

When you deploy using the CDK CLI `cdk deploy` command, you can provide parameter values at deployment with the `--parameters` option.

The following is an example of the `cdk deploy` command structure:

```
$ cdk deploy <stack-logical-id> --parameters <stack-name>:<parameter-name>=<parameter-value>
```

If your CDK app contains a single stack, you don't have to provide the stack logical ID argument or the `stack-name` value in the `--parameters` option. The CDK CLI will automatically find and provide these values. The following is an example that specifies an `uploadbucket` value for the `uploadBucketName` parameter of the single stack in our CDK app:

```
$ cdk deploy --parameters <uploadBucketName>=<uploadbucket>
```

Provide parameter values with cdk deploy for multi-stack applications

The following is an example CDK application in TypeScript that contains two CDK stacks. Each stack contains an Amazon S3 bucket instance and a parameter to set the Amazon S3 bucket name:

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

// Define the CDK app
const app = new cdk.App();

// First stack
export class MyFirstStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Set a default parameter name
    const bucketNameParam = new cdk.CfnParameter(this, 'bucketNameParam', {
      type: 'String',
      default: 'myfirststackdefaultbucketname'
    });

    // Define an S3 bucket
```

```
    new s3.Bucket(this, 'MyFirstBucket', {
      bucketName: bucketNameParam.valueAsString
    });
  }
}

// Second stack
export class MySecondStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Set a default parameter name
    const bucketNameParam = new cdk.CfnParameter(this, 'bucketNameParam', {
      type: 'String',
      default: 'mysecondstackdefaultbucketname'
    });

    // Define an S3 bucket
    new s3.Bucket(this, 'MySecondBucket', {
      bucketName: bucketNameParam.valueAsString
    });
  }
}

// Instantiate the stacks
new MyFirstStack(app, 'MyFirstStack', {
  stackName: 'MyFirstDeployedStack',
});

new MySecondStack(app, 'MySecondStack', {
  stackName: 'MySecondDeployedStack',
});
```

For CDK apps that contain multiple stacks, you can do the following:

- **Deploy one stack with parameters** – To deploy a single stack from a multi-stack application, provide the stack logical ID as an argument.

  The following is an example that deploys `MySecondStack` with `mynewbucketname` as the parameter value for `bucketNameParam`:

  ```
  $ cdk deploy <MySecondStack> --parameters <bucketNameParam>='<mynewbucketname>'
  ```

- **Deploy all stacks and specify parameter values for each stack** – Provide the `'*'` wildcard or the `--all` option to deploy all stacks. Provide the `--parameters` option multiple times in a single command to specify parameter values for each stack. The following is an example:

```
$ cdk deploy <'*'> --parameters
 <MyFirstDeployedStack>:<bucketNameParam>='<mynewfirststackbucketname>' --parameters
 <MySecondDeployedStack>:<bucketNameParam>='<mynewsecondstackbucketname>'
```

- **Deploy all stacks and specify parameter values for a single stack** – Provide the `'*'` wildcard or the `--all` option to deploy all stacks. Then, specify the stack to define the parameter for in the `--parameters` option. The following are examples that deploys all stacks in a CDK app and specifies a parameter value for the `MySecondDeployedStack` AWS CloudFormation stack. All other stacks will deploy and use the default parameter value:

```
$ cdk deploy <'*'> --parameters
 <MySecondDeployedStack>:<bucketNameParam>='<mynewbucketname>'
$ cdk deploy <--all> --parameters
 <MySecondDeployedStack>:<bucketNameParam>='<mynewbucketname>'
```

Provide parameter values with `cdk deploy` for applications with nested stacks

The CDK CLI behavior when working with applications containing nested stacks is similar to multi-stack applications. The main difference is, if you want to deploy all nested stacks, use the `'**'` wildcard. The `'*'` wildcard deploys all stacks but will not deploy nested stacks. The `'**'` wildcard deploys all stacks, including nested stacks.

The following is an example that deploys nested stacks while specifying the parameter value for one nested stack:

```
$ cdk deploy '**' --parameters <MultiStackCdkApp/
SecondStack>:<bucketNameParam>='<mysecondstackbucketname>'
```

For more information on `cdk deploy` command options, see cdk deploy.

Import an existing AWS CloudFormation template

Import resources from an AWS CloudFormation template into your AWS Cloud Development Kit (AWS CDK) applications by using the `cloudformation-include.CfnInclude` construct to convert resources to L1 constructs.

After import, you can work with these resources in your app in the same way that you would if they were originally defined in AWS CDK code. You can also use these L1 constructs within higher-level AWS CDK constructs. For example, this can let you use the L2 permission grant methods with the resources they define.

The `cloudformation-include.CfnInclude` construct essentially adds an AWS CDK API wrapper to any resource in your AWS CloudFormation template. Use this capability to import your existing AWS CloudFormation templates to the AWS CDK a piece at a time. By doing this, you can manage your existing resources using AWS CDK constructs to utilize the benefits of higher-level abstractions. You can also use this feature to vend your AWS CloudFormation templates to AWS CDK developers by providing an AWS CDK construct API.

> **Note**
>
> AWS CDK v1 also included `aws-cdk-lib.CfnInclude`, which was previously used for the same general purpose. However, it lacks much of the functionality of `cloudformation-include.CfnInclude`.

Import an AWS CloudFormation template

The following is a sample AWS CloudFormation template that we will use to provide examples in this topic. Copy and save the template as `my-template.json` to follow along. After working through these examples, you can explore further by using any of your existing deployed AWS CloudFormation templates. You can obtain them from the AWS CloudFormation console.

```
{
  "Resources": {
    "amzn-s3-demo-bucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "amzn-s3-demo-bucket",
      }
```

```
    }
  }
}
```

You can work with either JSON or YAML templates. We recommend JSON if available since YAML parsers can vary slightly in what they accept.

The following is an example of how to import the sample template into your AWS CDK app using `cloudformation-include`. Templates are imported within the context of an CDK stack.

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as cfninc from 'aws-cdk-lib/cloudformation-include';
import { Construct } from 'constructs';

export class MyStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const template = new cfninc.CfnInclude(this, 'Template', {
      templateFile: 'my-template.json',
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const cfninc = require('aws-cdk-lib/cloudformation-include');

class MyStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const template = new cfninc.CfnInclude(this, 'Template', {
      templateFile: 'my-template.json',
    });
  }
}
```

```
module.exports = { MyStack }
```

Python

```
import aws_cdk as cdk
from aws_cdk import cloudformation_include as cfn_inc
from constructs import Construct

class MyStack(cdk.Stack):

    def __init__(self, scope: Construct, id: str, **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        template = cfn_inc.CfnInclude(self, "Template",
            template_file="my-template.json")
```

Java

```
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.cloudformation.include.CfnInclude;
import software.constructs.Construct;

public class MyStack extends Stack {
    public MyStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyStack(final Construct scope, final String id, final StackProps props) {
        super(scope, id, props);

        CfnInclude template = CfnInclude.Builder.create(this, "Template")
         .templateFile("my-template.json")
         .build();
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;
using cfnInc = Amazon.CDK.CloudFormation.Include;
```

```
namespace MyApp
{
    public class MyStack : Stack
    {
        internal MyStack(Construct scope, string id, IStackProps props = null) :
 base(scope, id, props)
        {
            var template = new cfnInc.CfnInclude(this, "Template", new
 cfnInc.CfnIncludeProps
            {
                TemplateFile = "my-template.json"
            });
        }
    }
}
```

By default, importing a resource preserves the resource's original logical ID from the template. This behavior is suitable for importing an AWS CloudFormation template into the AWS CDK, where logical IDs must be retained. AWS CloudFormation needs this information to recognize these imported resources as the same resources from the AWS CloudFormation template.

If you are developing an AWS CDK construct wrapper for the template so that it can be used by other AWS CDK developers, have the AWS CDK generate new resource IDs instead. By doing this, the construct can be used multiple times in a stack without name conflicts. To do this, set the `preserveLogicalIds` property to `false` when importing the template. The following is an example:

Example

TypeScript

```
const template = new cfninc.CfnInclude(this, 'MyConstruct', {
  templateFile: 'my-template.json',
  preserveLogicalIds: false
});
```

JavaScript

```
const template = new cfninc.CfnInclude(this, 'MyConstruct', {
  templateFile: 'my-template.json',
  preserveLogicalIds: false
```

```
});
```

Python

```
template = cfn_inc.CfnInclude(self, "Template",
    template_file="my-template.json",
    preserve_logical_ids=False)
```

Java

```
CfnInclude template = CfnInclude.Builder.create(this, "Template")
 .templateFile("my-template.json")
 .preserveLogicalIds(false)
 .build();
```

C#

```
var template = new cfnInc.CfnInclude(this, "Template", new cfn_inc.CfnIncludeProps
{
    TemplateFile = "my-template.json",
    PreserveLogicalIds = false
});
```

To put imported resources under the control of your AWS CDK app, add the stack to the App:

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { MyStack } from '../lib/my-stack';

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

JavaScript

```
const cdk = require('aws-cdk-lib');
const { MyStack } = require('../lib/my-stack');

const app = new cdk.App();
```

```
new MyStack(app, 'MyStack');
```

Python

```
import aws_cdk as cdk
from mystack.my_stack import MyStack

app = cdk.App()
MyStack(app, "MyStack")
```

Java

```
import software.amazon.awscdk.App;

public class MyApp {
    public static void main(final String[] args) {
        App app = new App();

        new MyStack(app, "MyStack");
    }
}
```

C#

```
using Amazon.CDK;

namespace CdkApp
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new MyStack(app, "MyStack");
        }
    }
}
```

To verify that there won't be any unintended changes to the AWS resources in the stack, you can perform a diff. Use the AWS CDK CLI `cdk diff` command and omit any AWS CDK-specific metadata. The following is an example:

```
cdk diff --no-version-reporting --no-path-metadata --no-asset-metadata
```

After you import an AWS CloudFormation template, the AWS CDK app should become the source of truth for your imported resources. To make changes to your resources, modify them in your AWS CDK app and deploy with the AWS CDK CLI `cdk deploy` command.

Access imported resources

The name `template` in the example code represents the imported AWS CloudFormation template. To access a resource from it, use the object's [`getResource()`] method. To access the returned resource as a specific kind of resource, cast the result to the desired type. This isn't necessary in Python or JavaScript. The following is an example:

Example

TypeScript

```
const cfnBucket = template.getResource('amzn-s3-demo-bucket') as s3.CfnBucket;
```

JavaScript

```
const cfnBucket = template.getResource('amzn-s3-demo-bucket');
```

Python

```
cfn_bucket = template.get_resource("amzn-s3-demo-bucket")
```

Java

```
CfnBucket cfnBucket = (CfnBucket)template.getResource("amzn-s3-demo-bucket");
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("amzn-s3-demo-bucket");
```

From this example, `cfnBucket` is now an instance of the [`aws-s3.CfnBucket`] class. This is an L1 construct that represents the corresponding AWS CloudFormation resource. You can treat it like any other resource of its type. For example, you can get its ARN value with the `bucket.attrArn` property.

To wrap the L1 `CfnBucket` resource in an L2 `aws-s3.CfnBucket` instance instead, use the static methods `fromBucketArn()`, `fromBucketAttributes()`, or `fromBucketName()`. Usually, the `fromBucketName()` method is most convenient. The following is an example:

Example

TypeScript

```
const bucket = s3.Bucket.fromBucketName(this, 'Bucket', cfnBucket.ref);
```

JavaScript

```
const bucket = s3.Bucket.fromBucketName(this, 'Bucket', cfnBucket.ref);
```

Python

```
bucket = s3.Bucket.from_bucket_name(self, "Bucket", cfn_bucket.ref)
```

Java

```
Bucket bucket = (Bucket)Bucket.fromBucketName(this, "Bucket", cfnBucket.getRef());
```

C#

```
var bucket = (Bucket)Bucket.FromBucketName(this, "Bucket", cfnBucket.Ref);
```

Other L2 constructs have similar methods for creating the construct from an existing resource.

When you wrap an L1 construct in an L2 construct, it doesn’t create a new resource. From our example, we are not creating a second S3; bucket. Instead, the new `Bucket` instance encapsulates the existing `CfnBucket`.

From the example, the `bucket` is now an L2 `Bucket` construct that behaves like any other L2 construct. For example, you can grant an AWS Lambda function write access to the bucket by using the bucket’s convenient `grantWrite()` method. You don’t have to define the necessary AWS Identity and Access Management (IAM) policy manually. The following is an example:

Example

TypeScript

```
bucket.grantWrite(lambdaFunc);
```

JavaScript

```
bucket.grantWrite(lambdaFunc);
```

Python

```
bucket.grant_write(lambda_func)
```

Java

```
bucket.grantWrite(lambdaFunc);
```

C#

```
bucket.GrantWrite(lambdaFunc);
```

Replace parameters

If your AWS CloudFormation template contains parameters, you can replace them with build time values at import by using the `parameters` property. In the following example, we replace the `UploadBucket` parameter with the ARN of a bucket defined elsewhere in our AWS CDK code.

Example

TypeScript

```
const template = new cfninc.CfnInclude(this, 'Template', {
  templateFile: 'my-template.json',
  parameters: {
    'UploadBucket': bucket.bucketArn,
  },
});
```

JavaScript

```
const template = new cfninc.CfnInclude(this, 'Template', {
  templateFile: 'my-template.json',
  parameters: {
    'UploadBucket': bucket.bucketArn,
  },
});
```

Python

```
template = cfn_inc.CfnInclude(self, "Template",
    template_file="my-template.json",
    parameters=dict(UploadBucket=bucket.bucket_arn)
)
```

Java

```
CfnInclude template = CfnInclude.Builder.create(this, "Template")
 .templateFile("my-template.json")
 .parameters(java.util.Map.of(    // Map.of requires Java 9+
   "UploadBucket", bucket.getBucketArn()))
 .build();
```

C#

```
var template = new cfnInc.CfnInclude(this, "Template", new cfnInc.CfnIncludeProps
{
    TemplateFile = "my-template.json",
    Parameters = new Dictionary<string, string>
    {
        { "UploadBucket", bucket.BucketArn }
    }
});
```

Import other template elements

You can import any AWS CloudFormation template element, not just resources. The imported elements become a part of the AWS CDK stack. To import these elements, use the following methods of the `CfnInclude` object:

- `getCondition()` – AWS CloudFormation conditions.
- `getHook()` – AWS CloudFormation hooks for blue/green deployments.
- `getMapping()` – AWS CloudFormation mappings.
- `getOutput()` – AWS CloudFormation outputs.
- `getParameter()` – AWS CloudFormation parameters.
- `getRule()` – AWS CloudFormation rules for AWS Service Catalog templates.

Each of these methods return an instance of a class that represents the specific type of AWS CloudFormation element. These objects are mutable. Changes that you make to them will appear in the template that gets generated from the AWS CDK stack. The following is an example that imports a parameter from the template and modifies its default value:

Example

TypeScript

```
const param = template.getParameter('MyParameter');
param.default = "AWS CDK"
```

JavaScript

```
const param = template.getParameter('MyParameter');
param.default = "AWS CDK"
```

Python

```
param = template.get_parameter("MyParameter")
param.default = "AWS CDK"
```

Java

```
CfnParameter param = template.getParameter("MyParameter");
param.setDefaultValue("AWS CDK")
```

C#

```
var cfnBucket = (CfnBucket)template.GetResource("amzn-s3-demo-bucket");
var param = template.GetParameter("MyParameter");
```

```
param.Default = "AWS CDK";
```

Import nested stacks

You can import nested stacks by specifying them either when you import their main template, or at some later point. The nested template must be stored in a local file, but referenced as a `NestedStack` resource in the main template. Also, the resource name used in the AWS CDK code must match the name used for the nested stack in the main template.

Given this resource definition in the main template, the following code shows how to import the referenced nested stack both ways.

```
"NestedStack": {
  "Type": "AWS::CloudFormation::Stack",
  "Properties": {
    "TemplateURL": "https://my-s3-template-source.s3.amazonaws.com/nested-stack.json"
  }
}
```

Example

TypeScript

```
// include nested stack when importing main stack
const mainTemplate = new cfninc.CfnInclude(this, 'MainStack', {
  templateFile: 'main-template.json',
  loadNestedStacks: {
    'NestedStack': {
      templateFile: 'nested-template.json',
    },
  },
});

// or add it some time after importing the main stack
const nestedTemplate = mainTemplate.loadNestedStack('NestedTemplate', {
  templateFile: 'nested-template.json',
});
```

JavaScript

```
// include nested stack when importing main stack
```

```
const mainTemplate = new cfninc.CfnInclude(this, 'MainStack', {
  templateFile: 'main-template.json',
  loadNestedStacks: {
    'NestedStack': {
      templateFile: 'nested-template.json',
    },
  },
});

// or add it some time after importing the main stack
const nestedTemplate = mainTemplate.loadNestedStack('NestedStack', {
  templateFile: 'my-nested-template.json',
});
```

Python

```
# include nested stack when importing main stack
main_template = cfn_inc.CfnInclude(self, "MainStack",
    template_file="main-template.json",
    load_nested_stacks=dict(NestedStack=
        cfn_inc.CfnIncludeProps(template_file="nested-template.json")))

# or add it some time after importing the main stack
nested_template = main_template.load_nested_stack("NestedStack",
    template_file="nested-template.json")
```

Java

```
CfnInclude mainTemplate = CfnInclude.Builder.create(this, "MainStack")
 .templateFile("main-template.json")
 .loadNestedStacks(java.util.Map.of(    // Map.of requires Java 9+
    "NestedStack", CfnIncludeProps.builder()
    .templateFile("nested-template.json").build()))
 .build();

// or add it some time after importing the main stack
IncludedNestedStack nestedTemplate = mainTemplate.loadNestedStack("NestedTemplate",
 CfnIncludeProps.builder()
  .templateFile("nested-template.json")
  .build());
```

C#

```
// include nested stack when importing main stack
var mainTemplate = new cfnInc.CfnInclude(this, "MainStack", new
 cfnInc.CfnIncludeProps
{
    TemplateFile = "main-template.json",
    LoadNestedStacks = new Dictionary<string, cfnInc.ICfnIncludeProps>
    {
        { "NestedStack", new cfnInc.CfnIncludeProps { TemplateFile = "nested-
template.json" } }
    }
});

// or add it some time after importing the main stack
var nestedTemplate = mainTemplate.LoadNestedStack("NestedTemplate", new
 cfnInc.CfnIncludeProps {
    TemplateFile = 'nested-template.json'
});
```

You can import multiple nested stacks with either methods. When importing the main template, you provide a mapping between the resource name of each nested stack and its template file. This mapping can contain any number of entries. To do it after the initial import, call `loadNestedStack()` once for each nested stack.

After importing a nested stack, you can access it using the main template's [`getNestedStack()`] method.

Example

TypeScript

```
const nestedStack = mainTemplate.getNestedStack('NestedStack').stack;
```

JavaScript

```
const nestedStack = mainTemplate.getNestedStack('NestedStack').stack;
```

Python

```
nested_stack = main_template.get_nested_stack("NestedStack").stack
```

Java

```
NestedStack nestedStack = mainTemplate.getNestedStack("NestedStack").getStack();
```

C#

```
var nestedStack = mainTemplate.GetNestedStack("NestedStack").Stack;
```

The `getNestedStack()` method returns an [`IncludedNestedStack`] instance. From this instance, you can access the AWS CDK [`NestedStack`] instance via the `stack` property, as shown in the example. You can also access the original AWS CloudFormation template object via `includedTemplate`, from which you can load resources and other AWS CloudFormation elements.

Get a value from the Systems Manager Parameter Store

The AWS Cloud Development Kit (AWS CDK) can retrieve the value of AWS Systems Manager Parameter Store attributes. During synthesis, the AWS CDK produces a token that is resolved by AWS CloudFormation during deployment.

The AWS CDK supports retrieving both plain and secure values. You may request a specific version of either kind of value. For plain values, you may omit the version from your request to retrieve the latest version. For secure values, you must specify the version when requesting the value of the secure attribute.

> **Note**
>
> This topic shows how to read attributes from the AWS Systems Manager Parameter Store. You can also read secrets from the AWS Secrets Manager (see Get a value from AWS Secrets Manager).

Read Systems Manager values at deployment time

To read values from the Systems Manager Parameter Store, use the `valueForStringParameter` and `valueForSecureStringParameter` methods. Choose a method based on whether the attribute you want is a plain string or a secure string value. These methods return tokens, not the actual value. The value is resolved by AWS CloudFormation during deployment. The following is an example:

Example

TypeScript

```
import * as ssm from 'aws-cdk-lib/aws-ssm';

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(
    this, 'my-secure-parameter-name', 1);   // must specify version
```

JavaScript

```
const ssm = require('aws-cdk-lib/aws-ssm');

// Get latest version or specified version of plain string attribute
const latestStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name');      // latest version
const versionOfStringToken = ssm.StringParameter.valueForStringParameter(
    this, 'my-plain-parameter-name', 1);   // version 1

// Get specified version of secure string attribute
const secureStringToken = ssm.StringParameter.valueForSecureStringParameter(
    this, 'my-secure-parameter-name', 1);   // must specify version
```

Python

```
import aws_cdk.aws_ssm as ssm
```

```
# Get latest version or specified version of plain string attribute
latest_string_token = ssm.StringParameter.value_for_string_parameter(
    self, "my-plain-parameter-name")
latest_string_token = ssm.StringParameter.value_for_string_parameter(
    self, "my-plain-parameter-name", 1)

# Get specified version of secure string attribute
secure_string_token = ssm.StringParameter.value_for_secure_string_parameter(
    self, "my-secure-parameter-name", 1)   # must specify version
```

Java

```
import software.amazon.awscdk.services.ssm.StringParameter;

//Get latest version or specified version of plain string attribute
String latestStringToken = StringParameter.valueForStringParameter(
            this, "my-plain-parameter-name");       // latest version
String versionOfStringToken = StringParameter.valueForStringParameter(
            this, "my-plain-parameter-name", 1);    // version 1

//Get specified version of secure string attribute
String secureStringToken = StringParameter.valueForSecureStringParameter(
            this, "my-secure-parameter-name", 1);   // must specify version
```

C#

```
using Amazon.CDK.AWS.SSM;

// Get latest version or specified version of plain string attribute
var latestStringToken = StringParameter.ValueForStringParameter(
    this, "my-plain-parameter-name");      // latest version
var versionOfStringToken = StringParameter.ValueForStringParameter(
    this, "my-plain-parameter-name", 1);   // version 1

// Get specified version of secure string attribute
var secureStringToken = StringParameter.ValueForSecureStringParameter(
    this, "my-secure-parameter-name", 1);   // must specify version
```

A limited number of AWS services currently support this feature.

Read Systems Manager values at synthesis time

At times, it's useful to provide a parameter at synthesis time. By doing this, the AWS CloudFormation template will always use the same value instead of resolving the value during deployment.

To read a value from the Systems Manager Parameter Store at synthesis time, use the `valueFromLookup` method (Python: `value_from_lookup`). This method returns the actual value of the parameter as a Context values and the AWS CDK value. If the value is not already cached in `cdk.json` or passed on the command line, it is retrieved from the current AWS account. For this reason, the stack *must* be synthesized with explicit AWS environment information.

The following is an example:

Example

TypeScript

```
import * as ssm from 'aws-cdk-lib/aws-ssm';

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

JavaScript

```
const ssm = require('aws-cdk-lib/aws-ssm');

const stringValue = ssm.StringParameter.valueFromLookup(this, 'my-plain-parameter-
name');
```

Python

```
import aws_cdk.aws_ssm as ssm

string_value = ssm.StringParameter.value_from_lookup(self, "my-plain-parameter-
name")
```

Java

```
import software.amazon.awscdk.services.ssm.StringParameter;
```

```
String stringValue = StringParameter.valueFromLookup(this, "my-plain-parameter-
name");
```

C#

```
using Amazon.CDK.AWS.SSM;

var stringValue = StringParameter.ValueFromLookup(this, "my-plain-parameter-name");
```

Only plain Systems Manager strings may be retrieved. Secure strings cannot be retrieved. The latest version will always be returned. Specific versions cannot be requested.

> **Important**
>
> The retrieved value will end up in your synthesized AWS CloudFormation template. This might be a security risk, depending on who has access to your AWS CloudFormation templates and what kind of value it is. Generally, don't use this feature for passwords, keys, or other values you want to keep private.

Write values to Systems Manager

You can use the AWS CLI, the AWS Management Console, or an AWS SDK to set Systems Manager parameter values. The following examples use the [`ssm put-parameter`](https://docs.aws.amazon.com/cli/latest/reference/ssm/put-parameter.html) CLI command.

```
aws ssm put-parameter --name "parameter-name" --type "String" --value "parameter-value"
aws ssm put-parameter --name "secure-parameter-name" --type "SecureString" --value
 "secure-parameter-value"
```

When updating an SSM value that already exists, also include the `--overwrite` option.

```
aws ssm put-parameter --overwrite --name "parameter-name" --type "String" --value
 "parameter-value"
aws ssm put-parameter --overwrite --name "secure-parameter-name" --type "SecureString"
 --value "secure-parameter-value"
```

Get a value from AWS Secrets Manager

To use values from AWS Secrets Manager in your AWS CDK app, use the `fromSecretAttributes()` method. It represents a value that is retrieved from Secrets Manager and used at AWS CloudFormation deployment time. The following is an example:

Example

TypeScript

```
import * as sm from "aws-cdk-lib/aws-secretsmanager";

export class SecretsManagerStack extends cdk.Stack {
  constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretCompleteArn:
        "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-
<random-6-characters>"
      // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
      // encryptionKey: ...
      }
    );
  }
}
```

JavaScript

```
const sm = require("aws-cdk-lib/aws-secretsmanager");

class SecretsManagerStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const secret = sm.Secret.fromSecretAttributes(this, "ImportedSecret", {
      secretCompleteArn:
        "arn:aws:secretsmanager:<region>:<account-id-number>:secret:<secret-name>-
<random-6-characters>"
      // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
```

```
      // encryptionKey: ...
    });
  }
}

module.exports = { SecretsManagerStack }
```

Python

```
import aws_cdk.aws_secretsmanager as sm

class SecretsManagerStack(cdk.Stack):
    def __init__(self, scope: cdk.App, id: str, **kwargs):
      super().__init__(scope, name, **kwargs)

      secret = sm.Secret.from_secret_attributes(self, "ImportedSecret",
          secret_complete_arn="arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>",
          # If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
          # encryption_key=....
      )
```

Java

```
import software.amazon.awscdk.services.secretsmanager.Secret;
import software.amazon.awscdk.services.secretsmanager.SecretAttributes;

public class SecretsManagerStack extends Stack {
    public SecretsManagerStack(App scope, String id) {
        this(scope, id, null);
    }

    public SecretsManagerStack(App scope, String id, StackProps props) {
        super(scope, id, props);

        Secret secret = (Secret)Secret.fromSecretAttributes(this, "ImportedSecret",
 SecretAttributes.builder()
             .secretCompleteArn("arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>")
             // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
             // .encryptionKey(...)
```

```
                .build());
    }
}
```

C#

```
using Amazon.CDK.AWS.SecretsManager;

public class SecretsManagerStack : Stack
{
    public SecretsManagerStack(App scope, string id, StackProps props) : base(scope,
 id, props) {

        var secret = Secret.FromSecretAttributes(this, "ImportedSecret", new
 SecretAttributes {
            SecretCompleteArn = "arn:aws:secretsmanager:<region>:<account-id-
number>:secret:<secret-name>-<random-6-characters>"
            // If the secret is encrypted using a KMS-hosted CMK, either import or
 reference that key:
            // encryptionKey = ...,
        });
    }
}
```

Tip

Use the AWS CLI `create-secret` CLI command to create a secret from the command line, such as when testing:

```
aws secretsmanager create-secret --name ImportedSecret --secret-string
 mygroovybucket
```

The command returns an ARN that you can use with the preceding example.

Once you have created a `Secret` instance, you can get the secret's value from the instance's `secretValue` attribute. The value is represented by a `SecretValue` instance, a special type of Tokens and the AWS CDK. Because it's a token, it has meaning only after resolution. Your CDK app does not need to access its actual value. Instead, the app can pass the `SecretValue` instance (or its string or numeric representation) to whatever CDK method needs the value.

Set a CloudWatch alarm

Use the `aws-cloudwatch` package to set up Amazon CloudWatch alarms on CloudWatch metrics. You can use predefined metrics or create your own.

Use an existing metric

Many AWS Construct Library modules let you set an alarm on an existing metric by passing the metric's name to a convenience method on an instance of an object that has metrics. For example, given an Amazon SQS queue, you can get the metric **ApproximateNumberOfMessagesVisible** from the queue's `metric()` method:

Example

TypeScript

```
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

JavaScript

```
const metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

Python

```
metric = queue.metric("ApproximateNumberOfMessagesVisible")
```

Java

```
Metric metric = queue.metric("ApproximateNumberOfMessagesVisible");
```

C#

```
var metric = queue.Metric("ApproximateNumberOfMessagesVisible");
```

Create your own metric

Create your own metric as follows, where the *namespace* value should be something like **AWS/SQS** for an Amazon SQS queue. You also need to specify your metric's name and dimension:

Example

TypeScript

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensionsMap: { MyDimension: 'MyDimensionValue' }
});
```

JavaScript

```
const metric = new cloudwatch.Metric({
  namespace: 'MyNamespace',
  metricName: 'MyMetric',
  dimensionsMap: { MyDimension: 'MyDimensionValue' }
});
```

Python

```
metric = cloudwatch.Metric(
    namespace="MyNamespace",
    metric_name="MyMetric",
    dimensionsMap=dict(MyDimension="MyDimensionValue")
)
```

Java

```
Metric metric = Metric.Builder.create()
        .namespace("MyNamespace")
        .metricName("MyMetric")
        .dimensionsMap(java.util.Map.of(    // Java 9 or later
            "MyDimension", "MyDimensionValue"))
        .build();
```

C#

```
var metric = new Metric(this, "Metric", new MetricProps
{
    Namespace = "MyNamespace",
    MetricName = "MyMetric",
    Dimensions = new Dictionary<string, object>
```

```
    {
        { "MyDimension", "MyDimensionValue" }
    }
});
```

Create the alarm

Once you have a metric, either an existing one or one you defined, you can create an alarm. In this example, the alarm is raised when there are more than 100 of your metric in two of the last three evaluation periods. You can use comparisons such as less-than in your alarms via the `comparisonOperator` property. Greater-than-or-equal-to is the AWS CDK default, so we don't need to specify it.

Example

TypeScript

```
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

JavaScript

```
const alarm = new cloudwatch.Alarm(this, 'Alarm', {
  metric: metric,
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2
});
```

Python

```
alarm = cloudwatch.Alarm(self, "Alarm",
    metric=metric,
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
```

```
)
```

Java

```
import software.amazon.awscdk.services.cloudwatch.Alarm;
import software.amazon.awscdk.services.cloudwatch.Metric;

Alarm alarm = Alarm.Builder.create(this, "Alarm")
        .metric(metric)
        .threshold(100)
        .evaluationPeriods(3)
        .datapointsToAlarm(2).build();
```

C#

```
var alarm = new Alarm(this, "Alarm", new AlarmProps
{
    Metric = metric,
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

An alternative way to create an alarm is using the metric's `createAlarm()` method, which takes essentially the same properties as the `Alarm` constructor. You don't need to pass in the metric, because it's already known.

Example

TypeScript

```
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

JavaScript

```
metric.createAlarm(this, 'Alarm', {
  threshold: 100,
```

```
  evaluationPeriods: 3,
  datapointsToAlarm: 2,
});
```

Python

```
metric.create_alarm(self, "Alarm",
    threshold=100,
    evaluation_periods=3,
    datapoints_to_alarm=2
)
```

Java

```
metric.createAlarm(this, "Alarm", new CreateAlarmOptions.Builder()
        .threshold(100)
        .evaluationPeriods(3)
        .datapointsToAlarm(2)
        .build());
```

C#

```
metric.CreateAlarm(this, "Alarm", new CreateAlarmOptions
{
    Threshold = 100,
    EvaluationPeriods = 3,
    DatapointsToAlarm = 2
});
```

Save and retrieve context variable values

You can specify context variables with the AWS Cloud Development Kit (AWS CDK) CLI or in the `cdk.json` file. Then, use the `TryGetContext` method to retrieve values.

Specify context variables

You can specify a context variable either as part of an AWS CDK CLI command, or in `cdk.json`.

To create a command line context variable, use the `--context` (`-c`) option, as shown in the following example.

```
cdk synth -c bucket_name=mygroovybucket
```

To specify the same context variable and value in the `cdk.json` file, use the following code.

```
{
  "context": {
    "bucket_name": "myotherbucket"
  }
}
```

If you specify a context variable using both the AWS CDK CLI and `cdk.json` file, the AWS CDK CLI value takes precedence.

Retrieve context variable values

To get the value of a context variable in your app, use the `TryGetContext` method in the context of a construct. (That is, when `this`, or `self` in Python, is an instance of some construct.)

In this example, we retrieve the value of the `bucket_name` context variable. If the requested value is not defined, `TryGetContext` returns `undefined` (None in Python; `null` in Java and C#; `nil` in Go) rather than raising an exception.

Example

TypeScript

```
const bucket_name = this.node.tryGetContext('bucket_name');
```

JavaScript

```
const bucket_name = this.node.tryGetContext('bucket_name');
```

Python

```
bucket_name = self.node.try_get_context("bucket_name")
```

Java

```
String bucketName = (String)this.getNode().tryGetContext("bucket_name");
```

C#

```
var bucketName = this.Node.TryGetContext("bucket_name");
```

Outside the context of a construct, you can access the context variable from the app object, like this.

Example

TypeScript

```
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name')
```

JavaScript

```
const app = new cdk.App();
const bucket_name = app.node.tryGetContext('bucket_name');
```

Python

```
app = cdk.App()
bucket_name = app.node.try_get_context("bucket_name")
```

Java

```
App app = App();
String bucketName = (String)app.getNode().tryGetContext("bucket_name");
```

C#

```
app = App();
var bucketName = app.Node.TryGetContext("bucket_name");
```

For more details on working with context variables, see Context values and the AWS CDK.

Use resources from the AWS CloudFormation Public Registry

The AWS CloudFormation Public Registry lets you manage extensions, both public and private, such as resources, modules, and hooks that are available for use in your AWS account. You can use public resource extensions in your AWS Cloud Development Kit (AWS CDK) applications with the `CfnResource` construct.

To learn more about the AWS CloudFormation Public Registry, see Using the AWS CloudFormation registry in the *AWS CloudFormation User Guide*.

All public extensions published by AWS are available to all accounts in all Regions without any action on your part. However, you must activate each third-party extension you want to use, in each account and Region where you want to use it.

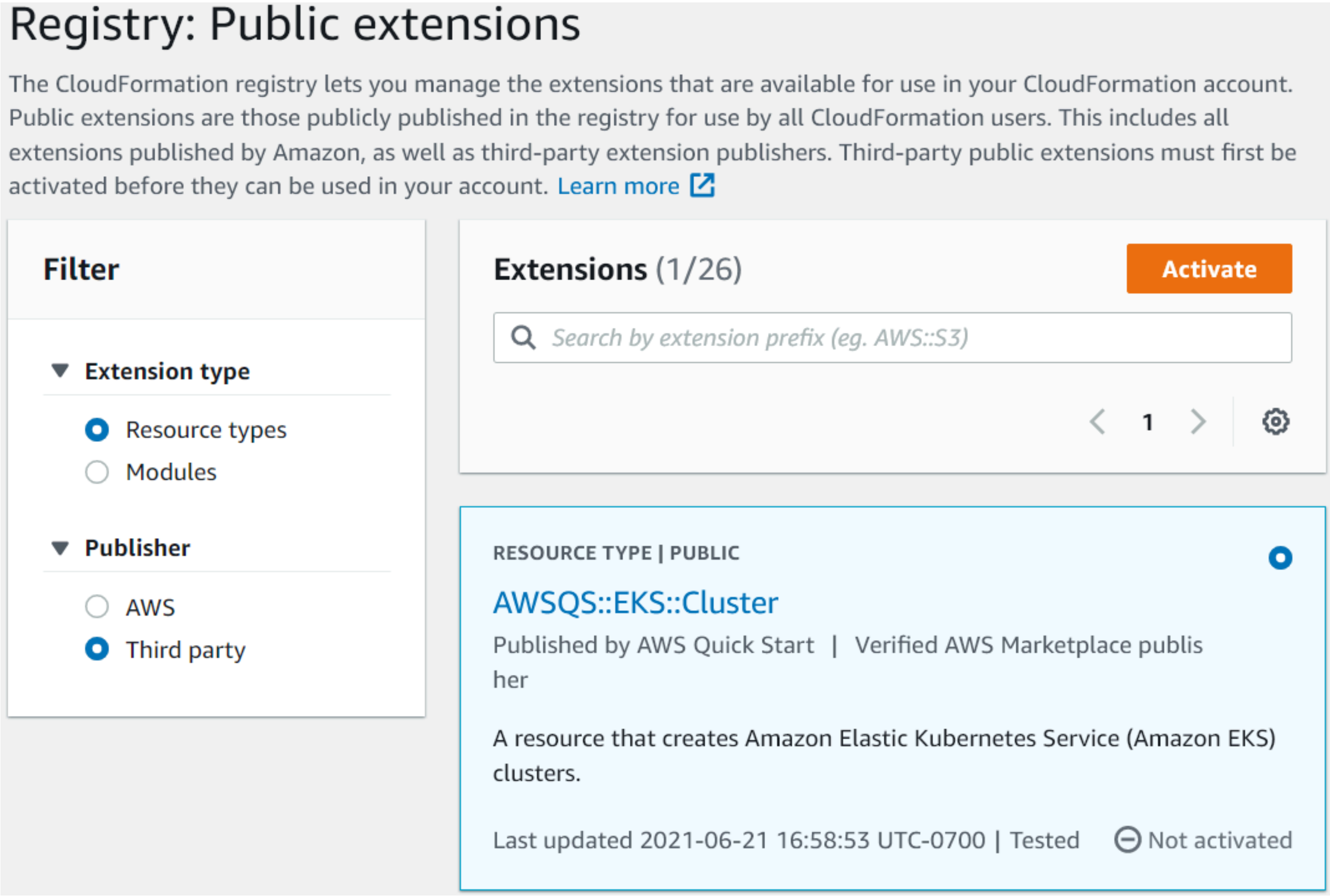
> **Note**
>
> When you use AWS CloudFormation with third-party resource types, you will incur charges. Charges are based on the number of handler operations you run per month and handler operation duration. See CloudFormation pricing for complete details.

To learn more about public extensions, see Using public extensions in CloudFormation in the *AWS CloudFormation User Guide*

Activate a third-party resource in your account and Region

Extensions published by AWS do not require activation. They are always available in every account and Region. You can activate a third-party extension through the AWS Management Console, via the AWS Command Line Interface, or by deploying a special AWS CloudFormation resource.

To activate a third-party extension through the AWS Management Console or see what resources are available



1. Sign in to the AWS account in which you want to use the extension, then switch to the Region where you want to use it.
2. Navigate to the CloudFormation console via the **Services** menu.
3. Choose **Public extensions** on the navigation bar, then activate the **Third party** radio button under **Publisher**. A list of the available third-party public extensions appears. (You may also choose **AWS** to see a list of the public extensions published by AWS, though you don't need to activate them.)
4. Browse the list and find the extension you want to activate. Alternatively, search for it, then activate the radio button in the upper right corner of the extension's card.
5. Choose the **Activate** button at the top of the list to activate the selected extension. The extension's **Activate** page appears.
6. In the **Activate** page, you can override the extension's default name and specify an execution role and logging configuration. You can also choose whether to automatically update the

extension when a new version is released. When you have set these options as you like, choose **Activate extension** at the bottom of the page.

To activate a third-party extension using the AWS CLI

- Use the `activate-type` command. Substitute the ARN of the custom type you want to use where indicated.

  The following is an example:

  ```
  aws cloudformation activate-type --public-type-arn <public_extension_ARN> --auto-update-activated
  ```

To activate a third-party extension through CloudFormation or CDK

1. Deploy a resource of type `AWS::CloudFormation::TypeActivation` and specify the following properties:

   a. `TypeName` - The name of the type, such as `AWSQS::EKS::Cluster`.

   b. `MajorVersion` - The major version number of the extension that you want. Omit if you want the latest version.

   c. `AutoUpdate` - Whether to automatically update this extension when a new minor version is released by the publisher. (Major version updates require explicitly changing the `MajorVersion` property.)

   d. `ExecutionRoleArn` - The ARN of the IAM role under which this extension will run.

   e. `LoggingConfig` - The logging configuration for the extension.

   The `TypeActivation` resource can be deployed by the CDK using the `CfnResource` construct. This is shown for the actual extensions in the following section.

Add a resource from the AWS CloudFormation Public Registry to your CDK app

Use the `CfnResource` construct to include a resource from the AWS CloudFormation Public Registry in your application. This construct is in the CDK's `aws-cdk-lib` module.

For example, suppose that there is a public resource named `MY::S5::UltimateBucket` that you want to use in your AWS CDK application. This resource takes one property: the bucket name. The corresponding `CfnResource` instantiation looks like this.

Example

TypeScript

```
const ubucket = new CfnResource(this, 'MyUltimateBucket', {
    type: 'MY::S5::UltimateBucket::MODULE',
    properties: {
           BucketName: 'UltimateBucket'
    }
});
```

JavaScript

```
const ubucket = new CfnResource(this, 'MyUltimateBucket', {
    type: 'MY::S5::UltimateBucket::MODULE',
    properties: {
           BucketName: 'UltimateBucket'
    }
});
```

Python

```
ubucket = CfnResource(self, "MyUltimateBucket",
            type="MY::S5::UltimateBucket::MODULE",
            properties=dict(
                BucketName="UltimateBucket"))
```

Java

```
CfnResource.Builder.create(this, "MyUltimateBucket")
 .type("MY::S5::UltimateBucket::MODULE")
 .properties(java.util.Map.of(    // Map.of requires Java 9+
     "BucketName", "UltimateBucket"))
 .build();
```

C#

```
new CfnResource(this, "MyUltimateBucket", new CfnResourceProps
{
    Type = "MY::S5::UltimateBucket::MODULE",
    Properties = new Dictionary<string, object>
    {
```

```
        ["BucketName"] = "UltimateBucket"
    }
});
```

Define permissions for L2 constructs with the AWS CDK

Define AWS Identity and Access Management (IAM) roles and policies for L2 constructs when using the AWS Cloud Development Kit (AWS CDK).

Use grant methods to define permissions

When you define your infrastructure using L2 constructs from the AWS Construct Library, you can use the provided *grant methods* to specify the permissions your resources will require. The AWS CDK will automatically create the IAM roles needed for all AWS resources that require them.

The following is an example that defines permissions between an AWS Lambda function and Amazon Simple Storage Service (Amazon S3) bucket. Here, the `grantRead` method of the Bucket L2 construct is used to define these permissions:

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as kms from 'aws-cdk-lib/aws-kms';

export class CdkDemoStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const key = new kms.Key(this, 'BucketKey');
    const bucket = new s3.Bucket(this, 'Bucket', {
      encryptionKey: key,
    });
    const handler = new lambda.Function(this, 'Handler', {
      runtime: lambda.Runtime.NODEJS_20_X,
      handler: 'index.handler',
      code: lambda.Code.fromAsset('lambda'),
```

```
    });

    // Define permissions between function and S3 bucket using grantRead method
    bucket.grantRead(handler);

  }
}
```

JavaScript

```
const { Stack, Duration } = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');
const lambda = require('aws-cdk-lib/aws-lambda');
const kms = require('aws-cdk-lib/aws-kms');

class CdkDemoStack extends Stack {

  constructor(scope, id, props) {
    super(scope, id, props);

    const key = new kms.Key(this, 'BucketKey');
    const bucket = new s3.Bucket(this, 'Bucket', {
      encryptionKey: key,
    });
    const handler = new lambda.Function(this, 'Handler', {
      runtime: lambda.Runtime.NODEJS_20_X,
      handler: 'index.handler',
      code: lambda.Code.fromAsset('lambda'),
    });

    // Define permissions between function and S3 bucket using grantRead method
    bucket.grantRead(handler);

  }
}

// ...
```

Python

```
from aws_cdk import (
    Stack,
    aws_s3 as s3,
```

```
    aws_lambda as _lambda,
    aws_kms as kms,
)
from constructs import Construct

class CdkDemoStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        key = kms.Key(self, 'BucketKey')
        bucket = s3.Bucket(self, 'Bucket')
        handler = _lambda.Function(
            self,
            'Handler',
            runtime = _lambda.Runtime.NODEJS_20_X,
            handler = 'index.handler',
            code = _lambda.Code.from_asset('lambda'),
        )

        # Define permissions between function and S3 bucket using grantRead method
        bucket.grantRead(handler)
```

Java

```
package com.myorg;

import software.amazon.awscdk.core.App;
import software.amazon.awscdk.core.Stack;
import software.amazon.awscdk.core.StackProps;
import software.amazon.awscdk.services.kms.Key;
import software.amazon.awscdk.services.kms.KeyProps;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketProps;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.FunctionProps;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Code;
import software.constructs.Construct;

public class CdkDemoStack extends Stack {
    public CdkDemoStack(final Construct scope, final String id) {
        this(scope, id, null);
```

```
    }

    public CdkDemoStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Key key = new Key(this, "BucketKey", KeyProps.builder().build());

        Bucket bucket = new Bucket(this, "Bucket", BucketProps.builder()
            .encryptionKey(key)
            .build());

        Function handler = new Function(this, "Handler", FunctionProps.builder()
            .runtime(Runtime.NODEJS_20_X)
            .handler("index.handler")
            .code(Code.fromAsset("lambda"))
            .build());

        // Define permissions between function and S3 bucket using grantRead method
        bucket.grantRead(handler);
    }

    public static void main(final String[] args) {
        App app = new App();

        new CdkDemoStack(app, "CdkDemoStack");

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.KMS;
using Amazon.CDK.AWS.S3;
using Amazon.CDK.AWS.Lambda;

namespace CdkDemo
{
    public class CdkDemoStack : Stack
    {
```

```
        internal CdkDemoStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var key = new Key(this, "BucketKey");

            var bucket = new Bucket(this, "Bucket", new BucketProps
            {
                EncryptionKey = key
            });

            var handler = new Function(this, "Handler", new FunctionProps
            {
                Runtime = Runtime.NODEJS_20_X,
                Handler = "index.handler",
                Code = Code.FromAsset("lambda")
            });

            // Define permissions between function and S3 bucket using grantRead
 method
            bucket.GrantRead(handler);
        }
    }
}
```

Go

```
package main
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/aws-cdk-go/awscdk/v2/awskms"
    "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
    "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)
// ...
func NewCdkDemoStack(scope constructs.Construct, id string, props
 *CdkDemoStackProps) awscdk.Stack {
    stack := awscdk.NewStack(scope, &id, &props.StackProps)
    key := awskms.NewKey(stack, jsii.String("BucketKey"), nil)
    bucket := awss3.NewBucket(stack, jsii.String("Bucket"), &awss3.BucketProps{
        EncryptionKey: key,
    })
```

```
    handler := awslambda.NewFunction(stack, jsii.String("Handler"),
 &awslambda.FunctionProps{
        Runtime: awslambda.Runtime_NODEJS_20_X(),
        Handler: jsii.String("index.handler"),
        Code:    awslambda.Code_FromAsset(jsii.String("lambda"),
 &awss3assets.AssetOptions{}),
    })
    bucket.GrantRead(handler)
    return stack
}
// ...
```

When you use grant methods of L2 constructs to define permissions between resources, the AWS CDK will create roles with least privilege policies based on the method you specify. As a security best practice, we recommend that you use the method that applies only the permissions that you require. For example, if you only need to grant permissions for a Lambda function to read from an Amazon S3 bucket, use the `grantRead` method instead of `grantReadWrite`.

For each method that you use, the CDK creates a unique IAM role for the specified resources. If necessary, you can also directly modify the policy that will be attached to the role. The following is an example:

Example

TypeScript

```
import { aws_iam as iam } from 'aws-cdk-lib';

handler.addToRolePolicy(new iam.PolicyStatement({
  actions: ['s3:GetObject', 's3:List*'],
  resources: [
    bucket.bucketArn,
    bucket.arnForObjects('*'),
  ]
}));
```

JavaScript

```
const iam = require('aws-cdk-lib/aws-iam');

handler.addToRolePolicy(new iam.PolicyStatement({
```

```
  actions: ['s3:GetObject', 's3:List*'],
  resources: [
    bucket.bucketArn,
    bucket.arnForObjects('*'),
  ]
}));
```

Python

```
from aws_cdk import aws_iam as iam

handler.add_to_role_policy(iam.PolicyStatement(
    actions=['s3:GetObject', 's3:List*'],
    resources=[
        bucket.bucket_arn,
        bucket.arn_for_objects('*'),
    ]
))
```

Java

```
import software.amazon.awscdk.services.iam.PolicyStatement;
import software.amazon.awscdk.services.iam.PolicyStatementProps;

handler.addToRolePolicy(PolicyStatement.Builder.create()
    .actions(Arrays.asList("s3:GetObject", "s3:List*"))
    .resources(Arrays.asList(
        bucket.getBucketArn(),
        bucket.arnForObjects("*")
    ))
    .build());
```

C#

```
using Amazon.CDK.AWS.IAM;
using Amazon.CDK.AWS.S3;
using Amazon.CDK.AWS.Lambda;

handler.AddToRolePolicy(new PolicyStatement(new PolicyStatementProps
{
    Actions = new[] { "s3:GetObject", "s3:List*" },
    Resources = new[] { bucket.BucketArn, bucket.ArnForObjects("*") }
```

```
}));
```

Go

```
package main

import (
 // ...
 "github.com/aws/aws-cdk-go/awscdk/v2/awsiam"
 // ...
)

// ...

func NewCdkDemoStack(scope constructs.Construct, id string, props
 *CdkDemoStackProps) awscdk.Stack {
 // ...

 handler.AddToRolePolicy(awsiam.NewPolicyStatement(&awsiam.PolicyStatementProps{
  Actions:   jsii.Strings("s3:GetObject", "s3:List*"),
  Resources: jsii.Strings(bucket.BucketArn(), bucket.ArnForObjects("*")),
 }))

 // ...
}
```

However, we recommend that you use the `grant` methods when available.

Manually create and use IAM roles

If you prefer not to use the CDK `grant` methods to create and manage permissions, you must manually create and configure them. You can create IAM roles using the AWS Management Console, AWS CLI, or AWS SDKs. Then, you can pass them into your CDK application manually or use the *role customization* feature.

Reference and manage all roles manually

Constructs that require a role have an optional `role` property that you can use to pass in a role object.

To reference a role manually

1. Use `Role.fromRoleName()` to reference your pre-existing role. The following is an example:

```
const existingRole = Role.fromRoleName(stack, 'Role', 'my-pre-existing-role', {
  mutable: false // Prevent CDK from attempting to add policies to this role
})
```

2. Pass the pre-existing role when defining your resource. The following is an example:

```
const handler = new lambda.Function(stack, 'Handler', { runtime:
 lambda.Runtime.NODEJS_20_XZ, handler:
         'index.handler', code: lambda.Code.fromAsset(path.join(__dirname,
 'lambda-handler')), // Pass in pre-existing role
         role: existingRole, });
```

Use the role customization feature

The AWS CDK *role customization* feature generates a report of roles and policies in your CDK app. You can use this feature to generate a report. Then you can substitute pre-created roles for them.

To use the role customization feature

1. Add `Role.customizeRoles()` somewhere towards the top of your CDK application. The following is an example:

```
const stack = new Stack(app, 'LambdaStack');

// Add this to use the role customization feature
iam.Role.customizeRoles(stack);

// Define your resources using L2 constructs
const key = new kms.Key(stack, 'BucketKey');
const bucket = new s3.Bucket(stack, 'Bucket', {
  encryptionKey: key,
});
const handler = new lambda.Function(stack, 'Handler', {
  runtime: lambda.Runtime.NODEJS_16_X,
  handler: 'index.handler',
  code: lambda.Code.fromAsset(path.join(__dirname, 'lambda-handler')),
});
```

```
// The grantRead() is still important. Even though it actually doesn't mutate
// any policies, it indicates the need for them.
bucket.grantRead(handler);
```

2. When you synthesize your application, the CDK will throw an error, indicating that you need to provide the pre-created role name to `Role.customizeRoles()`. The following is an example of the generated report:

```
<missing role> (LambdaStack/Handler/ServiceRole)

AssumeRole Policy:
[
  {
    "Action": "sts:AssumeRole",
    "Effect": "Allow",
    "Principal": {
      "Service": "lambda.amazonaws.com"
    }
  }
]

Managed Policy ARNs:
[
  "arn:(PARTITION):iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
]

Managed Policies Statements:
NONE

Identity Policy Statements:
[
  {
    "Action": [
      "s3:GetObject*",
      "s3:GetBucket*",
      "s3:List*"
    ],
    "Effect": "Allow",
    "Resource": [
      "(LambdaStack/Bucket/Resource.Arn)",
      "(LambdaStack/Bucket/Resource.Arn)/*"
    ]
```

```
  }
]
```

3. Once the role is created, you can pass it into your application for the resource that it applies to. For example, if the name of the role created for `LambdaStack/Handler/ServiceRole` is `lambda-service-role`, you would update your CDK app as follows:

```
const stack = new Stack(app, 'LambdaStack');

// Add this to pass in the role
iam.Role.customizeRoles(stack, {
  usePrecreatedRoles: {
    'LambdaStack/Handler/ServiceRole': 'lambda-service-role',
  },
});
```

   The CDK will now use the pre-created role name anywhere that the role is referenced in the CDK application. It will also continue to generate the report so that any future policy changes can be referenced.

   You will notice that the reference to the Amazon S3 bucket ARN in the report is rendered as (`LambdaStack/Bucket/Resource.Arn`) instead of the actual ARN of the bucket. This is because the bucket ARN is a deploy time value that is not known at synthesis (the bucket hasn't been created yet). This is another example of why we recommend allowing CDK to manage IAM roles and permissions by using the provided `grant` methods. In order to create the role with the initial policy, the admin will have to create the policy with broader permissions (for example, `arn:aws:s3:::*`).

Configure and perform CDK stack synthesis

Before you can deploy an AWS Cloud Development Kit (AWS CDK) stack, it must first be synthesized. *Stack synthesis* is the process of producing an AWS CloudFormation template and deployment artifacts from a CDK stack. The template and artifacts are known as the *cloud assembly*. The cloud assembly is what gets deployed to provision your resources on AWS. For more information on how deployments work, see How AWS CDK deployments work.

How synthesis and bootstrapping work together

For your CDK apps to properly deploy, the CloudFormation templates produced during synthesis must correctly specify the resources created during bootstrapping. Therefore, bootstrapping and synthesis must complement one another for a deployment to be successful:

- Bootstrapping is a one-time process of setting up an AWS environment for AWS CDK deployments. It configures specific AWS resources in your environment that are used by the CDK for deployments. These are commonly referred to as *bootstrap resources*. For instructions on bootstrapping, see Bootstrap your environment for use with the AWS CDK.
- CloudFormation templates produced during synthesis include information on which bootstrap resources to use. During synthesis, the CDK CLI doesn't know specifically how your AWS environment has been bootstrapped. Instead, the CDK CLI produces CloudFormation templates based on the synthesizer you configure for each CDK stack. For a deployment to be successful, the synthesizer must produce CloudFormation templates that reference the correct bootstrap resources to use.

The CDK comes with a default synthesizer and bootstrapping configuration that are designed to work together. If you customize one, you must apply relevant customizations to the other.

How to configure CDK stack synthesis

You configure CDK stack synthesis using the `synthesizer` property of your `Stack` instance. This property specifies how your CDK stacks will be synthesized. You provide an instance of a class that implements `IStackSynthesizer` or `IReusableStackSynthesizer`. Its methods will be invoked every time an asset is added to the stack or when the stack is synthesized. The following is a basic example of using this property within your stack:

Example

TypeScript

```
new MyStack(this, 'MyStack', {
  // stack properties
  synthesizer: new DefaultStackSynthesizer({
    // synthesizer properties
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  // stack properties
  synthesizer: new DefaultStackSynthesizer({
    // synthesizer properties
  }),
});
```

Python

```
MyStack(self, "MyStack",
    # stack properties
    synthesizer=DefaultStackSynthesizer(
        # synthesizer properties
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  // stack properties
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    // synthesizer properties
    .build())
  .build();
)
```

C#

```
new MyStack(app, "MyStack", new StackProps
```

```
// stack properties
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        // synthesizer properties
    })
});
```

Go

```
func main() {
  app := awscdk.NewApp(nil)

  NewMyStack(app, "MyStack", &MyStackProps{
    StackProps: awscdk.StackProps{
      Synthesizer:
 awscdk.NewDefaultStackSynthesizer(&awscdk.DefaultStackSynthesizerProps{
        // synthesizer properties
      }),
    },
  })

  app.Synth(nil)
}
```

You can also configure a synthesizer for all CDK stacks in your CDK app using the `defaultStackSynthesizer` property of your App instance:

Example

TypeScript

```
import { App, Stack, DefaultStackSynthesizer } from 'aws-cdk-lib';

const app = new App({
  // Configure for all stacks in this app
  defaultStackSynthesizer: new DefaultStackSynthesizer({
    /* ... */
  }),
});
```

JavaScript

```
const { App, Stack, DefaultStackSynthesizer } = require('aws-cdk-lib');

const app = new App({
  // Configure for all stacks in this app
  defaultStackSynthesizer: new DefaultStackSynthesizer({
    /* ... */
  }),
});
```

Python

```
from aws_cdk import App, Stack, DefaultStackSynthesizer

app = App(
    default_stack_synthesizer=DefaultStackSynthesizer(
        # Configure for all stacks in this app
        # ...
    )
)
```

Java

```
import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.DefaultStackSynthesizer;

public class Main {
    public static void main(final String[] args) {
        App app = new App(AppProps.builder()
            // Configure for all stacks in this app

 .defaultStackSynthesizer(DefaultStackSynthesizer.Builder.create().build())
            .build()
        );
    }
}
```

C#

```
using Amazon.CDK;
```

```
using Amazon.CDK.Synthesizers;

namespace MyNamespace
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App(new AppProps
            {
                // Configure for all stacks in this app
                DefaultStackSynthesizer = new DefaultStackSynthesizer(new
 DefaultStackSynthesizerProps
                {
                    // ...
                })
            });
        }
    }
}
```

Go

```
package main

import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)

func main() {
    defer jsii.Close()

    app := awscdk.NewApp(&awscdk.AppProps{
        // Configure for all stacks in this app
        DefaultStackSynthesizer:
 awscdk.NewDefaultStackSynthesizer(&awscdk.DefaultStackSynthesizerProps{
            // ...
        }),
    })
}
```

By default, the AWS CDK uses `DefaultStackSynthesizer`. If you don't configure a synthesizer, this synthesizer will be used.

If you don't modify bootstrapping, such as making changes to the bootstrap stack or template, you don't have to modify stack synthesis. You don't even have to provide a synthesizer. The CDK will use the default `DefaultStackSynthesizer` class to configure CDK stack synthesis to properly interact with your bootstrap stack.

How to synthesize a CDK stack

To synthesize a CDK stack, use the AWS CDK Command Line Interface (AWS CDK CLI) `cdk synth` command. For more information about this command, including options that you can use with this command, see cdk synthesize.

If your CDK app contains a single stack, or to synthesize all stacks, you don't have to provide the CDK stack name as an argument. By default, the CDK CLI will synthesize your CDK stacks into AWS CloudFormation templates. A `json` formatted template for each stack is saved to the `cdk.out` directory. If your app contains a single stack, a `yaml` formatted template is printed to `stdout`. The following is an example:

```
$ cdk synth
Resources:
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Analytics: v2:deflate64:H4sIAAAAAAAA/unique-identifier
    Metadata:
      aws:cdk:path: CdkAppStack/CDKMetadata/Default
    Condition: CDKMetadataAvailable
    ...
```

If your CDK app contains multiple stacks, you can provide the logical ID of a stack to synthesize a single stack. The following is an example:

```
$ cdk synth MyStackName
```

If you don't synthesize a stack and run `cdk deploy`, the CDK CLI will automatically synthesize your stack before deployment.

How synthesis works by default

Generated logical IDs in your AWS CloudFormation template

When you synthesize a CDK stack to produce a CloudFormation template, logical IDs are generated from the following sources, formatted as `construct-pathconstruct-IDunique-hash`:

- **Construct path** – The entire path to the construct in your CDK app. This path excludes the ID of the L1 construct, which is always `Resource` or `Default`, and the ID of the top-level stack that it's a part of.
- **Construct ID** – The ID that you provide as the second argument when instantiating your construct.
- **Unique hash** – The AWS CDK generates an 8 character unique hash using a deterministic hashing algorithm. This unique hash helps to ensure that logical ID values in your template are unique from one another. The deterministic behavior of this hash generation ensures that the generated logical ID value for each construct remains the same every time that you perform synthesis. The hash value will only change if you modify specific construct values such as your construct's ID or its path.

  Logical IDs have a maximum length of 255 characters. Therefore, the AWS CDK will truncate the construct path and construct ID if necessary to keep within that limit.

  The following is an example of a construct that defines an Amazon Simple Storage Service (Amazon S3) bucket. Here, we pass `myBucket` as the ID for our construct:

  Example

  TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct} from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

export class MyCdkAppStack extends cdk.Stack {
  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define the S3 bucket
    new s3.Bucket(this, 'myBucket', {
      versioned: true,
```

```
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class MyCdkAppStack extends cdk.Stack {

  constructor(scope, id, props) {
    super(scope, id, props);

    new s3.Bucket(this, 'myBucket', {
      versioned: true,
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });
  }
}

module.exports = { MyCdkAppStack }
```

Python

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk import Stack
from aws_cdk import aws_s3 as s3

class MyCdkAppStack(Stack):
  def __init__(self, scope: Construct, construct_id: str, **kwargs) - None:
    super().__init__(scope, construct_id, **kwargs)

    s3.Bucket(self, 'MyBucket',
      versioned=True,
      removal_policy=cdk.RemovalPolicy.DESTROY
    )
```

Java

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketProps;
import software.amazon.awscdk.RemovalPolicy;

public class MyCdkAppStack extends Stack {
    public MyCdkAppStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyCdkAppStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "myBucket")
            .versioned(true)
            .removalPolicy(RemovalPolicy.DESTROY)
            .build();
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.S3;

namespace MyCdkApp
{
    public class MyCdkAppStack : Stack
    {
        public MyCdkAppStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            new Bucket(this, "myBucket", new BucketProps
            {
```

```
                Versioned = true,
                RemovalPolicy = RemovalPolicy.DESTROY
            });
        }
    }
}
```

Go

```
package main

import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)

type MyCdkAppStackProps struct {
    awscdk.StackProps
}

func NewMyCdkAppStack(scope constructs.Construct, id string, props
 *MyCdkAppStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, id, sprops)

    awss3.NewBucket(stack, jsii.String("myBucket"), awss3.BucketProps{
      Versioned: jsii.Bool(true),
      RemovalPolicy: awscdk.RemovalPolicy_DESTROY,
    })

    return stack
}

// ...
```

When we run `cdk synth`, a logical ID in the format of `myBucketunique-hash` gets generated. The following is an example of this resource in the generated AWS CloudFormation template:

```
Resources:
  myBucket5AF9C99B:
    Type: AWS::S3::Bucket
    Properties:
      VersioningConfiguration:
        Status: Enabled
    UpdateReplacePolicy: Delete
    DeletionPolicy: Delete
    Metadata:
      aws:cdk:path: S3BucketAppStack/myBucket/Resource
```

The following is an example of a custom construct named `Bar` that defines an Amazon S3 bucket. The `Bar` construct includes the custom construct `Foo` in its path:

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

// Define the Bar construct
export class Bar extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);

    // Define an S3 bucket inside of Bar
    new s3.Bucket(this, 'Bucket', {
       versioned: true,
       removalPolicy: cdk.RemovalPolicy.DESTROY,
      } );
  }
}

// Define the Foo construct
export class Foo extends Construct {
  constructor(scope: Construct, id: string) {
```

```
    super(scope, id);

    // Create an instance of Bar inside Foo
    new Bar(this, 'Bar');
  }
}

// Define the CDK stack
export class MyCustomAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Instantiate Foo construct in the stack
    new Foo(this, 'Foo');
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');
const { Construct } = require('constructs');

// Define the Bar construct
class Bar extends Construct {
  constructor(scope, id) {
    super(scope, id);

    // Define an S3 bucket inside of Bar
    new s3.Bucket(this, 'Bucket', {
      versioned: true,
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });
  }
}

// Define the Foo construct
class Foo extends Construct {
  constructor(scope, id) {
    super(scope, id);

    // Create an instance of Bar inside Foo
    new Bar(this, 'Bar');
```

```
  }
}

// Define the CDK stack
class MyCustomAppStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Instantiate Foo construct in the stack
    new Foo(this, 'Foo');
  }
}

module.exports = { MyCustomAppStack }
```

Python

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk import (
    Stack,
    aws_s3 as s3,
    RemovalPolicy,
)

# Define the Bar construct
class Bar(Construct):
    def __init__(self, scope: Construct, id: str) - None:
        super().__init__(scope, id)

        # Define an S3 bucket inside of Bar
        s3.Bucket(self, 'Bucket',
            versioned=True,
            removal_policy=RemovalPolicy.DESTROY
        )

# Define the Foo construct
class Foo(Construct):
    def __init__(self, scope: Construct, id: str) - None:
        super().__init__(scope, id)

        # Create an instance of Bar inside Foo
        Bar(self, 'Bar')
```

```
# Define the CDK stack
class MyCustomAppStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs) - None:
        super().__init__(scope, id, **kwargs)

        # Instantiate Foo construct in the stack
        Foo(self, 'Foo')
```

Java

In `my-custom-app/src/main/java/com/myorg/Bar.java`:

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketProps;
import software.amazon.awscdk.RemovalPolicy;

public class Bar extends Construct {
    public Bar(final Construct scope, final String id) {
        super(scope, id);

        // Define an S3 bucket inside Bar
        Bucket.Builder.create(this, "Bucket")
            .versioned(true)
            .removalPolicy(RemovalPolicy.DESTROY)
            .build();
    }
}
```

In `my-custom-app/src/main/java/com/myorg/Foo.java`:

```
package com.myorg;

import software.constructs.Construct;

public class Foo extends Construct {
    public Foo(final Construct scope, final String id) {
        super(scope, id);
```

```
        // Create an instance of Bar inside Foo
        new Bar(this, "Bar");
    }
}
```

In `my-custom-app/src/main/java/com/myorg/MyCustomAppStack.java`:

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class MyCustomAppStack extends Stack {
    public MyCustomAppStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Instantiate Foo construct in the stack
        new Foo(this, "Foo");
    }

    // Overload constructor in case StackProps is not provided
    public MyCustomAppStack(final Construct scope, final String id) {
        this(scope, id, null);
    }
}
```

C#

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.S3;

namespace MyCustomApp
{
    // Define the Bar construct
    public class Bar : Construct
    {
        public Bar(Construct scope, string id) : base(scope, id)
        {
            // Define an S3 bucket inside Bar
```

```
            new Bucket(this, "Bucket", new BucketProps
            {
                Versioned = true,
                RemovalPolicy = RemovalPolicy.DESTROY
            });
        }
    }

    // Define the Foo construct
    public class Foo : Construct
    {
        public Foo(Construct scope, string id) : base(scope, id)
        {
            // Create an instance of Bar inside Foo
            new Bar(this, "Bar");
        }
    }

    // Define the CDK Stack
    public class MyCustomAppStack : Stack
    {
        public MyCustomAppStack(Construct scope, string id, StackProps props =
 null) : base(scope, id, props)
        {
            // Instantiate Foo construct in the stack
            new Foo(this, "Foo");
        }
    }
}
```

Go

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 "github.com/aws/aws-cdk-go/awscdk/v2/awss3"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
)

// Define the Bar construct
type Bar struct {
```

```
 constructs.Construct
}

func NewBar(scope constructs.Construct, id string) constructs.Construct {
 bar := constructs.NewConstruct(scope, id)

 // Define an S3 bucket inside Bar
 awss3.NewBucket(bar, jsii.String("Bucket"), awss3.BucketProps{
  Versioned:     jsii.Bool(true),
  RemovalPolicy: awscdk.RemovalPolicy_DESTROY,
 })

 return bar
}

// Define the Foo construct
type Foo struct {
 constructs.Construct
}

func NewFoo(scope constructs.Construct, id string) constructs.Construct {
 foo := constructs.NewConstruct(scope, id)

 // Create an instance of Bar inside Foo
 NewBar(foo, "Bar")

 return foo
}

// Define the CDK Stack
type MyCustomAppStackProps struct {
 awscdk.StackProps
}

func NewMyCustomAppStack(scope constructs.Construct, id string, props
 *MyCustomAppStackProps) awscdk.Stack {
 stack := awscdk.NewStack(scope, id, props.StackProps)

 // Instantiate Foo construct in the stack
 NewFoo(stack, "Foo")

 return stack
}
```

```
// Define the CDK App
func main() {
 app := awscdk.NewApp(nil)

 NewMyCustomAppStack(app, "MyCustomAppStack", MyCustomAppStackProps{
  StackProps: awscdk.StackProps{},
 })

 app.Synth(nil)
}
```

When we run `cdk synth`, a logical ID in the format of `FooBarBucketunique-hash` gets generated. The following is an example of this resource in the generated AWS CloudFormation template:

```
Resources:
  FooBarBucketBA3ED1FA:
    Type: AWS::S3::Bucket
    Properties:
      VersioningConfiguration:
        Status: Enabled
    UpdateReplacePolicy: Delete
    DeletionPolicy: Delete
    # ...
```

Customize CDK stack synthesis

If the default CDK synthesis behavior doesn't suit your needs, you can customize CDK synthesis. To do this, you modify `DefaultStackSynthesizer`, use other available built-in synthesizers, or create your own synthesizer. For instructions, see Customize CDK stack synthesis.

Customize CDK stack synthesis

You can customize AWS Cloud Development Kit (AWS CDK) stack synthesis by modifying the default synthesizer, using other available built-in synthesizers, or creating your own synthesizer.

The AWS CDK includes the following built-in synthesizers that you can use to customize synthesis behavior:

- `DefaultStackSynthesizer` – If you don't specify a synthesizer, this one is used automatically. It supports cross-account deployments and deployments using the CDK Pipelines construct. Its bootstrap contract requires an existing Amazon S3 bucket with a known name, an existing Amazon ECR repository with a known name, and five existing IAM roles with known names. The default bootstrapping template meets these requirements.
- `CliCredentialsStackSynthesizer` – This synthesizer's bootstrap contract requires an existing Amazon S3 bucket and existing Amazon ECR repository. It does not require any IAM roles. To perform deployments, this synthesizer relies on the permissions of the CDK CLI user and is recommend for organizations that want to restrict IAM deployment credentials. This synthesizer doesn't support cross-account deployments or CDK Pipelines.
- `LegacyStackSynthesizer` – This synthesizer emulates CDK v1 synthesis behavior. Its bootstrap contract requires an existing Amazon S3 bucket of an arbitrary name and expects the locations of assets to be passed in as CloudFormation stack parameters. If you use this synthesizer, you must use the CDK CLI to perform deployment.

If none of these built-in synthesizers are appropriate for your use case, you can write your own synthesizer as a class that implements `IStackSynthesizer` or look at synthesizers from the Construct Hub.

Customize the `DefaultStackSynthesizer`

The `DefaultStackSynthesizer` is the default synthesizer for the AWS CDK. It is designed to allow cross-account deployments of CDK applications, as well as deploying CDK apps from a CI/CD system that does not have explicit support for the AWS CDK, but supports regular CloudFormation deployments, such as AWS CodePipeline. This synthesizer is the best option for most use cases.

`DefaultStackSynthesizer` bootstrap contract

`DefaultStackSynthesizer` requires the following bootstrap contract. These are the resources that must be created during bootstrapping:

Bootstrap resource	Description	Default expected resource name	Purpose
Amazon S3 bucket	Staging bucket	cdk-hnb659fds-assets-<ACCOUNT>-<REGION>	Stores file assets.

Bootstrap resource	Description	Default expected resource name	Purpose
Amazon ECR repository	Staging repository	cdk-hnb659fds-container-assets-<ACCOUNT>-<REGION>	Stores and manages Docker image assets.
IAM role	Deploy role	cdk-hnb659fds-deploy-role-<ACCOUNT>-<REGION>	Assumed by the CDK CLI and potentially CodePipeline to assume other roles and start the AWS CloudFormation deployment. The trust policy of this role controls who can deploy with the AWS CDK in this AWS environment.
IAM role	AWS CloudFormation execution role	cdk-hnb659fds-cfn-exec-role-<ACCOUNT>-<REGION>	This role is used by AWS CloudFormation to perform the deployment. The policies of this role control what operations the CDK deployment can perform.

Bootstrap resource	Description	Default expected resource name	Purpose
IAM role	Lookup role	cdk-hnb659fds-lookup-role-<ACCOUNT>-<REGION>	This role is used when the CDK CLI needs to perform environmental context lookups. The trust policy of this role controls who can look up information in the environment.
IAM role	File publishing role	cdk-hnb659fds-file-publishing-role-<ACCOUNT>-<REGION>	This role is used to upload assets to the Amazon S3 staging bucket. It is assumed from the deploy role.
IAM role	Image publishing role	cdk-hnb659fds-image-publishing-role-<ACCOUNT>-<REGION>	This role is used to upload Docker images to the Amazon ECR staging repository. It is assumed from the deploy role.
SSM parameter	Bootstrap version parameter	/cdk-bootstrap/hnb659fds/<version>	The version of the bootstrap template. It is used by the bootstrap template and the CDK CLI to validate requirements.

One way to customize CDK stack synthesis, is by modifying the `DefaultStackSynthesizer`. You can customize this synthesizer for a single CDK stack using the `synthesizer` property of your `Stack` instance. You can also modify `DefaultStackSynthesizer` for all stacks in your CDK app using the `defaultStackSynthesizer` property of your `App` instance.

Change the qualifier

The *qualifier* is added to the name of resources created during bootstrapping. By default, this value is `hnb659fds`. When you modify the qualifier during bootstrapping, you need to customize CDK stack synthesis to use the same qualifier.

To change the qualifier, configure the `qualifier` property of `DefaultStackSynthesizer` or configure the qualifier as a context key in your CDK project's `cdk.json` file.

The following is an example of configuring the `qualifier` property of the `DefaultStackSynthesizer`:

Example

TypeScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new DefaultStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
})
```

Python

```
MyStack(self, "MyStack",
    synthesizer=DefaultStackSynthesizer(
        qualifier="MYQUALIFIER"
```

```
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  .synthesizer(DefaultStackSynthesizer.Builder.create()
    .qualifier("MYQUALIFIER")
    .build())
  .build();
)
```

C#

```
new MyStack(app, "MyStack", new StackProps
{
    Synthesizer = new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
    {
        Qualifier = "MYQUALIFIER"
    })
});
```

Go

```
func NewMyStack(scope constructs.Construct, id string, props *MyStackProps)
 awscdk.Stack {
 var sprops awscdk.StackProps
 if props != nil {
  sprops = props.StackProps
 }
 stack := awscdk.NewStack(scope, &id, &sprops)

 synth := awscdk.NewDefaultStackSynthesizer(&awscdk.DefaultStackSynthesizerProps{
  Qualifier: jsii.String("MYQUALIFIER"),
 })

 stack.SetSynthesizer(synth)

 return stack
}
```

The following is an example of configuring the qualifier as a context key in `cdk.json`:

```
{
  "app": "...",
  "context": {
    "@aws-cdk/core:bootstrapQualifier": "MYQUALIFIER"
  }
}
```

Change resource names

All of the other `DefaultStackSynthesizer` properties relate to the names of the resources in the bootstrap template. You only need to provide any of these properties if you modified the bootstrap template and changed the resource names or naming scheme.

All properties accept the special placeholders `${Qualifier}`, `${AWS::Partition}`, `${AWS::AccountId}`, and `${AWS::Region}`. These placeholders are replaced with the values of the `qualifier` parameter and the AWS partition, account ID, and AWS Region values for the stack's environment, respectively.

The following example shows the most commonly used properties for `DefaultStackSynthesizer` along with their default values, as if you were instantiating the synthesizer. For a complete list, see [`DefaultStackSynthesizerProps`]:

Example

TypeScript

```
new DefaultStackSynthesizer({
  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',

  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',
  dockerTagPrefix: '',

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  deployRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}',
  deployRoleExternalId: '',
```

```
  // ARN of the role used for file asset publishing (assumed from the CLI role)
  fileAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}',
  fileAssetPublishingExternalId: '',

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  imageAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}',
  imageAssetPublishingExternalId: '',

  // ARN of the role passed to CloudFormation to execute the deployments
  cloudFormationExecutionRole: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}',

  // ARN of the role used to look up context information in an environment
  lookupRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}',
  lookupRoleExternalId: '',

  // Name of the SSM parameter which describes the bootstrap stack version number
  bootstrapStackVersionSsmParameter: '/cdk-bootstrap/${Qualifier}/version',

  // Add a rule to every template which verifies the required bootstrap stack
 version
  generateBootstrapVersionRule: true,

})
```

JavaScript

```
new DefaultStackSynthesizer({
  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',

  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',
  dockerTagPrefix: '',

  // ARN of the role assumed by the CLI and Pipeline to deploy here
  deployRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}',
```

```
  deployRoleExternalId: '',

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  fileAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}',
  fileAssetPublishingExternalId: '',

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  imageAssetPublishingRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}',
  imageAssetPublishingExternalId: '',

  // ARN of the role passed to CloudFormation to execute the deployments
  cloudFormationExecutionRole: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}',

  // ARN of the role used to look up context information in an environment
  lookupRoleArn: 'arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}',
  lookupRoleExternalId: '',

  // Name of the SSM parameter which describes the bootstrap stack version number
  bootstrapStackVersionSsmParameter: '/cdk-bootstrap/${Qualifier}/version',

  // Add a rule to every template which verifies the required bootstrap stack
 version
  generateBootstrapVersionRule: true,
})
```

Python

```
DefaultStackSynthesizer(
  # Name of the S3 bucket for file assets
  file_assets_bucket_name="cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
  bucket_prefix="",

  # Name of the ECR repository for Docker image assets
  image_assets_repository_name="cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}",
  docker_tag_prefix="",

  # ARN of the role assumed by the CLI and Pipeline to deploy here
```

```
  deploy_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}",
  deploy_role_external_id="",

  # ARN of the role used for file asset publishing (assumed from the CLI role)
  file_asset_publishing_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}",
  file_asset_publishing_external_id="",

  # ARN of the role used for Docker asset publishing (assumed from the CLI role)
  image_asset_publishing_role_arn="arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-
${AWS::Region}",
  image_asset_publishing_external_id="",

  # ARN of the role passed to CloudFormation to execute the deployments
  cloud_formation_execution_role="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}",

  # ARN of the role used to look up context information in an environment
  lookup_role_arn="arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}",
  lookup_role_external_id="",

  # Name of the SSM parameter which describes the bootstrap stack version number
  bootstrap_stack_version_ssm_parameter="/cdk-bootstrap/${Qualifier}/version",

  # Add a rule to every template which verifies the required bootstrap stack version
  generate_bootstrap_version_rule=True,
)
```

Java

```
DefaultStackSynthesizer.Builder.create()
  // Name of the S3 bucket for file assets
  .fileAssetsBucketName("cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}")
  .bucketPrefix('')

  // Name of the ECR repository for Docker image assets
  .imageAssetsRepositoryName("cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}")
  .dockerTagPrefix('')
```

```
  // ARN of the role assumed by the CLI and Pipeline to deploy here
  .deployRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}")
  .deployRoleExternalId("")

  // ARN of the role used for file asset publishing (assumed from the CLI role)
  .fileAssetPublishingRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}")
  .fileAssetPublishingExternalId("")

  // ARN of the role used for Docker asset publishing (assumed from the CLI role)
  .imageAssetPublishingRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-${AWS::Region}")
  .imageAssetPublishingExternalId("")

  // ARN of the role passed to CloudFormation to execute the deployments
  .cloudFormationExecutionRole("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-${AWS::Region}")

  .lookupRoleArn("arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}")
  .lookupRoleExternalId("")

  // Name of the SSM parameter which describes the bootstrap stack version number
  .bootstrapStackVersionSsmParameter("/cdk-bootstrap/${Qualifier}/version")

  // Add a rule to every template which verifies the required bootstrap stack
 version
  .generateBootstrapVersionRule(true)
.build()
```

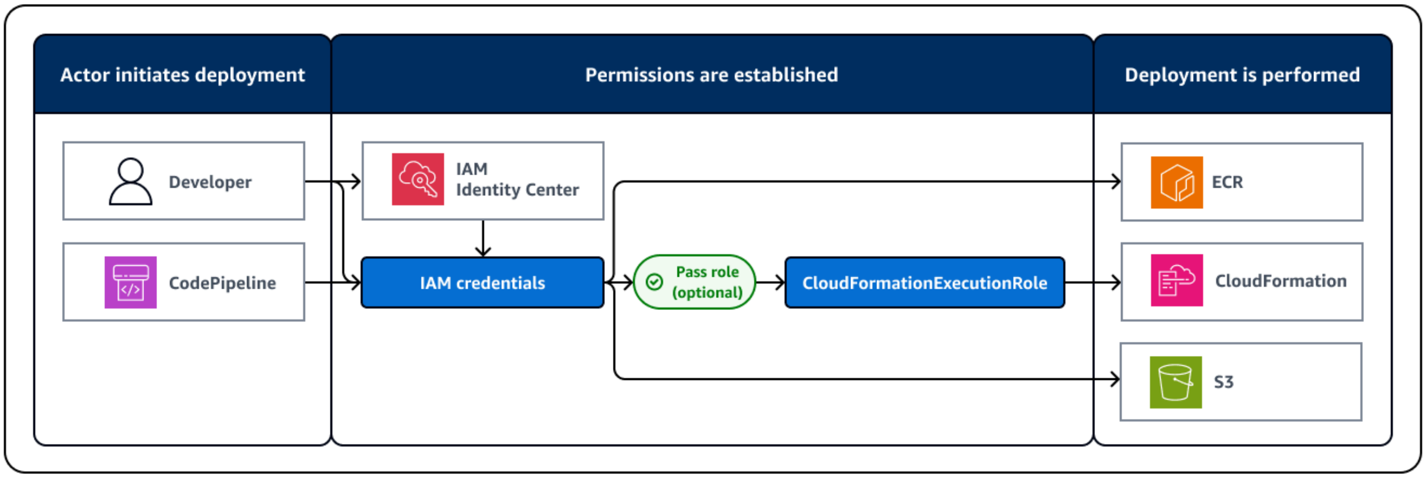
C#

```
new DefaultStackSynthesizer(new DefaultStackSynthesizerProps
{
    // Name of the S3 bucket for file assets
    FileAssetsBucketName = "cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
    BucketPrefix = "",

    // Name of the ECR repository for Docker image assets
    ImageAssetsRepositoryName = "cdk-${Qualifier}-container-assets-
${AWS::AccountId}-${AWS::Region}",
```

```
    DockerTagPrefix = "",

    // ARN of the role assumed by the CLI and Pipeline to deploy here
    DeployRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-deploy-role-${AWS::AccountId}-${AWS::Region}",
    DeployRoleExternalId = "",

    // ARN of the role used for file asset publishing (assumed from the CLI role)
    FileAssetPublishingRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/
cdk-${Qualifier}-file-publishing-role-${AWS::AccountId}-${AWS::Region}",
    FileAssetPublishingExternalId = "",

    // ARN of the role used for Docker asset publishing (assumed from the CLI role)
    ImageAssetPublishingRoleArn = "arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-image-publishing-role-${AWS::AccountId}-
${AWS::Region}",
    ImageAssetPublishingExternalId = "",

    // ARN of the role passed to CloudFormation to execute the deployments
    CloudFormationExecutionRole = "arn:${AWS::Partition}:iam::
${AWS::AccountId}:role/cdk-${Qualifier}-cfn-exec-role-${AWS::AccountId}-
${AWS::Region}",

    LookupRoleArn = "arn:${AWS::Partition}:iam::${AWS::AccountId}:role/cdk-
${Qualifier}-lookup-role-${AWS::AccountId}-${AWS::Region}",
    LookupRoleExternalId = "",

    // Name of the SSM parameter which describes the bootstrap stack version number
    BootstrapStackVersionSsmParameter = "/cdk-bootstrap/${Qualifier}/version",

    // Add a rule to every template which verifies the required bootstrap stack
 version
    GenerateBootstrapVersionRule = true,
})
```

Use `CliCredentialsStackSynthesizer`

To modify the security credentials used to provide permissions during CDK deployments, you can customize synthesis by using `CliCredentialsStackSynthesizer`. This synthesizer works with the default AWS resources that are created during bootstrapping to store assets, such as the Amazon S3 bucket and Amazon ECR repository. Instead of using the default IAM roles created by the CDK during bootstrapping, it uses the security credentials of the actor initiating deployment. Therefore, the security credentials of the actor must have valid permissions to perform all deployment actions. The following diagram illustrates the deployment process when using this synthesizer:



When using `CliCredentialsStackSynthesizer`:

- By default, CloudFormation performs API calls in your account using the permissions of the actor. Therefore, the current identity must have permission to make necessary changes to the AWS resources in the CloudFormation stack, along with the permissions to perform necessary CloudFormation operations, such as `CreateStack` or `UpdateStack`. Deployment capabilities will be limited to the permissions of the actor.
- Asset publishing and CloudFormation deployments will be done using the current IAM identity. This identity must have sufficient permissions to both read from and write to the asset bucket and repository.
- Lookups are performed using the current IAM identity, and lookups are subject to its policies.

When using this synthesizer, you can use a separate CloudFormation execution role by specifying it using the `--role-arn` option with any CDK CLI command.

`CliCredentialsStackSynthesizer` bootstrap contract

`CliCredentialsStackSynthesizer` requires the following bootstrap contract. These are the resources that must be created during bootstrapping:

Bootstrap resource	Description	Default expected resource name	Purpose
Amazon S3 bucket	Staging bucket	cdk-hnb659fds-assets-<ACCOUNT>-<REGION>	Stores file assets.
Amazon ECR repository	Staging repository	cdk-hnb659fds-container-assets-<ACCOUNT>-<REGION>	Stores and manages Docker image assets.

The string `hnb659fds` in the resource name is called the *qualifier*. Its default value has no special significance. You can have multiple copies of the bootstrap resources in a single environment as long as they have a different qualifier. Having multiple copies can be useful for keeping assets of different applications in the same environment separated.

You can deploy the default bootstrap template to satisfy `CliCredentialsStackSynthesizer`'s bootstrap contract. The default bootstrap template will create IAM roles, but this synthesizer will not use them. You can also customize the bootstrap template to remove the IAM roles.

Modify `CliCredentialsStackSynthesizer`

If you change the qualifier or any of the default bootstrap resource names during bootstrapping, you have to modify the synthesizer to use the same names. You can modify the synthesizer for a single stack or for all stacks in your app. The following is an example:

Example

TypeScript

```
new MyStack(this, 'MyStack', {
  synthesizer: new CliCredentialsStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
});
```

JavaScript

```
new MyStack(this, 'MyStack', {
```

```
  synthesizer: new CliCredentialsStackSynthesizer({
    qualifier: 'MYQUALIFIER',
  }),
})
```

Python

```
MyStack(self, "MyStack",
    synthesizer=CliCredentialsStackSynthesizer(
        qualifier="MYQUALIFIER"
))
```

Java

```
new MyStack(app, "MyStack", StackProps.builder()
  .synthesizer(CliCredentialsStackSynthesizer.Builder.create()
    .qualifier("MYQUALIFIER")
    .build())
  .build();
)
```

C#

```
new MyStack(app, "MyStack", new StackProps
{
    Synthesizer = new CliCredentialsStackSynthesizer(new
 CliCredentialsStackSynthesizerProps
    {
        Qualifier = "MYQUALIFIER"
    })
});
```

The following example shows the most commonly used properties for `CliCredentialsStackSynthesizer` along with their default values. For a complete list, see [`CliCredentialsStackSynthesizerProps`]:

Example

TypeScript

```
new CliCredentialsStackSynthesizer({
```

```
  // Value for '${Qualifier}' in the resource names
  qualifier: 'hnb659fds',

  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',

  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',
  dockerTagPrefix: '',
})
```

JavaScript

```
new CliCredentialsStackSynthesizer({
  // Value for '${Qualifier}' in the resource names
  qualifier: 'hnb659fds',

  // Name of the S3 bucket for file assets
  fileAssetsBucketName: 'cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}',
  bucketPrefix: '',

  // Name of the ECR repository for Docker image assets
  imageAssetsRepositoryName: 'cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}',
  dockerTagPrefix: '',
})
```

Python

```
CliCredentialsStackSynthesizer(
  # Value for '${Qualifier}' in the resource names
  qualifier="hnb659fds",

  # Name of the S3 bucket for file assets
  file_assets_bucket_name="cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
  bucket_prefix="",

  # Name of the ECR repository for Docker image assets
  image_assets_repository_name="cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}",
```

```
  docker_tag_prefix="",
)
```

Java

```
CliCredentialsStackSynthesizer.Builder.create()
  // Value for '${Qualifier}' in the resource names
  .qualifier("hnb659fds")

  // Name of the S3 bucket for file assets
  .fileAssetsBucketName("cdk-${Qualifier}-assets-${AWS::AccountId}-${AWS::Region}")
  .bucketPrefix('')

  // Name of the ECR repository for Docker image assets
  .imageAssetsRepositoryName("cdk-${Qualifier}-container-assets-${AWS::AccountId}-
${AWS::Region}")
  .dockerTagPrefix('')
.build()
```

C#

```
new CliCredentialsStackSynthesizer(new CliCredentialsStackSynthesizerProps
{

    // Value for '${Qualifier}' in the resource names
    Qualifier = "hnb659fds",

    // Name of the S3 bucket for file assets
    FileAssetsBucketName = "cdk-${Qualifier}-assets-${AWS::AccountId}-
${AWS::Region}",
    BucketPrefix = "",

    // Name of the ECR repository for Docker image assets
    ImageAssetsRepositoryName = "cdk-${Qualifier}-container-assets-
${AWS::AccountId}-${AWS::Region}",
    DockerTagPrefix = "",
})
```

Use `LegacyStackSynthesizer`

The `LegacyStackSynthesizer` emulates the behavior of CDK v1 deployments. The security credentials of the actor performing deployment will be used to establish permissions. File assets will be uploaded to a bucket that must be created using a AWS CloudFormation stack named `CDKToolkit`. The CDK CLI will create an unmanaged Amazon ECR repository named `aws-cdk/assets` to store Docker image assets. You will be responsible to clean up and manage this repository. Stacks synthesized using the `LegacyStackSynthesizer` can only be deployed using the CDK CLI.

You can use the `LegacyStackSynthesizer` if you are migrating from CDK v1 to CDK v2, and are unable to re-bootstrap your environments. For new projects, we recommend that you don't use `LegacyStackSynthesizer`.

`LegacyStackSynthesizer` bootstrap contract

`LegacyStackSynthesizer` requires the following bootstrap contract. These are the resources that must be created during bootstrapping:

Bootstrap resource	Description	Default expected resource name	Purpose
Amazon S3 bucket	Staging bucket	cdk-hnb659fds-assets-<ACCOUNT>-<REGION>	Stores file assets.
CloudFormation output	Bucket name output	Stack – `CDKToolkit` Output name – `BucketName`	A CloudFormation output describing the name of the staging bucket

The `LegacyStackSynthesizer` does not assume the existence of an Amazon S3 bucket with a fixed name. Instead, the synthesized CloudFormation template will contain three CloudFormation parameters for each file asset. These parameters will store the Amazon S3 bucket name, Amazon S3 object key, and artifact hash for each file asset.

Docker image assets will be published to an Amazon ECR repository named `aws-cdk/assets`. This name can be changed per asset. The repositories will be created if they do not exist.

A CloudFormation stack must exist with the default name `CDKToolkit`. This stack must have a CloudFormation export named `BucketName` that refers to the staging bucket.

The default bootstrap template satisfies the `LegacyStackSynthesizer` bootstrap contract. However, only the Amazon S3 bucket from the bootstrap resources of the bootstrap template will be used. You can customize the bootstrap template to remove the Amazon ECR, IAM, and SSM bootstrap resources.

`LegacyStackSynthesizer` deployment process

When you use this synthesizer, the following process is performed during deployment:

- The CDK CLI looks for a CloudFormation stack named `CDKToolkit` in your environment. From this stack, the CDK CLI reads the CloudFormation output named `BucketName`. You can use the `--toolkit-stack-name` option with `cdk deploy` to specify a different stack name.
- The security credentials of the actor initiating deployment will be used to establish permissions for deployment. Therefore, the actor must have sufficient permissions to perform all deployment actions. This includes reading and writing to the Amazon S3 staging bucket, creating and writing to the Amazon ECR repository, starting and monitoring AWS CloudFormation deployments, and performing any API calls necessary for deployment.
- If necessary, and if permissions are valid, file assets will be published to the Amazon S3 staging bucket.
- If necessary, and if permissions are valid, Docker image assets are published to the repository named by the `repositoryName` property of the asset. The default value is `'aws-cdk/assets'` if you don’t provide a repository name.
- If permissions are valid, the AWS CloudFormation deployment is performed. The locations of the Amazon S3 staging bucket and keys are passed as CloudFormation parameters.

Deploy AWS CDK applications

An AWS Cloud Development Kit (AWS CDK) deployment is the process of provisioning your infrastructure on AWS.

How AWS CDK deployments work

The AWS CDK utilizes the AWS CloudFormation service to perform deployments. Before you deploy, you synthesize your CDK stacks. This creates a CloudFormation template and deployment artifacts for each CDK stack in your app. Deployments are initiated from a local development machine or from a *continuous integration and continuous delivery (CI/CD)* environment. During deployment, assets are uploaded to the bootstrapped resources and the CloudFormation template is submitted to CloudFormation to provision your AWS resources.

For a deployment to be successful, the following is required:

- The AWS CDK Command Line Interface (AWS CDK CLI) must be provided with valid permissions.
- The AWS environment must be bootstrapped.
- The AWS CDK must know the bootstrapped resources to upload assets into.

Prerequisites for CDK deployments

Before you can deploy an AWS CDK application, you must complete the following:

- Configure security credentials for the CDK CLI.
- Bootstrap your AWS environment.
- Configure an AWS environment for each of your CDK stacks.
- Develop your CDK app.

Configure security credentials

To use the CDK CLI to interact with AWS, you must configure security credentials on your local machine. For instructions, see Configure security credentials for the AWS CDK CLI.

Bootstrap your AWS environment

A deployment is always associated with one or more AWS environments. Before you can deploy, the environment must first be bootstrapped. Bootstrapping provisions resources in your environment that the CDK uses to perform and manage deployments. These resources include an Amazon Simple Storage Service (Amazon S3) bucket and Amazon Elastic Container Registry (Amazon ECR) repository to store and manage assets. These resources also include AWS Identity and Access Management (IAM) roles that are used to provide permissions during development and deployment.

We recommend that you use the AWS CDK Command Line Interface (AWS CDK CLI) `cdk bootstrap` command to bootstrap your environment. You can customize bootstrapping or manually create these resources in your environment if necessary. For instructions, see Bootstrap your environment for use with the AWS CDK.

Configure AWS environments

Each CDK stack must be associated with an environment to determine where the stack is deployed to. For instructions, see Configure environments to use with the AWS CDK.

Develop your CDK app

Within a CDK project, you create and develop your CDK app. Within your app, you create one or more CDK stacks. Within your stacks, you import and use constructs from the AWS Construct Library to define your infrastructure. Before you can deploy, your CDK app must contain at least one stack.

CDK app synthesis

To perform synthesis, we recommend that you use the CDK CLI `cdk synth` command. The `cdk deploy` command will also perform synthesis before initiating deployment. However, by using `cdk synth`, you can validate your CDK app and catch errors before initiating deployment.

Synthesis behavior is determined by the stack synthesizer that you configure for your CDK stack. If you don't configure a synthesizer, `DefaultStackSynthesizer` will be used. You can also configure and customize synthesis to meet your needs. For instructions, see Configure and perform CDK stack synthesis.

For your synthesized CloudFormation template to deploy successfully into your environment, it must be compatible with how your environment was bootstrapped. For example, your CloudFormation template must specify the correct Amazon S3 bucket to deploy assets into. If you use the default method of bootstrapping your environment, the default stack synthesizer will work. If you customize CDK behavior, such as customizing bootstrapping or synthesis, CDK deployment behavior may vary.

The app lifecycle

When you perform synthesis, your CDK app is run through the following phases, known as the *app lifecycle*:

Construction (or Initialization)

Your code instantiates all of the defined constructs and then links them together. In this stage, all of the constructs (app, stacks, and their child constructs) are instantiated and the constructor chain is run. Most of your app code is run in this stage.

Preparation

All constructs that have implemented the `prepare` method participate in a final round of modifications, to set up their final state. The preparation phase happens automatically. As a user, you don't see any feedback from this phase. It's rare to need to use the "prepare" hook, and generally not recommended. Be very careful when mutating the construct tree during this phase, because the order of operations could impact behavior.

During this phase, once the construct tree has been built, any aspects that you have configured are applied as well.

Validation

All constructs that have implemented the `validate` method can validate themselves to ensure that they're in a state that will correctly deploy. You will get notified of any validation failures that happen during this phase. Generally, we recommend performing validation as soon as possible (usually as soon as you get some input) and throwing exceptions as early as possible. Performing validation early improves reliability as stack traces will be more accurate, and ensures that your code can continue to execute safely.

Synthesis

This is the final stage of running your CDK app. It's triggered by a call to `app.synth()`, and it traverses the construct tree and invokes the `synthesize` method on all constructs. Constructs that implement `synthesize` can participate in synthesis and produce deployment artifacts to the resulting cloud assembly. These artifacts include CloudFormation templates, AWS Lambda application bundles, file and Docker image assets, and other deployment artifacts. In most cases, you won't need to implement the `synthesize` method.

Running your app

The CDK CLI needs to know how to run your CDK app. If you created the project from a template using the `cdk init` command, your app's `cdk.json` file includes an `app` key. This key specifies the necessary command for the language that the app is written in. If your language requires compilation, the command line performs this step before running the app automatically.

Example

TypeScript

```
{
  "app": "npx ts-node --prefer-ts-exts bin/my-app.ts"
}
```

JavaScript

```
{
  "app": "node bin/my-app.js"
}
```

Python

```
{
    "app": "python app.py"
}
```

Java

```
{
  "app": "mvn -e -q compile exec:java"
}
```

C#

```
{
  "app": "dotnet run -p src/MyApp/MyApp.csproj"
}
```

Go

```
{
  "app": "go mod download && go run my-app.go"
}
```

If you didn't create your project using the CDK CLI, or if you want to override the command line given in `cdk.json`, you can provide the `--app` option when running the `cdk` command.

```
$ cdk --app '<executable>' <cdk-command> ...
```

The `<executable>` part of the command indicates the command that should be run to execute your CDK application. Use quotation marks as shown, since such commands contain spaces. The `<cdk-command>` is a subcommand like `synth` or `deploy` that tells the CDK CLI what you want to do with your app. Follow this with any additional options needed for that subcommand.

The CDK CLI can also interact directly with an already-synthesized cloud assembly. To do that, pass the directory in which the cloud assembly is stored in `--app`. The following example lists the stacks defined in the cloud assembly stored under `./my-cloud-assembly`.

```
$ cdk --app <./my-cloud-assembly> ls
```

Cloud assemblies

The call to `app.synth()` is what tells the AWS CDK to synthesize a cloud assembly from an app. Typically you don't interact directly with cloud assemblies. They are files that include everything needed to deploy your app to a cloud environment. For example, it includes an AWS CloudFormation template for each stack in your app. It also includes a copy of any file assets or Docker images that you reference in your app.

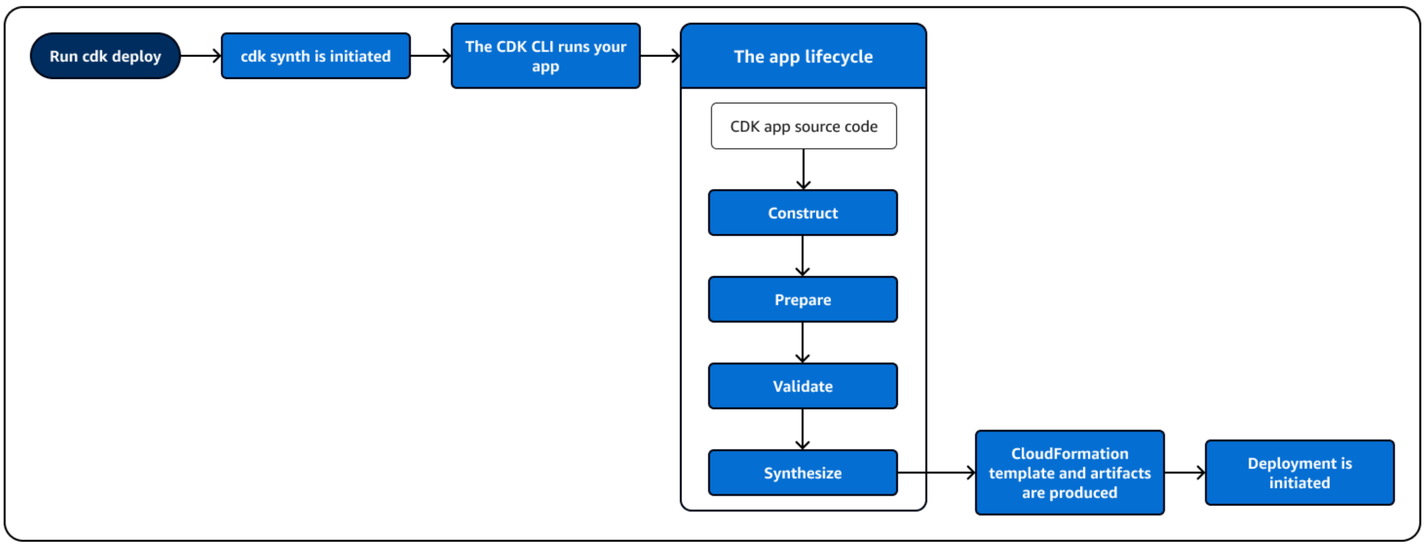
See the cloud assembly specification for details on how cloud assemblies are formatted.

To interact with the cloud assembly that your AWS CDK app creates, you typically use the AWS CDK CLI. However, any tool that can read the cloud assembly format can be used to deploy your app.

Deploy your application

To deploy your application, we recommend that you use the CDK CLI `cdk deploy` command to initiate deployments or to configure automated deployments.

When you run `cdk deploy`, the CDK CLI initiates `cdk synth` to prepare for deployment. The following diagram illustrates the app lifecycle in the context of a deployment:



During deployment, the CDK CLI takes the cloud assembly produced by synthesis and deploys it to an AWS environment. Assets are uploaded to Amazon S3 and Amazon ECR and the CloudFormation template is submitted to AWS CloudFormation for deployment.

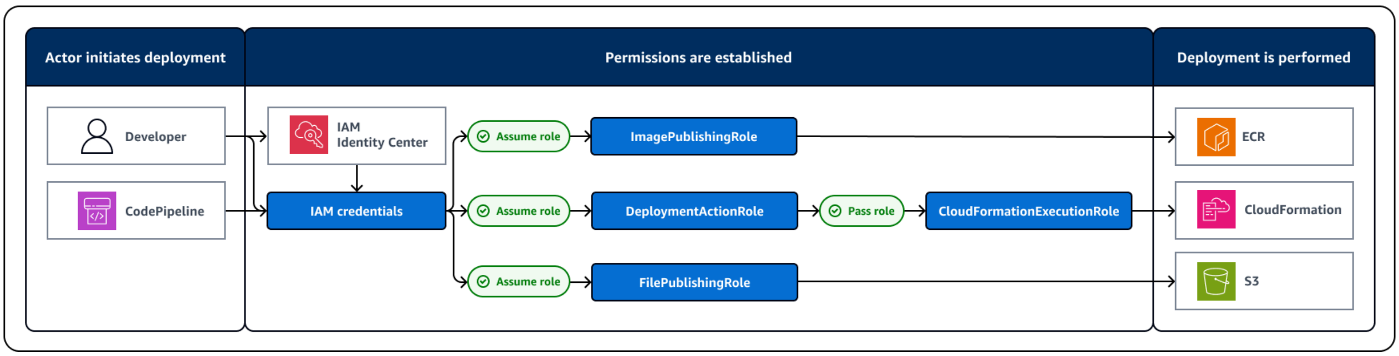
By the time the AWS CloudFormation deployment phase starts, your CDK app has already finished running and exited. This has the following implications:

- The CDK app can't respond to events that happen during deployment, such as a resource being created or the whole deployment finishing. To run code during the deployment phase, you must inject it into the AWS CloudFormation template as a custom resource. For more information about adding a custom resource to your app, see the AWS CloudFormation module, or the custom-resource example. You can also configure the Triggers module to run code during deployments.

- The CDK app might have to work with values that can't be known at the time it runs. For example, if the AWS CDK app defines an Amazon S3 bucket with an automatically generated name, and you retrieve the `bucket.bucketName` (Python: `bucket_name`) attribute, that value is not the name of the deployed bucket. Instead, you get a `Token` value. To determine whether a particular value is available, call `cdk.isUnresolved(value)` (Python: `is_unresolved`). See Tokens and the AWS CDK for details.

Deployment permissions

Before deployment can be performed, permissions must be established. The following diagram illustrates the permissions that are used during a default deployment, when using the default bootstrapping process and stack synthesizer:



Actor initiates deployment

Deployments are initiated by an *actor*, using the CDK CLI. An actor can either be a person, or a service such as AWS CodePipeline.

If necessary, the CDK CLI runs `cdk synth` when you run `cdk deploy`. During synthesis, the AWS identity assumes the `LookupRole` to perform context lookups in the AWS environment.

Permissions are established

First, the actor's security credentials are used to authenticate to AWS and obtain the first IAM identity in the process. For human actors, how security credentials are configured and obtained depends on how you or your organization manages users. For more information, see Configure security credentials for the AWS CDK CLI. For service actors, such as CodePipeline, an IAM execution role is assumed and used.

Next, the IAM roles created in your AWS environment during bootstrapping are used to establish permissions to perform the actions needed for deployment. For more information about these roles and what they grant permissions for, see IAM roles created during bootstrapping. This process includes the following:

- The AWS identity assumes the `DeploymentActionRole` role and passes the `CloudFormationExecutionRole` role to CloudFormation, ensuring that CloudFormation assumes the role when it performs any actions in your AWS environment. `DeploymentActionRole` grants permission to perform deployments into your environment and `CloudFormationExecutionRole` determines what actions CloudFormation can perform.
- The AWS identity assumes the `FilePublishingRole`, which determines the actions that can be performed on the Amazon S3 bucket created during bootstrapping.
- The AWS identity assumes the `ImagePublishingRole`, which determines the actions that can be performed on the Amazon ECR repository created during bootstrapping.
- If necessary, the AWS identity assumes the `LookupRole` to perform context lookups in the AWS environment. This action may also be performed during template synthesis.

Deployment is performed

During deployment, the CDK CLI reads the bootstrap version parameter to confirm the bootstrap version number. AWS CloudFormation also reads this parameter at deployment time to confirm. If permissions across the deployment workflow are valid, deployment is performed. Assets are uploaded to the bootstrapped resources and the CloudFormation template produced at synthesis is deployed using the CloudFormation service as a CloudFormation stack to provision your resources.

AWS CDK policy validation at synthesis time

Policy validation at synthesis time

If you or your organization use any policy validation tool, such as AWS CloudFormation Guard or OPA, to define constraints on your AWS CloudFormation template, you can integrate them with the AWS CDK at synthesis time. By using the appropriate policy validation plugin, you can make the AWS CDK application check the generated AWS CloudFormation template against your policies immediately after synthesis. If there are any violations, the synthesis will fail and a report will be printed to the console.

The validation performed by the AWS CDK at synthesis time validate controls at one point in the deployment lifecycle, but they can’t affect actions that occur outside synthesis. Examples include actions taken directly in the console or via service APIs. They aren't resistant to alteration of AWS CloudFormation templates after synthesis. Some other mechanism to validate the same rule set more authoritatively should be set up independently, like [AWS CloudFormation hooks] or [AWS Config]. Nevertheless, the ability of the AWS CDK to evaluate the rule set during development is still useful as it will improve detection speed and developer productivity.

The goal of AWS CDK policy validation is to minimize the amount of set up needed during development, and make it as easy as possible.

> **Note**
>
> This feature is considered experimental, and both the plugin API and the format of the validation report are subject to change in the future.

For application developers

To use one or more validation plugins in your application, use the `policyValidationBeta1` property of `Stage`:

```
import { CfnGuardValidator } from '@cdklabs/cdk-validator-cfnguard';
const app = new App({
  policyValidationBeta1: [
    new CfnGuardValidator()
  ],
});
// only apply to a particular stage
const prodStage = new Stage(app, 'ProdStage', {
  policyValidationBeta1: [...],
});
```

Immediately after synthesis, all plugins registered this way will be invoked to validate all the templates generated in the scope you defined. In particular, if you register the templates in the App object, all templates will be subject to validation.

> **Warning**
>
> Other than modifying the cloud assembly, plugins can do anything that your AWS CDK application can. They can read data from the filesystem, access the network etc. It's your responsibility as the consumer of a plugin to verify that it's secure to use.

AWS CloudFormation Guard plugin

Using the `CfnGuardValidator` plugin allows you to use AWS CloudFormation Guard to perform policy validations. The `CfnGuardValidator` plugin comes with a select set of AWS Control Tower proactive controls built in. The current set of rules can be found in the project documentation. As mentioned in Policy validation at synthesis time, we recommend that organizations set up a more authoritative method of validation using AWS CloudFormation hooks.

For AWS Control Tower customers, these same proactive controls can be deployed across your organization. When you enable AWS Control Tower proactive controls in your AWS Control Tower environment, the controls can stop the deployment of non-compliant resources deployed via AWS CloudFormation. For more information about managed proactive controls and how they work, see the AWS Control Tower documentation.

These AWS CDK bundled controls and managed AWS Control Tower proactive controls are best used together. In this scenario you can configure this validation plugin with the same proactive controls that are active in your AWS Control Tower cloud environment. You can then quickly gain confidence that your AWS CDK application will pass the AWS Control Tower controls by running `cdk synth` locally.

Validation Report

When you synthesize the AWS CDK app the validator plugins will be called and the results will be printed. An example report is showing below.

```
Validation Report (CfnGuardValidator)
-------------------------------------
(Summary)
#######################################
# Status    # failure                 #
#######################################
# Plugin    # CfnGuardValidator       #
```

```
######################################
(Violations)
Ensure S3 Buckets are encrypted with a KMS CMK (1 occurrences)
Severity: medium
  Occurrences:

    - Construct Path: MyStack/MyCustomL3Construct/Bucket
    - Stack Template Path: ./cdk.out/MyStack.template.json
    - Creation Stack:
        ###  MyStack (MyStack)
             # Library: aws-cdk-lib.Stack
             # Library Version: 2.50.0
             # Location: Object.<anonymous> (/home/johndoe/tmp/cdk-tmp-app/src/
main.ts:25:20)
             ###  MyCustomL3Construct (MyStack/MyCustomL3Construct)
                  # Library: N/A - (Local Construct)
                  # Library Version: N/A
                  # Location: new MyStack (/home/johndoe/tmp/cdk-tmp-app/src/
main.ts:15:20)
                  ###  Bucket (MyStack/MyCustomL3Construct/Bucket)
                       # Library: aws-cdk-lib/aws-s3.Bucket
                       # Library Version: 2.50.0
                       # Location: new MyCustomL3Construct (/home/johndoe/tmp/cdk-tmp-
app/src/main.ts:9:20)
    - Resource Name: amzn-s3-demo-bucket
    - Locations:
      > BucketEncryption/ServerSideEncryptionConfiguration/0/
ServerSideEncryptionByDefault/SSEAlgorithm
  Recommendation: Missing value for key `SSEAlgorithm` - must specify `aws:kms`
  How to fix:
    > Add to construct properties for `cdk-app/MyStack/Bucket`
      `encryption: BucketEncryption.KMS`

Validation failed. See above reports for details
```

By default, the report will be printed in a human readable format. If you want a report in JSON format, enable it using the `@aws-cdk/core:validationReportJson` via the CLI or passing it directly to the application:

```
const app = new App({
  context: { '@aws-cdk/core:validationReportJson': true },
});
```

Alternatively, you can set this context key-value pair using the `cdk.json` or `cdk.context.json` files in your project directory (see Context values and the AWS CDK).

If you choose the JSON format, the AWS CDK will print the policy validation report to a file called `policy-validation-report.json` in the cloud assembly directory. For the default, human-readable format, the report will be printed to the standard output.

For plugin authors

Plugins

The AWS CDK core framework is responsible for registering and invoking plugins and then displaying the formatted validation report. The responsibility of the plugin is to act as the translation layer between the AWS CDK framework and the policy validation tool. A plugin can be created in any language supported by AWS CDK. If you are creating a plugin that might be consumed by multiple languages then it's recommended that you create the plugin in `TypeScript` so that you can use JSII to publish the plugin in each AWS CDK language.

Creating plugins

The communication protocol between the AWS CDK core module and your policy tool is defined by the `IPolicyValidationPluginBeta1` interface. To create a new plugin you must write a class that implements this interface. There are two things you need to implement: the plugin name (by overriding the `name` property), and the `validate()` method.

The framework will call `validate()`, passing an `IValidationContextBeta1` object. The location of the templates to be validated is given by `templatePaths`. The plugin should return an instance of `ValidationPluginReportBeta1`. This object represents the report that the user wil receive at the end of the synthesis.

```
validate(context: IPolicyValidationContextBeta1): PolicyValidationReportBeta1 {
  // First read the templates using context.templatePaths...
  // ...then perform the validation, and then compose and return the report.
  // Using hard-coded values here for better clarity:
  return {
    success: false,
    violations: [{
      ruleName: 'CKV_AWS_117',
      description: 'Ensure that AWS Lambda function is configured inside a VPC',
      fix: 'https://docs.bridgecrew.io/docs/ensure-that-aws-lambda-function-is-
configured-inside-a-vpc-1',
```

```
      violatingResources: [{
        resourceName: 'MyFunction3BAA72D1',
        templatePath: '/home/johndoe/myapp/cdk.out/MyService.template.json',
        locations: 'Properties/VpcConfig',
      }],
    }],
  };
}
```

Note that plugins aren't allowed to modify anything in the cloud assembly. Any attempt to do so will result in synthesis failure.

If your plugin depends on an external tool, keep in mind that some developers may not have that tool installed in their workstations yet. To minimize friction, we highly recommend that you provide some installation script along with your plugin package, to automate the whole process. Better yet, run that script as part of the installation of your package. With `npm`, for example, you can add it to the `postinstall` [script](https://docs.npmjs.com/cli/v9/using-npm/scripts) in the `package.json` file.

Handling Exemptions

If your organization has a mechanism for handling exemptions, it can be implemented as part of the validator plugin.

An example scenario to illustrate a possible exemption mechanism:

- An organization has a rule that public Amazon S3 buckets aren't allowed, *except* for under certain scenarios.
- A developer is creating an Amazon S3 bucket that falls under one of those scenarios and requests an exemption (create a ticket for example).
- Security tooling knows how to read from the internal system that registers exemptions

In this scenario the developer would request an exception in the internal system and then will need some way of "registering" that exception. Adding on to the guard plugin example, you could create a plugin that handles exemptions by filtering out the violations that have a matching exemption in an internal ticketing system.

See the existing plugins for example implementations.

- @cdklabs/cdk-validator-cfnguard

Continuous integration and delivery (CI/CD) using CDK Pipelines

Use the CDK Pipelines module from the AWS Construct Library to configure continuous delivery of AWS CDK applications. When you commit your CDK app's source code into AWS CodeCommit, GitHub, or AWS CodeStar, CDK Pipelines can automatically build, test, and deploy your new version.

CDK Pipelines are self-updating. If you add application stages or stacks, the pipeline automatically reconfigures itself to deploy those new stages or stacks.

> **Note**
>
> CDK Pipelines supports two APIs. One is the original API that was made available in the CDK Pipelines Developer Preview. The other is a modern API that incorporates feedback from CDK customers received during the preview phase. The examples in this topic use the modern API. For details on the differences between the two supported APIs, see CDK Pipelines original API in the *aws-cdk GitHub repository*.

Bootstrap your AWS environments

Before you can use CDK Pipelines, you must bootstrap the AWS environment that you will deploy your stacks to.

A CDK Pipeline involves at least two environments. The first environment is where the pipeline is provisioned. The second environment is where you want to deploy the application's stacks or stages to (stages are groups of related stacks). These environments can be the same, but a best practice recommendation is to isolate stages from each other in different environments.

> **Note**
>
> See AWS CDK bootstrapping for more information on the kinds of resources created by bootstrapping and how to customize the bootstrap stack.

Continuous deployment with CDK Pipelines requires the following to be included in the CDK Toolkit stack:

- An Amazon Simple Storage Service (Amazon S3) bucket.
- An Amazon ECR repository.
- IAM roles to give the various parts of a pipeline the permissions they need.

The CDK Toolkit will upgrade your existing bootstrap stack or creates a new one if necessary.

To bootstrap an environment that can provision an AWS CDK pipeline, invoke `cdk bootstrap` as shown in the following example. Invoking the AWS CDK Toolkit via the `npx` command temporarily installs it if necessary. It will also use the version of the Toolkit installed in the current project, if one exists.

`--cloudformation-execution-policies` specifies the ARN of a policy under which future CDK Pipelines deployments will execute. The default `AdministratorAccess` policy makes sure that your pipeline can deploy every type of AWS resource. If you use this policy, make sure you trust all the code and dependencies that make up your AWS CDK app.

Most organizations mandate stricter controls on what kinds of resources can be deployed by automation. Check with the appropriate department within your organization to determine the policy your pipeline should use.

You can omit the `--profile` option if your default AWS profile contains the necessary authentication configuration and AWS Region.

Example

macOS/Linux

```
npx cdk bootstrap aws://<ACCOUNT-NUMBER>/<REGION> --profile <ADMIN-PROFILE> \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
```

Windows

```
npx cdk bootstrap aws://<ACCOUNT-NUMBER></REGION> --profile< ADMIN-PROFILE> ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
```

To bootstrap additional environments into which AWS CDK applications will be deployed by the pipeline, use the following commands instead. The `--trust` option indicates which other account should have permissions to deploy AWS CDK applications into this environment. For this option, specify the pipeline's AWS account ID.

Again, you can omit the `--profile` option if your default AWS profile contains the necessary authentication configuration and AWS Region.

Example

macOS/Linux

```
npx cdk bootstrap aws://<ACCOUNT-NUMBER>/<REGION> --profile <ADMIN-PROFILE> \
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
 \
    --trust <PIPELINE-ACCOUNT-NUMBER>
```

Windows

```
npx cdk bootstrap aws://<ACCOUNT-NUMBER>/<REGION> --profile <ADMIN-PROFILE> ^
    --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess
 ^
    --trust <PIPELINE-ACCOUNT-NUMBER>
```

> **Tip**
>
> Use administrative credentials only to bootstrap and to provision the initial pipeline. Afterward, use the pipeline itself, not your local machine, to deploy changes.

If you are upgrading a legacy bootstrapped environment, the previous Amazon S3 bucket is orphaned when the new bucket is created. Delete it manually by using the Amazon S3 console.

Protecting your bootstrap stack from deletion

If a bootstrap stack is deleted, the AWS resources that were originally provisioned in the environment to support CDK deployments will also be deleted. This will cause the pipeline to stop working. If this happens, there is no general solution for recovery.

After your environment is bootstrapped, do not delete and recreate the environment's bootstrap stack. Instead, try to update the bootstrap stack to a new version by running the `cdk bootstrap` command again.

To protect against accidental deletion of your bootstrap stack, we recommend that you provide the `--termination-protection` option with the `cdk bootstrap` command to enable termination protection. You can enable termination protection on new or existing bootstrap stacks. To learn more about this option, see `--termination-protection`.

After enabling termination protection, you can use the AWS CLI or CloudFormation console to verify.

1. Run the following command to enable termination protection on a new or existing bootstrap stack:

```
$ cdk bootstrap --termination-protection
```

2. Use the AWS CLI or CloudFormation console to verify. The following is an example, using the AWS CLI. If you modified your bootstrap stack name, replace `CDKToolkit` with your stack name:

```
$ aws cloudformation describe-stacks --stack-name <CDKToolkit> --query
 "Stacks[0].EnableTerminationProtection"
true
```

Initialize a project

Create a new, empty GitHub project and clone it to your workstation in the `my-pipeline` directory. (Our code examples in this topic use GitHub. You can also use AWS CodeStar or AWS CodeCommit.)

```
git clone <GITHUB-CLONE-URL> my-pipeline
cd my-pipeline
```

> **Note**
>
> You can use a name other than `my-pipeline` for your app's main directory. However, if you do so, you will have to tweak the file and class names later in this topic. This is because the AWS CDK Toolkit bases some file and class names on the name of the main directory.

After cloning, initialize the project as usual.

Example

TypeScript

```
$ cdk init app --language typescript
```

JavaScript

```
$ cdk init app --language javascript
```

Python

```
$ cdk init app --language python
```

After the app has been created, also enter the following two commands. These activate the app's Python virtual environment and install the AWS CDK core dependencies.

```
$ source .venv/bin/activate # On Windows, run `.\venv\Scripts\activate` instead
$ python -m pip install -r requirements.txt
```

Java

```
$ cdk init app --language java
```

If you are using an IDE, you can now open or import the project. In Eclipse, for example, choose **File** > **Import** > **Maven** > **Existing Maven Projects**. Make sure that the project settings are set to use Java 8 (1.8).

C#

```
$ cdk init app --language csharp
```

If you are using Visual Studio, open the solution file in the `src` directory.

Go

```
$ cdk init app --language go
```

After the app has been created, also enter the following command to install the AWS Construct Library modules that the app requires.

```
$ go get
```

Important

Be sure to commit your `cdk.json` and `cdk.context.json` files to source control. The context information (such as feature flags and cached values retrieved from your AWS account) are part of your project's state. The values may be different in another environment, which can cause unexpected changes in your results. For more information, see Context values and the AWS CDK.

Define a pipeline

Your CDK Pipelines application will include at least two stacks: one that represents the pipeline itself, and one or more stacks that represent the application deployed through it. Stacks can also be grouped into *stages*, which you can use to deploy copies of infrastructure stacks to different environments. For now, we'll consider the pipeline, and later delve into the application it will deploy.

The construct `CodePipeline` is the construct that represents a CDK Pipeline that uses AWS CodePipeline as its deployment engine. When you instantiate `CodePipeline` in a stack, you define the source location for the pipeline (such as a GitHub repository). You also define the commands to build the app.

For example, the following defines a pipeline whose source is stored in a GitHub repository. It also includes a build step for a TypeScript CDK application. Fill in the information about your GitHub repo where indicated.

Note

By default, the pipeline authenticates to GitHub using a personal access token stored in Secrets Manager under the name `github-token`.

You'll also need to update the instantiation of the pipeline stack to specify the AWS account and Region.

Example

TypeScript

In `lib/my-pipeline-stack.ts` (may vary if your project folder isn't named `my-pipeline`):

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { CodePipeline, CodePipelineSource, ShellStep } from 'aws-cdk-lib/pipelines';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });
  }
}
```

In `bin/my-pipeline.ts` (may vary if your project folder isn't named `my-pipeline`):

```
#!/usr/bin/env node
import * as cdk from 'aws-cdk-lib';
import { MyPipelineStack } from '../lib/my-pipeline-stack';

const app = new cdk.App();
new MyPipelineStack(app, 'MyPipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```

JavaScript

In `lib/my-pipeline-stack.js` (may vary if your project folder isn't named `my-pipeline`):

```
const cdk = require('aws-cdk-lib');
const { CodePipeline, CodePipelineSource, ShellStep } = require('aws-cdk-lib/
pipelines');

 class MyPipelineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });
  }
}

module.exports = { MyPipelineStack }
```

In `bin/my-pipeline.js` (may vary if your project folder isn't named `my-pipeline`):

```
#!/usr/bin/env node

const cdk = require('aws-cdk-lib');
const { MyPipelineStack } = require('../lib/my-pipeline-stack');

const app = new cdk.App();
new MyPipelineStack(app, 'MyPipelineStack', {
  env: {
    account: '111111111111',
    region: 'eu-west-1',
  }
});

app.synth();
```

Python

In `my-pipeline/my-pipeline-stack.py` (may vary if your project folder isn't named `my-pipeline`):

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep

class MyPipelineStack(cdk.Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        pipeline =  CodePipeline(self, "Pipeline",
                        pipeline_name="MyPipeline",
                        synth=ShellStep("Synth",
                            input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                            commands=["npm install -g aws-cdk",
                                "python -m pip install -r requirements.txt",
                                "cdk synth"]
                        )
                    )
```

In `app.py`:

```
#!/usr/bin/env python3
import aws_cdk as cdk
from my_pipeline.my_pipeline_stack import MyPipelineStack

app = cdk.App()
MyPipelineStack(app, "MyPipelineStack",
    env=cdk.Environment(account="111111111111", region="eu-west-1")
)

app.synth()
```

Java

In `src/main/java/com/myorg/MyPipelineStack.java` (may vary if your project folder isn't named `my-pipeline`):

```
package com.myorg;

import java.util.Arrays;
import software.constructs.Construct;
import software.amazon.awscdk.Stack;
```

```
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.pipelines.CodePipeline;
import software.amazon.awscdk.pipelines.CodePipelineSource;
import software.amazon.awscdk.pipelines.ShellStep;

public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
             .pipelineName("MyPipeline")
             .synth(ShellStep.Builder.create("Synth")
                .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
             .build();
    }
}
```

In `src/main/java/com/myorg/MyPipelineApp.java` (may vary if your project folder isn't named `my-pipeline`):

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

public class MyPipelineApp {
    public static void main(final String[] args) {
        App app = new App();

        new MyPipelineStack(app, "PipelineStack", StackProps.builder()
            .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build());
```

```
        app.synth();
    }
}
```

C#

In `src/MyPipeline/MyPipelineStack.cs` (may vary if your project folder isn't named `my-pipeline`):

```
using Amazon.CDK;
using Amazon.CDK.Pipelines;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
            {
                PipelineName = "MyPipeline",
                Synth = new ShellStep("Synth", new ShellStepProps
                {
                    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
                    Commands = new string[] { "npm install -g aws-cdk", "cdk
 synth" }
                })
            });
        }
    }
}
```

In `src/MyPipeline/Program.cs` (may vary if your project folder isn't named `my-pipeline`):

```
using Amazon.CDK;

namespace MyPipeline
{
    sealed class Program
    {
        public static void Main(string[] args)
```

```
        {
            var app = new App();
            new MyPipelineStack(app, "MyPipelineStack", new StackProps
            {
                Env = new Amazon.CDK.Environment {
                    Account = "111111111111", Region = "eu-west-1" }
            });

            app.Synth();
        }
    }
}
```

Go

```
package main

import (
 "github.com/aws/aws-cdk-go/awscdk/v2"
 codebuild "github.com/aws/aws-cdk-go/awscdk/v2/awscodebuild"
 ssm "github.com/aws/aws-cdk-go/awscdk/v2/awsssm"
 pipeline "github.com/aws/aws-cdk-go/awscdk/v2/pipelines"
 "github.com/aws/constructs-go/constructs/v10"
 "github.com/aws/jsii-runtime-go"
 "os"
)

// my CDK Stack with resources
func NewCdkStack(scope constructs.Construct, id *string, props *awscdk.StackProps)
 awscdk.Stack {
 stack := awscdk.NewStack(scope, id, props)

 // create an example ssm parameter
 _ = ssm.NewStringParameter(stack, jsii.String("ssm-test-param"),
 &ssm.StringParameterProps{
  ParameterName: jsii.String("/testparam"),
  Description:   jsii.String("ssm parameter for demo"),
  StringValue:   jsii.String("my test param"),
 })

 return stack
}
```

```
// my CDK Application
func NewCdkApplication(scope constructs.Construct, id *string, props
 *awscdk.StageProps) awscdk.Stage {
 stage := awscdk.NewStage(scope, id, props)

 _ = NewCdkStack(stage, jsii.String("cdk-stack"), &awscdk.StackProps{Env:
 props.Env})

 return stage
}

// my CDK Pipeline
func NewCdkPipeline(scope constructs.Construct, id *string, props
 *awscdk.StackProps) awscdk.Stack {
 stack := awscdk.NewStack(scope, id, props)

 // GitHub repo with owner and repository name
 githubRepo := pipeline.CodePipelineSource_GitHub(jsii.String("owner/repo"),
 jsii.String("main"), &pipeline.GitHubSourceOptions{
  Authentication: awscdk.SecretValue_SecretsManager(jsii.String("my-github-token"),
 nil),
 })

 // self mutating pipeline
 myPipeline := pipeline.NewCodePipeline(stack, jsii.String("cdkPipeline"),
 &pipeline.CodePipelineProps{
  PipelineName: jsii.String("CdkPipeline"),
  // self mutation true - pipeline changes itself before application deployment
  SelfMutation: jsii.Bool(true),
  CodeBuildDefaults: &pipeline.CodeBuildOptions{
   BuildEnvironment: &codebuild.BuildEnvironment{
    // image version 6.0 recommended for newer go version
    BuildImage: codebuild.LinuxBuildImage_FromCodeBuildImageId(jsii.String("aws/
codebuild/standard:6.0")),
   },
  },
  Synth: pipeline.NewCodeBuildStep(jsii.String("Synth"),
 &pipeline.CodeBuildStepProps{
   Input: githubRepo,
   Commands: &[]*string{
    jsii.String("npm install -g aws-cdk"),
    jsii.String("cdk synth"),
   },
  }),
```

```
 })

 // deployment of actual CDK application
 myPipeline.AddStage(NewCdkApplication(stack, jsii.String("MyApplication"),
 &awscdk.StageProps{
  Env: targetAccountEnv(),
 }), &pipeline.AddStageOpts{
  Post: &[]pipeline.Step{
   pipeline.NewCodeBuildStep(jsii.String("Manual Steps"),
 &pipeline.CodeBuildStepProps{
    Commands: &[]*string{
     jsii.String("echo \"My CDK App deployed, manual steps go here ... \""),
    },
   }),
  },
 })

 return stack
}

// main app
func main() {
 defer jsii.Close()

 app := awscdk.NewApp(nil)

 // call CDK Pipeline
 NewCdkPipeline(app, jsii.String("CdkPipelineStack"), &awscdk.StackProps{
  Env: pipelineEnv(),
 })

 app.Synth(nil)
}

// env determines the AWS environment (account+region) in which our stack is to
// be deployed. For more information see: https://docs.aws.amazon.com/cdk/latest/
guide/environments.html
func pipelineEnv() *awscdk.Environment {
 return &awscdk.Environment{
  Account: jsii.String(os.Getenv("CDK_DEFAULT_ACCOUNT")),
  Region:  jsii.String(os.Getenv("CDK_DEFAULT_REGION")),
 }
}
```

```
func targetAccountEnv() *awscdk.Environment {
 return &awscdk.Environment{
  Account: jsii.String(os.Getenv("CDK_DEFAULT_ACCOUNT")),
  Region:  jsii.String(os.Getenv("CDK_DEFAULT_REGION")),
 }
}
```

You must deploy a pipeline manually once. After that, the pipeline keeps itself up to date from the source code repository. So be sure that the code in the repo is the code you want deployed. Check in your changes and push to GitHub, then deploy:

```
git add --all
git commit -m "initial commit"
git push
cdk deploy
```

> **Tip**
>
> Now that you've done the initial deployment, your local AWS account no longer needs administrative access. This is because all changes to your app will be deployed via the pipeline. All you need to be able to do is push to GitHub.

Application stages

To define a multi-stack AWS application that can be added to the pipeline all at once, define a subclass of `Stage`. (This is different from `CdkStage` in the CDK Pipelines module.)

The stage contains the stacks that make up your application. If there are dependencies between the stacks, the stacks are automatically added to the pipeline in the right order. Stacks that don't depend on each other are deployed in parallel. You can add a dependency relationship between stacks by calling `stack1.addDependency(stack2)`.

Stages accept a default `env` argument, which becomes the default environment for the stacks inside it. (Stacks can still have their own environment specified.).

An application is added to the pipeline by calling `addStage()` with instances of Stage. A stage can be instantiated and added to the pipeline multiple times to define different stages of your DTAP or multi-Region application pipeline.

We will create a stack containing a simple Lambda function and place that stack in a stage. Then we will add the stage to the pipeline so it can be deployed.

Example

TypeScript

Create the new file `lib/my-pipeline-lambda-stack.ts` to hold our application stack containing a Lambda function.

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { Function, InlineCode, Runtime } from 'aws-cdk-lib/aws-lambda';

export class MyLambdaStack extends cdk.Stack {
    constructor(scope: Construct, id: string, props?: cdk.StackProps) {
      super(scope, id, props);

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_18_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}
```

Create the new file `lib/my-pipeline-app-stage.ts` to hold our stage.

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from "constructs";
import { MyLambdaStack } from './my-pipeline-lambda-stack';

export class MyPipelineAppStage extends cdk.Stage {

    constructor(scope: Construct, id: string, props?: cdk.StageProps) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}
```

Edit `lib/my-pipeline-stack.ts` to add the stage to our pipeline.

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { CodePipeline, CodePipelineSource, ShellStep } from 'aws-cdk-lib/pipelines';
import { MyPipelineAppStage } from './my-pipeline-app-stage';

export class MyPipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });

    pipeline.addStage(new MyPipelineAppStage(this, "test", {
      env: { account: "111111111111", region: "eu-west-1" }
    }));
  }
}
```

JavaScript

Create the new file `lib/my-pipeline-lambda-stack.js` to hold our application stack containing a Lambda function.

```
const cdk = require('aws-cdk-lib');
const { Function, InlineCode, Runtime } = require('aws-cdk-lib/aws-lambda');

class MyLambdaStack extends cdk.Stack {
    constructor(scope, id, props) {
      super(scope, id, props);

      new Function(this, 'LambdaFunction', {
        runtime: Runtime.NODEJS_18_X,
        handler: 'index.handler',
        code: new InlineCode('exports.handler = _ => "Hello, CDK";')
      });
    }
}
```

```
module.exports = { MyLambdaStack }
```

Create the new file `lib/my-pipeline-app-stage.js` to hold our stage.

```
const cdk = require('aws-cdk-lib');
const { MyLambdaStack } = require('./my-pipeline-lambda-stack');

class MyPipelineAppStage extends cdk.Stage {

    constructor(scope, id, props) {
      super(scope, id, props);

      const lambdaStack = new MyLambdaStack(this, 'LambdaStack');
    }
}

module.exports = { MyPipelineAppStage };
```

Edit `lib/my-pipeline-stack.ts` to add the stage to our pipeline.

```
const cdk = require('aws-cdk-lib');
const { CodePipeline, CodePipelineSource, ShellStep } = require('aws-cdk-lib/
pipelines');
const { MyPipelineAppStage } = require('./my-pipeline-app-stage');

 class MyPipelineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const pipeline = new CodePipeline(this, 'Pipeline', {
      pipelineName: 'MyPipeline',
      synth: new ShellStep('Synth', {
        input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
        commands: ['npm ci', 'npm run build', 'npx cdk synth']
      })
    });

    pipeline.addStage(new MyPipelineAppStage(this, "test", {
      env: { account: "111111111111", region: "eu-west-1" }
    }));

  }
}
```

```
module.exports = { MyPipelineStack }
```

Python

Create the new file `my_pipeline/my_pipeline_lambda_stack.py` to hold our application stack containing a Lambda function.

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.aws_lambda import Function, InlineCode, Runtime

class MyLambdaStack(cdk.Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        Function(self, "LambdaFunction",
            runtime=Runtime.NODEJS_18_X,
            handler="index.handler",
            code=InlineCode("exports.handler = _ => 'Hello, CDK';")
        )
```

Create the new file `my_pipeline/my_pipeline_app_stage.py` to hold our stage.

```
import aws_cdk as cdk
from constructs import Construct
from my_pipeline.my_pipeline_lambda_stack import MyLambdaStack

class MyPipelineAppStage(cdk.Stage):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        lambdaStack = MyLambdaStack(self, "LambdaStack")
```

Edit `my_pipeline/my-pipeline-stack.py` to add the stage to our pipeline.

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk.pipelines import CodePipeline, CodePipelineSource, ShellStep
from my_pipeline.my_pipeline_app_stage import MyPipelineAppStage

class MyPipelineStack(cdk.Stack):
```

```
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        pipeline =  CodePipeline(self, "Pipeline",
                        pipeline_name="MyPipeline",
                        synth=ShellStep("Synth",
                            input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                            commands=["npm install -g aws-cdk",
                                "python -m pip install -r requirements.txt",
                                "cdk synth"]))

        pipeline.add_stage(MyPipelineAppStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))
```

Java

Create the new file `src/main/java/com.myorg/MyPipelineLambdaStack.java` to hold our application stack containing a Lambda function.

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.InlineCode;

public class MyPipelineLambdaStack extends Stack {
    public MyPipelineLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineLambdaStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        Function.Builder.create(this, "LambdaFunction")
          .runtime(Runtime.NODEJS_18_X)
          .handler("index.handler")
          .code(new InlineCode("exports.handler = _ => 'Hello, CDK';"))
```

```
            .build();

    }

}
```

Create the new file `src/main/java/com.myorg/MyPipelineAppStage.java` to hold our stage.

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.Stage;
import software.amazon.awscdk.StageProps;

public class MyPipelineAppStage extends Stage {
    public MyPipelineAppStage(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineAppStage(final Construct scope, final String id, final
 StageProps props) {
        super(scope, id, props);

        Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
    }

}
```

Edit `src/main/java/com.myorg/MyPipelineStack.java` to add the stage to our pipeline.

```
package com.myorg;

import java.util.Arrays;
import software.constructs.Construct;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.StageProps;
import software.amazon.awscdk.pipelines.CodePipeline;
import software.amazon.awscdk.pipelines.CodePipelineSource;
import software.amazon.awscdk.pipelines.ShellStep;
```

```
public class MyPipelineStack extends Stack {
    public MyPipelineStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public MyPipelineStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
            .pipelineName("MyPipeline")
            .synth(ShellStep.Builder.create("Synth")
                .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
            .build();

        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
            .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
                .build())
            .build()));
    }
}
```

C#

Create the new file `src/MyPipeline/MyPipelineLambdaStack.cs` to hold our application stack containing a Lambda function.

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.AWS.Lambda;

namespace MyPipeline
{
    class MyPipelineLambdaStack : Stack
    {
        public MyPipelineLambdaStack(Construct scope, string id, StackProps
 props=null) : base(scope, id, props)
        {
```

```
            new Function(this, "LambdaFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_18_X,
                Handler = "index.handler",
                Code = new InlineCode("exports.handler = _ => 'Hello, CDK';")
            });
        }
    }
}
```

Create the new file `src/MyPipeline/MyPipelineAppStage.cs` to hold our stage.

```
using Amazon.CDK;
using Constructs;

namespace MyPipeline
{
    class MyPipelineAppStage : Stage
    {
        public MyPipelineAppStage(Construct scope, string id, StageProps
 props=null) : base(scope, id, props)
        {
            Stack lambdaStack = new MyPipelineLambdaStack(this, "LambdaStack");
        }
    }
}
```

Edit `src/MyPipeline/MyPipelineStack.cs` to add the stage to our pipeline.

```
using Amazon.CDK;
using Constructs;
using Amazon.CDK.Pipelines;

namespace MyPipeline
{
    public class MyPipelineStack : Stack
    {
        internal MyPipelineStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
            {
                PipelineName = "MyPipeline",
```

```
                Synth = new ShellStep("Synth", new ShellStepProps
                {
                    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
                    Commands = new string[] { "npm install -g aws-cdk", "cdk
 synth" }
                })
            });

            pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
            {
                Env = new Environment
                {
                    Account = "111111111111", Region = "eu-west-1"
                }
            }));
        }
    }
}
```

Every application stage added by `addStage()` results in the addition of a corresponding pipeline stage, represented by a [`StageDeployment`](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.pipelines.StageDeployment.html) instance returned by the `addStage()` call. You can add pre-deployment or post-deployment actions to the stage by calling its `addPre()` or `addPost()` method.

Example

TypeScript

```
// import { ManualApprovalStep } from 'aws-cdk-lib/pipelines';

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, 'testing', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

    testingStage.addPost(new ManualApprovalStep('approval'));
```

JavaScript

```
// const { ManualApprovalStep } = require('aws-cdk-lib/pipelines');

const testingStage = pipeline.addStage(new MyPipelineAppStage(this, 'testing', {
```

```
  env: { account: '111111111111', region: 'eu-west-1' }
}));

testingStage.addPost(new ManualApprovalStep('approval'));
```

Python

```
# from aws_cdk.pipelines import ManualApprovalStep

testing_stage = pipeline.add_stage(MyPipelineAppStage(self, "testing",
    env=cdk.Environment(account="111111111111", region="eu-west-1")))

testing_stage.add_post(ManualApprovalStep('approval'))
```

Java

```
// import software.amazon.awscdk.pipelines.StageDeployment;
// import software.amazon.awscdk.pipelines.ManualApprovalStep;

StageDeployment testingStage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

testingStage.addPost(new ManualApprovalStep("approval"));
```

C#

```
var testingStage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new
 StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

testingStage.AddPost(new ManualApprovalStep("approval"));
```

You can add stages to a `Wave` to deploy them in parallel, for example when deploying a stage to multiple accounts or Regions.

Example

TypeScript

```
const wave = pipeline.addWave('wave');
wave.addStage(new MyApplicationStage(this, 'MyAppEU', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));
wave.addStage(new MyApplicationStage(this, 'MyAppUS', {
  env: { account: '111111111111', region: 'us-west-1' }
}));
```

JavaScript

```
const wave = pipeline.addWave('wave');
wave.addStage(new MyApplicationStage(this, 'MyAppEU', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));
wave.addStage(new MyApplicationStage(this, 'MyAppUS', {
  env: { account: '111111111111', region: 'us-west-1' }
}));
```

Python

```
wave = pipeline.add_wave("wave")
wave.add_stage(MyApplicationStage(self, "MyAppEU",
    env=cdk.Environment(account="111111111111", region="eu-west-1")))
wave.add_stage(MyApplicationStage(self, "MyAppUS",
    env=cdk.Environment(account="111111111111", region="us-west-1")))
```

Java

```
// import software.amazon.awscdk.pipelines.Wave;
final Wave wave = pipeline.addWave("wave");
wave.addStage(new MyPipelineAppStage(this, "MyAppEU", StageProps.builder()
        .env(Environment.builder()
                .account("111111111111")
                .region("eu-west-1")
```

```
                .build())
        .build()));
wave.addStage(new MyPipelineAppStage(this, "MyAppUS", StageProps.builder()
        .env(Environment.builder()
                .account("111111111111")
                .region("us-west-1")
                .build())
        .build()));
```

C#

```
var wave = pipeline.AddWave("wave");
wave.AddStage(new MyPipelineAppStage(this, "MyAppEU", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));
wave.AddStage(new MyPipelineAppStage(this, "MyAppUS", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "us-west-1"
    }
}));
```

Testing deployments

You can add steps to a CDK Pipeline to validate the deployments that you're performing. For example, you can use the CDK Pipeline library's `ShellStep` to perform tasks such as the following:

- Trying to access a newly deployed Amazon API Gateway backed by a Lambda function
- Checking a setting of a deployed resource by issuing an AWS CLI command

In its simplest form, adding validation actions looks like this:

Example

TypeScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['../tests/validate.sh'],
}));
```

JavaScript

```
// stage was returned by pipeline.addStage

stage.addPost(new ShellStep("validate", {
  commands: ['../tests/validate.sh'],
}));
```

Python

```
# stage was returned by pipeline.add_stage

stage.add_post(ShellStep("validate",
  commands=[''../tests/validate.sh'']
))
```

Java

```
// stage was returned by pipeline.addStage

stage.addPost(ShellStep.Builder.create("validate")
        .commands(Arrays.asList("'../tests/validate.sh'"))
        .build());
```

C#

```
// stage was returned by pipeline.addStage

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Commands = new string[] { "'../tests/validate.sh'" }
```

```
}));
```

Many AWS CloudFormation deployments result in the generation of resources with unpredictable names. Because of this, CDK Pipelines provide a way to read AWS CloudFormation outputs after a deployment. This makes it possible to pass (for example) the generated URL of a load balancer to a test action.

To use outputs, expose the `CfnOutput` object you're interested in. Then, pass it in a step's `envFromCfnOutputs` property to make it available as an environment variable within that step.

Example

TypeScript

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
this.loadBalancerAddress = new cdk.CfnOutput(lbStack, 'LbAddress', {
  value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
});

// pass the load balancer address to a shell step
stage.addPost(new ShellStep("lbaddr", {
  envFromCfnOutputs: {lb_addr: lbStack.loadBalancerAddress},
  commands: ['echo $lb_addr']
}));
```

JavaScript

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
this.loadBalancerAddress = new cdk.CfnOutput(lbStack, 'LbAddress', {
  value: `https://${lbStack.loadBalancer.loadBalancerDnsName}/`
});

// pass the load balancer address to a shell step
stage.addPost(new ShellStep("lbaddr", {
  envFromCfnOutputs: {lb_addr: lbStack.loadBalancerAddress},
  commands: ['echo $lb_addr']
}));
```

Python

```
# given a stack lb_stack that exposes a load balancer construct as load_balancer
```

```
self.load_balancer_address = cdk.CfnOutput(lb_stack, "LbAddress",
    value=f"https://{lb_stack.load_balancer.load_balancer_dns_name}/")

# pass the load balancer address to a shell step
stage.add_post(ShellStep("lbaddr",
    env_from_cfn_outputs={"lb_addr": lb_stack.load_balancer_address}
    commands=["echo $lb_addr"]))
```

Java

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
loadBalancerAddress = CfnOutput.Builder.create(lbStack, "LbAddress")
                            .value(String.format("https://%s/",
                                    lbStack.loadBalancer.loadBalancerDnsName))
                            .build();

stage.addPost(ShellStep.Builder.create("lbaddr")
    .envFromCfnOutputs(     // Map.of requires Java 9 or later
        java.util.Map.of("lbAddr", loadBalancerAddress))
    .commands(Arrays.asList("echo $lbAddr"))
    .build());
```

C#

```
// given a stack lbStack that exposes a load balancer construct as loadBalancer
loadBalancerAddress = new CfnOutput(lbStack, "LbAddress", new CfnOutputProps
{
    Value = string.Format("https://{0}/", lbStack.loadBalancer.LoadBalancerDnsName)
});

stage.AddPost(new ShellStep("lbaddr", new ShellStepProps
{
    EnvFromCfnOutputs = new Dictionary<string, CfnOutput>
    {
        {  "lbAddr", loadBalancerAddress }
    },
    Commands = new string[] { "echo $lbAddr" }
}));
```

You can write simple validation tests right in the `ShellStep`, but this approach becomes unwieldy when the test is more than a few lines. For more complex tests, you can bring additional files (such as complete shell scripts, or programs in other languages) into the `ShellStep` via the `inputs` property. The inputs can be any step that has an output, including a source (such as a GitHub repo) or another `ShellStep`.

Bringing in files from the source repository is appropriate if the files are directly usable in the test (for example, if they are themselves executable). In this example, we declare our GitHub repo as `source` (rather than instantiating it inline as part of the `CodePipeline`). Then, we pass this fileset to both the pipeline and the validation test.

Example

TypeScript

```
const source = CodePipelineSource.gitHub('OWNER/REPO', 'main');

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: new ShellStep('Synth', {
    input: source,
    commands: ['npm ci', 'npm run build', 'npx cdk synth']
  })
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

stage.addPost(new ShellStep('validate', {
  input: source,
  commands: ['sh ../tests/validate.sh']
}));
```

JavaScript

```
const source = CodePipelineSource.gitHub('OWNER/REPO', 'main');

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: new ShellStep('Synth', {
    input: source,
    commands: ['npm ci', 'npm run build', 'npx cdk synth']
  })
```

```
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

stage.addPost(new ShellStep('validate', {
  input: source,
  commands: ['sh ../tests/validate.sh']
}));
```

Python

```
source   = CodePipelineSource.git_hub("OWNER/REPO", "main")

pipeline =  CodePipeline(self, "Pipeline",
                pipeline_name="MyPipeline",
                synth=ShellStep("Synth",
                    input=source,
                    commands=["npm install -g aws-cdk",
                        "python -m pip install -r requirements.txt",
                        "cdk synth"]))

stage = pipeline.add_stage(MyApplicationStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))

stage.add_post(ShellStep("validate", input=source,
    commands=["sh ../tests/validate.sh"],
))
```

Java

```
final CodePipelineSource source = CodePipelineSource.gitHub("OWNER/REPO", "main");

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
        .pipelineName("MyPipeline")
        .synth(ShellStep.Builder.create("Synth")
                .input(source)
                .commands(Arrays.asList("npm install -g aws-cdk", "cdk synth"))
                .build())
        .build();

final StageDeployment stage =
```

```
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
                .build()));

stage.addPost(ShellStep.Builder.create("validate")
        .input(source)
        .commands(Arrays.asList("sh ../tests/validate.sh"))
        .build());
```

C#

```
var source = CodePipelineSource.GitHub("OWNER/REPO", "main");

var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
{
    PipelineName = "MyPipeline",
    Synth = new ShellStep("Synth", new ShellStepProps
    {
        Input = source,
        Commands = new string[] { "npm install -g aws-cdk", "cdk synth" }
    })
});

var stage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Input = source,
    Commands = new string[] { "sh ../tests/validate.sh" }
}));
```

Getting the additional files from the synth step is appropriate if your tests need to be compiled, which is done as part of synthesis.

Example

TypeScript

```
const synthStep = new ShellStep('Synth', {
  input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
  commands: ['npm ci', 'npm run build', 'npx cdk synth'],
});

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: synthStep
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, 'test', {
  env: { account: '111111111111', region: 'eu-west-1' }
}));

// run a script that was transpiled from TypeScript during synthesis
stage.addPost(new ShellStep('validate', {
  input: synthStep,
  commands: ['node tests/validate.js']
}));
```

JavaScript

```
const synthStep = new ShellStep('Synth', {
  input: CodePipelineSource.gitHub('OWNER/REPO', 'main'),
  commands: ['npm ci', 'npm run build', 'npx cdk synth'],
});

const pipeline = new CodePipeline(this, 'Pipeline', {
  pipelineName: 'MyPipeline',
  synth: synthStep
});

const stage = pipeline.addStage(new MyPipelineAppStage(this, "test", {
  env: { account: "111111111111", region: "eu-west-1" }
}));

// run a script that was transpiled from TypeScript during synthesis
stage.addPost(new ShellStep('validate', {
  input: synthStep,
```

```
  commands: ['node tests/validate.js']
}));
```

Python

```
synth_step = ShellStep("Synth",
                input=CodePipelineSource.git_hub("OWNER/REPO", "main"),
                commands=["npm install -g aws-cdk",
                  "python -m pip install -r requirements.txt",
                  "cdk synth"])

pipeline   = CodePipeline(self, "Pipeline",
                pipeline_name="MyPipeline",
                synth=synth_step)

stage = pipeline.add_stage(MyApplicationStage(self, "test",
            env=cdk.Environment(account="111111111111", region="eu-west-1")))

# run a script that was compiled during synthesis
stage.add_post(ShellStep("validate",
    input=synth_step,
    commands=["node test/validate.js"],
))
```

Java

```
final ShellStep synth = ShellStep.Builder.create("Synth")
                            .input(CodePipelineSource.gitHub("OWNER/REPO", "main"))
                            .commands(Arrays.asList("npm install -g aws-cdk", "cdk
 synth"))
                            .build();

final CodePipeline pipeline = CodePipeline.Builder.create(this, "pipeline")
        .pipelineName("MyPipeline")
        .synth(synth)
        .build();

final StageDeployment stage =
        pipeline.addStage(new MyPipelineAppStage(this, "test", StageProps.builder()
                .env(Environment.builder()
                        .account("111111111111")
                        .region("eu-west-1")
                        .build())
```

```
                .build()));

stage.addPost(ShellStep.Builder.create("validate")
        .input(synth)
        .commands(Arrays.asList("node ./tests/validate.js"))
        .build());
```

C#

```
var synth = new ShellStep("Synth", new ShellStepProps
{
    Input = CodePipelineSource.GitHub("OWNER/REPO", "main"),
    Commands = new string[] { "npm install -g aws-cdk", "cdk synth" }
});

var pipeline = new CodePipeline(this, "pipeline", new CodePipelineProps
{
    PipelineName = "MyPipeline",
    Synth = synth
});

var stage = pipeline.AddStage(new MyPipelineAppStage(this, "test", new StageProps
{
    Env = new Environment
    {
        Account = "111111111111", Region = "eu-west-1"
    }
}));

stage.AddPost(new ShellStep("validate", new ShellStepProps
{
    Input = synth,
    Commands = new string[] { "node ./tests/validate.js" }
}));
```

Security notes

Any form of continuous delivery has inherent security risks. Under the AWS Shared Responsibility Model, you are responsible for the security of your information in the AWS Cloud. The CDK Pipelines library gives you a head start by incorporating secure defaults and modeling best practices.

However, by its very nature, a library that needs a high level of access to fulfill its intended purpose cannot assure complete security. There are many attack vectors outside of AWS and your organization.

In particular, keep in mind the following:

- Be mindful of the software you depend on. Vet all third-party software you run in your pipeline, because it can change the infrastructure that gets deployed.
- Use dependency locking to prevent accidental upgrades. CDK Pipelines respects `package-lock.json` and `yarn.lock` to make sure that your dependencies are the ones you expect.
- CDK Pipelines runs on resources created in your own account, and the configuration of those resources is controlled by developers submitting code through the pipeline. Therefore, CDK Pipelines by itself cannot protect against malicious developers trying to bypass compliance checks. If your threat model includes developers writing CDK code, you should have external compliance mechanisms in place like AWS CloudFormation Hooks (preventive) or AWS Config (reactive) that the AWS CloudFormation Execution Role does not have permissions to disable.
- Credentials for production environments should be short-lived. After bootstrapping and initial provisioning, there is no need for developers to have account credentials at all. Changes can be deployed through the pipeline. Reduce the possibility of credentials leaking by not needing them in the first place.

Troubleshooting

The following issues are commonly encountered while getting started with CDK Pipelines.

Pipeline: Internal Failure

```
CREATE_FAILED  | {aws}::CodePipeline::Pipeline | Pipeline/Pipeline
Internal Failure
```

Check your GitHub access token. It might be missing, or might not have the permissions to access the repository.

Key: Policy contains a statement with one or more invalid principals

```
CREATE_FAILED | {aws}::KMS::Key | Pipeline/Pipeline/ArtifactsBucketEncryptionKey
Policy contains a statement with one or more invalid principals.
```

One of the target environments has not been bootstrapped with the new bootstrap stack. Make sure all your target environments are bootstrapped.

Stack is in ROLLBACK_COMPLETE state and cannot be updated

```
Stack <STACK_NAME> is in ROLLBACK_COMPLETE state and cannot be updated. (Service:
AmazonCloudFormation; Status Code: 400; Error Code: ValidationError; Request
ID: ...)
```

The stack failed its previous deployment and is in a non-retryable state. Delete the stack from the AWS CloudFormation console and retry the deployment.

Build and deploy container image assets in CDK apps

When you build container image assets with the AWS Cloud Development Kit (AWS CDK), Docker is utilized by default to perform these actions. If you want to use a different container management tool, you can replace Docker through the `CDK_DOCKER` environment variable.

Example: Build and publish a container image asset with the AWS CDK

The following is a simple example of an AWS CDK app that builds and publishes a container asset to Amazon Elastic Container Registry (Amazon ECR) using Docker by default:

Project structure

```
my-cdk-app/
### lib/
#   ### my-stack.ts
#   ### docker/
#       ### Dockerfile
#       ### app/
#           ### index.js
### bin/
#   ### my-cdk-app.ts
### package.json
### tsconfig.json
### cdk.json
```

Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:16
```

```
# Copy application code
COPY app/ /var/task/

# (Optional) Install dependencies
# RUN npm install

# The Lambda Node.js base image looks for index.handler by default
```

Application code

In `lib/docker/app/index.js`:

```
console.log("Hello from inside the container!");
```

CDK stack

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ecr_assets from 'aws-cdk-lib/aws-ecr-assets';

export class MyStack extends cdk.Stack {
  constructor(scope: Construct, id: string) {
    super(scope, id);

    // Define a Docker image asset
    const dockerImageAsset = new ecr_assets.DockerImageAsset(this, 'MyDockerImage',
 {
      directory: 'lib/docker', // Path to the directory containing the Dockerfile
    });

    // Output the ECR URI
    new cdk.CfnOutput(this, 'ECRImageUri', {
      value: dockerImageAsset.imageUri,
    });
  }
}
```

CDK app

```
#!/usr/bin/env node
import * as cdk from 'aws-cdk-lib';
```

```
import { MyStack } from '../lib/my-stack';

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

When we run `cdk deploy`, the AWS Cloud Development Kit (AWS CDK) Command Line Interface (CLI) does the following:

1. **Build the Docker image** – Run `docker build` locally based on the `Dockerfile` in the specified directory (`lib/docker`).
2. **Tag the image** – Run `docker tag` to tag the built image with a unique hash, based on the image contents.
3. **Publish to Amazon ECR** – Run `docker push` to publish the container image to an Amazon ECR repository. This repository must already exist. It is created during the default bootstrapping process.
4. **Output the Image URI** – After a successful deployment, the Amazon ECR URI of the published container image is output in your command prompt. This is the URI of our Docker image in Amazon ECR.

How to replace Docker with another container management tool

Use the `CDK_DOCKER` environment variable to specify the path to the binary of your replacement container management tool. The following is an example of replacing Docker with Finch:

```
$ which finch
/usr/local/bin/finch # Locate the path to the binary

$ export CDK_DOCKER='/usr/local/bin/finch' # Set the environment variable

$ cdk deploy # Deploy using the replacement
```

Aliasing or linking is not supported. To replace Docker, you must use the `CDK_DOCKER` environment variable.

Supported Docker drop-in replacement engines

Finch is supported, although there may be some Docker features that are unavailable or may work differently as the tool evolves. For more information on Finch, see [Ready for Flight: Announcing Finch 1.0 GA!](#) in the *AWS Open Source Blog*.

Other container management tools may work. The CDK doesn't check which Docker replacement you are using to determine if it's supported. If the tool has equivalent Docker commands and behaves similarly, it should work.

Troubleshoot AWS CDK deployments

Troubleshoot common issues when deploying AWS Cloud Development Kit (AWS CDK) applications.

Incorrect service principals are being created at deployment

When deploying CDK applications that contain AWS Identity and Access Management (IAM) roles with service principals, you find that incorrect domains for the service principals are being created.

The following is a basic example of creating an IAM role that can be assumed by Amazon CloudWatch Logs using its service principal:

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

export class MyCdkProjectStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create an IAM role for CloudWatch Logs to assume
    const cloudWatchLogsRole = new iam.Role(this, 'CloudWatchLogsRole', {
      assumedBy: new iam.ServicePrincipal('logs.amazonaws.com'), // This is for
 CloudWatch Logs
      managedPolicies: [
        iam.ManagedPolicy.fromAwsManagedPolicyName('service-role/
AWSCloudWatchLogsFullAccess')
      ]
    });
```

```
    // You can then use this role in other constructs or configurations where
 CloudWatch Logs needs to assume a role
  }
}
```

When you deploy this stack, a service principal named `logs.amazonaws.com` should be created. In most cases, AWS services use the following naming for service principals: `<service>.amazonaws.com`.

Common causes

If you are using a version of the AWS CDK older than v2.150.0, you may encounter this bug. In older AWS CDK versions, the naming of service principals were not standardized, which could lead to the creation of service principals with incorrect domains.

In AWS CDK v2.51.0, a fix was implemented by standardizing all automatically created service principals to use `<service>.amazonaws.com` when possible. This fix was available by allowing the `@aws-cdk/aws-iam:standardizedServicePrincipals` feature flag.

Starting in AWS CDK v2.150.0, this became default behavior.

Resolution

Upgrade to AWS CDK v2.150.0 or newer.

If you are unable to upgrade to AWS CDK v2.150.0 or newer, you must upgrade to at least v2.51.0 or newer. Then, allow the following feature flag in your `cdk.json` file: `@aws-cdk/aws-iam:standardizedServicePrincipals`.

Configure constructs with CDK Blueprints

Use AWS CDK Blueprints to standardize and distribute L2 construct configurations across your organization. With Blueprints, you can ensure that AWS resources are configured consistently according to your organizational standards and best practices. For example, you can automatically enable encryption for all Amazon S3 buckets, apply specific logging configurations to all AWS Lambda functions, or enforce standard security rules for all security groups.

Blueprints are powered by *property injection*, a mechanism introduced in AWS CDK v2.196.0 that allows you to modify construct properties at instantiation time. A Blueprint is a collection of property injectors, where each property injector specifies optimal configuration for a specific L2 construct. The Blueprint represents the overall best practices for your organization.

Blueprints are not a compliance enforcement mechanism. Developers can still override the defaults if needed. For strict compliance enforcement, consider using AWS CloudFormation Guard, Service Control Policies, or CDK Aspects in addition to Blueprints.

For detailed implementation information, see the Property Injection RFC.

Key components of Blueprints

Blueprints are collections of property injectors that apply default properties to constructs when they're instantiated. A property injector is a component that implements the `IPropertyInjector` interface, which intercepts construct creation and modifies or adds properties before the construct is created.

- **IPropertyInjector** - An `IPropertyInjector` defines a way to inject additional properties that are not specified in the props. It is specific to one L2 construct and operates on that construct's properties.
- **PropertyInjectors** - `PropertyInjectors` are a collection of injectors attached to the construct tree. Injectors can be attached to any construct, but in practice we expect most of them will be attached to `App`, `Stage` or `Stack`.

Common use cases for Blueprints

You can use CDK Blueprints to standardize many aspects of your AWS resources. Here are some common use cases:

Security standards

- Ensure all Amazon S3 buckets have server-side encryption enabled.
- Configure all security groups to block public access by default.
- Apply least-privilege AWS Identity and Access Management (IAM) permissions to AWS Lambda functions.
- Enforce SSL for all network communications.

Operational excellence

- Configure standardized logging for all AWS Lambda functions.
- Apply consistent tagging strategies across resources.
- Set up default monitoring and alerting thresholds.
- Implement standard retention policies for logs and backups.

Cost optimization

- Configure appropriate instance sizes based on environment.
- Apply auto-scaling policies with organizational defaults.
- Set lifecycle rules for Amazon S3 buckets to transition objects to cheaper storage classes.
- Configure default provisioned throughput for databases.

Compliance requirements

- Implement required encryption settings for regulated data.
- Apply necessary backup policies for data retention requirements.
- Configure default Amazon VPC settings that meet security requirements.
- Ensure resources have required tags for cost allocation.

Developer productivity

- Provide sensible defaults that reduce the need for boilerplate code.
- Create organization-specific Stack classes with built-in injectors.
- Share best practices across teams through reusable injectors.
- Simplify onboarding by encoding organizational knowledge in code.

Getting started with Blueprints

Here’s a simple example of how to create and use a property injector:

First, create a property injector for Amazon S3 buckets:

```
import { IPropertyInjector, InjectionContext } from 'aws-cdk-lib';
import { Bucket, BucketProps, BlockPublicAccess } from 'aws-cdk-lib/aws-s3';

export class SecureBucketDefaults implements IPropertyInjector {
  public readonly constructUniqueId: string;

  constructor() {
    this.constructUniqueId = Bucket.PROPERTY_INJECTION_ID;
  }

  public inject(originalProps: BucketProps, _context: InjectionContext): BucketProps {
    return {
      // Set security defaults
      blockPublicAccess: BlockPublicAccess.BLOCK_ALL,
      enforceSSL: true,

      // Include original props to allow overrides
      ...originalProps,
    };
  }
}
```

Then, use the injector in your CDK application:

```
import { App, Stack } from 'aws-cdk-lib';
import { Bucket } from 'aws-cdk-lib/aws-s3';
import { SecureBucketDefaults } from './secure-bucket-defaults';

// Attach injectors when creating the App
const app = new App({
  propertyInjectors: [new SecureBucketDefaults()]
});

const stack = new Stack(app, 'MyStack');

// This bucket automatically gets the default properties
const myBucket = new Bucket(stack, 'MyBucket');
```

Alternatively, you can use the `PropertyInjectors.of()` method:

```
import { App, Stack, PropertyInjectors } from 'aws-cdk-lib';
import { SecureBucketDefaults } from './secure-bucket-defaults';
```

```
const app = new App();
PropertyInjectors.of(app).add(new SecureBucketDefaults());

const stack = new Stack(app, 'MyStack');
const myBucket = new Bucket(stack, 'MyBucket');
```

Best practices

- Place default properties before ...`originalProps` to allow overrides.
- Place forced properties after ...`originalProps` to prevent overrides.
- Use a skip flag when creating resources to prevent infinite recursion. For an example, see What happens when you need to create a accessLogBucket for a Bucket? in the *Property Injection RFC*.
- Add logging for debugging.
- Use CDK context to enable/disable injectors for testing.

For more detailed information about property injection, including implementation details, troubleshooting tips, and reference information, see the Property Injection RFC.

Configure plugins for the CDK Toolkit

The AWS CDK Toolkit supports plugins that add new capabilities to your CDK workflows. Plugins are primarily designed for use with the CDK Command Line Interface (CDK CLI), though they can also be used with the CDK Toolkit Library programmatically. They provide a standardized way to extend functionality, such as alternative credential sources.

> **Note**
>
> While the plugin system works with both the CDK CLI and CDK Toolkit Library, it is primarily intended for CLI usage. When using the CDK Toolkit Library programmatically, there are often simpler, more direct ways to accomplish the same tasks without plugins, particularly for credential management.

Currently, the CDK Toolkit supports one plugin capability:

- **Custom AWS credential providers** - Create alternative methods to obtain AWS credentials beyond the built-in mechanisms.

How to create plugins

To create a CDK Toolkit plugin, you first create a Node.js module, authored in TypeScript or JavaScript, that can be loaded by the CDK Toolkit. This module exports an object with a specific structure that the CDK Toolkit can recognize. At minimum, the exported object must include a version identifier and an initialization function that receives an `IPluginHost` instance, which the plugin uses to register its capabilities.

Example

TypeScript

CommonJS

```
// Example plugin structure
import type { IPluginHost, Plugin } from '@aws-cdk/cli-plugin-contract';
```

```
const plugin: Plugin = {
    // Version of the plugin infrastructure (currently always '1')
    version: '1',

    // Initialization function called when the plugin is loaded
    init(host: IPluginHost): void {
        // Register your plugin functionality with the host
        // For example, register a custom credential provider
    }
};

export = plugin;
```

ESM

```
// Example plugin structure
import type { IPluginHost, Plugin } from '@aws-cdk/cli-plugin-contract';

const plugin: Plugin = {
    // Version of the plugin infrastructure (currently always '1')
    version: '1',

    // Initialization function called when the plugin is loaded
    init(host: IPluginHost): void {
        // Register your plugin functionality with the host
        // For example, register a custom credential provider
    }
};

export { plugin as 'module.exports' };
```

JavaScript

CommonJS

```
const plugin = {
    // Version of the plugin infrastructure (currently always '1')
    version: '1',

    // Initialization function called when the plugin is loaded
    init(host) {
        // Register your plugin functionality with the host
```

```
        // For example, register a custom credential provider
    }
};

module.exports = plugin;
```

ESM

```
const plugin = {
    // Version of the plugin infrastructure (currently always '1')
    version: '1',

    // Initialization function called when the plugin is loaded
    init(host) {
        // Register your plugin functionality with the host
        // For example, register a custom credential provider
    }
};

export { plugin as 'module.exports' };
```

Note

CDK Toolkit plugins are loaded as CommonJS modules. You can build your plugin as an ECMAScript module (ESM), but have to adhere to some constraints:

- The module cannot contain top-level `await`, or import any modules that contain top-level `await`.
- The module must ensure compatibility by exporting the plugin object under the `module.exports` export name: `export { plugin as 'module.exports' }`.

How to load plugins with the CDK CLI

You have two options to specify plugins:

Option 1: Use the `--plugin` command line option

```
# Load a single plugin
```

```
$ cdk list --plugin=my-custom-plugin

# Load multiple plugins
$ cdk deploy --plugin=custom-plugin-1 --plugin=custom-plugin-2
```

The value of the `--plugin` argument should be a JavaScript file that, when imported via the Node.js `require()` function, returns an object implementing the `Plugin` interface.

Option 2: Add entries to configuration files

You can add plugin specifications to the project-specific `cdk.json` file or the `~/.cdk.json` global configuration file:

```
{
  "plugin": [
    "custom-plugin-1",
    "custom-plugin-2"
  ]
}
```

With either approach, the CDK CLI will load the specified plugins before running commands.

Load plugins in code with the CDK Toolkit Library

When working with the CDK Toolkit Library programmatically, you can load plugins directly in your code using the `PluginHost` instance from the `@aws-cdk/toolkit-lib` package. The `PluginHost` provides a `load()` method for loading plugins by module name or path.

Example

TypeScript

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Create a Toolkit instance
const toolkit = new Toolkit();

// Load a plugin by module name or path
// The module must export an object matching the Plugin interface
await toolkit.pluginHost.load('my-custom-plugin');
```

```
// You can load multiple plugins if needed
await toolkit.pluginHost.load('./path/to/another-plugin');

// Now proceed with other CDK Toolkit Library operations
// The plugin's functionality will be available to the toolkit
```

JavaScript

```
const { Toolkit } = require('@aws-cdk/toolkit-lib');

// Create a Toolkit instance
const toolkit = new Toolkit();

// Load a plugin by module name or path
// The module must export an object matching the Plugin interface
await toolkit.pluginHost.load('my-custom-plugin');

// You can load multiple plugins if needed
await toolkit.pluginHost.load('./path/to/another-plugin');

// Now proceed with other CDK Toolkit Library operations
// The plugin's functionality will be available to the toolkit
```

The `load()` method takes a single parameter, `moduleSpec`, which is the name or path of the plugin module to load. This can be either:

- A Node.js module name installed in the `node_modules` directory.
- A relative or absolute file path to a JavaScript or TypeScript module.

Implementing credential provider plugins

The current primary use case for plugins is to create custom AWS credential providers. Like the AWS CLI, the CDK CLI needs AWS credentials for authentication and authorization. However, there are several scenarios where the standard credential resolution might fail:

- The initial set of credentials cannot be obtained.
- The account to which the initial credentials belong cannot be obtained.
- The account associated with the credentials is different from the account on which the CLI is trying to operate.

To address these scenarios, the CDK Toolkit supports credential provider plugins. These plugins implement the `CredentialProviderSource` interface from the `@aws-cdk/cli-plugin-contract` package and are registered to the Toolkit using the `registerCredentialProviderSource` method. This enables the CDK Toolkit to obtain AWS credentials from non-standard sources like specialized authentication systems or custom credential stores.

To implement a custom credential provider, create a class that implements the required interface:

Example

TypeScript

CommonJS

```
import type { CredentialProviderSource, ForReading, ForWriting, IPluginHost,
 Plugin, PluginProviderResult, SDKv3CompatibleCredentials } from '@aws-cdk/cli-
plugin-contract';

class CustomCredentialProviderSource implements CredentialProviderSource {
  // Friendly name for the provider, used in error messages
  public readonly name: string = 'custom-credential-provider';

  // Check if this provider is available on the current system
  public async isAvailable(): Promise<boolean> {
    // Return false if the plugin cannot be used
    // For example, if it depends on files not present on the host
    return true;
  }

  // Check if this provider can provide credentials for a specific account
  public async canProvideCredentials(accountId: string): Promise<boolean> {
    // Return false if the plugin cannot provide credentials for this account
    // For example, if the account is not managed by this credential system
    return true;
    // You can use patterns to filter specific accounts
    // return accountId.startsWith('123456');
  }

  // Get credentials for the specified account and access mode
  // Returns PluginProviderResult which can be one of:
  // - SDKv2CompatibleCredentials (AWS SDK v2 entered maintenance on Sept 8, 2024
 and will reach end-of-life on Sept 8, 2025)
```

```
  // - SDKv3CompatibleCredentialProvider
  // - SDKv3CompatibleCredentials
  public async getProvider(accountId: string, mode: ForReading | ForWriting):
 Promise<PluginProviderResult> {
    // The access mode can be used to provide different credential sets
    const readOnly = mode === 0 satisfies ForReading;

    // Create appropriate credentials based on your authentication mechanism
    const credentials: SDKv3CompatibleCredentials = {
      accessKeyId: 'AKIAIOSFODNN7EXAMPLE',
      secretAccessKey: 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
      // Add sessionToken if using temporary credentials
      // sessionToken: 'AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT
+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/
IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk',
      // expireTime: new Date(Date.now() + 3600 * 1000), // 1 hour from now
    };

    return credentials;
  }
}

const plugin: Plugin = {
  version: '1',
  init(host: IPluginHost): void {
    // Register the credential provider to the PluginHost.
    host.registerCredentialProviderSource(new CustomCredentialProviderSource());
  }
};

export = plugin;
```

ESM

```
import type { CredentialProviderSource, ForReading, ForWriting, IPluginHost,
 Plugin, PluginProviderResult, SDKv3CompatibleCredentials } from '@aws-cdk/cli-
plugin-contract';

class CustomCredentialProviderSource implements CredentialProviderSource {
  // Friendly name for the provider, used in error messages
  public readonly name: string = 'custom-credential-provider';

  // Check if this provider is available on the current system
```

```
  public async isAvailable(): Promise<boolean> {
    // Return false if the plugin cannot be used
    // For example, if it depends on files not present on the host
    return true;
  }

  // Check if this provider can provide credentials for a specific account
  public async canProvideCredentials(accountId: string): Promise<boolean> {
    // Return false if the plugin cannot provide credentials for this account
    // For example, if the account is not managed by this credential system
    return true;
    // You can use patterns to filter specific accounts
    // return accountId.startsWith('123456');
  }

  // Get credentials for the specified account and access mode
  // Returns PluginProviderResult which can be one of:
  // - SDKv2CompatibleCredentials (AWS SDK v2 entered maintenance on Sept 8, 2024
 and will reach end-of-life on Sept 8, 2025)
  // - SDKv3CompatibleCredentialProvider
  // - SDKv3CompatibleCredentials
  public async getProvider(accountId: string, mode: ForReading | ForWriting):
 Promise<PluginProviderResult> {
    // The access mode can be used to provide different credential sets
    const readOnly = mode === 0 satisfies ForReading;

    // Create appropriate credentials based on your authentication mechanism
    const credentials: SDKv3CompatibleCredentials = {
      accessKeyId: 'AKIAIOSFODNN7EXAMPLE',
      secretAccessKey: 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
      // Add sessionToken if using temporary credentials
      // sessionToken: 'AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT
+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/
IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk',
      // expireTime: new Date(Date.now() + 3600 * 1000), // 1 hour from now
    };

    return credentials;
  }
}

const plugin: Plugin = {
  version: '1',
  init(host: IPluginHost): void {
```

```
    // Register the credential provider to the PluginHost.
    host.registerCredentialProviderSource(new CustomCredentialProviderSource());
  }
};

export { plugin as 'module.exports' };
```

JavaScript

CommonJS

```
// Implement the CredentialProviderSource interface
class CustomCredentialProviderSource {
  constructor() {
    // Friendly name for the provider, used in error messages
    this.name = 'custom-credential-provider';
  }

  // Check if this provider is available on the current system
  async isAvailable() {
    // Return false if the plugin cannot be used
    // For example, if it depends on files not present on the host
    return true;
  }

  // Check if this provider can provide credentials for a specific account
  async canProvideCredentials(accountId) {
    // Return false if the plugin cannot provide credentials for this account
    // For example, if the account is not managed by this credential system
    return true;
    // You can use patterns to filter specific accounts
    // return accountId.startsWith('123456');
  }

  // Get credentials for the specified account and access mode
  // Returns PluginProviderResult which can be one of:
  // - SDKv2CompatibleCredentials (AWS SDK v2 entered maintenance on Sept 8, 2024
 and will reach end-of-life on Sept 8, 2025)
  // - SDKv3CompatibleCredentialProvider
  // - SDKv3CompatibleCredentials
  async getProvider(accountId, mode) {
    // The access mode can be used to provide different credential sets
```

```
    const readOnly = mode === 0; // 0 indicates ForReading; 1 indicates
 ForWriting

    // Create appropriate credentials based on your authentication mechanism
    const credentials = {
      accessKeyId: 'ASIAIOSFODNN7EXAMPLE',
      secretAccessKey: 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
      // Add sessionToken if using temporary credentials
      // sessionToken: 'AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT
+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/
IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk',
      // expireTime: new Date(Date.now() + 3600 * 1000), // 1 hour from now
    };

    return credentials;
  }
}

const plugin = {
  version: '1',
  init(host) {
    // Register the credential provider to the PluginHost.
    host.registerCredentialProviderSource(new CustomCredentialProviderSource());
  }
};

module.exports = plugin;
```

ESM

```
// Implement the CredentialProviderSource interface
class CustomCredentialProviderSource {
  constructor() {
    // Friendly name for the provider, used in error messages
    this.name = 'custom-credential-provider';
  }

  // Check if this provider is available on the current system
  async isAvailable() {
    // Return false if the plugin cannot be used
    // For example, if it depends on files not present on the host
    return true;
  }
```

```
  // Check if this provider can provide credentials for a specific account
  async canProvideCredentials(accountId) {
    // Return false if the plugin cannot provide credentials for this account
    // For example, if the account is not managed by this credential system
    return true;
    // You can use patterns to filter specific accounts
    // return accountId.startsWith('123456');
  }

  // Get credentials for the specified account and access mode
  // Returns PluginProviderResult which can be one of:
  // - SDKv2CompatibleCredentials (AWS SDK v2 entered maintenance on Sept 8, 2024
 and will reach end-of-life on Sept 8, 2025)
  // - SDKv3CompatibleCredentialProvider
  // - SDKv3CompatibleCredentials
  async getProvider(accountId, mode) {
    // The access mode can be used to provide different credential sets
    const readOnly = mode === 0; // 0 indicates ForReading; 1 indicates
 ForWriting

    // Create appropriate credentials based on your authentication mechanism
    const credentials = {
      accessKeyId: 'ASIAIOSFODNN7EXAMPLE',
      secretAccessKey: 'wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY',
      // Add sessionToken if using temporary credentials
      // sessionToken: 'AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT
+FvwqnKwRcOIfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/
IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R4Olgk',
      // expireTime: new Date(Date.now() + 3600 * 1000), // 1 hour from now
    };

    return credentials;
  }
}

const plugin = {
  version: '1',
  init(host) {
    // Register the credential provider to the PluginHost.
    host.registerCredentialProviderSource(new CustomCredentialProviderSource());
  }
};
```

```
export { plugin as 'module.exports' };
```

Important

Credentials obtained from providers are cached by the CDK Toolkit. It's strongly recommended that credential objects returned by your provider are self-refreshing to prevent expiration issues during long-running operations. For more information, see Credentials in the *AWS SDK for JavaScript v3 documentation*.

Learn more

To learn more about CDK Toolkit plugins, see the AWS CDK Toolkit Plugin Contract in the *aws-cdk-cli GitHub repository*.

Perform programmatic actions using the CDK Toolkit Library

Understanding the CDK Toolkit Library

The CDK Toolkit Library enables you to perform CDK actions programmatically through code instead of using CLI commands. You can use this library to create custom tools, build specialized CLI applications, and integrate CDK capabilities into your development workflows.

Manage your infrastructure lifecycle with programmatic control

The CDK Toolkit Library provides programmatic interfaces for the following CDK actions:

- **Synthesis** - Generate AWS CloudFormation templates and deployment artifacts.
- **Deployment** - Provision or update infrastructure using CloudFormation templates.
- **List** - View information about stacks and their dependencies.
- **Watch** - Monitor CDK apps for local changes.
- **Rollback** - Return stacks to their last stable state.
- **Destroy** - Remove CDK stacks and associated resources.

Enhance and customize your infrastructure management

- **Control through code** - Integrate infrastructure management directly into your applications and build responsive deployment pipelines.
- **Manage cloud assemblies** - Create, inspect, and transform your infrastructure definitions before deployment.
- **Customize deployments** - Configure parameters, rollback behavior, and monitoring to match your requirements.
- **Handle errors precisely** - Implement structured error handling with detailed diagnostic information.
- **Tailor communications** - Configure custom progress indicators and logging through `IoHost` implementations.
- **Connect with AWS** - Configure profiles, Regions, and authentication flows programmatically.

Choosing when to use the CDK Toolkit Library

The CDK Toolkit Library is particularly valuable when you need to:

- Automate infrastructure deployments as part of CI/CD pipelines.
- Build custom deployment tools tailored to your organization's needs.
- Integrate CDK actions into existing applications or platforms.
- Create specialized deployment workflows with custom validation or approval steps.
- Implement advanced infrastructure management patterns across multiple environments.

Using the CDK Toolkit Library

The following example shows how to create and deploy a simple S3 bucket using the CDK Toolkit Library:

```
// Import required packages
import { Toolkit } from '@aws-cdk/toolkit-lib';
import { App, Stack } from 'aws-cdk-lib';
import * as s3 from 'aws-cdk-lib/aws-s3';

// Create and configure the CDK Toolkit
const toolkit = new Toolkit();

// Create a cloud assembly source with an inline app
const cloudAssemblySource = await toolkit.fromAssemblyBuilder(async () => {
   const app = new App();
   const stack = new Stack(app, 'SimpleStorageStack');

   // Create an S3 bucket in the stack
   new s3.Bucket(stack, 'MyFirstBucket', {
      versioned: true
   });

   return app.synth();
});

// Deploy the stack
await toolkit.deploy(cloudAssemblySource);
```

What you can do next

- **Automate deployments** - Trigger deployments programmatically and add pre/post deployment steps.
- **Integrate with systems** - Connect with CI/CD workflows, custom tools, and monitoring solutions.
- **Control deployment details** - Configure fine-grained options for stack selection and multi-environment deployments.
- **Enhance reliability** - Implement production-ready error handling and deployment progress tracking.

Next steps

To begin using the CDK Toolkit Library, see Get started with the CDK Toolkit Library.

Learn more

To learn more about the CDK Toolkit Library, see the following:

- ReadMe in the *@aws-cdk/toolkit-lib* `npm` package.
- AWS CDK Toolkit Library API Reference.

Getting started with the CDK Toolkit Library

Get started with using the AWS CDK Toolkit Library to programmatically perform CDK actions, such as synthesis and deployment, in your code.

Prerequisites

1. Supported version of Node.js installed.
2. AWS credentials configured.
3. Basic familiarity with the AWS CDK.

For more information, see AWS CDK prerequisites.

Step 1: Installing the CDK Toolkit Library

Install the CDK Toolkit Library package in your project's development environment by running the following:

```
npm install --save @aws-cdk/toolkit-lib
```

Step 2: Initializing the CDK Toolkit Library

Create a CDK Toolkit instance to perform programmatic actions on your CDK app.

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

const toolkit = new Toolkit({
    // Optional configuration options go here
});
```

You can customize the CDK Toolkit instance during creation. For instructions, see Configure your CDK Toolkit instance.

Step 3: Creating a cloud assembly source for your CDK app

A cloud assembly source provides instructions for generating CloudFormation templates from your CDK app. You can create one in multiple ways. The following are a few examples:

1. **An inline assembly builder function**:

   ```
   import * as cdk from 'aws-cdk-lib';

   const cloudAssemblySource = await toolkit.fromAssemblyBuilder(async () => {
     const app = new cdk.App();
     new MyStack(app, 'MyStack');
     return app.synth();
   });
   ```

2. **An existing CDK app file**:

   ```
   const cloudAssemblySource = await toolkit.fromCdkApp("ts-node app.ts");
   ```

For more information, see Configure cloud assembly sources.

Step 4: Defining programmatic actions for your CDK app

Now that you've created a CDK Toolkit instance and cloud assembly source, you can start to define programmatic actions. The following is a basic example that creates a deployment of the MyStack stack:

```
import { StackSelectionStrategy } from '@aws-cdk/toolkit-lib';

await toolkit.deploy(cloudAssemblySource, {
  stacks: {
    strategy: StackSelectionStrategy.PATTERN_MUST_MATCH, // Deploy only stacks that
 exactly match the provided patterns
    patterns: ["MyStack"],
  },
});
```

Step 5: Customizing the CDK Toolkit further

You can configure and customize the CDK Toolkit further for your needs:

- **Messages and interactions** - Configure how the CDK Toolkit communicates with users and applications. See Configure messages & interactions.
- **Error handling** - Implement structured error handling for CDK operations. See Configure error handling.

Additional resources

For more information on the CDK Toolkit Library npm package, see the ReadMe in the *@aws-cdk/toolkit-lib* npm package.

For API reference information, see the CDK Toolkit Library API reference.

Configuring CDK Toolkit programmatic actions

The AWS CDK Toolkit Library provides programmatic interfaces for application lifecycle actions such as synthesis, deployment, and stack management. This guide explains how to use each action in your code.

Generating cloud assemblies with synth

The synth action generates a cloud assembly from your cloud assembly source. For more information about synthesis, see Configure and perform CDK stack synthesis. A cloud assembly contains the following deployment artifacts from your CDK app:

- AWS CloudFormation templates that define your infrastructure.
- Assets such as Lambda function code or Docker images.
- Deployment metadata and configuration.

Here's how to use the synth action to create a cloud assembly:

```
// Create a toolkit instance
const toolkit = new Toolkit();

// Create a cloud assembly source from a TypeScript app
const cloudAssemblySource = await toolkit.fromCdkApp("ts-node app.ts");

// Generate a cloud assembly
const cloudAssembly = await toolkit.synth(cloudAssemblySource);

// Use the cloud assembly for operations
await toolkit.list(cloudAssembly);
await toolkit.deploy(cloudAssembly);
await toolkit.diff(cloudAssembly);

// Query information from the cloud assembly
const template = cloudAssembly.getStack("my-stack").template;
```

> **Tip**
>
> Using a cloud assembly can optimize performance when you need to perform multiple operations, since synthesis only happens once. For more information about managing cloud assemblies, including caching and disposal, see Create and manage cloud assemblies.

Viewing stack information with list

The `list` action retrieves information about the stacks in your CDK application, including their dependencies and current status. Use this action to inspect your infrastructure before deployment or to generate reports.

```
import { StackSelectionStrategy } from '@aws-cdk/toolkit-lib';

// Get information about specific stacks
const stackDetails = await toolkit.list(cloudAssemblySource, {
  stacks: {
    strategy: StackSelectionStrategy.PATTERN_MUST_MATCH,
    patterns: ["my-stack"], // Only include stacks matching this pattern
  }
});

// Process the returned stack information
for (const stack of stackDetails) {
  console.log(`Stack: ${stack.id}, Dependencies: ${stack.dependencies}`);
}
```

Provisioning infrastructure with deploy

The `deploy` action provisions or updates your infrastructure in AWS using the cloud assembly produced during synthesis. For an introduction to deploying, see [Deploy AWS CDK applications](). You can control deployment options such as stack selection, parameter values, and rollback behavior.

```
// Deploy stacks with parameter values
await toolkit.deploy(cloudAssemblySource, {
  parameters: StackParameters.exactly({
    "MyStack": {
      "BucketName": "amzn-s3-demo-bucket"
    }
  })
});
```

The deploy action supports different deployment methods to accommodate various workflows. For most scenarios, especially in production environments, we recommend using the default deployment method which uses CloudFormation change sets. For development environments where iteration speed is important, you can use alternative methods like hotswap.

```
import { StackSelectionStrategy } from '@aws-cdk/toolkit-lib';

// Deploy using default deployment method (recommended for production)
await toolkit.deploy(cloudAssemblySource, {
  parameters: StackParameters.exactly({
    "MyStack": {
      "BucketName": "amzn-s3-demo-bucket"
    }
  })
});

// For development environments only: Deploy with hotswap for faster iterations
// Note: We recommend using default deployment methods for production environments
await toolkit.deploy(cloudAssemblySource, {
  deploymentMethod: { method: "hotswap", fallback: true }, // Faster but introduces
 drift
  stacks: {
    strategy: StackSelectionStrategy.PATTERN_MUST_MATCH,
    patterns: ["dev-stack"]
  }
});
```

Reverting failed deployments with rollback

The `rollback` action returns a stack to its last stable state when a deployment fails and cannot be automatically reversed. Use this action to recover from failed deployments that require manual intervention.

```
import { StackSelectionStrategy } from '@aws-cdk/toolkit-lib';

// Roll back stacks to their last stable state
await toolkit.rollback(cloudAssemblySource, {
  orphanFailedResources: false, // When true, removes failed resources from
 CloudFormation management
  stacks: {
    strategy: StackSelectionStrategy.PATTERN_MUST_MATCH,
    patterns: ["failed-stack"]
  }
});
```

Monitoring changes with watch

The `watch` action continuously monitors your CDK app for local file changes and automatically performs deployments or hotswaps. This creates a file watcher that runs until your code exits or is terminated.

> **Warning**
>
> Hotswap deployments update resources directly without going through CloudFormation when possible, making updates faster during development. This is enabled by default for the `watch` command. While this speeds up the development cycle, it introduces drift between your CloudFormation templates and deployed resources. Therefore, we recommend that you don't use hotswaps in production environments.

```
import { StackSelectionStrategy } from '@aws-cdk/toolkit-lib';

// Start watching for changes
const watcher = await toolkit.watch(cloudAssemblySource, {
  include: ["lib/**/*.ts"], // Only watch TypeScript files in the lib directory
  exclude: ["**/*.test.ts"], // Ignore test files
  deploymentMethod: { method: "hotswap" }, // This is the default, shown here for
 clarity
  stacks: {
    strategy: StackSelectionStrategy.ALL // Watch all stacks
  }
});

// Later in your code, you can explicitly stop watching:
// await watcher.dispose();
```

The watch function returns an `IWatcher` object that allows you to explicitly control when to stop watching. Call the `dispose()` method on this object when you want to end the watch process.

Removing infrastructure with destroy

The `destroy` action removes CDK stacks and their associated resources from AWS. Use this action to clean up resources when they're no longer needed.

> **Important**
>
> The destroy action permanently removes resources without prompting for confirmation, unlike the CLI version of this command. Make sure you have backups of any important data before destroying stacks.

```
import { StackSelectionStrategy } from '@aws-cdk/toolkit-lib';

// Remove specific stacks and their resources
await toolkit.destroy(cloudAssemblySource, {
  stacks: {
    strategy: StackSelectionStrategy.PATTERN_MUST_MATCH,
    patterns: ["dev-stack"], // Only destroy stacks matching this pattern
  }
});
```

Configuring your CDK Toolkit instance

Learn how to customize your AWS CDK Toolkit Library instance with options for message handling, AWS profile selection, and stack selection strategies. This guide explains the available configuration options and how to implement them effectively to meet your specific deployment requirements.

Configuring your AWS profile

When you use the CDK Toolkit Library, it makes API calls to AWS using the SDK. While authentication is loaded automatically from your environment, you can explicitly specify which profile to use:

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Create a toolkit instance with a specific AWS profile
const toolkit = new Toolkit({
  sdkConfig: { profile: "my-profile" },
});
```

Configuring stack selection

Most CDK Toolkit actions require you to specify which stacks to operate on. The `StackSelector` configuration controls this selection.

Select all stacks

Use this when you want to operate on every stack in your CDK app:

```
import { StackSelectionStrategy } from '@aws-cdk/toolkit-lib';

// Select all stacks in the cloud assembly
await toolkit.deploy(cloudAssemblySource, {
  stacks: {
    strategy: StackSelectionStrategy.ALL_STACKS
  }
});
```

Select only main assembly stacks

Use this to select only the top-level stacks from the main assembly:

```
// Select only top-level stacks
await toolkit.deploy(cloudAssemblySource, {
  stacks: {
    strategy: StackSelectionStrategy.MAIN_ASSEMBLY
  }
});
```

Select a single stack

Use this when your assembly contains exactly one stack and you want to assert this condition. If the assembly includes a single stack, it returns that stack. Otherwise, it throws an exception:

```
// Ensure there's exactly one stack and select it
await toolkit.deploy(cloudAssemblySource, {
  stacks: {
    strategy: StackSelectionStrategy.ONLY_SINGLE
  }
});
```

Select stacks by pattern

Use this to select specific stacks by name pattern:

```
// Select stacks matching specific patterns
await toolkit.deploy(cloudAssemblySource, {
  stacks: {
    strategy: StackSelectionStrategy.PATTERN_MUST_MATCH,
    patterns: ["Dev-*", "Test-Backend"],  // Supports wildcards
  }
});
```

> **Tip**
>
> Use `PATTERN_MUST_MATCH_SINGLE` to ensure exactly one stack matches your patterns, or `PATTERN_MATCH` if it's acceptable for no stacks to match. Pattern matching supports wildcards like "*" to match multiple stacks with similar names.

Configuring error handling

The CDK Toolkit uses structured errors to help you identify and handle issues. Each error includes:

- A **source** indicating where the error originated (toolkit or user).
- A specific **error type** (authentication, validation, etc.).
- A descriptive **message**.

Handling errors

Use the helper methods provided by the CDK Toolkit to detect and handle specific error types:

```
import { ToolkitError } from '@aws-cdk/toolkit-lib';

try {
  // Attempt a CDK Toolkit operation
  await toolkit.deploy(cloudAssemblySource, {
    stacks: { strategy: StackSelectionStrategy.ALL_STACKS }
  });
```

```
} catch (error) {
  // Handle specific error types
  if (ToolkitError.isAuthenticationError(error)) {
    // Example: AWS credentials are missing or invalid
    console.error('Authentication failed. Check your AWS credentials.');

  } else if (ToolkitError.isAssemblyError(error)) {
    // Example: Your CDK app has errors in stack definitions
    console.error('CDK app error:', error.message);

  } else if (ToolkitError.isDeploymentError(error)) {
    // Example: CloudFormation deployment failed
    console.error('Deployment failed:', error.message);

  } else if (ToolkitError.isToolkitError(error)) {
    // Handle all other Toolkit errors
    console.error('CDK Toolkit error:', error.message);

  } else {
    // Handle unexpected errors
    console.error('Unexpected error:', error);
  }
}
```

> **Important**
>
> Don’t rely on `instanceof` checks for error types, as they can behave unexpectedly when working with multiple copies of the same package. Always use the provided helper methods like `ToolkitError.isAuthenticationError()`.

Configuring Toolkit actions

Each CDK Toolkit action (deploy, synth, list, etc.) has its own specific configuration options. These actions allow you to manage the complete lifecycle of your CDK infrastructure. For detailed information on configuring individual actions, see Configure CDK Toolkit programmatic actions.

> **Tip**
>
> When building automation workflows, consider combining multiple actions in sequence. For example, you might want to `synth` your app, `list` the stacks to verify what will be deployed, and then `deploy` the infrastructure.

Managing AWS CDK Toolkit Library cloud assembly sources

Use the AWS CDK Toolkit Library to configure cloud assembly sources and customize how you deploy your CDK applications. This guide shows you how to configure cloud assembly sources to meet your deployment requirements and workflow needs.

Before you use the CDK Toolkit, specify a cloud assembly source. A *cloud assembly source* provides instructions for generating a cloud assembly from your CDK app. The resulting cloud assembly contains the synthesized infrastructure artifacts that the CDK Toolkit deploys to AWS.

The CDK Toolkit Library offers several approaches to configure cloud assembly sources, each suited for different scenarios and workflows.

Selecting a cloud assembly source

Method	Best For	Consideration
`fromCdkApp`	Working with existing CDK applications in any supported language.	Requires the appropriate language runtime to be installed.
`fromAssemblyBuilder`	Creating CDK constructs inline with full control over the synthesis process.	Provides low-level access to CDK functionality and can be used to build custom versions of other methods like `fromCdkApp` .
`fromAssemblyDirectory`	Using pre-synthesized cloud assemblies.	Faster execution as synthesis step is skipped.

Method	Best For	Consideration
Custom Source	Extremely specialized scenarios requiring complete custom implementation.	Requires implementing the `ICloudAssemblySource` interface from scratch; lacks built-in functionality like context lookups; rarely needed for most use cases.

Configuring your cloud assembly source

From an existing CDK app

Use the `fromCdkApp` method to work with CDK apps written in any supported language. This approach is ideal when you have an existing CDK application and want to deploy it programmatically.

```
import { App } from 'aws-cdk-lib';
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Create a toolkit instance
const toolkit = new Toolkit();

// TypeScript app
const cloudAssemblySource = await toolkit.fromCdkApp("ts-node app.ts");

// Deploy a specific stack from the assembly
await toolkit.deploy(cloudAssemblySource, {
    stacks: ['MyStack']
});

// Other language examples:
// JavaScript app
// const cloudAssemblySource = await toolkit.fromCdkApp("node app.js");

// Python app
// const cloudAssemblySource = await toolkit.fromCdkApp("python app.py");

// Java app
```

```
// const cloudAssemblySource = await toolkit.fromCdkApp("mvn -e -q exec:java -
Dexec.mainClass=com.mycompany.app.App");
```

From an inline assembly builder

Create a CDK app directly in your code using an assembly builder function. This approach is useful for simple deployments or testing scenarios where you want to define your infrastructure inline.

```
import { App, Stack, RemovalPolicy, StackProps } from 'aws-cdk-lib';
import { Bucket } from 'aws-cdk-lib/aws-s3';
import { Toolkit } from '@aws-cdk/toolkit-lib';
import { Construct } from 'constructs';

// Create a cloud assembly source from an inline CDK app
const cloudAssemblySource = await toolkit.fromAssemblyBuilder(async () => {
    const app = new App();

    // Define a simple stack with an S3 bucket
    class MyStack extends Stack {
        constructor(scope: Construct, id: string, props?: StackProps) {
            super(scope, id, props);

            // Create an S3 bucket
            new Bucket(this, 'MyBucket', {
                versioned: true,
                removalPolicy: RemovalPolicy.DESTROY,
                autoDeleteObjects: true
            });
        }
    }

    // Instantiate the stack
    new MyStack(app, 'MyInlineStack');

    return app.synth();
});

// Deploy using the cloud assembly source
await toolkit.deploy(cloudAssemblySource, {
    stacks: ['MyInlineStack']
});
```

From an existing assembly directory

If you already have a synthesized cloud assembly, you can use it directly. This is useful when you've already run `cdk synth` or when working with cloud assemblies generated by CI/CD pipelines.

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Create a toolkit instance
const toolkit = new Toolkit();

// Use an existing cloud assembly directory
const cloudAssemblySource = await toolkit.fromAssemblyDirectory("cdk.out");

// Deploy using the cloud assembly source
await toolkit.deploy(cloudAssemblySource, {
    stacks: ['MyStack']
});
```

Working with cached cloud assemblies

When working with cloud assemblies, you have two options:

1. Use a cloud assembly source directly (simple but may be slower):

    ```
    // Each operation triggers a new synthesis
    await toolkit.deploy(cloudAssemblySource, { /* options */ });
    await toolkit.list(cloudAssemblySource, { /* options */ });
    ```

2. Cache the cloud assembly (faster for multiple operations):

    ```
    // Synthesize once and reuse
    const cloudAssembly = await toolkit.synth(cloudAssemblySource);
    try {
      // Multiple operations use the same assembly
      await toolkit.deploy(cloudAssembly, { /* options */ });
      await toolkit.list(cloudAssembly, { /* options */ });
    } finally {
      // Clean up when done
      await cloudAssembly.dispose();
    }
    ```

Use cached assemblies when:

- You're performing multiple operations (deploy, list, diff, etc.).
- Your CDK app doesn't change frequently during operations.
- You want faster performance.

Use cloud assembly sources directly when:

- You're performing a single operation.
- Your CDK app changes frequently.
- You want simpler code and don't need to prioritze Toolkit operation speed.

> **Important**
>
> Most Toolkit interactions should use a cached assembly for better performance. The only time to avoid caching is when your source changes frequently and checking for changes would be expensive.

How to create, cache, and reuse cloud assemblies

After you create a cloud assembly source, you can generate a cloud assembly by synthesizing it. A cloud assembly contains the AWS CloudFormation templates and assets needed for deployment.

We recommend that you generate a cloud assembly once and reuse it for multiple Toolkit operations. This caching approach is more efficient than regenerating the assembly for each operation. Consider regenerating the assembly only when your source changes frequently.

Here's how to create a cached cloud assembly:

```
// Generate a cloud assembly from your source
const cloudAssembly = await toolkit.synth(cloudAssemblySource);
```

You can then perform various Toolkit actions on the cached cloud assembly, such as `list()`, `deploy()`, and `diff()`. By caching cloud assemblies, subsequent Toolkit actions are performed faster since synthesis occurs only once. For more information, see [synth - Generate cloud assemblies](#).

Dispose of cloud assembly resources

Always dispose of cloud assemblies when you're done using them to clean up temporary resources. We recommend using a try/finally block to ensure proper cleanup, especially when performing multiple operations:

```
// Generate a cloud assembly
const cloudAssembly = await toolkit.synth(cloudAssemblySource);

try {
    // Use the cloud assembly for multiple operations
    await toolkit.list(cloudAssembly);
    await toolkit.deploy(cloudAssembly);
} finally {
    // Always dispose when done
    await cloudAssembly.dispose();
}
```

Here's an example showing how to create and dispose of a cached cloud assembly:

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

const toolkit = new Toolkit();

// Create cloud assembly source from a CDK app
const cloudAssemblySource = await toolkit.fromCdkApp("ts-node app.ts");

// Create cloud assembly from source
const cloudAssembly = await toolkit.synth(cloudAssemblySource);

try {
    // List stacks in the assembly
    await toolkit.list(cloudAssembly);

    // Check for changes
    await toolkit.diff(cloudAssembly);

    // Deploy if needed
    await toolkit.deploy(cloudAssembly);
} finally {
    // Always dispose when done
    await cloudAssembly.dispose();
```

```
}
```

Understanding cloud assembly lifetimes

When you create a cached cloud assembly using `synth()`, you get a special type that serves as both a readable `CloudAssembly` and a `CloudAssemblySource`. Any cloud assemblies produced from this cached assembly (for example, from list or deploy operations) are tied to the parent's lifetime:

- Only the parent's dispose() call actually cleans up resources
- Cloud assemblies from list/deploy operations are managed by their parent
- Failing to dispose of a cached cloud assembly is considered a bug

Best practices for cloud assembly sources

When working with cloud assembly sources, consider these best practices:

- **Choose the right source method**: Select the approach that best fits your workflow and requirements.
- **Cache cloud assemblies**: Generate a cloud assembly once using `synth()` and reuse it for multiple operations to avoid unnecessary synthesis, especially for large applications.
- **Error handling**: Implement basic error handling to catch and display errors to users. Keep error handling simple and focus on providing clear error messages.
- **Version compatibility**: Ensure that your CDK Toolkit Library version can support the cloud assemblies you're working with. If the Construct Library used to create the cloud assembly is newer than what your Toolkit Library supports, you'll receive an error.
- **Environment variables**: Be aware that certain environment variables can affect cloud assembly synthesis and deployment. Variables like `CDK_DEFAULT_ACCOUNT`, `CDK_DEFAULT_REGION`, `CDK_OUTDIR`, and `CDK_CONTEXT_JSON` can override default behaviors. Ensure these are set appropriately for your deployment environment.

The following example demonstrates how to implement error handling and proper cleanup while reusing a cloud assembly for multiple operations:

```
import { Toolkit } from '@aws-cdk/toolkit-lib';
```

```
// Example with error handling and proper cleanup
async function deployInfrastructure(): Promise<void> {
    let cloudAssembly;

    try {
        // Generate a cloud assembly once
        cloudAssembly = await toolkit.synth(cloudAssemblySource);

        // Reuse the same cloud assembly for multiple operations
        await toolkit.list(cloudAssembly);    // Uses existing assembly
        await toolkit.deploy(cloudAssembly);   // Uses existing assembly
        await toolkit.diff(cloudAssembly);     // Uses existing assembly
    } catch (error) {
        console.error("Failed to deploy:", error);
    } finally {
        // Always dispose when done
        if (cloudAssembly) {
            await cloudAssembly.dispose();
        }
    }
}

// Call the async function
deployInfrastructure().catch(error => {
    console.error("Deployment failed:", error);
    process.exit(1);
});
```

Resolving potential issues

Follow these steps to resolve potential issues with cloud assembly sources:

- **Install missing dependencies**: Run `npm install` to install required dependencies for your CDK app.
- **Fix path issues**: Check that paths to CDK apps and assembly directories exist and are accessible.
- **Resolve version mismatches**: Update your CDK Toolkit Library version to match your CDK app version.
- **Fix synthesis errors**: Review your CDK app code for syntax errors or invalid configurations.

When errors occur during Toolkit operations, keep error handling simple and focus on providing clear error messages to users. Always dispose of cloud assemblies when you're done using them. Here's an example showing basic error handling with proper cleanup:

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Example with simple error handling
try {
    // Create the cloud assembly source
    const cloudAssemblySource = await toolkit.fromCdkApp("ts-node app.ts");

    // Synthesize the cloud assembly
    const cloudAssembly = await toolkit.synth(cloudAssemblySource);

    // Use the cloud assembly
    await toolkit.list(cloudAssembly);
} catch (error) {
    // Display the error message
    console.error("Operation failed:", error.message);
} finally {
    // Clean up resources
    await cloudAssembly.dispose();
}
```

Configuring CDK Toolkit messages and interactions

The AWS CDK Toolkit Library provides the `IIoHost` interface to customize how messages and interactions are handled during CDK operations, enabling you to control the display of deployment progress, error messages, and user prompts to better integrate with your application's user experience.

Before performing operations like deployment or synthesis, you need to understand how the CDK Toolkit communicates with users. The `IIoHost` interface serves as the communication channel between the CDK Toolkit and your application, handling both outgoing messages and incoming user responses.

When the CDK Toolkit runs operations, it communicates through two primary mechanisms:

- **Messages**: Informational outputs that notify you about operation progress (like "Starting deployment" or "Resource created").

- **Requests**: Decision points that require input or confirmation (like "Do you want to deploy these changes?"), giving you an opportunity to provide information that wasn't known to be needed beforehand.

Using the `IIoHost` interface

The `IIoHost` interface consists of two primary methods:

1. `notify`: Handles one-way informational messages.
2. `requestResponse`: Handles interactive requests that require a response.

```
import { IoMessage, IoRequest } from '@aws-cdk/toolkit-lib';

interface IIoHost {
  // Handle informational messages
  notify(message: IoMessage): Promise<void>;

  // Handle requests that need responses
  requestResponse(request: IoRequest): Promise<any>;
}
```

Message levels and request types

The CDK Toolkit generates several types of messages and requests:

Message levels

- **Debug**: Detailed messages for troubleshooting.
- **Error**: Error messages that may affect operation.
- **Info**: General informational messages.
- **Result** - Primary message of an operation.
- **Trace**: Very detailed execution flow information.
- **Warning**: Warning messages that don't prevent operation.

For a complete list, see IoMessages Registry in the *AWS CDK Toolkit Library API Reference*.

Request types

The CDK Toolkit sends requests when it needs input or confirmation from the user. These are special messages that allow for a response. If no response is provided, the Toolkit will use a default response when available.

Basic `IIoHost` implementation

Here's a simple example of implementing a custom io host:

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Create a toolkit with custom message handling
const toolkit = new Toolkit({
  ioHost: {
    // Implementing the IIoHost interface
    // Handle informational messages
    notify: async function (msg) {
      // Example: Handle different message levels appropriately
      switch (msg.level) {
        case 'error':
          console.error(`[${msg.time}] ERROR: ${msg.message}`);
          break;
        case 'warning':
          console.warn(`[${msg.time}] WARNING: ${msg.message}`);
          break;
        case 'info':
          console.info(`[${msg.time}] INFO: ${msg.message}`);
          break;
        case 'debug':
          console.debug(`[${msg.time}] DEBUG: ${msg.message}`);
          break;
        case 'trace':
          console.debug(`[${msg.time}] TRACE: ${msg.message}`);
          break;
        default:
          console.log(`[${msg.time}] ${msg.level}: ${msg.message}`);
      }
    },

    // Handle requests that need responses
    requestResponse: async function (msg) {
      // Example: Log the request and use default response
```

```
      console.log(`Request: ${msg.message}, using default: ${msg.defaultResponse}`);
      return msg.defaultResponse;

      // Or implement custom logic to provide responses
      // if (msg.type === 'deploy') {
      //   return promptUserForDeployment(msg);
      // }
    }
  } as IIoHost // Explicitly cast to IIoHost interface
});
```

The CDK Toolkit awaits the completion of each call, allowing you to perform asynchronous operations like HTTP requests or user prompts when handling messages.

Default `IIoHost` behavior

If you don't provide a custom io host, the CDK Toolkit Library uses a default non-interactive implementation:

- Console output for messages (using appropriate colors for different message types).
- Fully non-interactive with no prompts for user input.
- Automatically uses default responses where possible (equivalent to answering "yes" to prompts).
- Fails when input is required but no default response is available.

This default behavior is suitable for unattended operations, but not for interactive command-line applications. For command-line applications that require user interaction, you'll need to implement a custom io host. Custom implementations are also useful for integrating with logging systems, UIs, or other specialized environments.

Advanced io host implementation

For more complex scenarios, we recommend extending the `NonInteractiveIoHost` class as a starting point. This approach allows you to leverage the existing non-interactive implementation while customizing only the specific behaviors you need to change.

Here's an example of a custom io host that extends the base implementation:

```
import { NonInteractiveIoHost } from '@aws-cdk/toolkit-lib';
```

```
class MyCustomIoHost extends NonInteractiveIoHost {
  // Override only the methods you need to customize

  // Example: Custom implementation for notify
  public async notify(msg: IoMessage<unknown>): Promise<void> {
    // Add custom notification handling logic
    if (msg.level === 'error') {
      console.error(`ERROR: ${msg.message}`);
      // Optionally log to a service or notify a monitoring system
      await this.logToMonitoringService(msg);
    } else {
      await super.notify(msg);
    }
  }

  // Example: Custom implementation for requestResponse
  public async requestResponse<T, U>(request: IoRequest<T, U>): Promise<U> {
    // Implement custom request handling
    console.log(`Received request: ${request.message}`);
    return request.defaultResponse;
  }

  private async logToMonitoringService(msg: IoMessage<unknown>): Promise<void> {
    // Implementation for monitoring service integration
    console.log(`Logging to monitoring service: ${msg.level} - ${msg.message}`);
  }
}
```

This approach is more maintainable than implementing the entire `IIoHost` interface from scratch, as you only need to override the specific methods that require custom behavior.

Integrating with different environments

Web application integration

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Example for integrating with a web application
const toolkit = new Toolkit({
  ioHost: {
    notify: async function (msg) {
      // Send message to frontend via WebSocket
      webSocketServer.send(JSON.stringify({
```

```
        type: 'cdk-notification',
        messageLevel: msg.level,
        message: msg.message,
        time: msg.time
      }));
    },

    requestResponse: async function (msg) {
      // Create a promise that will be resolved when the user responds
      return new Promise((resolve) => {
        const requestId = generateUniqueId();

        // Store the resolver function
        pendingRequests[requestId] = resolve;

        // Send request to frontend
        webSocketServer.send(JSON.stringify({
          type: 'cdk-request',
          requestId: requestId,
          requestType: msg.type,
          message: msg.message,
          defaultResponse: msg.defaultResponse
        }));

        // Frontend would call an API endpoint with the response,
        // which would then call pendingRequests[requestId](response)
      });
    }
  } as IIoHost // Explicitly cast to IIoHost interface
});
```

CI/CD environment integration

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Example for CI/CD environments (non-interactive)
const toolkit = new Toolkit({
  ioHost: {
    notify: async function (msg) {
      // Log all messages with appropriate level
      switch (msg.level) {
        case 'error':
          console.error(msg.message);
```

```
          break;
        case 'warning':
          console.warn(msg.message);
          break;
        default:
          console.log(msg.message);
      }
    },

    requestResponse: async function (msg) {
      // In CI/CD, always use default responses or predefined answers
      console.log(`Auto-responding to request: ${msg.message} with
 ${msg.defaultResponse}`);
      return msg.defaultResponse;
    }
  } as IIoHost // Explicitly cast to IIoHost interface
});
```

Best practices for io host implementation

When implementing a custom io host, consider these best practices:

- **Error handling**: Implement robust error handling in your io host methods to prevent failures from affecting CDK operations.
- **Timeouts**: Consider implementing timeouts for user interactions to prevent indefinite waiting.
- **Logging**: Store important messages in logs for troubleshooting, especially in non-interactive environments.
- **Default responses**: Provide sensible default responses for automated environments where user interaction isn't possible.
- **Progress indication**: For long-running operations, provide clear progress indicators to improve user experience.

The following example demonstrates these best practices by implementing a custom io host with error handling, timeouts for user interactions, and proper logging. This implementation is suitable for interactive applications where you need to balance user responsiveness with reliable operation:

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Example with error handling and timeouts
```

```
const toolkit = new Toolkit({
  ioHost: {
    notify: async function (msg) {
      try {
        console.log(`[${msg.time}] ${msg.level}: ${msg.message}`);
        // Additional logging or UI updates
      } catch (error) {
        // Ensure errors in notification handling don't break the CDK operation
        console.error("Error handling notification:", error);
      }
    },

    requestResponse: async function (msg) {
      try {
        // Implement timeout for user response
        const response = await Promise.race([
          getUserResponse(msg),
          new Promise(resolve => setTimeout(() => resolve(msg.defaultResponse), 60000))
        ]);
        return response;
      } catch (error) {
        console.error("Error handling request:", error);
        return msg.defaultResponse;
      }
    }
  } as IIoHost // Explicitly cast to IIoHost interface
});
```

Troubleshooting

Considerations when implementing a custom io host:

- **Unhandled promise rejections**: Ensure all async operations are properly handled with try/catch blocks.
- **Infinite waiting**: Implement timeouts for all user interactions to prevent the application from hanging.
- **Missing message levels**: Be prepared to handle new message levels that might be added in future CDK versions.
- **Inconsistent responses**: Ensure your requestResponse implementation returns values in the expected format.

The following example demonstrates a robust approach to handling message levels, including graceful handling of unknown message types that might be introduced in future CDK versions. This implementation ensures your io host remains compatible with CDK updates while maintaining proper logging:

```
import { Toolkit } from '@aws-cdk/toolkit-lib';

// Example of handling unknown message levels
const toolkit = new Toolkit({
  ioHost: {
    notify: async function (msg) {
      // Handle known levels
      const knownLevels = ['info', 'warning', 'error', 'debug', 'trace', 'status'];

      if (knownLevels.includes(msg.level)) {
        // Handle known level
        handleKnownMessageLevel(msg);
      } else {
        // Handle unknown level as info
        console.log(`Unknown message level "${msg.level}": ${msg.message}`);
      }
    },

    requestResponse: async function (msg) {
      // Default implementation
      return msg.defaultResponse;
    }
  } as IIoHost // Explicitly cast to IIoHost interface
});
```

Advanced CDK Toolkit Library examples

Learn how to use advanced features of the AWS CDK Toolkit Library through practical examples. This guide provides detailed code samples for error handling, deployment monitoring, and cloud assembly management that build upon the basic concepts covered in other sections.

Integrating features

The following example demonstrates how to combine cloud assembly sources, custom io host implementation, and deployment options:

```
import { Toolkit, StackSelectionStrategy, IIoHost } from '@aws-cdk/toolkit-lib';

async function deployApplication(appPath, environment, options = {}) {
  // Create toolkit with custom message handling
  const toolkit = new Toolkit({
    ioHost: {
      notify: async (msg) => {
        // Add environment to all messages
        console.log(`[${environment}][${msg.time}] ${msg.level}: ${msg.message}`);
      },
      requestResponse: async (msg) => {
        // In production environments, use default responses
        if (environment === 'production') {
          console.log(`Auto-approving for production: ${msg.message}`);
          return msg.defaultResponse;
        }

        // For other environments, implement custom approval logic
        return promptForApproval(msg);
      }
    } as IIoHost
  });

  try {
    // Create cloud assembly source from the CDK app
    console.log(`Creating cloud assembly source from ${appPath}`);
    const cloudAssemblySource = await toolkit.fromCdkApp(appPath);

    // Synthesize the cloud assembly
    console.log(`Synthesizing cloud assembly`);
    const cloudAssembly = await toolkit.synth(cloudAssemblySource);

    try {
      // Deploy with environment-specific options
      console.log(`Deploying to ${environment} environment`);
      return await toolkit.deploy(cloudAssembly, {
        stacks: options.stacks || { strategy: StackSelectionStrategy.ALL_STACKS },
        parameters: options.parameters || {},
        tags: {
          Environment: environment,
          DeployedBy: 'CDK-Toolkit-Library',
          DeployTime: new Date().toISOString()
        }
```

```
      });
    } finally {
      // Always dispose when done
      await cloudAssembly.dispose();
    }
  } catch (error) {
    console.error(`Deployment to ${environment} failed:`, error);
    throw error;
  }
}

// Example usage
await deployApplication('ts-node app.ts', 'staging', {
  parameters: {
    MyStack: {
      InstanceType: 't3.small'
    }
  }
});
```

Tracking deployment progress

Track deployment progress with detailed status updates:

```
import { Toolkit, StackSelectionStrategy, IIoHost } from '@aws-cdk/toolkit-lib';

// Create a progress tracker
class DeploymentTracker {
  private startTime: Date;
  private resources = new Map<string, string>();

  constructor() {
    this.startTime = new Date();
  }

  onStackEvent(stackName: string, event: string, timestamp: string) {
    // Calculate elapsed time if needed, or use the provided timestamp
    const elapsed = (new Date().getTime() - this.startTime.getTime()) / 1000;
    console.log(`[${timestamp}] (${elapsed.toFixed(1)}s elapsed) Stack ${stackName}:
 ${event}`);
  }

  onResourceEvent(resourceId: string, status: string) {
```

```
    this.resources.set(resourceId, status);
    this.printProgress();
  }

  private printProgress() {
    console.log('\nResource Status:');
    for (const [id, status] of this.resources.entries()) {
      console.log(`- ${id}: ${status}`);
    }
    console.log();
  }
}

// Use the tracker with the toolkit
const tracker = new DeploymentTracker();
const toolkit = new Toolkit({
  ioHost: {
    notify: async (msg) => {
      if (msg.code.startsWith('CDK_DEPLOY')) {
        // Track deployment events
        if (msg.data && 'stackName' in msg.data) {
          tracker.onStackEvent(msg.data.stackName, msg.message, msg.time);
        }
      } else if (msg.code.startsWith('CDK_RESOURCE')) {
        // Track resource events
        if (msg.data && 'resourceId' in msg.data) {
          tracker.onResourceEvent(msg.data.resourceId, msg.message);
        }
      }
    }
  } as IIoHost
});

// Example usage with progress tracking
async function deployWithTracking(cloudAssemblySource: any) {
  try {
    // Synthesize the cloud assembly
    const cloudAssembly = await toolkit.synth(cloudAssemblySource);

    try {
      // Deploy using the cloud assembly
      await toolkit.deploy(cloudAssembly, {
        stacks: {
          strategy: StackSelectionStrategy.ALL_STACKS
```

```
        }
      });
    } finally {
      // Always dispose when done
      await cloudAssembly.dispose();
    }
  } catch (error) {
    // Display the error message
    console.error("Operation failed:", error.message);
    throw error;
  }
}
```

Handling errors with recovery

Implement robust error handling with recovery strategies:

```
import { Toolkit, ToolkitError, StackSelectionStrategy } from '@aws-cdk/toolkit-lib';

async function deployWithRetry(toolkit: Toolkit, cloudAssemblySource: any) {
  try {
    // Synthesize the cloud assembly
    const cloudAssembly = await toolkit.synth(cloudAssemblySource);

    try {
      // Deploy using the cloud assembly
      await toolkit.deploy(cloudAssembly, {
        stacks: {
          strategy: StackSelectionStrategy.ALL_STACKS
        }
      });
    } finally {
      // Always dispose when done
      await cloudAssembly.dispose();
    }
  } catch (error) {
    // Simply show the error to the user
    console.error("Operation failed:", error.message);
    throw error;
  }
}

// Example usage
```

```
try {
  await deployWithRetry(toolkit, cloudAssemblySource);
} catch (error) {
  console.error("Operation failed:", error.message);
  process.exit(1);
}
```

Integrating with CI/CD pipelines

Integrate the CDK Toolkit Library into a CI/CD pipeline:

```
import { Toolkit, StackSelectionStrategy, IIoHost } from '@aws-cdk/toolkit-lib';
import * as fs from 'fs';
import * as path from 'path';

async function cicdDeploy() {
  // Create a non-interactive toolkit for CI/CD environments
  const toolkit = new Toolkit({
    ioHost: {
      notify: async (msg) => {
        // Write to both console and log file
        const logMessage = `${msg.time} [${msg.level}] ${msg.message}`;
        console.log(logMessage);

        // Append to deployment log file
        fs.appendFileSync('deployment.log', logMessage + '\n');
      },
      requestResponse: async (msg) => {
        // Always use default responses in CI/CD
        console.log(`Auto-responding to: ${msg.message} with: ${msg.defaultResponse}`);
        return msg.defaultResponse;
      }
    } as IIoHost
  });

  // Determine environment from CI/CD variables
  const environment = process.env.DEPLOYMENT_ENV || 'development';

  // Load environment-specific parameters
  const paramsPath = path.join(process.cwd(), `params.${environment}.json`);
  const parameters = fs.existsSync(paramsPath)
    ? JSON.parse(fs.readFileSync(paramsPath, 'utf8'))
    : {};
```

```
  try {
    // Use pre-synthesized cloud assembly from build step
    const cloudAssemblySource = await toolkit.fromAssemblyDirectory('cdk.out');

    // Synthesize the cloud assembly
    const cloudAssembly = await toolkit.synth(cloudAssemblySource);

    try {
      // Deploy with CI/CD specific options
      const result = await toolkit.deploy(cloudAssembly, {
        stacks: { strategy: StackSelectionStrategy.ALL_STACKS },
        parameters,
        tags: {
          Environment: environment,
          BuildId: process.env.BUILD_ID || 'unknown',
          CommitHash: process.env.COMMIT_HASH || 'unknown'
        }
      });

      // Write outputs to a file for subsequent pipeline steps
      fs.writeFileSync(
        'stack-outputs.json',
        JSON.stringify(result.outputs, null, 2)
      );

      return result;
    } finally {
      // Always dispose when done
      await cloudAssembly.dispose();
    }
  } catch (error) {
    // Display the error message
    console.error("Operation failed:", error.message);
    process.exit(1);
  }
}

// Run the CI/CD deployment
cicdDeploy().then(() => {
  console.log('CI/CD deployment completed successfully');
});
```

Additional resources

For more detailed information on specific components used in these examples, refer to:

- Managing cloud assembly sources - Learn how to create and manage cloud assembly sources.
- Configuring messages and interactions - Detailed guide on customizing the `IIoHost` interface.

Test AWS CDK applications

With the AWS CDK, your infrastructure can be as testable as any other code you write. You can test in the cloud and locally. This topic addresses how to test in the cloud. For guidance on local testing see Locally test and build AWS CDK applications with the AWS SAM CLI. The standard approach to testing AWS CDK apps uses the AWS CDK's assertions module and popular test frameworks like Jest for TypeScript and JavaScript or Pytest for Python.

There are two categories of tests that you can write for AWS CDK apps.

- **Fine-grained assertions** test specific aspects of the generated AWS CloudFormation template, such as "this resource has this property with this value." These tests can detect regressions. They're also useful when you're developing new features using test-driven development. (You can write a test first, then make it pass by writing a correct implementation.) Fine-grained assertions are the most frequently used tests.
- **Snapshot tests** test the synthesized AWS CloudFormation template against a previously stored baseline template. Snapshot tests let you 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, CDK upgrades can also cause synthesized templates to change, so you can't rely only on snapshots to make sure that your implementation is correct.

Note

Complete versions of the TypeScript, Python, and Java apps used as examples in this topic are available on GitHub.

Getting started

To illustrate how to write these tests, we'll create a stack that contains an AWS Step Functions state machine and an AWS Lambda function. The Lambda function is subscribed to an Amazon SNS topic and simply forwards the message to the state machine.

First, create an empty CDK application project using the CDK Toolkit and installing the libraries we'll need. The constructs we'll use are all in the main CDK package, which is a default dependency in projects created with the CDK Toolkit. However, you must install your testing framework.

Example

TypeScript

```
$ mkdir state-machine && cd state-machine
cdk init --language=typescript
npm install --save-dev jest @types/jest
```

Create a directory for your tests.

```
$ mkdir test
```

Edit the project's `package.json` to tell NPM how to run Jest, and to tell Jest what kinds of files to collect. The necessary changes are as follows.

- Add a new `test` key to the `scripts` section
- Add Jest and its types to the `devDependencies` section
- Add a new `jest` top-level key with a `moduleFileExtensions` declaration

These changes are shown in the following outline. Place the new text where indicated in `package.json`. The "..." placeholders indicate existing parts of the file that should not be changed.

```
{
  ...
  "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "@types/jest": "^24.0.18",
    "jest": "^24.9.0"
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

JavaScript

```
$ mkdir state-machine && cd state-machine
$ cdk init --language=javascript
$ npm install --save-dev jest
```

Create a directory for your tests.

```
$ mkdir test
```

Edit the project's `package.json` to tell NPM how to run Jest, and to tell Jest what kinds of files to collect. The necessary changes are as follows.

- Add a new `test` key to the `scripts` section
- Add Jest to the `devDependencies` section
- Add a new `jest` top-level key with a `moduleFileExtensions` declaration

These changes are shown in the following outline. Place the new text where indicated in `package.json`. The "..." placeholders indicate existing parts of the file that shouldn't be changed.

```
{
  ...
  "scripts": {
    ...
    "test": "jest"
  },
  "devDependencies": {
    ...
    "jest": "^24.9.0"
  },
  "jest": {
    "moduleFileExtensions": ["js"]
  }
}
```

Python

```
$ mkdir state-machine && cd state-machine
```

```
$ cdk init --language=python
$ source .venv/bin/activate # On Windows, run '.\venv\Scripts\activate' instead
$ python -m pip install -r requirements.txt
$ python -m pip install -r requirements-dev.txt
```

Java

```
$ mkdir state-machine && cd-state-machine
$ cdk init --language=java
```

Open the project in your preferred Java IDE. (In Eclipse, use **File** > **Import** > Existing Maven Projects.)

C#

```
$ mkdir state-machine && cd-state-machine
$ cdk init --language=csharp
```

Open `src\StateMachine.sln` in Visual Studio.

Right-click the solution in Solution Explorer and choose **Add** > **New Project**. Search for MSTest C# and add an **MSTest Test Project** for C#. (The default name `TestProject1`is fine.)

Right-click `TestProject1` and choose **Add** > **Project Reference**, and add the `StateMachine` project as a reference.

The example stack

Here's the stack that will be tested in this topic. As we've previously described, it contains a Lambda function and a Step Functions state machine, and accepts one or more Amazon SNS topics. The Lambda function is subscribed to the Amazon SNS topics and forwards them to the state machine.

You don't have to do anything special to make the app testable. In fact, this CDK stack is not different in any important way from the other example stacks in this Guide.

Example

TypeScript

Place the following code in `lib/state-machine-stack.ts`:

```
import * as cdk from "aws-cdk-lib";
import * as sns from "aws-cdk-lib/aws-sns";
import * as sns_subscriptions from "aws-cdk-lib/aws-sns-subscriptions";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as sfn from "aws-cdk-lib/aws-stepfunctions";
import { Construct } from "constructs";

export interface StateMachineStackProps extends cdk.StackProps {
  readonly topics: sns.Topic[];
}

export class StateMachineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props: StateMachineStackProps) {
    super(scope, id, props);

    // In the future this state machine will do some work...
    const stateMachine = new sfn.StateMachine(this, "StateMachine", {
      definition: new sfn.Pass(this, "StartState"),
    });

    // This Lambda function starts the state machine.
    const func = new lambda.Function(this, "LambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "handler",
      code: lambda.Code.fromAsset("./start-state-machine"),
      environment: {
        STATE_MACHINE_ARN: stateMachine.stateMachineArn,
      },
    });
    stateMachine.grantStartExecution(func);

    const subscription = new sns_subscriptions.LambdaSubscription(func);
    for (const topic of props.topics) {
      topic.addSubscription(subscription);
    }
  }
}
```

JavaScript

Place the following code in `lib/state-machine-stack.js`:

```
const cdk = require("aws-cdk-lib");
```

```
const sns = require("aws-cdk-lib/aws-sns");
const sns_subscriptions = require("aws-cdk-lib/aws-sns-subscriptions");
const lambda = require("aws-cdk-lib/aws-lambda");
const sfn = require("aws-cdk-lib/aws-stepfunctions");

class StateMachineStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // In the future this state machine will do some work...
    const stateMachine = new sfn.StateMachine(this, "StateMachine", {
      definition: new sfn.Pass(this, "StartState"),
    });

    // This Lambda function starts the state machine.
    const func = new lambda.Function(this, "LambdaFunction", {
      runtime: lambda.Runtime.NODEJS_18_X,
      handler: "handler",
      code: lambda.Code.fromAsset("./start-state-machine"),
      environment: {
        STATE_MACHINE_ARN: stateMachine.stateMachineArn,
      },
    });
    stateMachine.grantStartExecution(func);

    const subscription = new sns_subscriptions.LambdaSubscription(func);
    for (const topic of props.topics) {
      topic.addSubscription(subscription);
    }
  }
}

module.exports = { StateMachineStack }
```

Python

Place the following code in `state_machine/state_machine_stack.py`:

```
from typing import List

import aws_cdk.aws_lambda as lambda_
import aws_cdk.aws_sns as sns
import aws_cdk.aws_sns_subscriptions as sns_subscriptions
import aws_cdk.aws_stepfunctions as sfn
```

```
import aws_cdk as cdk

class StateMachineStack(cdk.Stack):
    def __init__(
        self,
        scope: cdk.Construct,
        construct_id: str,
        *,
        topics: List[sns.Topic],
        **kwargs
    ) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # In the future this state machine will do some work...
        state_machine = sfn.StateMachine(
            self, "StateMachine", definition=sfn.Pass(self, "StartState")
        )

        # This Lambda function starts the state machine.
        func = lambda_.Function(
            self,
            "LambdaFunction",
            runtime=lambda_.Runtime.NODEJS_18_X,
            handler="handler",
            code=lambda_.Code.from_asset("./start-state-machine"),
            environment={
                "STATE_MACHINE_ARN": state_machine.state_machine_arn,
            },
        )
        state_machine.grant_start_execution(func)

        subscription = sns_subscriptions.LambdaSubscription(func)
        for topic in topics:
            topic.add_subscription(subscription)
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.lambda.Code;
```

```
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.sns.ITopicSubscription;
import software.amazon.awscdk.services.sns.Topic;
import software.amazon.awscdk.services.sns.subscriptions.LambdaSubscription;
import software.amazon.awscdk.services.stepfunctions.Pass;
import software.amazon.awscdk.services.stepfunctions.StateMachine;

import java.util.Collections;
import java.util.List;

public class StateMachineStack extends Stack {
    public StateMachineStack(final Construct scope, final String id, final
 List<Topic> topics) {
        this(scope, id, null, topics);
    }

    public StateMachineStack(final Construct scope, final String id, final
 StackProps props, final List<Topic> topics) {
        super(scope, id, props);

        // In the future this state machine will do some work...
        final StateMachine stateMachine = StateMachine.Builder.create(this,
 "StateMachine")
                .definition(new Pass(this, "StartState"))
                .build();

        // This Lambda function starts the state machine.
        final Function func = Function.Builder.create(this, "LambdaFunction")
                .runtime(Runtime.NODEJS_18_X)
                .handler("handler")
                .code(Code.fromAsset("./start-state-machine"))
                .environment(Collections.singletonMap("STATE_MACHINE_ARN",
 stateMachine.getStateMachineArn()))
                .build();
        stateMachine.grantStartExecution(func);

        final ITopicSubscription subscription = new LambdaSubscription(func);
        for (final Topic topic : topics) {
            topic.addSubscription(subscription);
        }
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.StepFunctions;
using Amazon.CDK.AWS.SNS;
using Amazon.CDK.AWS.SNS.Subscriptions;
using Constructs;

using System.Collections.Generic;

namespace AwsCdkAssertionSamples
{
    public class StateMachineStackProps : StackProps
    {
        public Topic[] Topics;
    }

    public class StateMachineStack : Stack
    {

        internal StateMachineStack(Construct scope, string id,
 StateMachineStackProps props = null) : base(scope, id, props)
        {
            // In the future this state machine will do some work...
            var stateMachine = new StateMachine(this, "StateMachine", new
 StateMachineProps
            {
                Definition = new Pass(this, "StartState")
            });

            // This Lambda function starts the state machine.
            var func = new Function(this, "LambdaFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_18_X,
                Handler = "handler",
                Code = Code.FromAsset("./start-state-machine"),
                Environment = new Dictionary<string, string>
                {
                    { "STATE_MACHINE_ARN", stateMachine.StateMachineArn }
                }
            });
            stateMachine.GrantStartExecution(func);
```

```
            foreach (Topic topic in props?.Topics ?? new Topic[0])
            {
                var subscription = new LambdaSubscription(func);
            }

        }
    }
}
```

We'll modify the app's main entry point so that we don't actually instantiate our stack. We don't want to accidentally deploy it. Our tests will create an app and an instance of the stack for testing. This is a useful tactic when combined with test-driven development: make sure that the stack passes all tests before you enable deployment.

Example

TypeScript

In `bin/state-machine.ts`:

```
#!/usr/bin/env node
import * as cdk from "aws-cdk-lib";

const app = new cdk.App();

// Stacks are intentionally not created here -- this application isn't meant to
// be deployed.
```

JavaScript

In `bin/state-machine.js`:

```
#!/usr/bin/env node
const cdk = require("aws-cdk-lib");

const app = new cdk.App();

// Stacks are intentionally not created here -- this application isn't meant to
// be deployed.
```

Python

In `app.py`:

```
#!/usr/bin/env python3
import os

import aws_cdk  as cdk

app = cdk.App()

# Stacks are intentionally not created here -- this application isn't meant to
# be deployed.

app.synth()
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import software.amazon.awscdk.App;

public class SampleApp {
    public static void main(final String[] args) {
        App app = new App();

        // Stacks are intentionally not created here -- this application isn't meant
 to be deployed.

        app.synth();
    }
}
```

C#

```
using Amazon.CDK;

namespace AwsCdkAssertionSamples
{
    sealed class Program
    {
        public static void Main(string[] args)
```

```
        {
            var app = new App();

            // Stacks are intentionally not created here -- this application isn't
 meant to be deployed.

            app.Synth();
        }
    }
}
```

The Lambda function

Our example stack includes a Lambda function that starts our state machine. We must provide the source code for this function so the CDK can bundle and deploy it as part of creating the Lambda function resource.

- Create the folder `start-state-machine` in the app's main directory.
- In this folder, create at least one file. For example, you can save the following code in `start-state-machines/index.js`.

  ```
  exports.handler = async function (event, context) {
   return 'hello world';
  };
  ```

  However, any file will work, since we won't actually be deploying the stack.

Running tests

For reference, here are the commands you use to run tests in your AWS CDK app. These are the same commands that you'd use to run the tests in any project using the same testing framework. For languages that require a build step, include that to make sure that your tests have compiled.

Example

TypeScript

```
$ tsc && npm test
```

JavaScript

```
$ npm test
```

Python

```
$ python -m pytest
```

Java

```
$ mvn compile && mvn test
```

C#

Build your solution (F6) to discover the tests, then run the tests (**Test** > **Run All Tests**). To choose which tests to run, open Test Explorer (**Test** > **Test Explorer**).

Or:

```
$ dotnet test src
```

Fine-grained assertions

The first step for testing a stack with fine-grained assertions is to synthesize the stack, because we're writing assertions against the generated AWS CloudFormation template.

Our `StateMachineStackStack` requires that we pass it the Amazon SNS topic to be forwarded to the state machine. So in our test, we'll create a separate stack to contain the topic.

Ordinarily, when writing a CDK app, you can subclass `Stack` and instantiate the Amazon SNS topic in the stack's constructor. In our test, we instantiate `Stack` directly, then pass this stack as the `Topic`'s scope, attaching it to the stack. This is functionally equivalent and less verbose. It also helps make stacks that are used only in tests "look different" from the stacks that you intend to deploy.

Example

TypeScript

```
import { Capture, Match, Template } from "aws-cdk-lib/assertions";
```

```
import * as cdk from "aws-cdk-lib";
import * as sns from "aws-cdk-lib/aws-sns";
import { StateMachineStack } from "../lib/state-machine-stack";

describe("StateMachineStack", () => {
  test("synthesizes the way we expect", () => {
    const app = new cdk.App();

    // Since the StateMachineStack consumes resources from a separate stack
    // (cross-stack references), we create a stack for our SNS topics to live
    // in here. These topics can then be passed to the StateMachineStack later,
    // creating a cross-stack reference.
    const topicsStack = new cdk.Stack(app, "TopicsStack");

    // Create the topic the stack we're testing will reference.
    const topics = [new sns.Topic(topicsStack, "Topic1", {})];

    // Create the StateMachineStack.
    const stateMachineStack = new StateMachineStack(app, "StateMachineStack", {
      topics: topics, // Cross-stack reference
    });

    // Prepare the stack for assertions.
    const template = Template.fromStack(stateMachineStack);
  })
})
```

JavaScript

```
const { Capture, Match, Template } = require("aws-cdk-lib/assertions");
const cdk = require("aws-cdk-lib");
const sns = require("aws-cdk-lib/aws-sns");
const { StateMachineStack } = require("../lib/state-machine-stack");

describe("StateMachineStack", () => {
  test("synthesizes the way we expect", () => {
    const app = new cdk.App();

    // Since the StateMachineStack consumes resources from a separate stack
    // (cross-stack references), we create a stack for our SNS topics to live
    // in here. These topics can then be passed to the StateMachineStack later,
    // creating a cross-stack reference.
    const topicsStack = new cdk.Stack(app, "TopicsStack");
```

```
    // Create the topic the stack we're testing will reference.
    const topics = [new sns.Topic(topicsStack, "Topic1", {})];

    // Create the StateMachineStack.
    const StateMachineStack = new StateMachineStack(app, "StateMachineStack", {
      topics: topics, // Cross-stack reference
    });

    // Prepare the stack for assertions.
    const template = Template.fromStack(stateMachineStack);
  })
})
```

Python

```
from aws_cdk import aws_sns as sns
import aws_cdk as cdk
from aws_cdk.assertions import Template

from app.state_machine_stack import StateMachineStack

def test_synthesizes_properly():
    app = cdk.App()

    # Since the StateMachineStack consumes resources from a separate stack
    # (cross-stack references), we create a stack for our SNS topics to live
    # in here. These topics can then be passed to the StateMachineStack later,
    # creating a cross-stack reference.
    topics_stack = cdk.Stack(app, "TopicsStack")

    # Create the topic the stack we're testing will reference.
    topics = [sns.Topic(topics_stack, "Topic1")]

    # Create the StateMachineStack.
    state_machine_stack = StateMachineStack(
        app, "StateMachineStack", topics=topics  # Cross-stack reference
    )

    # Prepare the stack for assertions.
    template = Template.from_stack(state_machine_stack)
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test;
import software.amazon.awscdk.assertions.Capture;
import software.amazon.awscdk.assertions.Match;
import software.amazon.awscdk.assertions.Template;
import software.amazon.awscdk.App;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.services.sns.Topic;

import java.util.*;

import static org.assertj.core.api.Assertions.assertThat;

public class StateMachineStackTest {
    @Test
    public void testSynthesizesProperly() {
        final App app = new App();

        // Since the StateMachineStack consumes resources from a separate stack
 (cross-stack references), we create a stack
        // for our SNS topics to live in here. These topics can then be passed to
 the StateMachineStack later, creating a
        // cross-stack reference.
        final Stack topicsStack = new Stack(app, "TopicsStack");

        // Create the topic the stack we're testing will reference.
        final List<Topic> topics =
 Collections.singletonList(Topic.Builder.create(topicsStack, "Topic1").build());

        // Create the StateMachineStack.
        final StateMachineStack stateMachineStack = new StateMachineStack(
                app,
                "StateMachineStack",
                topics // Cross-stack reference
        );

        // Prepare the stack for assertions.
        final Template template = Template.fromStack(stateMachineStack)
    }
}
```

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK;
using Amazon.CDK.AWS.SNS;
using Amazon.CDK.Assertions;
using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1
{
    [TestClass]
    public class StateMachineStackTest
    {
        [TestMethod]
        public void TestMethod1()
        {
            var app = new App();

            // Since the StateMachineStack consumes resources from a separate stack
 (cross-stack references), we create a stack
            // for our SNS topics to live in here. These topics can then be passed
 to the StateMachineStack later, creating a
            // cross-stack reference.
            var topicsStack = new Stack(app, "TopicsStack");

            // Create the topic the stack we're testing will reference.
            var topics = new Topic[] { new Topic(topicsStack, "Topic1") };

            // Create the StateMachineStack.
            var StateMachineStack = new StateMachineStack(app, "StateMachineStack",
 new StateMachineStackProps
            {
                Topics = topics
            });

            // Prepare the stack for assertions.
            var template = Template.FromStack(stateMachineStack);

            // test will go here
        }
```

```
    }
}
```

Now we can assert that the Lambda function and the Amazon SNS subscription were created.

Example

TypeScript

```
    // Assert it creates the function with the correct properties...
    template.hasResourceProperties("AWS::Lambda::Function", {
      Handler: "handler",
      Runtime: "nodejs14.x",
    });

    // Creates the subscription...
    template.resourceCountIs("AWS::SNS::Subscription", 1);
```

JavaScript

```
    // Assert it creates the function with the correct properties...
    template.hasResourceProperties("AWS::Lambda::Function", {
      Handler: "handler",
      Runtime: "nodejs14.x",
    });

    // Creates the subscription...
    template.resourceCountIs("AWS::SNS::Subscription", 1);
```

Python

```
# Assert that we have created the function with the correct properties
    template.has_resource_properties(
        "AWS::Lambda::Function",
        {
            "Handler": "handler",
            "Runtime": "nodejs14.x",
        },
    )

    # Assert that we have created a subscription
```

```
    template.resource_count_is("AWS::SNS::Subscription", 1)
```

Java

```
        // Assert it creates the function with the correct properties...
        template.hasResourceProperties("AWS::Lambda::Function", Map.of(
                "Handler", "handler",
                "Runtime", "nodejs14.x"
        ));

         // Creates the subscription...
        template.resourceCountIs("AWS::SNS::Subscription", 1);
```

C#

```
            // Prepare the stack for assertions.
            var template = Template.FromStack(stateMachineStack);

            // Assert it creates the function with the correct properties...
            template.HasResourceProperties("AWS::Lambda::Function", new StringDict {
                { "Handler", "handler"},
                { "Runtime", "nodejs14x" }
            });

            // Creates the subscription...
            template.ResourceCountIs("AWS::SNS::Subscription", 1);
```

Our Lambda function test asserts that two particular properties of the function resource have specific values. By default, the `hasResourceProperties` method performs a partial match on the resource's properties as given in the synthesized CloudFormation template. This test requires that the provided properties exist and have the specified values, but the resource can also have other properties, which are not tested.

Our Amazon SNS assertion asserts that the synthesized template contains a subscription, but nothing about the subscription itself. We included this assertion mainly to illustrate how to assert on resource counts. The `Template` class offers more specific methods to write assertions against the `Resources`, `Outputs`, and `Mapping` sections of the CloudFormation template.

Matchers

The default partial matching behavior of `hasResourceProperties` can be changed using *matchers* from the `Match` class.

Matchers range from lenient (`Match.anyValue`) to strict (`Match.objectEquals`). They can be nested to apply different matching methods to different parts of the resource properties. Using `Match.objectEquals` and `Match.anyValue` together, for example, we can test the state machine's IAM role more fully, while not requiring specific values for properties that may change.

Example

TypeScript

```
// Fully assert on the state machine's IAM role with matchers.
template.hasResourceProperties(
  "AWS::IAM::Role",
  Match.objectEquals({
    AssumeRolePolicyDocument: {
      Version: "2012-10-17",
      Statement: [
        {
          Action: "sts:AssumeRole",
          Effect: "Allow",
          Principal: {
            Service: {
              "Fn::Join": [
                "",
                ["states.", Match.anyValue(), ".amazonaws.com"],
              ],
            },
          },
        },
      ],
    },
  })
);
```

JavaScript

```
// Fully assert on the state machine's IAM role with matchers.
template.hasResourceProperties(
```

```
      "AWS::IAM::Role",
      Match.objectEquals({
        AssumeRolePolicyDocument: {
          Version: "2012-10-17",
          Statement: [
            {
              Action: "sts:AssumeRole",
              Effect: "Allow",
              Principal: {
                Service: {
                  "Fn::Join": [
                    "",
                    ["states.", Match.anyValue(), ".amazonaws.com"],
                  ],
                },
              },
            },
          ],
        },
      })
    );
```

Python

```
from aws_cdk.assertions import Match

    # Fully assert on the state machine's IAM role with matchers.
    template.has_resource_properties(
        "AWS::IAM::Role",
        Match.object_equals(
            {
                "AssumeRolePolicyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [
                        {
                            "Action": "sts:AssumeRole",
                            "Effect": "Allow",
                            "Principal": {
                                "Service": {
                                    "Fn::Join": [
                                        "",
                                        [
                                            "states.",
```

```
                                                Match.any_value(),
                                                ".amazonaws.com",
                                            ],
                                        ],
                                    },
                                },
                            },
                        ],
                    },
                }
            ),
        )
```

Java

```
        // Fully assert on the state machine's IAM role with matchers.
        template.hasResourceProperties("AWS::IAM::Role", Match.objectEquals(
                Collections.singletonMap("AssumeRolePolicyDocument", Map.of(
                        "Version", "2012-10-17",
                        "Statement", Collections.singletonList(Map.of(
                                "Action", "sts:AssumeRole",
                                "Effect", "Allow",
                                "Principal", Collections.singletonMap(
                                        "Service", Collections.singletonMap(
                                                "Fn::Join", Arrays.asList(
                                                        "",
                                                        Arrays.asList("states.",
Match.anyValue(), ".amazonaws.com")
                                                )
                                        )
                                )
                        ))
                ))
        ));
```

C#

```
            // Fully assert on the state machine's IAM role with matchers.
            template.HasResource("AWS::IAM::Role", Match.ObjectEquals(new
ObjectDict
            {
                { "AssumeRolePolicyDocument", new ObjectDict
                    {
```

```
                        { "Version", "2012-10-17" },
                        { "Action", "sts:AssumeRole" },
                        { "Principal", new ObjectDict
                            {
                                { "Version", "2012-10-17" },
                                { "Statement", new object[]
                                    {
                                        new ObjectDict {
                                            { "Action", "sts:AssumeRole" },
                                            { "Effect", "Allow" },
                                            { "Principal", new ObjectDict
                                                {
                                                    { "Service", new ObjectDict
                                                        {
                                                            { "", new object[]
                                                                { "states",
Match.AnyValue(), ".amazonaws.com" }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }));
```

Many CloudFormation resources include serialized JSON objects represented as strings. The `Match.serializedJson()` matcher can be used to match properties inside this JSON.

For example, Step Functions state machines are defined using a string in the JSON-based Amazon States Language. We'll use `Match.serializedJson()` to make sure that our initial state is the only step. Again, we'll use nested matchers to apply different kinds of matching to different parts of the object.

Example

TypeScript

```
// Assert on the state machine's definition with the Match.serializedJson()
// matcher.
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    // Match.objectEquals() is used implicitly, but we use it explicitly
    // here for extra clarity.
    Match.objectEquals({
      StartAt: "StartState",
      States: {
        StartState: {
          Type: "Pass",
          End: true,
          // Make sure this state doesn't provide a next state -- we can't
          // provide both Next and set End to true.
          Next: Match.absent(),
        },
      },
    })
  ),
});
```

JavaScript

```
// Assert on the state machine's definition with the Match.serializedJson()
// matcher.
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    // Match.objectEquals() is used implicitly, but we use it explicitly
    // here for extra clarity.
    Match.objectEquals({
      StartAt: "StartState",
      States: {
        StartState: {
          Type: "Pass",
          End: true,
          // Make sure this state doesn't provide a next state -- we can't
          // provide both Next and set End to true.
          Next: Match.absent(),
        },
      },
```

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

Python

```
  # Assert on the state machine's definition with the serialized_json matcher.
  template.has_resource_properties(
      "AWS::StepFunctions::StateMachine",
      {
          "DefinitionString": Match.serialized_json(
              # Match.object_equals() is the default, but specify it here for
clarity
              Match.object_equals(
                  {
                      "StartAt": "StartState",
                      "States": {
                          "StartState": {
                              "Type": "Pass",
                              "End": True,
                              # Make sure this state doesn't provide a next
state --
                              # we can't provide both Next and set End to true.
                              "Next": Match.absent(),
                          },
                      },
                  }
              )
          ),
      },
  )
```

Java

```
        // Assert on the state machine's definition with the
Match.serializedJson() matcher.
        template.hasResourceProperties("AWS::StepFunctions::StateMachine",
Collections.singletonMap(
                "DefinitionString", Match.serializedJson(
                        // Match.objectEquals() is used implicitly, but we use it
explicitly here for extra clarity.
                        Match.objectEquals(Map.of(
                                "StartAt", "StartState",
```

```
                              "States", Collections.singletonMap(
                                      "StartState", Map.of(
                                              "Type", "Pass",
                                              "End", true,
                                              // Make sure this state doesn't
provide a next state -- we can't provide
                                              // both Next and set End to true.
                                              "Next", Match.absent()
                                      )
                              )
                      ))
              )
      ));
```

C#

```
           // Assert on the state machine's definition with the
Match.serializedJson() matcher
           template.HasResourceProperties("AWS::StepFunctions::StateMachine",
new ObjectDict
           {
               { "DefinitionString", Match.SerializedJson(
                   // Match.objectEquals() is used implicitly, but we use it
explicitly here for extra clarity.
                   Match.ObjectEquals(new ObjectDict {
                       { "StartAt", "StartState" },
                       { "States", new ObjectDict
                       {
                           { "StartState", new ObjectDict {
                               { "Type", "Pass" },
                               { "End", "True" },
                               // Make sure this state doesn't provide a next
state -- we can't provide
                               // both Next and set End to true.
                               { "Next", Match.Absent() }
                           }}
                       }}
                   })
               )}});
```

Capturing

It's often useful to test properties to make sure they follow specific formats, or have the same value as another property, without needing to know their exact values ahead of time. The `assertions` module provides this capability in its [Capture](https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.assertions.Capture.html) class.

By specifying a `Capture` instance in place of a value in `hasResourceProperties`, that value is retained in the `Capture` object. The actual captured value can be retrieved using the object's `as` methods, including `asNumber()`, `asString()`, and `asObject`, and subjected to test. Use `Capture` with a matcher to specify the exact location of the value to be captured within the resource's properties, including serialized JSON properties.

The following example tests to make sure that the starting state of our state machine has a name beginning with `Start`. It also tests that this state is present within the list of states in the machine.

Example

TypeScript

```
// Capture some data from the state machine's definition.
const startAtCapture = new Capture();
const statesCapture = new Capture();
template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
  DefinitionString: Match.serializedJson(
    Match.objectLike({
      StartAt: startAtCapture,
      States: statesCapture,
    })
  ),
});

// Assert that the start state starts with "Start".
expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

// Assert that the start state actually exists in the states object of the
// state machine definition.
expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString());
```

JavaScript

```
// Capture some data from the state machine's definition.
```

```
    const startAtCapture = new Capture();
    const statesCapture = new Capture();
    template.hasResourceProperties("AWS::StepFunctions::StateMachine", {
      DefinitionString: Match.serializedJson(
        Match.objectLike({
          StartAt: startAtCapture,
          States: statesCapture,
        })
      ),
    });

    // Assert that the start state starts with "Start".
    expect(startAtCapture.asString()).toEqual(expect.stringMatching(/^Start/));

    // Assert that the start state actually exists in the states object of the
    // state machine definition.
    expect(statesCapture.asObject()).toHaveProperty(startAtCapture.asString());
```

Python

```
import re

    from aws_cdk.assertions import Capture

    # ...

    # Capture some data from the state machine's definition.
    start_at_capture = Capture()
    states_capture = Capture()
    template.has_resource_properties(
        "AWS::StepFunctions::StateMachine",
        {
            "DefinitionString": Match.serialized_json(
                Match.object_like(
                    {
                        "StartAt": start_at_capture,
                        "States": states_capture,
                    }
                )
            ),
        },
    )
```

```
    # Assert that the start state starts with "Start".
    assert re.match("^Start", start_at_capture.as_string())

    # Assert that the start state actually exists in the states object of the
    # state machine definition.
    assert start_at_capture.as_string() in states_capture.as_object()
```

Java

```
        // Capture some data from the state machine's definition.
        final Capture startAtCapture = new Capture();
        final Capture statesCapture = new Capture();
        template.hasResourceProperties("AWS::StepFunctions::StateMachine",
Collections.singletonMap(
                "DefinitionString", Match.serializedJson(
                        Match.objectLike(Map.of(
                                "StartAt", startAtCapture,
                                "States", statesCapture
                        ))
                )
        ));

        // Assert that the start state starts with "Start".
        assertThat(startAtCapture.asString()).matches("^Start.+");

        // Assert that the start state actually exists in the states object of
the state machine definition.

assertThat(statesCapture.asObject()).containsKey(startAtCapture.asString());
```

C#

```
            // Capture some data from the state machine's definition.
            var startAtCapture = new Capture();
            var statesCapture = new Capture();
            template.HasResourceProperties("AWS::StepFunctions::StateMachine",
new ObjectDict
            {
                { "DefinitionString", Match.SerializedJson(
                    new ObjectDict
                    {
                        { "StartAt", startAtCapture },
                        { "States", statesCapture }
```

```
                    }
                )}
            });

            Assert.IsTrue(startAtCapture.ToString().StartsWith("Start"));

Assert.IsTrue(statesCapture.AsObject().ContainsKey(startAtCapture.ToString()));
```

Snapshot tests

In *snapshot testing*, you compare the entire synthesized CloudFormation template against a previously stored baseline (often called a "master") template. Unlike fine-grained assertions, snapshot testing isn't useful in catching regressions. This is because snapshot testing applies to the entire template, and things besides code changes can cause small (or not-so-small) differences in synthesis results. These changes may not even affect your deployment, but they will still cause a snapshot test to fail.

For example, you might update a CDK construct to incorporate a new best practice, which can cause changes to the synthesized resources or how they're organized. Alternatively, you might update the CDK Toolkit to a version that reports additional metadata. Changes to context values can also affect the synthesized template.

Snapshot tests can be of great help in refactoring, though, as long as you hold constant all other factors that might affect the synthesized template. You will know immediately if a change you made has unintentionally changed the template. If the change is intentional, simply accept the new template as the baseline.

For example, if we have this `DeadLetterQueue` construct:

Example

TypeScript

```
export class DeadLetterQueue extends sqs.Queue {
  public readonly messagesInQueueAlarm: cloudwatch.IAlarm;

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

    // Add the alarm
```

```
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
      alarmDescription: 'There are messages in the Dead Letter Queue',
      evaluationPeriods: 1,
      threshold: 1,
      metric: this.metricApproximateNumberOfMessagesVisible(),
    });
  }
}
```

JavaScript

```
class DeadLetterQueue extends sqs.Queue {

  constructor(scope, id) {
    super(scope, id);

    // Add the alarm
    this.messagesInQueueAlarm = new cloudwatch.Alarm(this, 'Alarm', {
      alarmDescription: 'There are messages in the Dead Letter Queue',
      evaluationPeriods: 1,
      threshold: 1,
      metric: this.metricApproximateNumberOfMessagesVisible(),
    });
  }
}

module.exports = { DeadLetterQueue }
```

Python

```
class DeadLetterQueue(sqs.Queue):
    def __init__(self, scope: Construct, id: str):
        super().__init__(scope, id)

        self.messages_in_queue_alarm = cloudwatch.Alarm(
            self,
            "Alarm",
            alarm_description="There are messages in the Dead Letter Queue.",
            evaluation_periods=1,
            threshold=1,
            metric=self.metric_approximate_number_of_messages_visible(),
        )
```

Java

```
public class DeadLetterQueue extends Queue {
    private final IAlarm messagesInQueueAlarm;

    public DeadLetterQueue(@NotNull Construct scope, @NotNull String id) {
        super(scope, id);

        this.messagesInQueueAlarm = Alarm.Builder.create(this, "Alarm")
                .alarmDescription("There are messages in the Dead Letter Queue.")
                .evaluationPeriods(1)
                .threshold(1)
                .metric(this.metricApproximateNumberOfMessagesVisible())
                .build();
    }

    public IAlarm getMessagesInQueueAlarm() {
        return messagesInQueueAlarm;
    }
}
```

C#

```
namespace AwsCdkAssertionSamples
{
    public class DeadLetterQueue : Queue
    {
        public IAlarm messagesInQueueAlarm;

        public DeadLetterQueue(Construct scope, string id) : base(scope, id)
        {
            messagesInQueueAlarm = new Alarm(this, "Alarm", new AlarmProps
            {
                AlarmDescription = "There are messages in the Dead Letter Queue.",
                EvaluationPeriods = 1,
                Threshold = 1,
                Metric = this.MetricApproximateNumberOfMessagesVisible()
            });
        }
    }
}
```

We can test it like this:

Example

TypeScript

```
import { Match, Template } from "aws-cdk-lib/assertions";
import * as cdk from "aws-cdk-lib";
import { DeadLetterQueue } from "../lib/dead-letter-queue";

describe("DeadLetterQueue", () => {
  test("matches the snapshot", () => {
    const stack = new cdk.Stack();
    new DeadLetterQueue(stack, "DeadLetterQueue");

    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
});
```

JavaScript

```
const { Match, Template } = require("aws-cdk-lib/assertions");
const cdk = require("aws-cdk-lib");
const { DeadLetterQueue } = require("../lib/dead-letter-queue");

describe("DeadLetterQueue", () => {
  test("matches the snapshot", () => {
    const stack = new cdk.Stack();
    new DeadLetterQueue(stack, "DeadLetterQueue");

    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
});
```

Python

```
import aws_cdk_lib as cdk
from aws_cdk_lib.assertions import Match, Template

from app.dead_letter_queue import DeadLetterQueue
```

```
def snapshot_test():
    stack = cdk.Stack()
    DeadLetterQueue(stack, "DeadLetterQueue")

    template = Template.from_stack(stack)
    assert template.to_json() == snapshot
```

Java

```
package software.amazon.samples.awscdkassertionssamples;

import org.junit.jupiter.api.Test;
import au.com.origin.snapshots.Expect;
import software.amazon.awscdk.assertions.Match;
import software.amazon.awscdk.assertions.Template;
import software.amazon.awscdk.Stack;

import java.util.Collections;
import java.util.Map;

public class DeadLetterQueueTest {
    @Test
    public void snapshotTest() {
        final Stack stack = new Stack();
        new DeadLetterQueue(stack, "DeadLetterQueue");

        final Template template = Template.fromStack(stack);
        expect.toMatchSnapshot(template.toJSON());
    }
}
```

C#

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

using Amazon.CDK;
using Amazon.CDK.Assertions;
using AwsCdkAssertionSamples;

using ObjectDict = System.Collections.Generic.Dictionary<string, object>;
using StringDict = System.Collections.Generic.Dictionary<string, string>;

namespace TestProject1
```

```
{
    [TestClass]
    public class StateMachineStackTest

    [TestClass]
    public class DeadLetterQueueTest
    {
    [TestMethod]
        public void SnapshotTest()
        {
            var stack = new Stack();
            new DeadLetterQueue(stack, "DeadLetterQueue");

            var template = Template.FromStack(stack);

            return Verifier.Verify(template.ToJSON());
        }
    }
}
```

Tips for tests

Remember, your tests will live just as long as the code they test, and they will be read and modified just as often. Therefore, it pays to take a moment to consider how best to write them.

Don’t copy and paste setup lines or common assertions. Instead, refactor this logic into fixtures or helper functions. Use good names that reflect what each test actually tests.

Don’t try to do too much in one test. Preferably, a test should test one and only one behavior. If you accidentally break that behavior, exactly one test should fail, and the name of the test should tell you what failed. This is more an ideal to be striven for, however; sometimes you will unavoidably (or inadvertently) write tests that test more than one behavior. Snapshot tests are, for reasons we’ve already described, especially prone to this problem, so use them sparingly.

Locally test and build AWS CDK applications with the AWS SAM CLI

You can use the AWS SAM CLI to locally test and build serverless applications defined using the AWS Cloud Development Kit (AWS CDK). Because the AWS SAM CLI works within the AWS

CDK project structure, you can still use the AWS CDK CLI reference for creating, modifying, and deploying your AWS CDK applications.

For details on using AWS SAM, see Getting started with AWS SAM in the *AWS Serverless Application Model Developer Guide*.

Topics

Getting started with locally testing

This topic describes what you need to use the AWS SAM CLI with AWS CDK applications, and it provides instructions for building and locally testing a simple AWS CDK application.

Prerequisites

To test locally, you must install the AWS SAM CLI. see Install the AWS SAM CLI for installation instructions.

Creating and locally testing an AWS CDK application

To locally test an AWS CDK application using the AWS SAM CLI, you must have an AWS CDK application that contains a Lambda function. Use the following steps to create a basic AWS CDK application with a Lambda function. For more information, see Creating a serverless application using the AWS CDK in the *AWS Cloud Development Kit (AWS CDK) Developer Guide*.

Step 1: Create an AWS CDK application

For this tutorial, initialize an AWS CDK application that uses TypeScript.

Command to run:

```
$ mkdir cdk-sam-example
$ cd cdk-sam-example
$ cdk init app --language typescript
```

Step 2: Add a Lambda function to your application

Replace the code in `lib/cdk-sam-example-stack.ts` with the following:

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as lambda from 'aws-cdk-lib/aws-lambda';

export class CdkSamExampleStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    new lambda.Function(this, 'MyFunction', {
      runtime: lambda.Runtime.PYTHON_3_12,
      handler: 'app.lambda_handler',
      code: lambda.Code.fromAsset('./my_function'),
    });
  }
}
```

Step 3: Add your Lambda function code

Create a directory named `my_function`. In that directory, create a file named `app.py`.

Command to run:

Example

macOS / Linux

```
$ mkdir my_function
$ cd my_function
$ touch app.py
```

Windows

```
$ mkdir my_function
$ cd my_function
$ type nul > app.py
```

PowerShell

```
$ mkdir my_function
```

```
$ cd my_function
$ New-Item -Path "app.py”
```

Add the following code to `app.py`:

```
def lambda_handler(event, context):
    return "Hello from SAM and the CDK!"
```

Step 4: Test your Lambda function

You can use the AWS SAM CLI to locally invoke a Lambda function that you define in an AWS CDK application. To do this, you need the function construct identifier and the path to your synthesized AWS CloudFormation template.

Run the following command to go back to the `lib` directory:

```
$  cd ..
```

Command to run:

```
$  cdk synth --no-staging
```

```
$  sam local invoke MyFunction --no-event -t ./cdk.out/
CdkSamExampleStack.template.json
```

Example output:

```
Invoking app.lambda_handler (python3.9)

START RequestId: 5434c093-7182-4012-9b06-635011cac4f2 Version: $LATEST
"Hello from SAM and the CDK!"
END RequestId: 5434c093-7182-4012-9b06-635011cac4f2
REPORT RequestId: 5434c093-7182-4012-9b06-635011cac4f2 Init Duration: 0.32 ms
 Duration: 177.47 ms Billed Duration: 178 ms Memory Size: 128 MB Max Memory Used:
 128 MB
```

Local testing AWS CDK applications with AWS SAM

You can use the AWS SAM CLI to locally test your AWS CDK applications by running the following commands from the project root directory of your AWS CDK application:

- `sam local invoke`
- `sam local start-api`
- `sam local start-lambda`

Before you run any of the `sam local` commands with a AWS CDK application, you must run `cdk synth`.

When running `sam local invoke` you need the function construct identifier that you want to invoke, and the path to your synthesized AWS CloudFormation template. If your application uses nested stacks, to resolve naming conflicts, you also need the stack name where the function is defined.

Usage

```
# Invoke the function FUNCTION_IDENTIFIER declared in the stack STACK_NAME
$  sam local invoke <OPTIONS> <STACK_NAME/FUNCTION_IDENTIFIER>

# Start all APIs declared in the AWS CDK application
$  sam local start-api -t <./cdk.out/CdkSamExampleStack.template.json> <OPTIONS>

# Start a local endpoint that emulates AWS Lambda
$  sam local start-lambda -t <./cdk.out/CdkSamExampleStack.template.json> <OPTIONS>
```

Example

Consider stacks and functions that are declared with the following sample:

```
app = new HelloCdkStack(app, "HelloCdkStack",
   ...
)
class HelloCdkStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    ...
    new lambda.Function(this, 'MyFunction', {
```

```
        ...
    });

    new HelloCdkNestedStack(this, 'HelloNestedStack' ,{
        ...
    });
  }
}

class HelloCdkNestedStack extends cdk.NestedStack {
  constructor(scope: Construct, id: string, props?: cdk.NestedStackProps) {
    ...
    new lambda.Function(this, 'MyFunction', {
        ...
    });
    new lambda.Function(this, 'MyNestedFunction', {
        ...
    });
  }
}
```

The following commands locally invokes the Lambda functions defined in example presented above:

```
# Invoke MyFunction from the HelloCdkStack
$ sam local invoke -t <./cdk.out/HelloCdkStack.template.json> <MyFunction>
```

```
# Invoke MyNestedFunction from the HelloCdkNestedStack
$ sam local invoke -t <./cdk.out/HelloCdkStack.template.json> <MyNestedFunction>
```

```
# Invoke MyFunction from the HelloCdkNestedStack
$ sam local invoke -t <./cdk.out/HelloCdkStack.template.json> <HelloNestedStack/
MyFunction>
```

Building AWS CDK applications with AWS SAM

The AWS SAM CLI provides support for building Lambda functions and layers defined in your AWS CDK application with `sam build`.

For Lambda functions that use zip artifacts, run `cdk synth` before you run `sam local` commands. `sam build` isn't required.

If your AWS CDK application uses functions with the image type, run `cdk synth` and then run `sam build` before you run `sam local` commands. When you run `sam build`, AWS SAM doesn't build Lambda functions or layers that use runtime-specific constructs, for example, [`NodejsFunction`](). `sam build` doesn't support [bundled assets]().

Example

Running the following command from the AWS CDK project root directory builds the application.

```
$ sam build -t <./cdk.out/CdkSamExampleStack.template.json>
```

AWS CDK CLI reference

The AWS Cloud Development Kit (AWS CDK) Command Line Interface (AWS CDK CLI), also known as the CDK Toolkit, is the primary tool for interacting with your AWS CDK app. It executes your app, interrogates the application model you defined, and produces and deploys the AWS CloudFormation templates generated by the AWS CDK. It also provides other features useful for creating and working with AWS CDK projects. This topic contains information about common use cases of the CDK CLI.

The CDK CLI is installed with the Node Package Manager. In most cases, we recommend installing it globally.

```
npm install -g aws-cdk             # install latest version
npm install -g aws-cdk@X.YY.Z      # install specific version
```

Tip

If you regularly work with multiple versions of the AWS CDK, consider installing a matching version of the CDK CLI in individual CDK projects. To do this, omit `-g` from the `npm install` command. Then use `npx aws-cdk` to invoke it. This runs the local version if one exists, falling back to a global version if not.

CDK CLI commands

All CDK CLI commands start with `cdk`, which is followed by a subcommand (`list`, `synthesize`, `deploy`, etc.). Some subcommands have a shorter version (`ls`, `synth`, etc.) that is equivalent. Options and arguments follow the subcommand in any order.

For a description of all subcommands, options, and arguments, see AWS CDK CLI command reference.

Specify options and their values

Command line options begin with two hyphens (`--`). Some frequently used options have single-letter synonyms that begin with a single hyphen (for example, `--app` has a synonym `-a`). The order of options in an CDK CLI command is not important.

All options accept a value, which must follow the option name. The value may be separated from the name by white space or by an equals sign =. The following two options are equivalent.

```
--toolkit-stack-name MyBootstrapStack
--toolkit-stack-name=MyBootstrapStack
```

Some options are flags (Booleans). You may specify `true` or `false` as their value. If you do not provide a value, the value is taken to be `true`. You may also prefix the option name with `no-` to imply `false`.

```
# sets staging flag to true
--staging
--staging=true
--staging true

# sets staging flag to false
--no-staging
--staging=false
--staging false
```

A few options, namely `--context`, `--parameters`, `--plugin`, `--tags`, and `--trust`, may be specified more than once to specify multiple values. These are noted as having `[array]` type in the CDK CLI help. For example:

```
cdk bootstrap --tags costCenter=0123 --tags responsibleParty=jdoe
```

Built-in help

The CDK CLI has integrated help. You can see general help about the utility and a list of the provided subcommands by issuing:

```
cdk --help
```

To see help for a particular subcommand, for example `deploy`, specify it before the `--help` flag.

```
cdk deploy --help
```

Issue `cdk version` to display the version of the CDK CLI. Provide this information when requesting support.

Version reporting

To gain insight into how the AWS CDK is used, the constructs used by AWS CDK applications are collected and reported by using a resource identified as `AWS::CDK::Metadata`. To learn more, see Configure AWS CDK usage data reporting.

Authentication with AWS

There are different ways in which you can configure programmatic access to AWS resources, depending on the environment and the AWS access available to you.

To choose your method of authentication and configure it for the CDK CLI, see Configure security credentials for the AWS CDK CLI.

The recommended approach for new users developing locally, who are not given a method of authentication by their employer, is to set up AWS IAM Identity Center. This method includes installing the AWS CLI for ease of configuration and for regularly signing in to the AWS access portal. If you choose this method, your environment should contain the following elements after you complete the procedure for IAM Identity Center authentication in the *AWS SDKs and Tools Reference Guide*:

- The AWS CLI, which you use to start an AWS access portal session before you run your application.
- A shared AWS config file having a `[default]` profile with a set of configuration values that can be referenced from the AWS CDK. To find the location of this file, see Location of the shared files in the *AWS SDKs and Tools Reference Guide*.
- The shared `config` file sets the region setting. This sets the default AWS Region the AWS CDK and CDK CLI use for AWS requests.
- The CDK CLI uses the profile's SSO token provider configuration to acquire credentials before sending requests to AWS. The `sso_role_name` value, which is an IAM role connected to an IAM Identity Center permission set, should allow access to the AWS services used in your application.

  The following sample `config` file shows a default profile set up with SSO token provider configuration. The profile's `sso_session` setting refers to the named `sso-session` section. The `sso-session` section contains settings to initiate an AWS access portal session.

  ```
  [default]
  ```

```
sso_session = <my-sso>
sso_account_id = <111122223333>
sso_role_name = <SampleRole>
region = <us-east-1>
output = <json>

[sso-session <my-sso>]
sso_region = <us-east-1>
sso_start_url = <https://provided-domain.awsapps.com/start>
sso_registration_scopes = sso:account:access
```

Start an AWS access portal session

Before accessing AWS services, you need an active AWS access portal session for the CDK CLI to use IAM Identity Center authentication to resolve credentials. Depending on your configured session lengths, your access will eventually expire and the CDK CLI will encounter an authentication error. Run the following command in the AWS CLI to sign in to the AWS access portal.

```
aws sso login
```

If your SSO token provider configuration is using a named profile instead of the default profile, the command is `aws sso login --profile <NAME>`. Also specify this profile when issuing `cdk` commands using the `--profile` option or the `AWS_PROFILE` environment variable.

To test if you already have an active session, run the following AWS CLI command.

```
aws sts get-caller-identity
```

The response to this command should report the IAM Identity Center account and permission set configured in the shared `config` file.

Note

If you already have an active AWS access portal session and run `aws sso login`, you will not be required to provide credentials.
The sign in process may prompt you to allow the AWS CLI access to your data. Since the AWS CLI is built on top of the SDK for Python, permission messages may contain variations of the `botocore` name.

Specify Region and other configuration

The CDK CLI needs to know the AWS Region that you're deploying into and how to authenticate with AWS. This is needed for deployment operations and to retrieve context values during synthesis. Together, your account and Region make up the *environment*.

Region may be specified using environment variables or in configuration files. These are the same variables and files used by other AWS tools such as the AWS CLI and the various AWS SDKs. The CDK CLI looks for this information in the following order.

- The `AWS_DEFAULT_REGION` environment variable.
- A named profile defined in the standard AWS `config` file and specified using the `--profile` option on `cdk` commands.
- The `[default]` section of the standard AWS `config` file.

Besides specifying AWS authentication and a Region in the `[default]` section, you can also add one or more `[profile <NAME>]` sections, where `<NAME>` is the name of the profile. For more information about named profiles, see Shared config and credentials files in the *AWS SDKs and Tools Reference Guide*.

The standard AWS `config` file is located at `~/.aws/config` (macOS/Linux) or `%USERPROFILE%\.aws\config` (Windows). For details and alternate locations, see Location of the shared config and credentials files in the *AWS SDKs and Tools Reference Guide*

The environment that you specify in your AWS CDK app by using the stack's `env` property is used during synthesis. It's used to generate an environment-specific AWS CloudFormation template, and during deployment, it overrides the account or Region specified by one of the preceding methods. For more information, see Environments for the AWS CDK.

Note

The AWS CDK uses credentials from the same source files as other AWS tools and SDKs, including the AWS Command Line Interface. However, the AWS CDK might behave somewhat differently from these tools. It uses the AWS SDK for JavaScript under the hood. For complete details on setting up credentials for the AWS SDK for JavaScript, see Setting credentials.

You may optionally use the `--role-arn` (or `-r`) option to specify the ARN of an IAM role that should be used for deployment. This role must be assumable by the AWS account being used.

Specify the app command

Many features of the CDK CLI require one or more AWS CloudFormation templates be synthesized, which in turn requires running your application. The AWS CDK supports programs written in a variety of languages. Therefore, it uses a configuration option to specify the exact command necessary to run your app. This option can be specified in two ways.

First, and most commonly, it can be specified using the `app` key inside the file `cdk.json`. This is in the main directory of your AWS CDK project. The CDK CLI provides an appropriate command when creating a new project with `cdk init`. Here is the `cdk.json` from a fresh TypeScript project, for instance.

```
{
  "app": "npx ts-node bin/hello-cdk.ts"
}
```

The CDK CLI looks for `cdk.json` in the current working directory when attempting to run your app. Because of this, you might keep a shell open in your project's main directory for issuing CDK CLI commands.

The CDK CLI also looks for the app key in `~/.cdk.json` (that is, in your home directory) if it can't find it in `./cdk.json`. Adding the app command here can be useful if you usually work with CDK code in the same language.

If you are in some other directory, or to run your app using a command other than the one in `cdk.json`, use the `--app` (or `-a`) option to specify it.

```
cdk --app "npx ts-node bin/hello-cdk.ts" ls
```

When deploying, you may also specify a directory containing synthesized cloud assemblies, such as `cdk.out`, as the value of `--app`. The specified stacks are deployed from this directory; the app is not synthesized.

Specify stacks

Many CDK CLI commands (for example, `cdk deploy`) work on stacks defined in your app. If your app contains only one stack, the CDK CLI assumes you mean that one if you don't specify a stack explicitly.

Otherwise, you must specify the stack or stacks you want to work with. You can do this by specifying the desired stacks by ID individually on the command line. Recall that the ID is the value specified by the second argument when you instantiate the stack.

```
cdk synth PipelineStack LambdaStack
```

You may also use wildcards to specify IDs that match a pattern.

- ? matches any single character
- * matches any number of characters (* alone matches all stacks)
- ** matches everything in a hierarchy

You may also use the `--all` option to specify all stacks.

If your app uses CDK Pipelines, the CDK CLI understands your stacks and stages as a hierarchy. Also, the `--all` option and the * wildcard only match top-level stacks. To match all the stacks, use **. Also use ** to indicate all the stacks under a particular hierarchy.

When using wildcards, enclose the pattern in quotes, or escape the wildcards with \. If you don't, your shell may try to expand the pattern to the names of files in the current directory. At best, this won't do what you expect; at worst, you could deploy stacks you didn't intend to. This isn't strictly necessary on Windows because `cmd.exe` does not expand wildcards, but is good practice nonetheless.

```
cdk synth "*Stack"    # PipelineStack, LambdaStack, etc.
cdk synth 'Stack?'    # StackA, StackB, Stack1, etc.
cdk synth \*          # All stacks in the app, or all top-level stacks in a CDK
 Pipelines app
cdk synth '**'        # All stacks in a CDK Pipelines app
cdk synth 'PipelineStack/Prod/**'   # All stacks in Prod stage in a CDK Pipelines app
```

Note

The order in which you specify the stacks is not necessarily the order in which they will be processed. The CDK CLI accounts for dependencies between stacks when deciding the order in which to process them. For example, let's say that one stack uses a value produced by another (such as the ARN of a resource defined in the second stack). In this case, the second stack is synthesized before the first one because of this dependency. You can add dependencies between stacks manually using the stack's `addDependency()` method.

Bootstrap your AWS environment

Deploying stacks with the CDK requires special dedicated AWS CDK resources to be provisioned. The `cdk bootstrap` command creates the necessary resources for you. You only need to bootstrap if you are deploying a stack that requires these dedicated resources. See AWS CDK bootstrapping for details.

```
cdk bootstrap
```

If issued with no arguments, as shown here, the `cdk bootstrap` command synthesizes the current app and bootstraps the environments its stacks will be deployed to. If the app contains environment-agnostic stacks, which don't explicitly specify an environment, the default account and Region are bootstrapped, or the environment specified using `--profile`.

Outside of an app, you must explicitly specify the environment to be bootstrapped. You may also do so to bootstrap an environment that's not specified in your app or local AWS profile. Credentials must be configured (e.g. in `~/.aws/credentials`) for the specified account and Region. You may specify a profile that contains the required credentials.

```
cdk bootstrap <ACCOUNT-NUMBER>/<REGION> # e.g.
cdk bootstrap 1111111111/us-east-1
cdk bootstrap --profile test 1111111111/us-east-1
```

Important

Each environment (account/region combination) to which you deploy such a stack must be bootstrapped separately.

You may incur AWS charges for what the AWS CDK stores in the bootstrapped resources. Additionally, if you use `--bootstrap-customer-key`, an AWS KMS key will be created, which also incurs charges per environment.

Note

Earlier versions of the bootstrap template created a KMS key by default. To avoid charges, re-bootstrap using `--no-bootstrap-customer-key`.

Note

CDK CLI v2 does not support the original bootstrap template, dubbed the legacy template, used by default with CDK v1.

Important

The modern bootstrap template effectively grants the permissions implied by the `--cloudformation-execution-policies` to any AWS account in the `--trust` list. By default, this extends permissions to read and write to any resource in the bootstrapped account. Make sure to configure the bootstrapping stack with policies and trusted accounts that you are comfortable with.

Create a new app

To create a new app, create a directory for it, then, inside the directory, issue `cdk init`.

```
mkdir my-cdk-app
cd my-cdk-app
cdk init <TEMPLATE> --language <LANGUAGE>
```

The supported languages (<LANGUAGE>) are:

Code	Language
`typescript`	TypeScript

Code	Language
`javascript`	JavaScript
`python`	Python
`java`	Java
`csharp`	C#
`Go`	Go

<TEMPLATE> is an optional template. If the desired template is *app*, the default, you may omit it. The available templates are:

Template	Description
`app` (default)	Creates an empty AWS CDK app.
`sample-app`	Creates an AWS CDK app with a stack containing an Amazon SQS queue and an Amazon SNS topic.

The templates use the name of the project folder to generate names for files and classes inside your new app.

List stacks

To see a list of the IDs of the stacks in your AWS CDK application, enter one of the following equivalent commands:

```
cdk list
cdk ls
```

If your application contains CDK Pipelines stacks, the CDK CLI displays stack names as paths according to their location in the pipeline hierarchy. (For example, `PipelineStack`, `PipelineStack/Prod`, and `PipelineStack/Prod/MyService`.)

If your app contains many stacks, you can specify full or partial stack IDs of the stacks to be listed. For more information, see Specify stacks.

Add the `--long` flag to see more information about the stacks, including the stack names and their environments (AWS account and Region).

Synthesize stacks

The `cdk synthesize` command (almost always abbreviated `synth`) synthesizes a stack defined in your app into a CloudFormation template.

```
cdk synth         # if app contains only one stack
cdk synth MyStack
cdk synth Stack1 Stack2
cdk synth "*"     # all stacks in app
```

> **Note**
>
> The CDK CLI actually runs your app and synthesizes fresh templates before most operations (such as when deploying or comparing stacks). These templates are stored by default in the `cdk.out` directory. The `cdk synth` command simply prints the generated templates for one or more specified stacks.

See `cdk synth --help` for all available options. A few of the most frequently used options are covered in the following section.

Specify context values

Use the `--context` or `-c` option to pass runtime context values to your CDK app.

```
# specify a single context value
cdk synth --context key=value MyStack

# specify multiple context values (any number)
cdk synth --context key1=value1 --context key2=value2 MyStack
```

When deploying multiple stacks, the specified context values are normally passed to all of them. If you want, you can specify different values for each stack by prefixing the stack name to the context value.

```
# different context values for each stack
cdk synth --context Stack1:key=value Stack2:key=value Stack1 Stack2
```

Specify display format

By default, the synthesized template is displayed in YAML format. Add the `--json` flag to display it in JSON format instead.

```
cdk synth --json MyStack
```

Specify the output directory

Add the `--output` (`-o`) option to write the synthesized templates to a directory other than `cdk.out`.

```
cdk synth --output=~/templates
```

Deploy stacks

The `cdk deploy` subcommand deploys one or more specified stacks to your AWS account.

```
cdk deploy        # if app contains only one stack
cdk deploy MyStack
cdk deploy Stack1 Stack2
cdk deploy "*"    # all stacks in app
```

> **Note**
>
> The CDK CLI runs your app and synthesizes fresh AWS CloudFormation templates before deploying anything. Therefore, most command line options you can use with `cdk synth` (for example, `--context`) can also be used with `cdk deploy`.

See `cdk deploy --help` for all available options. A few of the most useful options are covered in the following section.

Skip synthesis

The `cdk deploy` command normally synthesizes your app's stacks before deploying to make sure that the deployment reflects the latest version of your app. If you know that you haven't changed your code since your last `cdk synth`, you can suppress the redundant synthesis step when deploying. To do so, specify your project's `cdk.out` directory in the `--app` option.

```
cdk deploy --app cdk.out StackOne StackTwo
```

Disable rollback

AWS CloudFormation has the ability to roll back changes so that deployments are atomic. This means that they either succeed or fail as a whole. The AWS CDK inherits this capability because it synthesizes and deploys AWS CloudFormation templates.

Rollback makes sure that your resources are in a consistent state at all times, which is vital for production stacks. However, while you're still developing your infrastructure, some failures are inevitable, and rolling back failed deployments can slow you down.

For this reason, the CDK CLI lets you disable rollback by adding `--no-rollback` to your `cdk deploy` command. With this flag, failed deployments are not rolled back. Instead, resources deployed before the failed resource remain in place, and the next deployment starts with the failed resource. You'll spend a lot less time waiting for deployments and a lot more time developing your infrastructure.

Hot swapping

Use the `--hotswap` flag with `cdk deploy` to attempt to update your AWS resources directly instead of generating an AWS CloudFormation change set and deploying it. Deployment falls back to AWS CloudFormation deployment if hot swapping is not possible.

Currently hot swapping supports Lambda functions, Step Functions state machines, and Amazon ECS container images. The `--hotswap` flag also disables rollback (i.e., implies `--no-rollback`).

Important

Hot-swapping is not recommended for production deployments.

Watch mode

The CDK CLI's watch mode (`cdk deploy --watch`, or `cdk watch` for short) continuously monitors your CDK app's source files and assets for changes. It immediately performs a deployment of the specified stacks when a change is detected.

By default, these deployments use the `--hotswap` flag, which fast-tracks deployment of changes to Lambda functions. It also falls back to deploying through AWS CloudFormation if you have changed infrastructure configuration. To have `cdk watch` always perform full AWS CloudFormation deployments, add the `--no-hotswap` flag to `cdk watch`.

Any changes made while `cdk watch` is already performing a deployment are combined into a single deployment, which begins as soon as the in-progress deployment is complete.

Watch mode uses the `"watch"` key in the project's `cdk.json` to determine which files to monitor. By default, these files are your application files and assets, but this can be changed by modifying the `"include"` and `"exclude"` entries in the `"watch"` key. The following `cdk.json` file shows an example of these entries.

```
{
  "app": "mvn -e -q compile exec:java",
  "watch": {
    "include": "src/main/**",
    "exclude": "target/*"
  }
}
```

`cdk watch` executes the `"build"` command from `cdk.json` to build your app before synthesis. If your deployment requires any commands to build or package your Lambda code (or anything else that's not in your CDK app), add it here.

Git-style wildcards, both `*` and `**`, can be used in the `"watch"` and `"build"` keys. Each path is interpreted relative to the parent directory of `cdk.json`. The default value of `include` is `**/*`, meaning all files and directories in the project root directory. `exclude` is optional.

> **Important**
>
> Watch mode is not recommended for production deployments.

Specify AWS CloudFormation parameters

The CDK CLI supports specifying AWS CloudFormation parameters at deployment. You may provide these on the command line following the `--parameters` flag.

```
cdk deploy MyStack --parameters uploadBucketName=UploadBucket
```

To define multiple parameters, use multiple `--parameters` flags.

```
cdk deploy MyStack --parameters uploadBucketName=UpBucket --parameters
 downloadBucketName=DownBucket
```

If you are deploying multiple stacks, you can specify a different value of each parameter for each stack. To do so, prefix the name of the parameter with the stack name and a colon. Otherwise, the same value is passed to all stacks.

```
cdk deploy MyStack YourStack --parameters MyStack:uploadBucketName=UploadBucket --
parameters YourStack:uploadBucketName=UpBucket
```

By default, the AWS CDK retains values of parameters from previous deployments and uses them in later deployments if they are not specified explicitly. Use the `--no-previous-parameters` flag to require all parameters to be specified.

Specify outputs file

If your stack declares AWS CloudFormation outputs, these are normally displayed on the screen at the conclusion of deployment. To write them to a file in JSON format, use the `--outputs-file` flag.

```
cdk deploy --outputs-file outputs.json MyStack
```

Approve security-related changes

To protect you against unintended changes that affect your security posture, the CDK CLI prompts you to approve security-related changes before deploying them. You can specify the level of change that requires approval:

```
cdk deploy --require-approval <LEVEL>
```

<LEVEL> can be one of the following:

Term	Meaning
`never`	Approval is never required
`any-change`	Requires approval on any IAM or security-group-related change
`broadening` (default)	Requires approval when IAM statements or traffic rules are added; removals don't require approval

The setting can also be configured in the `cdk.json` file.

```
{
  "app": "...",
  "requireApproval": "never"
}
```

Compare stacks

The `cdk diff` command compares the current version of a stack (and its dependencies) defined in your app with the already-deployed versions, or with a saved AWS CloudFormation template, and displays a list of changes.

```
Stack HelloCdkStack
IAM Statement Changes
################################################################################################
```

```
#   # Resource                     # Effect # Action                        # Principal
                   # Condition #
#############################################################################################
# + # ${Custom::S3AutoDeleteObject # Allow  # sts:AssumeRole                #
 Service:lambda.amazonaws.com #           #
#   # sCustomResourceProvider/Role #        #                               #
                   #           #
#   # .Arn}                        #        #                               #
                   #           #
#############################################################################################
# + # ${MyFirstBucket.Arn}         # Allow  # s3:DeleteObject*              # {aws}:
${Custom::S3AutoDeleteOb #           #
#   # ${MyFirstBucket.Arn}/*       #        # s3:GetBucket*                 #
 jectsCustomResourceProvider/ #           #
#   #                              #        # s3:GetObject*                 # Role.Arn}
                   #           #
#   #                              #        # s3:List*                      #
                   #           #
#############################################################################################
IAM Policy Changes
#############################################################################################
#   # Resource                                                   # Managed Policy ARN
                               #
#############################################################################################
# + # ${Custom::S3AutoDeleteObjectsCustomResourceProvider/Ro # {"Fn::Sub":"arn:
${{aws}::Partition}:iam::aws:policy/serv #
#   # le}                                                        # ice-role/
AWSLambdaBasicExecutionRole"}                 #
#############################################################################################
(NOTE: There may be security-related changes not in this list. See https://github.com/
aws/aws-cdk/issues/1299)

Parameters
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3Bucket
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3BucketBF7A7F3
 {"Type":"String","Description":"S3 bucket for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
S3VersionKey
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392S3VersionKeyFAF
```

```
 {"Type":"String","Description":"S3 key for asset version
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}
[+] Parameter
 AssetParameters/4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392/
ArtifactHash
 AssetParameters4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392ArtifactHashE56
 {"Type":"String","Description":"Artifact hash for asset
 \"4cd61014b71160e8c66fe167e43710d5ba068b80b134e9bd84508cf9238b2392\""}

Resources
[+] {aws}::S3::BucketPolicy MyFirstBucket/Policy MyFirstBucketPolicy3243DEFD
[+] Custom::S3AutoDeleteObjects MyFirstBucket/AutoDeleteObjectsCustomResource
 MyFirstBucketAutoDeleteObjectsCustomResourceC52FCF6E
[+] {aws}::IAM::Role Custom::S3AutoDeleteObjectsCustomResourceProvider/Role
 CustomS3AutoDeleteObjectsCustomResourceProviderRole3B1BD092
[+] {aws}::Lambda::Function Custom::S3AutoDeleteObjectsCustomResourceProvider/Handler
 CustomS3AutoDeleteObjectsCustomResourceProviderHandler9D90184F
[~] {aws}::S3::Bucket MyFirstBucket MyFirstBucketB8884501
 ## [~] DeletionPolicy
 #   ## [-] Retain
 #   ## [+] Delete
 ## [~] UpdateReplacePolicy
     ## [-] Retain
     ## [+] Delete
```

To compare your app's stacks with the existing deployment:

```
cdk diff MyStack
```

To compare your app's stacks with a saved CloudFormation template:

```
cdk diff --template ~/stacks/MyStack.old MyStack
```

Import existing resources into a stack

You can use the `cdk import` command to bring resources under the management of CloudFormation for a particular AWS CDK stack. This is useful if you are migrating to AWS CDK, or are moving resources between stacks or changing their logical id. `cdk import` uses CloudFormation resource imports. See the list of resources that can be imported here.

To import an existing resource into a AWS CDK stack, follow the following steps:

- Make sure the resource is not currently being managed by any other CloudFormation stack. If it is, first set the removal policy to `RemovalPolicy.RETAIN` in the stack the resource is currently in and perform a deployment. Then, remove the resource from the stack and perform another deployment. This process will make sure that the resource is no longer managed by CloudFormation but does not delete it.
- Run a `cdk diff` to make sure there are no pending changes to the AWS CDK stack you want to import resources into. The only changes allowed in an "import" operation are the addition of new resources which you want to import.
- Add constructs for the resources you want to import to your stack. For example, if you want to import an Amazon S3 bucket, add something like `new s3.Bucket(this, 'ImportedS3Bucket', {});`. Do not make any modifications to any other resource.

  You must also make sure to exactly model the state that the resource currently has into the definition. For the example of the bucket, be sure to include AWS KMS keys, life cycle policies, and anything else that's relevant about the bucket. If you do not, subsequent update operations may not do what you expect.

  You can choose whether or not to include the physical bucket name. We usually recommend to not include resource names into your AWS CDK resource definitions so that it becomes easier to deploy your resources multiple times.
- Run `cdk import <STACKNAME>`.
- If the resource names are not in your model, the CLI will prompt you to pass in the actual names of the resources you are importing. After this, the import starts.
- When `cdk import` reports success, the resource is now managed by AWS CDK and CloudFormation. Any subsequent changes you make to the resource properties in your AWS CDK app the construct configuration will be applied on the next deployment.
- To confirm that the resource definition in your AWS CDK app matches the current state of the resource, you can start an [CloudFormation drift detection operation]().

This feature currently does not support importing resources into nested stacks.

Configuration (`cdk.json`)

Default values for many CDK CLI command line flags can be stored in a project's `cdk.json` file or in the `.cdk.json` file in your user directory. Following is an alphabetical reference to the supported configuration settings.

Key	Notes	CDK CLI option
`app`	The command that executes the CDK application.	`--app`
`assetMetadata`	If `false`, CDK does not add metadata to resources that use assets.	`--no-asset-metadata`
`bootstrapKmsKeyId`	Overrides the ID of the AWS KMS key used to encrypt the Amazon S3 deployment bucket.	`--bootstrap-kms-key-id`
`build`	The command that compiles or builds the CDK application before synthesis. Not permitted in `~/.cdk.json`.	`--build`
`browser`	The command for launching a Web browser for the `cdk docs` subcommand.	`--browser`
`context`	See Context values and the AWS CDK. Context values in a configuration file will not be erased by `cdk context --clear`. (The CDK CLI places cached context values in `cdk.context.json`.)	`--context`
`debug`	If `true`, CDK CLI emits more detailed information useful for debugging.	`--debug`
`language`	The language to be used for initializing new projects.	`--language`

Key	Notes	CDK CLI option
`lookups`	If `false`, no context lookups are permitted. Synthesis will fail if any context lookups need to be performed.	`--no-lookups`
`notices`	If `false`, suppresses the display of messages about security vulnerabilities, regressions, and unsupported versions.	`--no-notices`
`output`	The name of the directory into which the synthesized cloud assembly will be emitted (default `"cdk.out"`).	`--output`
`outputsFile`	The file to which AWS CloudFormation outputs from deployed stacks will be written (in `JSON` format).	`--outputs-file`
`pathMetadata`	If `false`, CDK path metadata is not added to synthesized templates.	`--no-path-metadata`
`plugin`	JSON array specifying the package names or local paths of packages that extend the CDK	`--plugin`
`profile`	Name of the default AWS profile used for specifying Region and account credentials.	`--profile`

Key	Notes	CDK CLI option
`progress`	If set to `"events"`, the CDK CLI displays all AWS CloudFormation events during deployment, rather than a progress bar.	`--progress`
`requireApproval`	Default approval level for security changes. See [Approve security-related changes](#)	`--require-approval`
`rollback`	If `false`, failed deployments are not rolled back.	`--no-rollback`
`staging`	If `false`, assets are not copied to the output directory (use for local debugging of the source files with AWS SAM).	`--no-staging`
`tags`	`JSON` object containing tags (key-value pairs) for the stack.	`--tags`
`toolkitBucketName`	The name of the Amazon S3 bucket used for deploying assets such as Lambda functions and container images (see [Bootstrap your AWS environment](#)).	`--toolkit-bucket-name`
`toolkitStackName`	The name of the bootstrap stack (see [Bootstrap your AWS environment](#)).	`--toolkit-stack-name`
`versionReporting`	If `false`, opts out of version reporting.	`--no-version-reporting`

Key	Notes	CDK CLI option
`watch`	JSON object containing `"include"` and `"exclude"` keys that indicate which files should (or should not) trigger a rebuild of the project when changed. See Watch mode.	`--watch`

AWS CDK CLI command reference

This section contains command reference information for the AWS Cloud Development Kit (AWS CDK) Command Line Interface (CLI). The CDK CLI is also referred to as the CDK Toolkit.

Usage

```
$ cdk <command> <arguments> <options>
```

Commands

`acknowledge ack`

Acknowledge a notice by issue number and hide it from displaying again.

`bootstrap`

Prepare an AWS environment for CDK deployments by deploying the CDK bootstrap stack, named `CDKToolkit`, into the AWS environment.

`context`

Manage cached context values for your CDK application.

`deploy`

Deploy one or more CDK stacks into your AWS environment.

`destroy`

Delete one or more CDK stacks from your AWS environment.

`diff`

Perform a diff to see infrastructure changes between CDK stacks.

`docs, doc`

Open CDK documentation in your browser.

`doctor`

Inspect and display useful information about your local CDK project and development environment.

`drift`

Detect configuration drift for resources that you define, manage, and deploy using CDK.

`flags`

View and modify your feature flag configurations for the CDK CLI.

`import`

Use AWS CloudFormation resource imports to import existing AWS resources into a CDK stack.

`init`

Create a new CDK project from a template.

`list, ls`

List all CDK stacks and their dependencies from a CDK app.

`metadata`

Display metadata associated with a CDK stack.

`migrate`

Migrate AWS resources, AWS CloudFormation stacks, and AWS CloudFormation templates into a new CDK project.

`notices`

Display notices for your CDK application.

`synthesize, synth`

Synthesize a CDK app to produce a cloud assembly, including an AWS CloudFormation template for each stack.

`watch`

Continuously watch a local CDK project for changes to perform deployments and hotswaps.

Global options

The following options are compatible with all CDK CLI commands.

`--app, -a <STRING>`

Provide the command for running your app or cloud assembly directory.

Required: Yes

`--asset-metadata <BOOLEAN>`

Include `aws:asset:*` AWS CloudFormation metadata for resources that use assets.

Required: No

Default value: `true`

`--build <STRING>`

Command for running a pre-synthesis build.

Required: No

`--ca-bundle-path <STRING>`

Path to a CA certificate to use when validating HTTPS requests.

If this option is not provided, the CDK CLI will read from the `AWS_CA_BUNDLE` environment variable.

Required: Yes

`--ci <BOOLEAN>`

Indicate that CDK CLI commands are being run in a continuous integration (CI) environment.

This option modifies the behavior of the CDK CLI to better suit automated operations that are typical in CI pipelines.

When you provide this option, logs are sent to `stdout` instead of `stderr`.

Required: No

Default value: `false`

`--context, -c <ARRAY>`

Add contextual string parameters as key-value pairs.

`--debug <BOOLEAN>`

Enable detailed debugging information. This option produces a verbose output that includes a lot more detail about what the CDK CLI is doing behind the scenes.

Required: No

Default value: `false`

`--ec2creds, -i <BOOLEAN>`

Force the CDK CLI to try and fetch Amazon EC2 instance credentials.

By default, the CDK CLI guesses the Amazon EC2 instance status.

Required: No

Default value: `false`

`--help, -h <BOOLEAN>`

Show command reference information for the CDK CLI.

Required: No

Default value: `false`

`--ignore-errors <BOOLEAN>`

Ignore synthesis errors, which will likely produce an output that is not valid.

Required: No

Default value: `false`

`--json, -j <BOOLEAN>`

Use JSON instead of YAML for AWS CloudFormation templates that are printed to standard output (`stdout`).

Required: No

Default value: `false`

`--lookups <BOOLEAN>`

Perform context lookups.

Synthesis will fail if this value is `false` and context lookups need to be performed.

Required: No

Default value: `true`

`--no-color <BOOLEAN>`

Remove color and other styling from the console output.

Required: No

Default value: `false`

`--notices <BOOLEAN>`

Show relevant notices.

Required: No

Default value: `false`

`--output, -o <STRING>`

Specify the directory to output the synthesized cloud assembly to.

Required: Yes

Default value: `cdk.out`

`--path-metadata <BOOLEAN>`

Include `aws::cdk::path` AWS CloudFormation metadata for each resource.

Required: No

Default value: `true`

`--plugin, -p <ARRAY>`

Name or path of a node package that extends CDK features. This option can be provided multiple times in a single command.

You can configure this option in the project's `cdk.json` file or at `~/.cdk.json` on your local development machine:

```
{
   // ...
   "plugin": [
      "module_1",
      "module_2"
   ],
   // ...
}
```

Required: No

`--profile <STRING>`

Specify the name of the AWS profile, containing your AWS environment information, to use with the CDK CLI.

Required: Yes

`--proxy <STRING>`

Use the indicated proxy.

If this option is not provided, the CDK CLI will read from the `HTTPS_PROXY` environment variable.

Required: Yes

Default value: Read from `HTTPS_PROXY` environment variable.

`--role-arn, -r <STRING>`

The ARN of the IAM role that the CDK CLI will assume when interacting with AWS CloudFormation.

Required: No

`--staging <BOOLEAN>`

Copy assets to the output directory.

Specify `false` to prevent the copying of assets to the output directory. This allows the AWS SAM CLI to reference the original source files when performing local debugging.

Required: No

Default value: `true`

`--strict <BOOLEAN>`

Do not construct stacks that contain warnings.

Required: No

Default value: `false`

`--trace <BOOLEAN>`

Print trace for stack warnings.

Required: No

Default value: `false`

`--verbose, -v <COUNT>`

Show debug logs. You can specify this option multiple times to increase verbosity.

Required: No

`--version <BOOLEAN>`

Show the CDK CLI version number.

Required: No

Default value: `false`

`--version-reporting <BOOLEAN>`

Include the `AWS::CDK::Metadata` resource in synthesized AWS CloudFormation templates.

Required: No

Default value: `true`

Providing and configuring options

You can pass options through command-line arguments. For most options, you can configure them in a `cdk.json` configuration file. When you use multiple configuration sources, the CDK CLI adheres to the following precedence:

1. **Command-line values** – Any option provided at the command-line overrides options configured in `cdk.json` files.
2. **Project configuration file** – The `cdk.json` file in your CDK project's directory.
3. **User configuration file** – The `cdk.json` file located at `~/.cdk.json` on your local machine.

Passing options at the command line

Passing boolean values

For options that accept a boolean value, you can specify them in the following ways:

- Use `true` and `false` values – Provide the boolean value with the command. The following is an example:

```
$ cdk deploy --watch=true
$ cdk deploy --watch=false
```

- Provide the option's counterpart – Modify the option name by adding no to specify a `false` value. The following is an example:

```
$ cdk deploy --watch
$ cdk deploy --no-watch
```

- For options that default to `true` or `false`, you don't have to provide the option unless you want to change from the default.

`cdk acknowledge`

Acknowledge a notice by issue number and hide it from displaying again.

This is useful to hide notices that have been addressed or do not apply to you.

Acknowledgements are saved at a CDK project level. If you acknowledge a notice in one CDK project, it will still display in other projects until acknowledged there.

Usage

```
$ cdk acknowledge <arguments> <options>
```

Arguments

Notice ID

The ID of the notice.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk acknowledge` command.

Examples

Acknowledge and hide a notice that displays when running another CDK CLI command

```
$ cdk deploy
... # Normal output of the command

NOTICES

16603   Toggling off auto_delete_objects for Bucket empties the bucket

        Overview: If a stack is deployed with an S3 bucket with
                  auto_delete_objects=True, and then re-deployed with
                  auto_delete_objects=False, all the objects in the bucket
                  will be deleted.

        Affected versions: <1.126.0.

        More information at: https://github.com/aws/aws-cdk/issues/16603
```

```
17061   Error when building EKS cluster with monocdk import

        Overview: When using monocdk/aws-eks to build a stack containing
                  an EKS cluster, error is thrown about missing
                  lambda-layer-node-proxy-agent/layer/package.json.

        Affected versions: >=1.126.0 <=1.130.0.

        More information at: https://github.com/aws/aws-cdk/issues/17061

$ cdk acknowledge 16603
```

`cdk bootstrap`

Prepare an AWS environment for CDK deployments by deploying the CDK bootstrap stack, named `CDKToolkit`, into the AWS environment.

The bootstrap stack is a CloudFormation stack that provisions an Amazon S3 bucket and Amazon ECR repository in the AWS environment. The AWS CDK CLI uses these resources to store synthesized templates and related assets during deployment.

Usage

```
$ cdk bootstrap <arguments> <options>
```

Arguments

AWS environment

The target AWS environment to deploy the bootstrap stack to in the following format: `aws://<account-id>/<region>`.

Example: `aws://123456789012/us-east-1`

This argument can be provided multiple times in a single command to deploy the bootstrap stack to multiple environments.

By default, the CDK CLI will bootstrap all environments referenced in the CDK app or will determine an environment from default sources. This could be an environment specified using the `--profile` option, from environment variables, or default AWS CLI sources.

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--bootstrap-bucket-name, --toolkit-bucket-name, -b <STRING>`

The name of the Amazon S3 bucket that will be used by the CDK CLI. This bucket will be created and must not currently exist.

Provide this option to override the default name of the Amazon S3 bucket.

When you use this option, you may have to customize synthesis. To learn more, see Customize CDK stack synthesis.

Default value: Undefined

`--bootstrap-customer-key <BOOLEAN>`

Create a Customer Master Key (CMK) for the bootstrap bucket (you will be charged but can customize permissions, modern bootstrapping only).

This option is not compatible with `--bootstrap-kms-key-id`.

Default value: Undefined

`--bootstrap-kms-key-id <STRING>`

The AWS KMS master key ID to use for the `SSE-KMS` encryption.

Provide this option to override the default AWS KMS key used to encrypt the Amazon S3 bucket.

This option is not compatible with `--bootstrap-customer-key`.

Default value: Undefined

`--cloudformation-execution-policies <ARRAY>`

The managed IAM policy ARNs that should be attached to the deployment role assumed by AWS CloudFormation during deployment of your stacks.

By default, stacks are deployed with full administrator permissions using the `AdministratorAccess` policy.

You can provide this option multiple times in a single command. You can also provide multiple ARNs as a single string, with the individual ARNs separated by commas. The following is an example:

```
$ cdk bootstrap --cloudformation-execution-policies "arn:aws:iam::aws:policy/
AWSLambda_FullAccess,arn:aws:iam::aws:policy/AWSCodeDeployFullAccess"
```

To avoid deployment failures, be sure that the policies you specify are sufficient for any deployments that you will perform into the environment being bootstrapped.

This option applies to modern bootstrapping only.

Important

The modern bootstrap template effectively grants the permissions implied by the `--cloudformation-execution-policies` to any AWS account in the `--trust` list. By default, this extends permissions to read and write to any resource in the bootstrapped account. Make sure to configure the bootstrapping stack with policies and trusted accounts that you are comfortable with.

Default value: `[]`

`--custom-permissions-boundary, -cpb <STRING>`

Specify the name of a permissions boundary to use.

This option is not compatible with `--example-permissions-boundary`.

Default value: Undefined

`--example-permissions-boundary, -epb <BOOLEAN>`

Use the example permissions boundary, supplied by the AWS CDK.

This option is not compatible with `--custom-permissions-boundary`.

The CDK supplied permissions boundary policy should be regarded as an example. Edit the content and reference the example policy if you are testing out the feature. Convert it into a new policy for actual deployments, if one does not already exist. The concern is to avoid drift. Most likely, a permissions boundary is maintained and has dedicated conventions, naming included.

For more information on configuring permissions, including using permissions boundaries, see the [Security and Safety Dev Guide](#).

Default value: Undefined

`--execute <BOOLEAN>`

Configure whether to execute the change set.

Default value: `true`

`--force, -f <BOOLEAN>`

Always bootstrap, even if it would downgrade the bootstrap template version.

Default value: `false`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk bootstrap` command.

`--previous-parameters <BOOLEAN>`

Use previous values for existing parameters.

Once a bootstrap template is deployed with a set of parameters, you must set this option to `false` to change any parameters on future deployments. When `false`, you must re-supply all previously supplied parameters.

Default value: `true`

`--public-access-block-configuration <BOOLEAN>`

Block public access configuration on the Amazon S3 bucket that is created and used by the CDK CLI.

Default value: `true`

`--qualifier <STRING>`

Nine-digit string value that is unique for each bootstrap stack. This value is added to the physical ID of resources in the bootstrap stack.

By providing a qualifier, you avoid resource name clashes when provisioning multiple bootstrap stacks in the same environment.

When you change the qualifier, your CDK app must pass the changed value to the stack synthesizer. For more information, see Customize CDK stack synthesis.

Default value: `hnb659fds`. This value has no significance.

`--show-template <BOOLEAN>`

Instead of bootstrapping, print the current bootstrap template to the standard output (`stdout`). You can then copy and customize the template as necessary.

Default value: `false`

`--tags, -t <ARRAY>`

Tags to add to the bootstrap stack in the format of `KEY=VALUE`.

Default value: `[]`

`--template <STRING>`

Use the template from the given file instead of the built-in one.

`--termination-protection <BOOLEAN>`

Toggle AWS CloudFormation termination protection on the bootstrap stack.

When `true`, termination protection is enabled. This prevents the bootstrap stack from being accidentally deleted.

To learn more about termination protection, see Protecting a stack from being deleted in the *AWS CloudFormation User Guide*.

Default value: Undefined

`--toolkit-stack-name <STRING>`

The name of the bootstrap stack to create.

By default, `cdk bootstrap` deploys a stack named `CDKToolkit` into the specified AWS environment. Use this option to provide a different name for your bootstrap stack.

The CDK CLI uses this value to verify your bootstrap stack version.

Default value: `CDKToolkit`

Required: Yes

`--trust <ARRAY>`

The AWS account IDs that should be trusted to perform deployments into this environment.

The account performing the bootstrapping will always be trusted.

This option requires that you also provide `--cloudformation-execution-policies`.

You can provide this option multiple times in a single command.

This option applies to modern bootstrapping only.

To add trusted accounts to an existing bootstrap stack, you must specify all of the accounts to trust, including those that you may have previously provided. If you only provide new accounts to trust, the previously trusted accounts will be removed.

The following is an example that trusts two accounts:

```
$ cdk bootstrap aws://123456789012/us-west-2 --trust 234567890123 --trust
 987654321098 --cloudformation-execution-policies arn:aws:iam::aws:policy/
AdministratorAccess
 #  Bootstrapping environment aws://123456789012/us-west-2...
```

```
Trusted accounts for deployment: 234567890123, 987654321098
Trusted accounts for lookup: (none)
Execution policies: arn:aws:iam::aws:policy/AdministratorAccess
CDKToolkit: creating CloudFormation changeset...
 #  Environment aws://123456789012/us-west-2 bootstrapped.
```

> **Important**
>
> The modern bootstrap template effectively grants the permissions implied by the `--cloudformation-execution-policies` to any AWS account in the `--trust` list. By default, this extends permissions to read and write to any resource in the bootstrapped account. Make sure to configure the bootstrapping stack with policies and trusted accounts that you are comfortable with.

Default value: `[]`

`--trust-for-lookup <ARRAY>`

The AWS account IDs that should be trusted to look up values in this environment.

Use this option to give accounts permission to synthesize stacks that will be deployed into the environment, without actually giving them permission to deploy those stacks directly.

You can provide this option multiple times in a single command.

This option applies to modern bootstrapping only.

Default value: `[]`

Examples

Bootstrap the AWS environment specified in the prod profile

```
$ cdk bootstrap --profile prod
```

Deploy the bootstrap stack to environments foo and bar

```
$ cdk bootstrap --app='node bin/main.js' foo bar
```

Export the bootstrap template to customize it

If you have specific requirements that are not met by the bootstrap template, you can customize it to fit your needs.

You can export the bootstrap template, modify it, and deploy it using AWS CloudFormation. The following is an example of exporting the existing template:

```
$ cdk bootstrap --show-template > bootstrap-template.yaml
```

You can also tell the CDK CLI to use a custom template. The following is an example:

```
$ cdk bootstrap --template my-bootstrap-template.yaml
```

Bootstrap with a permissions boundary. Then, remove that permissions boundary

To bootstrap with a custom permissions boundary, we run the following:

```
$ cdk bootstrap --custom-permissions-boundary my-permissions-boundary
```

To remove the permissions boundary, we run the following:

```
$ cdk bootstrap --no-previous-parameters
```

Use a qualifier to distinguish resources that are created for a development environment

```
$ cdk bootstrap --qualifier dev2024
```

`cdk context`

Manage cached context values for your AWS CDK application.

Context represents the configuration and environment information that can influence how your stacks are synthesized and deployed. Use `cdk context` to do the following:

- View your configured context values.
- Set and manage context values.

- Remove context values.

Usage

```
$ cdk context <options>
```

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--clear <BOOLEAN>`

Clear all context.

`--force, -f <BOOLEAN>`

Ignore missing key error.

Default value: `false`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk context` command.

`--reset, -e <STRING>`

The context key, or its index, to reset.

`cdk deploy`

Deploy one or more AWS CDK stacks into your AWS environment.

During deployment, the CDK CLI will output progress indicators, similar to what can be observed from the AWS CloudFormation console.

If the AWS environment is not bootstrapped, only stacks without assets and with synthesized templates under 51,200 bytes will successfully deploy.

Usage

```
$ cdk deploy <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to deploy.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--all <BOOLEAN>`

Deploy all stacks in your CDK app.

Default value: `false`

`--asset-parallelism <BOOLEAN>`

Specify whether to build and publish assets in parallel.

`--asset-prebuild <BOOLEAN>`

Specify whether to build all assets before deploying the first stack. This option is useful for failing Docker builds.

Default value: `true`

`--build-exclude, -E <ARRAY>`

Do not rebuild asset with the given ID.

This option can be specified multiple times in a single command.

Default value: `[]`

`--change-set-name <STRING>`

The name of the AWS CloudFormation change set to create.

This option is not compatible with `--method='direct'`.

`--concurrency <NUMBER>`

Deploy multiple stacks in parallel while accounting for inter-stack dependencies. Use this option to speed up deployments. You must still factor in AWS CloudFormation and other AWS account rate limiting.

Provide a number to specify the maximum number of simultaneous deployments (dependency permitting) to perform.

Default value: `1`

`--exclusively, -e <BOOLEAN>`

Only deploy requested stacks and don't include dependencies.

`--force, -f <BOOLEAN>`

When you deploy to update an existing stack, the CDK CLI will compare the template and tags of the deployed stack to the stack about to be deployed. If no changes are detected, the CDK CLI will skip deployment.

To override this behavior and always deploy stacks, even if no changes are detected, use this option.

Default value: `false`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk deploy` command.

`--hotswap <BOOLEAN>`

Hotswap deployments for faster development. This option attempts to perform a faster, hotswap deployment if possible. For example, if you modify the code of a Lambda function in your CDK app, the CDK CLI will update the resource directly through service APIs instead of performing a CloudFormation deployment.

If the CDK CLI detects changes that don't support hotswapping, those changes will be ignored and a message will display. If you prefer to perform a full CloudFormation deployment as a fall back, use `--hotswap-fallback` instead.

The CDK CLI uses your current AWS credentials to perform the API calls. It does not assume the roles from your bootstrap stack, even if the `@aws-cdk/core:newStyleStackSynthesis` feature flag is set to `true`. Those roles do not have the necessary permissions to update AWS resources directly, without using CloudFormation. For that reason, make sure that your credentials are for the same AWS account of the stacks that you are performing hotswap deployments against and that they have the necessary IAM permissions to update the resources.

Hotswapping is currently supported for the following changes:

- Code assets (including Docker images and inline code), tag changes, and configuration changes (only description and environment variables are supported) of Lambda functions.
- Lambda versions and alias changes.
- Definition changes of AWS Step Functions state machines.
- Container asset changes of Amazon ECS services.
- Website asset changes of Amazon S3 bucket deployments.
- Source and environment changes of AWS CodeBuild projects.
- VTL mapping template changes for AWS AppSync resolvers and functions.
- Schema changes for AWS AppSync GraphQL APIs.

Usage of certain CloudFormation intrinsic functions are supported as part of a hotswapped deployment. These include:

- `Ref`
- `Fn::GetAtt` – Only partially supported. Refer to this implementation for supported resources and attributes.
- `Fn::ImportValue`
- `Fn::Join`

- `Fn::Select`
- `Fn::Split`
- `Fn::Sub`

This option is also compatible with nested stacks.

> **Note**
>
> - This option deliberately introduces drift in CloudFormation stacks in order to speed up deployments. For this reason, only use it for development purposes. Do not use this option for your production deployments.
> - This option is considered experimental and may have breaking changes in the future.
> - Defaults for certain parameters may be different with the hotswap parameter. For example, an Amazon ECS service's minimum healthy percentage will currently be set at `0`. Review the source accordingly if this occurs.

Default value: `false`

`--hotswap-fallback <BOOLEAN>`

This option is is similar to `--hotswap`. The difference being that `--hotswap-fallback` will fall back to perform a full CloudFormation deployment if a change is detected that requires it.

For more information about this option, see `--hotswap`.

Default value: `false`

`--ignore-no-stacks <BOOLEAN>`

Perform a deployment even if your CDK app doesn't contain any stacks.

This option is helpful in the following scenario: You may have an app with multiple environments, such as `dev` and `prod`. When starting development, your prod app may not have any resources, or the resources may be commented out. This will result in a deployment error with a message stating that the app has no stacks. Use `--ignore-no-stacks` to bypass this error.

Default value: `false`

`--import-existing-resources <BOOLEAN>`

Import existing, unmanaged AWS CloudFormation resources from your AWS account.

When you use this option, resources from your synthesized AWS CloudFormation template with the same custom name as existing unmanaged resources in the same account will be imported into your stack.

You can use this option to import existing resources into new or existing stacks.

You can import existing resources and deploy new resources in the same `cdk deploy` command.

To learn more about custom names, see Name type in the *AWS CloudFormation User Guide*.

To learn more about the `ImportExistingResources` CloudFormation parameter, see AWS CloudFormation simplifies resource import with a new parameter for ChangeSets.

For more information on using this option, see Import existing resources in the *aws-cdk-cli GitHub repository*.

`--logs <BOOLEAN>`

Show Amazon CloudWatch log in the standard output (`stdout`) for all events from all resources in the selected stacks.

This option is only compatible with `--watch`.

Default value: `true`

`--method, -m <STRING>`

Configure the method to perform a deployment.

- `change-set` – Default method. The CDK CLI creates a CloudFormation change set with the changes that will be deployed, then performs deployment.
- `direct` – Do not create a change set. Instead, apply the change immediately. This is typically faster than creating a change set, but you lose deployment progress details in the CLI output.
- `prepare-change-set` – Create change set but don't perform deployment. This is useful if you have external tools that will inspect the change set or if you have an approval process for change sets.

Valid values: `change-set`, `direct`, `prepare-change-set`

Default value: `change-set`

`--notification-arns <ARRAY>`

The ARNs of Amazon SNS topics that CloudFormation will notify for stack related events.

`--outputs-file, -O <STRING>`

The path to where stack outputs from deployments are written to.

After deployment, stack outputs will be written to the specified output file in JSON format.

You can configure this option in the project's `cdk.json` file or at `~/.cdk.json` on your local development machine:

```
{
   "app": "npx ts-node bin/myproject.ts",
   // ...
   "outputsFile": "outputs.json"
}
```

If multiple stacks are deployed, outputs are written to the same output file, organized by keys representing the stack name.

`--parameters <ARRAY>`

Pass additional parameters to CloudFormation during deployment.

This option accepts an array in the following format: `STACK:KEY=VALUE`.

- `STACK` – The name of the stack to associate the parameter with.
- `KEY` – The name of the parameter from your stack.
- `VALUE` – The value to pass at deployment.

If a stack name is not provided, or if `*` is provided as the stack name, parameters will be applied to all stacks being deployed. If a stack does not make use of the parameter, deployment will fail.

Parameters do not propagate to nested stacks. To pass parameters to nested stacks, use the `NestedStack` construct.

Default value: `{}`

`--previous-parameters <BOOLEAN>`

Use previous values for existing parameters.

When this option is set to `false`, you must specify all parameters on every deployment.

Default value: `true`

`--progress <STRING>`

Configure how the CDK CLI displays deployment progress.

- `bar` – Display stack deployment events as a progress bar, with the events for the resource currently being deployed.
- `events` – Provide a complete history, including all CloudFormation events.

You can also configure this option in the project's `cdk.json` file or at `~/.cdk.json` on your local development machine:

```
{
   "progress": "events"
}
```

Valid values: `bar`, `events`

Default value: `bar`

`--require-approval <STRING>`

Specify what security-sensitive changes require manual approval.

- `any-change` – Manual approval required for any change to the stack.
- `broadening` – Manual approval required if changes involve a broadening of permissions or security group rules.

- `never` – Approval is not required.

  Valid values: `any-change`, `broadening`, `never`

  Default value: `broadening`

`--rollback` | `--no-rollback`, `-R`

During deployment, if a resource fails to be created or updated, the deployment will roll back to the latest stable state before the CDK CLI returns. All changes made up to that point will be undone. Resources that were created will be deleted and updates that were made will be rolled back.

Specify `--no-rollback` to turn off this behavior. If a resource fails to be created or updated, the CDK CLI will leave changes made up to that point in place and return. This will leave your deployment in a failed, paused state. From here, you can update your code and try the deployment again. This may be helpful in development environments where you are iterating quickly.

If a deployment performed with `--no-rollback` fails, and you decide that you want to rollback the deployment, you can use the `cdk rollback` command. For more information, see cdk rollback.

> **Note**
>
> With `--no-rollback`, deployments that cause resource replacements will always fail. You can only use this option value for deployments that update or create new resources.

Default value: `--rollback`

`--toolkit-stack-name <STRING>`

The name of the existing CDK Toolkit stack.

By default, `cdk bootstrap` deploys a stack named `CDKToolkit` into the specified AWS environment. Use this option to provide a different name for your bootstrap stack.

The CDK CLI uses this value to verify your bootstrap stack version.

`--watch <BOOLEAN>`

Continuously observe CDK project files, and deploy the specified stacks automatically when changes are detected.

This option implies `--hotswap` by default.

This option has an equivalent CDK CLI command. For more information, see cdk watch.

Examples

Deploy the stack named MyStackName

```
$ cdk deploy MyStackName --app='node bin/main.js'
```

Deploy multiple stacks in an app

Use `cdk list` to list your stacks:

```
$ cdk list
CdkHelloWorldStack
CdkStack2
CdkStack3
```

To deploy all stacks, use the `--all` option:

```
$ cdk deploy --all
```

To choose which stacks to deploy, provide stack names as arguments:

```
$ cdk deploy CdkHelloWorldStack CdkStack3
```

Deploy pipeline stacks

Use `cdk list` to show stack names as paths, showing where they are in the pipeline hierarchy:

```
$ cdk list
PipelineStack
PiplelineStack/Prod
PipelineStack/Prod/MyService
```

Use the `--all` option or the wildcard `*` to deploy all stacks. If you have a hierarchy of stacks as described above, `--all` and `*` will only match stacks on the top level. To match all stacks in the hierarchy, use `**`.

You can combine these patterns. The following deploys all stacks in the `Prod` stage:

```
$ cdk deploy PipelineStack/Prod/**
```

Pass parameters at deployment

Define parameters in your CDK stack. The following is an example that creates a parameter named `TopicNameParam` for an Amazon SNS topic:

```
new sns.Topic(this, 'TopicParameter', {
    topicName: new cdk.CfnParameter(this, 'TopicNameParam').value.toString()
});
```

To provide a parameter value of `parameterized`, run the following:

```
$ cdk deploy --parameters "MyStackName:TopicNameParam=parameterized"
```

You can override parameter values by using the `--force` option. The following is an example of overriding the topic name from a previous deployment:

```
$ cdk deploy --parameters "MyStackName:TopicNameParam=parameterName" --force
```

Write stack outputs to a file after deployment

Define outputs in your CDK stack file. The following is an example that creates an output for a function ARN:

```
const fn = new lambda.Function(this, "fn", {
  handler: "index.handler",
  code: lambda.Code.fromInline(`exports.handler = \${handler.toString()}`),
  runtime: lambda.Runtime.NODEJS_LATEST
});

new cdk.CfnOutput(this, 'FunctionArn', {
  value: fn.functionArn,
});
```

Deploy the stack and write outputs to `outputs.json`:

```
$ cdk deploy --outputs-file outputs.json
```

The following is an example of `outputs.json` after deployment:

```
{
  "MyStack": {
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MyStack-fn5FF616E3-G632ITHSP5HK"
  }
}
```

From this example, the key `FunctionArn` corresponds to the logical ID of the `CfnOutput` instance.

The following is an example of `outputs.json` after deployment when multiple stacks are deployed:

```
{
  "MyStack": {
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:MyStack-fn5FF616E3-G632ITHSP5HK"
  },
  "AnotherStack": {
    "VPCId": "vpc-z0mg270fee16693f"
  }
}
```

Modify the deployment method

To deploy faster, without using change sets, use `--method='direct'`:

```
$ cdk deploy --method='direct'
```

To create a change set but don't deploy, use `--method='prepare-change-set'`. By default, a change set named `cdk-deploy-change-set` will be created. If a previous change set with this name exists, it will be overwritten. If no changes are detected, an empty change set is still created.

You can also name your change set. The following is an example:

```
$ cdk deploy --method='prepare-change-set' --change-set-name='MyChangeSetName'
```

cdk destroy

Delete one or more AWS CDK stacks from your AWS environment.

When you delete a stack, resources in the stack will be destroyed, unless they were configured with a `DeletionPolicy` of `Retain`.

During stack deletion, this command will output progress information similar to `cdk deploy` behavior.

Usage

```
$ cdk destroy <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to delete.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--all <BOOLEAN>`

Destroy all available stacks.

Default value: `false`

`--exclusively, -e <BOOLEAN>`

Only destroy requested stacks and don't include dependencies.

`--force, -f <BOOLEAN>`

Do not ask for confirmation before destroying the stacks.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk destroy` command.

Examples

Delete a stack named MyStackName

```
$ cdk destroy --app='node bin/main.js' <MyStackName>
```

cdk diff

Perform a diff to see infrastructure changes between AWS CDK stacks.

This command is typically used to compare differences between the current state of stacks in your local CDK app against deployed stacks. However, you can also compare a deployed stack with any local AWS CloudFormation template.

Usage

```
$ cdk diff <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to perform a diff.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--change-set <BOOLEAN>`

Specify whether to create a change set to analyze resource replacements.

When `true`, the CDK CLI will create an AWS CloudFormation change set to display the exact changes that will be made to your stack. This output includes whether resources will be updated or replaced. The CDK CLI uses the deploy role instead of the lookup role to perform this action.

When `false`, a quicker, but less-accurate diff is performed by comparing CloudFormation templates. Any change detected to properties that require resource replacement will be displayed as a resource replacement, even if the change is purely cosmetic, like replacing a resource reference with a hard-coded ARN.

Default value: `true`

`--context-lines <NUMBER>`

Number of context lines to include in arbitrary JSON diff rendering.

Default value: `3`

`--exclusively, -e <BOOLEAN>`

Only diff requested stacks and don't include dependencies.

`--fail <BOOLEAN>`

Fail and exit with a code of `1` if differences are detected.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk diff` command.

`--processed <BOOLEAN>`

Specify whether to compare against the template with CloudFormation transforms already processed.

Default value: `false`

`--quiet, -q <BOOLEAN>`

Do not print the CDK stack name and default `cdk diff` message to `stdout` when no changes are detected.

Default value: `false`

`--security-only <BOOLEAN>`

Only diff for broadened security changes.

Default value: `false`

`--strict <BOOLEAN>`

Modify `cdk diff` behavior to be more precise or stringent. When true, the CDK CLI will not filter out `AWS::CDK::Metadata` resources or unreadable non-ASCII characters.

Default value: `false`

`--template <STRING>`

The path to the CloudFormation template to compare a CDK stack with.

Examples

Diff against the currently deployed stack named MyStackName

The CDK CLI uses the following symbols in the diff output:

- `[+]` – Identifies code or resources that will be added if you deploy your changes.
- `[-]` – Identifies code or resources that will be removed if you deploy your changes.
- `[~]` – Identifies a resource or property that will be modified if you deploy your changes.

The following is an example that shows a diff of local changes to a Lambda function:

```
$ cdk diff MyStackName
```

```
start: Building <asset-hash>:<account:Region>
success: Built <asset-hash>:<account:Region>
start: Publishing <asset-hash>:<account:Region>
success: Published <asset-hash>:<account:Region>
Hold on while we create a read-only change set to get a diff with accurate replacement
 information (use --no-change-set to use a less accurate but faster template-only diff)
Stack MyStackName
Resources
[~] AWS::Lambda::Function HelloWorldFunction <resource-logical-ID>
 ## [~] Code
     ## [~] .ZipFile:
         ## [-]
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello World!'),
          };
        };

         ## [+]
        exports.handler = async function(event) {
          return {
            statusCode: 200,
            body: JSON.stringify('Hello from CDK!'),
          };
        };



#  Number of stacks with differences: 1
```

A [~] indicator for resources that will be modified does not always mean a full resource replacement:

- Some resource properties, like Code, will update the resource.
- Some resource properties, like FunctionName, may cause a full resource replacement.

Diff against a specific CloudFormation template

```
$ cdk diff MyStackName --app='node bin/main.js' --template-path='./
MyStackNameTemplate.yaml'
```

Diff a local stack with its deployed stack. don't print to stdout if no changes are detected

```
$ cdk diff MyStackName --app='node bin/main.js' --quiet
```

`cdk docs`

Open AWS CDK documentation in your browser.

Usage

```
$ cdk docs <options>
```

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--browser, -b <STRING>`

The command to use to open the browser, using `%u` as a placeholder for the path of the file to open.

Default value: `open %u`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk docs` command.

Examples

Open AWS CDK documentation in Google Chrome

```
$ cdk docs --browser='chrome %u'
```

`cdk doctor`

Inspect and display useful information about your local AWS CDK project and development environment.

This information can help with troubleshooting CDK issues and should be provided when submitting bug reports.

Usage

```
$ cdk doctor <options>
```

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk doctor` command.

Examples

Simple example of the `cdk doctor` command

```
$ cdk doctor
## CDK Version: 1.0.0 (build e64993a)
## AWS environment variables:
  - AWS_EC2_METADATA_DISABLED = 1
  - AWS_SDK_LOAD_CONFIG = 1
```

`cdk drift`

Detect configuration drift for resources that you define, manage, and deploy using the AWS Cloud Development Kit (AWS CDK). Drift occurs when a stack's actual configuration differs from its expected configuration, which happens when resources are modified outside of AWS CloudFormation.

This command identifies resources that have been modified (for example, through the AWS Console or AWS CLI) by comparing their current state against their expected configuration. These modifications can cause unexpected behavior in your infrastructure.

During drift detection, the CDK CLI will output progress indicators and results, showing:

- Resources that have drifted from their expected configuration.
- The total number of resources with drift.
- A summary indicating whether drift was detected in the stack.

> **Important**
>
> The `cdk drift` and `cdk diff` commands work differently:
>
> - `cdk drift` calls CloudFormation's drift detection operation to compare the actual state of resources in AWS ("reality") against their expected configuration in CloudFormation. Not all AWS resources support drift detection. For a list of supported resources, see Resource type support in the *AWS CloudFormation User Guide*.
> - `cdk diff` compares the CloudFormation template synthesized from your local CDK code against the template of the deployed CloudFormation stack.
>
> Use `cdk drift` when you need to verify if resources have been modified outside of CloudFormation (for example, through the AWS Console or AWS CLI). Use `cdk diff` when you want to preview how your local code changes would affect your infrastructure before deployment.

Usage

```
$ cdk drift <arguments> <options>
```

Arguments

Stack name

The name of the stack that you want to check for drift. The stack must be previously deployed to CloudFormation to perform drift detection.

Type: String

Required: No

If no stack is specified, drift detection will be performed on all stacks defined in your CDK app.

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--fail <BOOLEAN>`

Return with exit code 1 if drift is detected.

Default value: `false`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk drift` command.

Examples

Check drift for a specific stack

```
$ cdk drift MyStackName
```

The command will output results similar to:

```
Stack MyStackName
Modified Resources
[~] AWS::Lambda::Function MyFunction MyLambdaFunc1234ABCD
 ## [~] /Description
     ## [-] My original hello world Lambda function
     ## [+] My drifted hello world Lambda function

1 resource has drifted from their expected configuration

#  Number of resources with drift: 1
```

Check drift when resources have been deleted

The following example shows what the output looks like when resources have been both modified and deleted:

```
Stack MyStackName
```

```
Modified Resources
[~] AWS::Lambda::Function MyFunction MyLambdaFunc1234ABCD
 ## [~] /Description
     ## [-] My original hello world Lambda function
     ## [+] My drifted hello world Lambda function
Deleted Resources
[-] AWS::CloudWatch::Alarm MyAlarm MyCWAlarmABCD1234

2 resources have drifted from their expected configuration

#  Number of resources with drift: 2
```

Check drift with exit code

To have the command return a non-zero exit code if drift is detected:

```
$ cdk drift MyStackName --fail
```

This is useful in CI/CD pipelines to automatically detect and respond to infrastructure drift.

`cdk flags`

View and modify your feature flag configurations for the CDK CLI.

Feature flags control the behavior of the CDK CLI and you can use them to enable or disable specific features. Use the `cdk flags` command to view your current feature flag configurations and modify them as needed.

> **Warning**
>
> The `cdk flags` command is in development for the AWS CDK. The current features of this command are considered production-ready and safe to use. However, the scope of this command and its features are subject to change. Therefore, you must opt in by providing the `unstable=flags` option to use this command.

Usage

```
$ cdk flags <arguments> <options>
```

Arguments

FLAGNAME

The name of the specific feature flag that you want to view or modify.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--set <BOOLEAN>`

Modify a feature flag configuration.

`--all <BOOLEAN>`

Modify or view all feature flags.

`--recommended <BOOLEAN>`

Change flags to recommended states.

`--default <BOOLEAN>`

Change flags to default states.

`--unconfigured <BOOLEAN>`

Modify unconfigured feature flags.

`--value <STRING>`

The value to set the feature flag configuration to.

requiresArg: true

`--safe <BOOLEAN>`

Enable all feature flags that do not impact your application.

`--interactive, -i <BOOLEAN>`

Interactive option for the flags command.

Examples

View feature flag configurations

Run `cdk flags` to view a report of your feature flag configurations that differ from our recommended states. Unconfigured flags are labeled with `<unset>` to indicate that the flag currently has no value. The flags are displayed in the following order:

- Flags set to a value that doesn't match our recommended value
- Flags that you haven't configured

```
$ cdk flags --unstable=flags
Feature Flag                                       Recommended                        User
* @aws-cdk/...                                           true                         false
* @aws-cdk/...                                           true                         false
* @aws-cdk/...                                           true                         <unset>
```

You can also run `cdk flags --all` to see a report of all feature flags in the following order:

- Flags set to a value that matches our recommended value
- Flags set to a value that doesn't match our recommended value
- Flags that you haven't configured

```
$ cdk flags --unstable=flags --all
Feature Flag                                          Recommended                      User
@aws-cdk/...                                             true                         true
* @aws-cdk/...                                           true                         false
* @aws-cdk/...                                           true                         false
* @aws-cdk/...                                           true                         <unset>
```

Modify feature flag values

To modify your feature flags interactively, run `cdk flags --interactive` (or `cdk flags -i`) to view a list of menu options.

To change every feature flag to our recommended value, run `cdk flags --set --recommended --all`. This command keeps your feature flag configuration up-to-date with the latest CDK feature flag configurations. Be aware that running this command can potentially overwrite existing configured values.

```
$ cdk flags --unstable=flags --set --recommended --all
Feature Flag                              Recommended Value            User Value
* @aws-cdk/...                              true                         false
* @aws-cdk/...                              true                         false
* @aws-cdk/...                              true                         <unset>
  Synthesizing...
    Resources
    [~] AWS::S3::Bucket MyBucket
    ## [~] Properties
        ## [~] Encryption
                ...
    Number of stacks with differences: 2
  Do you want to accept these changes? (y/n) y
  Resynthesizing...
```

If you want to preserve the state of your existing configured flags, run `cdk flags --set --recommended --unconfigured`. This option changes only the unconfigured feature flags to our recommended values.

```
$ cdk flags --unstable=flags --set --recommended --unconfigured
Feature Flag                              Recommended Value            User Value
* @aws-cdk/...                              true                         <unset>
* @aws-cdk/...                              true                         <unset>
  Synthesizing...
    Resources
    [~] AWS::S3::Bucket MyBucket
    ## [~] Properties
        ## [~] Encryption
            ## [-] None
            ## [+] ServerSideEncryptionConfiguration:
                    - ...
            ...
```

```
    Number of stacks with differences: 2
  Do you want to accept these changes? (y/n) y
  Resynthesizing...
```

If you want to ensure that any unconfigured feature flags don't interfere with your application, run the `cdk flags --set --default --unconfigured` command to change the unconfigured flags to their default values. For example, if `@aws-cdk/aws-cloudfront:defaultSecurityPolicyTLSv1.2_2021` is unconfigured, a notification displays after running `cdk synth`. However, if you set the flag to its default state (false), it will be configured, turned off, and not impact your application.

```
$ cdk flags --unstable=flags --set --default --unconfigured
Feature Flag                                 Recommended Value            User Value
* @aws-cdk/...                                 true                           <unset>
* @aws-cdk/...                                 true                           <unset>
  Synthesizing...

  Do you want to accept these changes? (y/n) y
  Resynthesizing...
```

Inspect specific feature flags

View more information about a flag

In addition to running `cdk flags` and `cdk flags --all` to view your feature flag configuration, you can also utilize `cdk flags "FLAGNAME"` to inspect a specific feature flag and find out what the flag does. This can be helpful in cases where you want to understand a particular flag and its impact on your application.

```
$ cdk flags --unstable=flags "@aws-cdk/aws-
cloudfront:defaultSecurityPolicyTLSv1.2_2021"
    Description: Enable this feature flag to have cloudfront distributions use the
 security policy TLSv1.2_2021 by default.
    Recommended Value: true
    User Value: true
```

Filter flags by substring

You can also run `cdk flags substring` to view all matching feature flags. If there is only one feature flag that matches that substring, specific details are displayed.

```
$ cdk flags --unstable=flags ebs
@aws-cdk/aws-ec2:ebsDefaultGp3Volume
    Description: When enabled, the default volume type of the EBS volume will be GP3
    Recommended Value: true
    User Value: true
```

If multiple flags match the substring, all matching flags are displayed in a table. If you enter multiple substrings, all matching flags that contain any of the substrings are returned.

```
$ cdk flags --unstable=flags s3 lambda
Feature Flag                                  Recommended                     User
* @aws-cdk/s3...                                true                         false
* @aws-cdk/lambda...                            true                         false
* @aws-cdk/lambda...                            true                         <unset>
```

Modify a specific flag

If you need to modify the value of a flag and want to make sure you're setting it to a correct and supported state, run `cdk flags --set "FLAGNAME" --value="state"`.

```
$ cdk flags --unstable=flags --set "@aws-cdk/aws-
cloudfront:defaultSecurityPolicyTLSv1.2_2021" --value="true"
  Synthesizing...
    Resources
    [~] AWS::CloudFront::Distribution MyDistribution
    ## [~] Properties
        ## [~] DefaultSecurityPolicy
            ## [-] TLSv1.0
            ## [+] TLSv1.2_2021
                    - ...
    Number of stacks with differences: 2
  Do you want to accept these changes? (y/n) y
  Resynthesizing...
```

cdk gc

Use the AWS Cloud Development Kit (AWS CDK) command line interface (CLI) `cdk gc` command to perform garbage collection on unused assets stored in the resources of your bootstrap stack. Use this command to view, manage, and delete assets that you no longer need.

For Amazon Simple Storage Service (Amazon S3) assets, the CDK CLI will check existing AWS CloudFormation templates in the same environment to see if they are referenced. If not referenced, they will be considered unused and eligible for garbage collection actions.

> **Warning**
>
> The `cdk gc` command is in development for the AWS CDK. The current features of this command are considered production-ready and safe to use. However, the scope of this command and its features are subject to change. Therefore, you must opt in by providing the `unstable=gc` option to use this command.

Usage

```
$ cdk gc <arguments> <options>
```

Arguments

AWS environment

The target AWS environment to perform garbage collection actions on.

When providing an environment, use the following format: `aws://<account-id>/<region>`. For example, `aws://<123456789012>/<us-east-1>`.

This argument can be provided multiple times in a single command to perform garbage collection actions on multiple environments.

By default, the CDK CLI will perform garbage collection actions on all environments that you reference in your CDK app or provide as arguments. If you don't provide an environment, the CDK CLI will determine the environment from default sources. These sources include environments that you specify using the `--profile` option, environment variables, or default AWS CLI sources.

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--action <STRING>`

The action that `cdk gc` performs on your assets during a garbage collection.

- `delete-tagged` – Deletes assets that have been tagged with a date within the range of buffer days that you provide, but does not tag newly identified unused assets.
- `full` – Perform all garbage collection actions. This includes deleting assets within the range of buffer days that you provide and tagging newly identified unused assets.
- `print` – Outputs the number of unused assets at the command prompt but doesn't make any actual changes within your AWS environment.
- `tag` – Tags any newly identified unused assets, but doesn't delete any assets within the range of buffer days that you provide.

Accepted values: `delete-tagged`, `full`, `print`, `tag`

Default value: `full`

`--bootstrap-stack-name <STRING>`

The name of the CDK bootstrap stack in your AWS environment. Provide this option if you customized your bootstrap stack name. If you are using the default `CDKToolkit` stack name, you don't have to provide this option.

Default value: `CDKToolkit`

`--confirm <BOOLEAN>`

Specify whether the CDK CLI will request manual confirmation from you before deleting any assets.

Specify `false` to automatically delete assets without prompting you for manual confirmation.

Default value: `true`

`--created-buffer-days <NUMBER>`

The number of days an asset must exist before it is eligible for garbage collection actions.

When you provide a number, assets that have not existed beyond your specified number of days are filtered out from garbage collection actions.

Default value: `1`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk gc` command.

`--rollback-buffer-days <NUMBER>`

The number of days an asset must be tagged as isolated before it is eligible for garbage collection actions.

When you provide a number, the CDK CLI will tag unused assets with the current date instead of deleting them. The CDK CLI will also check if any assets have been tagged during previous runs of the `cdk gc` command. Previously tagged assets that fall within the range of buffer days you provide will be deleted.

Default value: `0`

`--type <STRING>`

The bootstrap resource type within your bootstrap stack to perform garbage collection actions on.

- `all` – Perform garbage collection actions on all bootstrapped resources.
- `ecr` – Perform garbage collection actions on assets in the Amazon Elastic Container Registry (Amazon ECR) repository of your bootstrap stack.
- `s3` – Perform garbage collection actions on assets in the Amazon S3 bucket of your bootstrap stack.

Accepted values: `all`, `ecr`, `s3`

Default value: `all`

`--unstable <STRING>`

Allow the usage of CDK CLI commands that are still in development.

This option is required to use any CDK CLI command that is still in development and subject to change.

This option can be provided multiple times in a single command.

To use `cdk gc`, provide `--unstable=gc`.

Examples

Basic examples

The following example prompts you for manual confirmation to perform default garbage collection actions on assets in the Amazon S3 bucket of your bootstrap stack:

```
$ cdk gc --unstable=gc --type=s3

 #  Garbage Collecting environment aws://<account-id>/<region>...
Found 99 assets to delete based off of the following criteria:
- assets have been isolated for > 0 days
- assets were created > 1 days ago

Delete this batch (yes/no/delete-all)?
```

The following example performs garbage collection actions on a range of assets in the Amazon S3 bucket of your bootstrap stack. This range includes assets that have been previously tagged by `cdk gc` for over 30 days and have been created 10 days or older. This command will prompt for manual confirmation before deleting any assets:

```
$ cdk gc --unstable=gc --type=s3 --rollback-buffer-days=30 --created-buffer-days=10
```

The following example performs the action of deleting previously tagged assets in the Amazon S3 bucket of your bootstrap stack that have been unused for longer than 30 days:

```
$ cdk gc --unstable=gc --type=s3 --action=delete-tagged --rollback-buffer-days=30
```

`cdk import`

Use AWS CloudFormation resource imports to import existing AWS resources into a CDK stack.

With this command, you can take existing resources that were created using other methods and start managing them using the AWS CDK.

When considering moving resources into CDK management, sometimes creating new resources is acceptable, such as with IAM roles, Lambda functions, and event rules. For other resources, such as stateful resources like Amazon S3 buckets and DynamoDB tables, creating new resources can cause impacts to your service. You can use `cdk import` to import existing resources with minimal disruption to your services. For a list of supported AWS resources, see [Resource type support] in the *AWS CloudFormation User Guide*.

To import an existing resource to a CDK stack

1. Run a `cdk diff` to make sure your CDK stack has no pending changes. When performing a `cdk import`, the only changes allowed in an import operation are the addition of new resources being imported.
2. Add constructs for the resources you want to import to your stack. For example, add the following for an Amazon S3 bucket:

   ```
   new s3.Bucket(this, 'ImportedS3Bucket', {});
   ```

   Do not add any other changes. You must also make sure to exactly model the state that the resource currently has. For the bucket example, be sure to include AWS KMS keys, lifecycle policies, and anything else that is relevant about the bucket. Otherwise, subsequent update operations may not do what you expect.
3. Run `cdk import`. If there are multiple stacks in the CDK app, pass a specific stack name as an argument.
4. The CDK CLI will prompt you to pass in the actual names of the resources you are importing. After you provide this information, import will begin.
5. When `cdk import` reports success, the resource will be managed by the CDK. Any subsequent changes in the construct configuration will be reflected on the resource.

This feature currently has the following limitations:

- Importing resources into nested stacks isn't possible.
- There is no check on whether the properties you specify are correct and complete for the imported resource. Try starting a drift detection operation after importing.
- Resources that depend on other resources must all be imported together, or individually, in the right order. Otherwise, the CloudFormation deployment will fail with unresolved references.

- This command uses the deploy role credentials, which is necessary to read the encrypted staging bucket. This requires version 12 of the bootstrap template, which includes the necessary IAM permissions for the deploy role.

Usage

```
$ cdk import <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to import resources to. This argument can be provided multiple times in a single command.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--change-set-name <STRING>`

The name of the CloudFormation change set to create.

`--execute <BOOLEAN>`

Specify whether to execute change set.

Default value: `true`

`--force, -f <BOOLEAN>`

By default, the CDK CLI exits the process if the template diff includes updates or deletions. Specify `true` to override this behavior and always continue with importing.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk import` command.

`--record-resource-mapping, -r <STRING>`

Use this option to generate a mapping of existing physical resources to the CDK resources that will be imported. The mapping will be written to the file path that you provide. No actual import operations will be performed.

`--resource-mapping, -m <STRING>`

Use this option to specify a file that defines your resource mapping. The CDK CLI will use this file to map physical resources to resources for import instead of interactively asking you.

This option can be run from scripts.

`--rollback <BOOLEAN>`

Roll back the stack to stable state on failure.

To specify `false`, you can use `--no-rollback` or `-R`.

Specify `false` to iterate more rapidly. Deployments containing resource replacements will always fail.

Default value: `true`

`--toolkit-stack-name <STRING>`

The name of the CDK Toolkit stack to create.

By default, `cdk bootstrap` deploys a stack named `CDKToolkit` into the specified AWS environment. Use this option to provide a different name for your bootstrap stack.

The CDK CLI uses this value to verify your bootstrap stack version.

cdk init

Create a new AWS CDK project from a template.

Usage

```
$ cdk init <arguments> <options>
```

Arguments

Template type

The CDK template type to initialize a new CDK project from.

- `app` – Template for a CDK application.
- `lib` – Template for an AWS Construct Library.
- `sample-app` – Example CDK application that includes some constructs.

Valid values: `app`, `lib`, `sample-app`

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--generate-only <BOOLEAN>`

Specify this option to generate project files without initiating additional operations such as setting up a git repository, installing dependencies, or compiling the project.

Default value: `false`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk init` command.

`--language, -l <STRING>`

The language to be used for the new project. This option can be configured in the project's `cdk.json` configuration file or at `~/.cdk.json` on your local development machine.

Valid values: `csharp`, `fsharp`, `go`, `java`, `javascript`, `python`, `typescript`

`--list <BOOLEAN>`

List the available template types and languages.

Examples

List the available template types and languages

```
$ cdk init --list
Available templates:
* app: Template for a CDK Application
   ## cdk init app --language=[csharp|fsharp|go|java|javascript|python|typescript]
* lib: Template for a CDK Construct Library
   ## cdk init lib --language=typescript
* sample-app: Example CDK Application with some constructs
   ## cdk init sample-app --language=[csharp|fsharp|go|java|javascript|python|
typescript]
```

Create a new CDK app in TypeScript from the library template

```
$ cdk init lib --language=typescript
```

`cdk list`

List all AWS CDK stacks and their dependencies from a CDK app.

Usage

```
$ cdk list <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to perform this command against.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk list` command.

`--long, -l <BOOLEAN>`

Display AWS environment information for each stack.

Default value: `false`

`--show-dependencies, -d <BOOLEAN>`

Display stack dependency information for each stack.

Default value: `false`

Examples

List all stacks in the CDK app "node bin/main.js"

```
$ cdk list --app='node bin/main.js'
Foo
Bar
Baz
```

List all stacks, including AWS environment details for each stack

```
$ cdk list --app='node bin/main.js' --long
-
    name: Foo
    environment:
        name: 000000000000/bermuda-triangle-1
```

```
        account: '000000000000'
        region: bermuda-triangle-1
-
    name: Bar
    environment:
        name: 111111111111/bermuda-triangle-2
        account: '111111111111'
        region: bermuda-triangle-2
-
    name: Baz
    environment:
        name: 333333333333/bermuda-triangle-3
        account: '333333333333'
        region: bermuda-triangle-3
```

cdk metadata

Display metadata associated with a CDK stack.

Usage

```
$ cdk metadata <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to display metadata for.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk metadata` command.

cdk migrate

Migrate deployed AWS resources, AWS CloudFormation stacks, and CloudFormation templates into a new AWS CDK project.

This command creates a new CDK app that includes a single stack that is named with the value you provide using `--stack-name`. You can configure the migration source using `--from-scan`, `--from-stack`, or `--from-path`.

For more information on using `cdk migrate`, see Migrate existing resources and AWS CloudFormation templates to the AWS CDK.

> **Note**
>
> The `cdk migrate` command is experimental and may have breaking changes in the future.

Usage

```
$ cdk migrate <options>
```

Options

For a list of global options that work with all CDK CLI commands, see Global options.

Required options

`--stack-name <STRING>`

The name of the AWS CloudFormation stack that will be created within the CDK app after migrating.

Required: Yes

Conditional options

`--from-path <PATH>`

The path to the AWS CloudFormation template to migrate. Provide this option to specify a local template.

Required: Conditional. Required if migrating from a local AWS CloudFormation template.

`--from-scan <STRING>`

When migrating deployed resources from an AWS environment, use this option to specify whether a new scan should be started or if the AWS CDK CLI should use the last successful scan.

Required: Conditional. Required when migrating from deployed AWS resources.

Accepted values: `most-recent`, `new`

`--from-stack <BOOLEAN>`

Provide this option to migrate from a deployed AWS CloudFormation stack. Use `--stack-name` to specify the name of the deployed AWS CloudFormation stack.

Required: Conditional. Required if migrating from a deployed AWS CloudFormation stack.

Optional options

`--account <STRING>`

The account to retrieve the AWS CloudFormation stack template from.

Required: No

Default: The AWS CDK CLI obtains account information from default sources.

`--compress <BOOLEAN>`

Provide this option to compress the generated CDK project into a `ZIP` file.

Required: No

`--filter <ARRAY>`

Use when migrating deployed resources from an AWS account and AWS Region. This option specifies a filter to determine which deployed resources to migrate.

This option accepts an array of key-value pairs, where **key** represents the filter type and **value** represents the value to filter.

The following are accepted keys:

- `resource-identifier` – An identifier for the resource. Value can be the resource logical or physical ID. For example, `resource-identifier="ClusterName"`.
- `resource-type-prefix` – The AWS CloudFormation resource type prefix. For example, specify `resource-type-prefix="AWS::DynamoDB::"` to filter all Amazon DynamoDB resources.
- `tag-key` – The key of a resource tag. For example, `tag-key="myTagKey"`.
- `tag-value` – The value of a resource tag. For example, `tag-value="myTagValue"`.

Provide multiple key-value pairs for `AND` conditional logic. The following example filters for any DynamoDB resource that is tagged with `myTagKey` as the tag key: `--filter resource-type-prefix="AWS::DynamoDB::", tag-key="myTagKey"`.

Provide the `--filter` option multiple times in a single command for `OR` conditional logic. The following example filters for any resource that is a DynamoDB resource or is tagged with `myTagKey` as the tag key: `--filter resource-type-prefix="AWS::DynamoDB::" --filter tag-key="myTagKey"`.

Required: No

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk migrate` command.

`--language <STRING>`

The programming language to use for the CDK project created during migration.

Required: No

Valid values: `typescript`, `python`, `java`, `csharp`, `go`.

Default: `typescript`

`--output-path <PATH>`

The output path for the migrated CDK project.

Required: No

Default: By default, the AWS CDK CLI will use your current working directory.

`--region <STRING>`

The AWS Region to retrieve the AWS CloudFormation stack template from.

Required: No

Default: The AWS CDK CLI obtains AWS Region information from default sources.

Examples

Simple example of migrating from a CloudFormation stack

Migrate from a deployed CloudFormation stack in a specific AWS environment using `--from-stack`. Provide `--stack-name` to name your new CDK stack. The following is an example that migrates `myCloudFormationStack` to a new CDK app that is using TypeScript:

```
$ cdk migrate --language typescript --from-stack --stack-name 'myCloudFormationStack'
```

Simple example of migrating from a local CloudFormation template

Migrate from a local JSON or YAML CloudFormation template using `--from-path`. Provide `--stack-name` to name your new CDK stack. The following is an example that creates a new CDK app in TypeScript that includes a `myCloudFormationStack` stack from a local `template.json` file:

```
$ cdk migrate --stack-name "myCloudFormationStack" --language typescript --from-path
 "./template.json"
```

Simple example of migrating from deployed AWS resources

Migrate deployed AWS resources from a specific AWS environment that are not associated with a CloudFormation stack using `--from-scan`. The CDK CLI utilizes the IaC generator service to scan for resources and generate a template. Then, the CDK CLI references the template to create the new CDK app. The following is an example that creates a new CDK app in TypeScript with a new `myCloudFormationStack` stack containing migrated AWS resources:

```
$ cdk migrate --language typescript --from-scan --stack-name "myCloudFormationStack"
```

`cdk notices`

Display notices for your CDK application.

Notices can include important messages regarding security vulnerabilities, regressions, and usage of unsupported versions.

This command displays relevant notices, regardless of whether they have been acknowledged or not. Relevant notices may also appear after every command by default.

You can suppress notices in the following ways:

- Through command options. The following is an example:

  ```
  $ cdk deploy --no-notices
  ```

- Suppress all notices indefinitely through context in the project's `cdk.json` file:

  ```
  {
    "notices": false,
    "context": {
      // ...
    }
  }
  ```

- Acknowledge each notice with the `cdk acknowledge` command.

Usage

```
$ cdk notices <options>
```

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk notices` command.

Examples

Example of a default notice that displays after running the cdk deploy command

```
$ cdk deploy

... # Normal output of the command

NOTICES

16603   Toggling off auto_delete_objects for Bucket empties the bucket

        Overview: If a stack is deployed with an S3 bucket with
                  auto_delete_objects=True, and then re-deployed with
                  auto_delete_objects=False, all the objects in the bucket
                  will be deleted.

        Affected versions: <1.126.0.

        More information at: https://github.com/aws/aws-cdk/issues/16603


17061   Error when building EKS cluster with monocdk import

        Overview: When using monocdk/aws-eks to build a stack containing
                  an EKS cluster, error is thrown about missing
                  lambda-layer-node-proxy-agent/layer/package.json.

        Affected versions: >=1.126.0 <=1.130.0.

        More information at: https://github.com/aws/aws-cdk/issues/17061


If you don’t want to see an notice anymore, use "cdk acknowledge ID". For example, "cdk
 acknowledge 16603"
```

Simple example of running the cdk notices command

```
$ cdk notices

NOTICES

16603   Toggling off auto_delete_objects for Bucket empties the bucket

        Overview: if a stack is deployed with an S3 bucket with
                  auto_delete_objects=True, and then re-deployed with
                  auto_delete_objects=False, all the objects in the bucket
                  will be deleted.

        Affected versions: framework: <=2.15.0 >=2.10.0

        More information at: https://github.com/aws/aws-cdk/issues/16603


If you don’t want to see a notice anymore, use "cdk acknowledge <id>". For example,
 "cdk acknowledge 16603"
```

`cdk rollback`

Use the AWS Cloud Development Kit (AWS CDK) Command Line Interface (CLI) `cdk rollback` command to rollback a failed or paused stack from an AWS CloudFormation deployment to its last stable state.

> **Note**
>
> To use this command, you must have v23 of the bootstrap template deployed to your environment. For more information, see Bootstrap template version history.

When you deploy using `cdk deploy`, the CDK CLI will rollback a failed deployment by default. If you specify `--no-rollback` with `cdk deploy`, you can then use the `cdk rollback` command to manually rollback a failed deployment. This will initiate a rollback to the last stable state of your stack.

Usage

```
$ cdk rollback <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to rollback.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--all <BOOLEAN>`

Rollback all stacks in your CDK app.

Default value: `false`

`--force, -f <BOOLEAN>`

When you use `cdk rollback`, some resources may fail to rollback. Provide this option to force the rollback of all resources. This is the same behavior as providing the `--orphan` option for each resource in your stack.

Default value: `false`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk rollback` command.

`--orphan <LogicalId>`

When you use `cdk rollback`, some resources may fail to rollback. When this happens, you can try to force the rollback of a resource by using this option and providing the logical ID of the resource that failed to rollback.

This option can be provided multiple times in a single command The following is an example:

```
$ cdk rollback MyStack --orphan MyLambdaFunction --orphan MyLambdaFunction2
```

To force the rollback of all resources, use the `--force` option instead.

`--toolkit-stack-name <STRING>`

The name of the existing CDK Toolkit stack that the environment is bootstrapped with.

By default, `cdk bootstrap` deploys a stack named `CDKToolkit` into the specified AWS environment. Use this option to provide a different name for your bootstrap stack.

The CDK CLI uses this value to verify your bootstrap stack version.

`--validate-bootstrap-version <BOOLEAN>`

Specify whether to validate the bootstrap stack version. Provide `--validate-bootstrap-version=false` or `--no-validate-bootsrap-version` to turn off this behavior.

Default value: `true`

cdk synthesize

Synthesize a CDK app to produce a cloud assembly, including an AWS CloudFormation template for each stack.

Cloud assemblies are files that include everything needed to deploy your app to your AWS environment. For example, it includes a CloudFormation template for each stack in your app, and a copy of the file assets or Docker images that you reference in your app.

If your app contains a single stack or if a single stack is provided as an argument, the CloudFormation template will also be displayed in the standard output (`stdout`) in YAML format.

If your app contains multiple stacks, `cdk synth` will synthesize the cloud assembly to `cdk.out`.

Usage

```
$ cdk synthesize <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to synthesize.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--exclusively, -e <BOOLEAN>`

Only synthesize requested stacks, don't include dependencies.

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk synthesize` command.

`--quiet, -q <BOOLEAN>`

Do not output the CloudFormation template to `stdout`.

This option can be configured in the CDK project's `cdk.json` file. The following is an example:

```
{
   "quiet": true
}
```

Default value: `false`

`--validation <BOOLEAN>`

Validate the generated CloudFormation templates after synthesis by performing additional checks.

You can also configure this option through the `validateOnSynth` attribute or `CDK_VALIDATION` environment variable.

Default value: `true`

Examples

Synthesize the cloud assembly for a CDK stack with logial ID MyStackName and output the CloudFormation template to stdout

```
$ cdk synth MyStackName
```

Synthesize the cloud assembly for all stacks in a CDK app and save them into cdk.out

```
$ cdk synth
```

Synthesize the cloud assembly for MyStackName, but don't include dependencies

```
$ cdk synth MyStackName --exclusively
```

Synthesize the cloud assembly for MyStackName, but don't output the CloudFormation template to stdout

```
$ cdk synth MyStackName --quiet
```

`cdk watch`

Continuously watch a local AWS CDK project for changes to perform deployments and hotswaps.

This command is similar to `cdk deploy`, except that it can perform continuous deployments and hotswaps through a single command.

This command is a shortcut for `cdk deploy --watch`.

To end a `cdk watch` session, interrupt the process by pressing `Ctrl+C`.

The files that are observed is determined by the `"watch"` setting in your `cdk.json` file. It has two sub-keys, `"include"` and `"exclude"`, that accepts a single string or an array of strings. Each entry is interpreted as a path relative to the location of the `cdk.json` file. Both `*` and `**` are accepted.

If you create a project using the `cdk init` command, the following default behavior is configured for `cdk watch` in your project's `cdk.json` file:

- `"include"` is set to `"**/*"`, which includes all files and directories in the root of the project.
- `"exclude"` is optional, except for files and folders already ignored by default. This consists of files and directories starting with `.`, the CDK output directory, and the `node_modules` directory.

The minimal setting to configure `watch` is `"watch": {}`.

If either your CDK code or application code requires a build step before deployment, `cdk watch` works with the `"build"` key in the `cdk.json` file.

> **Note**
>
> This command is considered experimental and may have breaking changes in the future.

The same limitations of `cdk deploy --hotswap` applies to `cdk watch`. For more information, see [`cdk deploy --hotswap`](#) .

Usage

```
$ cdk watch <arguments> <options>
```

Arguments

CDK stack ID

The construct ID of the CDK stack from your app to watch.

Type: String

Required: No

Options

For a list of global options that work with all CDK CLI commands, see Global options.

`--build-exclude, -E <ARRAY>`

Do not rebuild asset with the given ID.

This option can be specified multiple times in a single command.

Default value: `[]`

`--change-set-name <STRING>`

The name of the CloudFormation change set to create.

`--concurrency <NUMBER>`

Deploy and hotswap multiple stacks in parallel while accounting for inter-stack dependencies. Use this option to speed up deployments. You must still factor in CloudFormation and other AWS account rate limiting.

Provide a number to specify the maximum number of simultaneous deployments (dependency permitting) to perform.

Default value: `1`

`--exclusively, -e <BOOLEAN>`

Only deploy requested stacks and don't include dependencies.

`--force, -f <BOOLEAN>`

Always deploy stacks, even if templates are identical.

Default value: `false`

`--help, -h <BOOLEAN>`

Show command reference information for the `cdk watch` command.

`--hotswap <BOOLEAN>`

By default, `cdk watch` uses hotswap deployments when possible to update your resources. The CDK CLI will attempt to perform a hotswap deployment and will not fall back to a full CloudFormation deployment if unsuccessful. Any changes detected that cannot be updated through a hotswap are ignored.

Default value: `true`

`--hotswap-fallback <BOOLEAN>`

By default, `cdk watch` attempts to perform hotswap deployments and ignores changes that require CloudFormation deployments. Provide `--hotswap-fallback` to fall back and perform a full CloudFormation deployment if the hotswap deployment is unsuccessful.

`--logs <BOOLEAN>`

By default, `cdk watch` monitors all CloudWatch log groups in your application and streams the log events locally to `stdout`.

Default value: `true`

`--progress <STRING>`

Configure how the CDK CLI displays deployment progress.

- `bar` – Display stack deployment events as a progress bar, with the events for the resource currently being deployed.
- `events` – Provide a complete history, including all CloudFormation events.

You can also configure this option in the project's `cdk.json` file or at `~/.cdk.json` on your local development machine:

```
{
   "progress": "events"
```

```
}
```

Valid values: `bar`, `events`

Default value: `bar`

`--rollback <BOOLEAN>`

During deployment, if a resource fails to be created or updated, the deployment will roll back to the latest stable state before the CDK CLI returns. All changes made up to that point will be undone. Resources that were created will be deleted and updates that were made will be rolled back.

Use `--no-rollback` or `-R` to deactivate this behavior. If a resource fails to be created or updated, the CDK CLI will leave changes made up to that point in place and return. This may be helpful in development environments where you are iterating quickly.

> **Note**
>
> When `false`, deployments that cause resource replacements will always fail. You can only use this value for deployments that update or create new resources.

Default value: `true`

`--toolkit-stack-name <STRING>`

The name of the existing CDK Toolkit stack.

By default, `cdk bootstrap` deploys a stack named `CDKToolkit` into the specified AWS environment. Use this option to provide a different name for your bootstrap stack.

The CDK CLI uses this value to verify your bootstrap stack version.

Examples

Watch a CDK stack with logical ID DevelopmentStack for changes

```
$ cdk watch DevelopmentStack
```

```
Detected change to 'lambda-code/index.js' (type: change). Triggering 'cdk deploy'
DevelopmentStack: deploying...

 #  DevelopmentStack
```

Configure a cdk.json file for what to include and exclude from being watched for changes

```
{
   "app": "mvn -e -q compile exec:java",
   "watch": {
    "include": "src/main/**",
    "exclude": "target/*"
   }
}
```

Build a CDK project using Java before deployment by configuring the cdk.json file

```
{
  "app": "mvn -e -q exec:java",
  "build": "mvn package",
  "watch": {
    "include": "src/main/**",
    "exclude": "target/*"
  }
}
```

AWS CDK reference

This section contains reference information for the AWS Cloud Development Kit (AWS CDK).

API reference

The API Reference contains information about the AWS Construct Library and other APIs provided by the AWS Cloud Development Kit (AWS CDK). Most of the AWS Construct Library is contained in a single package called by its TypeScript name: `aws-cdk-lib`. The actual package name varies by language. Separate versions of the API reference are provided for each supported programming language.

The CDK API reference is organized into sub-modules. There are one or more sub-modules for each AWS service.

Each sub-module has an overview that includes information about how to use its APIs. For example, the S3 overview demonstrates how to set default encryption on an Amazon Simple Storage Service (Amazon S3) bucket.

AWS CDK versioning

This topic provides reference information on how the AWS Cloud Development Kit (AWS CDK) handles versioning.

Version numbers consist of three numeric version parts: *major.minor.patch*, and strictly adhere to the semantic versioning model. This means that breaking changes to stable APIs are limited to major releases.

Minor and patch releases are backward compatible. The code written in a previous version with the same major version can be upgraded to a newer version within the same major version. It will also continue to build and run, producing the same output.

AWS CDK CLI compatibility

Each version of the main AWS CDK library (`aws-cdk-lib`) is compatible with the AWS CDK CLI (`aws-cdk-cli`) version that was current at the time of the CDK library's release. It is also compatible with any newer version of the CDK CLI. Each version of the CDK library maintains this compatibility until the library's *End of Life* date. Therefore, as long as you're using a supported CDK library version, it is always safe to upgrade your CDK CLI version.

Each version of the CDK library may also work with CDK CLI versions older than the version that was current at the time of the CDK library's release. However, this is not guaranteed. Compatibility depends on the CDK library's cloud assembly schema version. The AWS CDK generates a cloud assembly during synthesis and the CDK CLI consumes it for deployment. The schema that defines the format of the cloud assembly is strictly specified and versioned. Therefore, an older version of the CDK CLI would need to support the cloud assembly schema version of the CDK library for them to be compatible.

When the cloud assembly version required by the CDK library is not compatible with the version supported by the CDK CLI, you receive an error message like the following:

```
Cloud assembly schema version mismatch: Maximum schema version supported is 3.0.0, but
 found 4.0.0.
    Please upgrade your CLI in order to interact with this app.
```

To resolve this error, update the CDK CLI to a version compatible with the required cloud assembly version, or to the latest available version. The alternative (downgrading the construct library modules your app uses) is generally not recommended.

Note

For more information on the exact combinations of versions that work together, see the compatibility table in the *aws-cdk-cli GitHub repository*.

AWS Construct Library versioning

The modules in the AWS Construct Library move through various stages as they are developed from concept to mature API. Different stages offer varying degrees of API stability in subsequent versions of the AWS CDK.

APIs in the main AWS CDK library, `aws-cdk-lib`, are stable, and the library is fully semantically versioned. This package includes AWS CloudFormation (L1) constructs for all AWS services and all stable higher-level (L2 and L3) modules. (It also includes the core CDK classes like `App` and `Stack`). APIs will not be removed from this package (though they may be deprecated) until the next major release of the CDK. No individual API will ever have breaking changes. When a breaking change is required, an entirely new API will be added.

New APIs under development for a service already incorporated in aws-cdk-lib are identified using a Beta<N> suffix, where N starts at 1 and is incremented with each breaking change to the new API. Beta<N> APIs are never removed, only deprecated, so your existing app continues to work with newer versions of aws-cdk-lib. When the API is deemed stable, a new API without the Beta<N> suffix is added.

When higher-level (L2 or L3) APIs begin to be developed for an AWS service that previously had only L1 APIs, those APIs are initially distributed in a separate package. The name of such a package has an "Alpha" suffix, and its version matches the first version of aws-cdk-lib it is compatible with, with an alpha sub-version. When the module supports the intended use cases, its APIs are added to aws-cdk-lib.

Language binding stability

Over time, we might add support to the AWS CDK for additional programming languages. Although the API described in all the languages is the same, the way that API is expressed varies by language and might change as the language support evolves. For this reason, language bindings are deemed experimental for a time until they are considered ready for production use.

Language	Stability
TypeScript	Stable
JavaScript	Stable
Python	Stable
Java	Stable
C#/.NET	Stable
Go	Stable

Supported Node.js versions for the AWS CDK

The AWS Cloud Development Kit (AWS CDK) depends on Node.js for its functionality. This includes core CDK components such as the AWS CDK CLI (`aws-cdk-cli`), AWS Construct Library (`aws-cdk-lib`), as well as broader tools such as JSII, Projen, and CDK8s. This page documents the Node.js runtime versions compatible with the AWS CDK, helping you maintain a supported development environment.

The AWS CDK aims to maintain compatibility with actively supported Node.js Long Term Support (LTS) versions. As Node.js versions reach their end-of-life, the AWS CDK also phases out support for these versions to ensure security, performance, and access to the latest features.

Keeping your CDK components and tools on supported Node.js versions ensures that you receive security updates, bug fixes, and access to the latest CDK features and improvements.

Node.js version support timeline

The AWS CDK supports Node.js versions beyond their official End-of-Life (EOL) dates to give you time to upgrade your environment. This support extends for 6 months (for example, when Node.js versions reach EOL in April, support continues until October). The version support timeline table below documents specific CDK support end dates as they are determined. This approach gives you a defined window to plan and implement version upgrades while maintaining access to CDK functionality and updates.

Node.js version	Node EOL date	CDK support status
22.x	2027-04-30	Support expected until October 2027.
20.x	2026-04-30	Support expected until October 2026.
18.x	2025-05-30	Support ends 2025-11-30.
16.x	2023-09-11	Support ends 2025-05-30.
14.x	2023-04-30	Support ends 2025-05-30.

AWS CDK video resources

Enjoy these videos presented by members of the AWS CDK team.

> **Note**
>
> Since the AWS CDK is always evolving, some of the code presented in these videos may not work quite the same way it did when the video was recorded. This is especially true for modules that were under active development at the time. It's also possible that we've since added a better way to achieve the same result. Consult this Developer Guide and the AWS CDK API Reference for up-to-date information.

Infrastructure *is* Code with the AWS CDK

Deep dive into AWS Cloud Development Kit (AWS CDK)

Contributing to the AWS Construct Library

Faster deployments with CDK Pipelines

How to contribute to the AWS CDK using GitPod

AWS CDK tutorials and examples

This section contains tutorials and examples for the AWS Cloud Development Kit (AWS CDK).

Tutorials

Tutorials provide you with step-by-step instructions that you can follow to implement a task using the AWS CDK.

Examples

Examples show and explain how specific tasks can be implemented using the AWS CDK. They do not provide direct step-by-step instructions for you to follow.

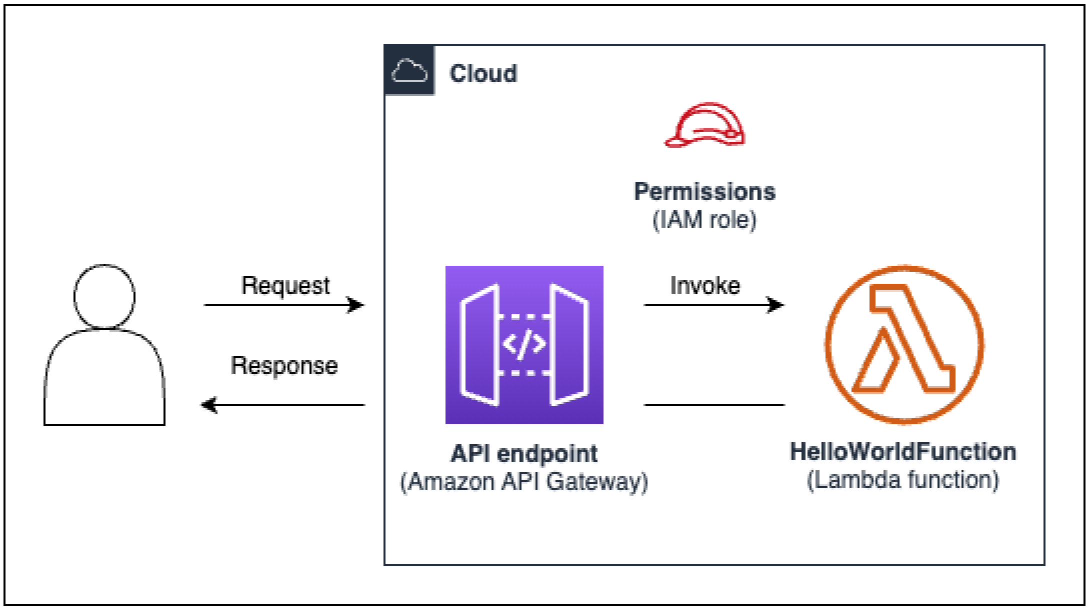
For more examples of AWS CDK stacks and apps in your favorite supported programming language, see the AWS CDK Examples repository on GitHub.

Tutorial: Create a serverless Hello World application

In this tutorial, you use the AWS Cloud Development Kit (AWS CDK) to create a simple serverless `Hello World` application that implements a basic API backend consisting of the following:

- **Amazon API Gateway REST API** – Provides an HTTP endpoint that is used to invoke your function through an HTTP GET request.
- **AWS Lambda function** – Function that returns a `Hello World!` message when invoked with the HTTP endpoint.
- **Integrations and permissions** – Configuration details and permissions for your resources to interact with one another and perform actions, such as writing logs to Amazon CloudWatch.

The following diagram shows the components of this application:



For this tutorial, you will create and interact with your application in the following steps:

1. Create an AWS CDK project.
2. Define a Lambda function and API Gateway REST API using L2 constructs from the AWS Construct Library.
3. Deploy your application to the AWS Cloud.
4. Interact with your application in the AWS Cloud.
5. Delete the sample application from the AWS Cloud.

Prerequisites

Before starting this tutorial, complete the following:

- Create an AWS account and have the AWS Command Line Interface (AWS CLI) installed and configured.
- Install Node.js and `npm`.
- Install the CDK Toolkit globally, using `npm install -g aws-cdk`.

For more information, see Getting started with the AWS CDK.

We also recommend a basic understanding of the following:

- What is the AWS CDK? for a basic introduction to the AWS CDK.
- Learn AWS CDK core concepts for an overview of core concepts of the AWS CDK.

Step 1: Create a CDK project

In this step, you create a new CDK project using the AWS CDK CLI `cdk init` command.

To create a CDK project

1. From a starting directory of your choice, create and navigate to a project directory named `cdk-hello-world` on your machine:

```
$ mkdir cdk-hello-world && cd cdk-hello-world
```

2. Use the `cdk init` command to create a new project in your preferred programming language:

Example

TypeScript

```
$ cdk init --language typescript
```

Install AWS CDK libraries:

```
$ npm install aws-cdk-lib constructs
```

JavaScript

```
$ cdk init --language javascript
```

Install AWS CDK libraries:

```
$ npm install aws-cdk-lib constructs
```

Python

```
$ cdk init --language python
```

Activate the virtual environment:

```
$ source .venv/bin/activate # On Windows, run '.\venv\Scripts\activate' instead
```

Install AWS CDK libraries and project dependencies:

```
(.venv)$ python3 -m pip install -r requirements.txt
```

Java

```
$ cdk init --language java
```

Install AWS CDK libraries and project dependencies:

```
$ mvn package
```

C#

```
$ cdk init --language csharp
```

Install AWS CDK libraries and project dependencies:

```
$ dotnet restore src
```

Go

```
$ cdk init --language go
```

Install project dependencies:

```
$ go get github.com/aws/aws-cdk-go/awscdk/v2
$ go get github.com/aws/aws-cdk-go/awscdk/v2/awslambda
$ go get github.com/aws/aws-cdk-go/awscdk/v2/awsapigateway
$ go mod tidy
```

The CDK CLI creates a project with the following structure:

Example

TypeScript

```
cdk-hello-world
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### cdk-hello-world.ts
### cdk.json
### jest.config.js
### lib
#   ### cdk-hello-world-stack.ts
### node_modules
### package-lock.json
### package.json
### test
#   ### cdk-hello-world.test.ts
### tsconfig.json
```

JavaScript

```
cdk-hello-world
### .git
### .gitignore
### .npmignore
### README.md
### bin
#   ### cdk-hello-world.js
### cdk.json
### jest.config.js
### lib
#   ### cdk-hello-world-stack.js
### node_modules
### package-lock.json
### package.json
### test
    ### cdk-hello-world.test.js
```

Python

```
cdk-hello-world
### .git
### .gitignore
### .venv
### README.md
### app.py
### cdk.json
### cdk_hello_world
#   ### __init__.py
#   ### cdk_hello_world_stack.py
### requirements-dev.txt
### requirements.txt
### source.bat
### tests
```

Java

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk.json
### pom.xml
### src
#   ### main
#   #   ### java
#   #       ### com
#   #           ### myorg
#   #               ### CdkHelloWorldApp.java
#   #               ### CdkHelloWorldStack.java
### target
```

C#

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk.json
### src
```

```
    ### CdkHelloWorld
    #   ### CdkHelloWorld.csproj
    #   ### CdkHelloWorldStack.cs
    #   ### GlobalSuppressions.cs
    #   ### Program.cs
    ### CdkHelloWorld.sln
```

Go

```
cdk-hello-world
### .git
### .gitignore
### README.md
### cdk-hello-world.go
### cdk-hello-world_test.go
### cdk.json
### go.mod
### go.sum
```

The CDK CLI automatically creates a CDK app that contains a single stack. The CDK app instance is created from the App class. The following is a portion of your CDK application file:

Example

TypeScript

Located in `bin/cdk-hello-world.ts`:

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { CdkHelloWorldStack } from '../lib/cdk-hello-world-stack';

const app = new cdk.App();
new CdkHelloWorldStack(app, 'CdkHelloWorldStack', {
});
```

JavaScript

Located in `bin/cdk-hello-world.js`:

```
#!/usr/bin/env node
```

```
const cdk = require('aws-cdk-lib');
const { CdkHelloWorldStack } = require('../lib/cdk-hello-world-stack');
const app = new cdk.App();
new CdkHelloWorldStack(app, 'CdkHelloWorldStack', {
});
```

Python

Located in `app.py`:

```
#!/usr/bin/env python3
import os
import aws_cdk as cdk
from cdk_hello_world.cdk_hello_world_stack import CdkHelloWorldStack

app = cdk.App()
CdkHelloWorldStack(app, "CdkHelloWorldStack",)
app.synth()
```

Java

Located in `src/main/java/…/CdkHelloWorldApp.java` :

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

import java.util.Arrays;

public class JavaApp {
    public static void main(final String[] args) {
        App app = new App();

        new JavaStack(app, "JavaStack", StackProps.builder()
                .build());

        app.synth();
    }
}
```

C#

Located in `src/CdkHelloWorld/Program.cs`:

```
using Amazon.CDK;
using System;
using System.Collections.Generic;
using System.Linq;

namespace CdkHelloWorld
{
    sealed class Program
    {
        public static void Main(string[] args)
        {
            var app = new App();
            new CdkHelloWorldStack(app, "CdkHelloWorldStack", new StackProps
            {

            });
            app.Synth();
        }
    }
}
```

Go

Located in `cdk-hello-world.go`:

```
package main
import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)

// ...

func main() {
    defer jsii.Close()
    app := awscdk.NewApp(nil)
    NewCdkHelloWorldStack(app, "CdkHelloWorldStack", &CdkHelloWorldStackProps{
        awscdk.StackProps{
```

```
            Env: env(),
        },
    })
    app.Synth(nil)
}

func env() *awscdk.Environment {
    return nil
}
```

Step 2: Create your Lambda function

Within your CDK project, create a `lambda` directory that includes a new `hello.js` file. The following is an example:

Example

TypeScript

From the root of your project, run the following:

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

The following should now be added to your CDK project:

```
cdk-hello-world
### lambda
    ### hello.js
```

JavaScript

From the root of your project, run the following:

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

The following should now be added to your CDK project:

```
cdk-hello-world
```

```
### lambda
    ### hello.js
```

Python

From the root of your project, run the following:

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

The following should now be added to your CDK project:

```
cdk-hello-world
### lambda
    ### hello.js
```

Java

From the root of your project, run the following:

```
$ mkdir -p src/main/resources/lambda
$ cd src/main/resources/lambda
$ touch hello.js
```

The following should now be added to your CDK project:

```
cdk-hello-world
### src
    ### main
        ###resources
            ###lambda
                ###hello.js
```

C#

From the root of your project, run the following:

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

The following should now be added to your CDK project:

```
cdk-hello-world
### lambda
    ### hello.js
```

Go

From the root of your project, run the following:

```
$ mkdir lambda && cd lambda
$ touch hello.js
```

The following should now be added to your CDK project:

```
cdk-hello-world
### lambda
    ### hello.js
```

Note

To keep this tutorial simple, we use a JavaScript Lambda function for all CDK programming languages.

Define your Lambda function by adding the following to the newly created file:

```
exports.handler = async (event) => {
    return {
        statusCode: 200,
        headers: { "Content-Type": "text/plain" },
        body: JSON.stringify({ message: "Hello, World!" }),
    };
};
```

Step 3: Define your constructs

In this step, you will define your Lambda and API Gateway resources using AWS CDK L2 constructs.

Open the project file that defines your CDK stack. You will modify this file to define your constructs. The following is an example of your starting stack file:

Example

TypeScript

Located in `lib/cdk-hello-world-stack.ts`:

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

export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Your constructs will go here

  }
}
```

JavaScript

Located in `lib/cdk-hello-world-stack.js`:

```
const { Stack, Duration } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigateway = require('aws-cdk-lib/aws-apigateway');

class CdkHelloWorldStack extends Stack {

  constructor(scope, id, props) {
    super(scope, id, props);

    // Your constructs will go here

  }
}

module.exports = { CdkHelloWorldStack }
```

Python

Located in `cdk_hello_world/cdk_hello_world_stack.py`:

```
from aws_cdk import Stack
```

```
from constructs import Construct

class CdkHelloWorldStack(Stack):
    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

          // Your constructs will go here
```

Java

Located in `src/main/java/.../CdkHelloWorldStack.java` :

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Your constructs will go here
    }
}
```

C#

Located in `src/CdkHelloWorld/CdkHelloWorldStack.cs`:

```
using Amazon.CDK;
using Constructs;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
```

```
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Your constructs will go here
        }
    }
}
```

Go

Located at `cdk-hello-world.go`:

```
package main

import (
    "github.com/aws/aws-cdk-go/awscdk/v2"
    "github.com/aws/constructs-go/constructs/v10"
    "github.com/aws/jsii-runtime-go"
)

type CdkHelloWorldStackProps struct {
    awscdk.StackProps
}

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Your constructs will go here

    return stack
}

func main() {

    // ...

}

func env() *awscdk.Environment {
```

```
    return nil

}
```

In this file, the AWS CDK is doing the following:

- Your CDK stack instance is instantiated from the `Stack` class.
- The `Constructs` base class is imported and provided as the scope or parent of your stack instance.

Define your Lambda function resource

To define your Lambda function resource, you import and use the `aws-lambda` L2 construct from the AWS Construct Library.

Modify your stack file as follows:

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// Import Lambda L2 construct
import * as lambda from 'aws-cdk-lib/aws-lambda';

export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Define the Lambda function resource
    const helloWorldFunction = new lambda.Function(this, 'HelloWorldFunction', {
      runtime: lambda.Runtime.NODEJS_20_X, // Choose any supported Node.js runtime
      code: lambda.Code.fromAsset('lambda'), // Points to the lambda directory
      handler: 'hello.handler', // Points to the 'hello' file in the lambda
 directory
    });
  }
}
```

JavaScript

```
const { Stack, Duration } = require('aws-cdk-lib');
// Import Lambda L2 construct
const lambda = require('aws-cdk-lib/aws-lambda');


class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    const helloWorldFunction = new lambda.Function(this, 'HelloWorldFunction', {
      runtime: lambda.Runtime.NODEJS_20_X, // Choose any supported Node.js runtime
      code: lambda.Code.fromAsset('lambda'), // Points to the lambda directory
      handler: 'hello.handler', // Points to the 'hello' file in the lambda
 directory
    });
  }
}

module.exports = { CdkHelloWorldStack }
```

Python

```
from aws_cdk import (
    Stack,
    # Import Lambda L2 construct
    aws_lambda as _lambda,
)
# ...

class CdkHelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Define the Lambda function resource
        hello_world_function = _lambda.Function(
            self,
            "HelloWorldFunction",
            runtime = _lambda.Runtime.NODEJS_20_X, # Choose any supported Node.js
 runtime
```

```
            code = _lambda.Code.from_asset("lambda"), # Points to the lambda
 directory
            handler = "hello.handler", # Points to the 'hello' file in the lambda
 directory
        )
```

> **Note**
>
> We import the `aws_lambda` module as `\_lambda` because `lambda` is a build-in identifier in Python.

Java

```
// ...
// Import Lambda L2 construct
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.Runtime;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // Define the Lambda function resource
        Function helloWorldFunction = Function.Builder.create(this,
 "HelloWorldFunction")
                .runtime(Runtime.NODEJS_20_X)  // Choose any supported Node.js
 runtime
                .code(Code.fromAsset("src/main/resources/lambda")) // Points to the
 lambda directory
                .handler("hello.handler")  // Points to the 'hello' file in the
 lambda directory
                .build();
    }
}
```

C#

```
// ...
// Import Lambda L2 construct
using Amazon.CDK.AWS.Lambda;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            // Define the Lambda function resource
            var helloWorldFunction = new Function(this, "HelloWorldFunction", new
 FunctionProps
            {
                Runtime = Runtime.NODEJS_20_X, // Choose any supported Node.js
 runtime
                Code = Code.FromAsset("lambda"), // Points to the lambda directory
                Handler = "hello.handler" // Points to the 'hello' file in the
 lambda directory
            });
        }
    }
}
```

Go

```
package main

import (
    // ...
    // Import Lambda L2 construct
    "github.com/aws/aws-cdk-go/awscdk/v2/awslambda"
    // Import S3 assets construct
    "github.com/aws/aws-cdk-go/awscdk/v2/awss3assets"
    // ...
)

// ...
```

```
func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Define the Lambda function resource
    helloWorldFunction := awslambda.NewFunction(stack,
 jsii.String("HelloWorldFunction"), &awslambda.FunctionProps{
        Runtime: awslambda.Runtime_NODEJS_20_X(), // Choose any supported Node.js
 runtime
        Code:    awslambda.Code_FromAsset(jsii.String("lambda"),
 &awss3assets.AssetOptions{}), // Points to the lambda directory
        Handler: jsii.String("hello.handler"), // Points to the 'hello' file in the
 lambda directory
    })

    return stack
}

// ...
```

Here, you create a Lambda function resource and define the following properties:

- `runtime` – The environment the function runs in. Here, we use Node.js version 20.x.
- `code` – The path to the function code on your local machine.
- `handler` – The name of the specific file that contains your function code.

Define your API Gateway REST API resource

To define your API Gateway REST API resource, you import and use the `aws-apigateway` L2 construct from the AWS Construct Library.

Modify your stack file as follows:

Example

TypeScript

```
// ...
//Import API Gateway L2 construct
import * as apigateway from 'aws-cdk-lib/aws-apigateway';


export class CdkHelloWorldStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // ...

    // Define the API Gateway resource
    const api = new apigateway.LambdaRestApi(this, 'HelloWorldApi', {
      handler: helloWorldFunction,
      proxy: false,
    });

    // Define the '/hello' resource with a GET method
    const helloResource = api.root.addResource('hello');
    helloResource.addMethod('GET');
  }
}
```

JavaScript

```
// ...
// Import API Gateway L2 construct
const apigateway = require('aws-cdk-lib/aws-apigateway');


class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // ...

    // Define the API Gateway resource
    const api = new apigateway.LambdaRestApi(this, 'HelloWorldApi', {
      handler: helloWorldFunction,
```

```
      proxy: false,
    });

    // Define the '/hello' resource with a GET method
    const helloResource = api.root.addResource('hello');
    helloResource.addMethod('GET');
  };
};

// ...
```

Python

```
from aws_cdk import (
    # ...
    # Import API Gateway L2 construct
    aws_apigateway as apigateway,
)
from constructs import Construct

class CdkHelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # ...

        # Define the API Gateway resource
        api = apigateway.LambdaRestApi(
            self,
            "HelloWorldApi",
            handler = hello_world_function,
            proxy = False,
        )

        # Define the '/hello' resource with a GET method
        hello_resource = api.root.add_resource("hello")
        hello_resource.add_method("GET")
```

Java

```
// ...
// Import API Gateway L2 construct
```

```
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.apigateway.Resource;

public class CdkHelloWorldStack extends Stack {
    public CdkHelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkHelloWorldStack(final Construct scope, final String id, final
 StackProps props) {
        super(scope, id, props);

        // ...

        // Define the API Gateway resource
        LambdaRestApi api = LambdaRestApi.Builder.create(this, "HelloWorldApi")
                .handler(helloWorldFunction)
                .proxy(false) // Turn off default proxy integration
                .build();

        // Define the '/hello' resource and its GET method
        Resource helloResource = api.getRoot().addResource("hello");
        helloResource.addMethod("GET");
    }
}
```

C#

```
// ...
// Import API Gateway L2 construct
using Amazon.CDK.AWS.APIGateway;

namespace CdkHelloWorld
{
    public class CdkHelloWorldStack : Stack
    {
        internal CdkHelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
           // ...

            // Define the API Gateway resource
```

```
            var api = new LambdaRestApi(this, "HelloWorldApi", new
 LambdaRestApiProps
            {
                Handler = helloWorldFunction,
                Proxy = false
            });

            // Add a '/hello' resource with a GET method
            var helloResource = api.Root.AddResource("hello");
            helloResource.AddMethod("GET");
        }
    }
}
```

Go

```
// ...

import (
    // ...
    // Import Api Gateway L2 construct
    "github.com/aws/aws-cdk-go/awscdk/v2/awsapigateway"
    // ...
)

// ...

func NewCdkHelloWorldStack(scope constructs.Construct, id string, props
 *CdkHelloWorldStackProps) awscdk.Stack {
    var sprops awscdk.StackProps
    if props != nil {
        sprops = props.StackProps
    }
    stack := awscdk.NewStack(scope, &id, &sprops)

    // Define the Lambda function resource
    // ...

    // Define the API Gateway resource
    api := awsapigateway.NewLambdaRestApi(stack, jsii.String("HelloWorldApi"),
 &awsapigateway.LambdaRestApiProps{
        Handler: helloWorldFunction,
        Proxy: jsii.Bool(false),
```

```
    })

    // Add a '/hello' resource with a GET method
    helloResource := api.Root().AddResource(jsii.String("hello"),
 &awsapigateway.ResourceOptions{})
    helloResource.AddMethod(jsii.String("GET"),
 awsapigateway.NewLambdaIntegration(helloWorldFunction,
 &awsapigateway.LambdaIntegrationOptions{}), &awsapigateway.MethodOptions{})

    return stack
}

// ...
```

Here, you create an API Gateway REST API resource, along with the following:

- An integration between the REST API and your Lambda function, allowing the API to invoke your function. This includes the creation of a Lambda permission resource.
- A new resource or path named `hello` that is added to the root of the API endpoint. This creates a new endpoint that adds `/hello` to your base URL.
- A GET method for the `hello` resource. When a GET request is sent to the `/hello` endpoint, the Lambda function is invoked and its response is returned.

Step 4: Prepare your application for deployment

In this step you prepare your application for deployment by building, if necessary, and performing basic validation with the AWS CDK CLI `cdk synth` command.

If necessary, build your application:

Example

TypeScript

From the root of your project, run the following:

```
$ npm run build
```

JavaScript

Building is not required.

Python

Building is not required.

Java

From the root of your project, run the following:

```
$ mvn package
```

C#

From the root of your project, run the following:

```
$ dotnet build src
```

Go

Building is not required.

Run `cdk synth` to synthesize an AWS CloudFormation template from your CDK code. By using L2 constructs, many of the configuration details required by AWS CloudFormation to facilitate the interaction between your Lambda function and REST API are provisioned for you by the AWS CDK.

From the root of your project, run the following:

```
$ cdk synth
```

Note

If you receive an error like the following, verify that you are in the `cdk-hello-world` directory and try again:

```
--app is required either in command-line, in cdk.json or in ~/.cdk.json
```

If successful, the AWS CDK CLI will output the AWS CloudFormation template in `YAML` format at the command prompt. A `JSON` formatted template is also saved in the `cdk.out` directory.

The following is an example output of the AWS CloudFormation template:

AWS CloudFormation template

```
Resources:
  HelloWorldFunctionServiceRoleunique-identifier:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldFunction/ServiceRole/Resource
  HelloWorldFunctionunique-identifier:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket:
          Fn::Sub: cdk-unique-identifier-assets-${AWS::AccountId}-${AWS::Region}
        S3Key: unique-identifier.zip
      Handler: hello.handler
      Role:
        Fn::GetAtt:
          - HelloWorldFunctionServiceRoleunique-identifier
          - Arn
      Runtime: nodejs20.x
    DependsOn:
      - HelloWorldFunctionServiceRoleunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldFunction/Resource
      aws:asset:path: asset.unique-identifier
```

```
      aws:asset:is-bundled: false
      aws:asset:property: Code
  HelloWorldApiunique-identifier:
    Type: AWS::ApiGateway::RestApi
    Properties:
      Name: HelloWorldApi
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Resource
  HelloWorldApiDeploymentunique-identifier:
    Type: AWS::ApiGateway::Deployment
    Properties:
      Description: Automatically created by the RestApi construct
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    DependsOn:
      - HelloWorldApihelloGETunique-identifier
      - HelloWorldApihellounique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Deployment/Resource
  HelloWorldApiDeploymentStageprod012345ABC:
    Type: AWS::ApiGateway::Stage
    Properties:
      DeploymentId:
        Ref: HelloWorldApiDeploymentunique-identifier
      RestApiId:
        Ref: HelloWorldApiunique-identifier
      StageName: prod
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/DeploymentStage.prod/Resource
  HelloWorldApihellounique-identifier:
    Type: AWS::ApiGateway::Resource
    Properties:
      ParentId:
        Fn::GetAtt:
          - HelloWorldApiunique-identifier
          - RootResourceId
      PathPart: hello
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/Resource
  HelloWorldApihelloGETApiPermissionCdkHelloWorldStackHelloWorldApiunique-identifier:
    Type: AWS::Lambda::Permission
    Properties:
```

```
      Action: lambda:InvokeFunction
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunctionunique-identifier
          - Arn
      Principal: apigateway.amazonaws.com
      SourceArn:
        Fn::Join:
          - ""
          - - "arn:"
            - Ref: AWS::Partition
            - ":execute-api:"
            - Ref: AWS::Region
            - ":"
            - Ref: AWS::AccountId
            - ":"
            - Ref: HelloWorldApi9E278160
            - /
            - Ref: HelloWorldApiDeploymentStageprodunique-identifier
            - /GET/hello
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/
ApiPermission.CdkHelloWorldStackHelloWorldApiunique-identifier.GET..hello
  HelloWorldApihelloGETApiPermissionTestCdkHelloWorldStackHelloWorldApiunique-
identifier:
    Type: AWS::Lambda::Permission
    Properties:
      Action: lambda:InvokeFunction
      FunctionName:
        Fn::GetAtt:
          - HelloWorldFunctionunique-identifier
          - Arn
      Principal: apigateway.amazonaws.com
      SourceArn:
        Fn::Join:
          - ""
          - - "arn:"
            - Ref: AWS::Partition
            - ":execute-api:"
            - Ref: AWS::Region
            - ":"
            - Ref: AWS::AccountId
            - ":"
            - Ref: HelloWorldApiunique-identifier
```

```
            - /test-invoke-stage/GET/hello
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/
ApiPermission.Test.CdkHelloWorldStackHelloWorldApiunique-identifier.GET..hello
  HelloWorldApihelloGETunique-identifier:
    Type: AWS::ApiGateway::Method
    Properties:
      AuthorizationType: NONE
      HttpMethod: GET
      Integration:
        IntegrationHttpMethod: POST
        Type: AWS_PROXY
        Uri:
          Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - ":apigateway:"
              - Ref: AWS::Region
              - :lambda:path/2015-03-31/functions/
              - Fn::GetAtt:
                  - HelloWorldFunctionunique-identifier
                  - Arn
              - /invocations
      ResourceId:
        Ref: HelloWorldApihellounique-identifier
      RestApiId:
        Ref: HelloWorldApiunique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/HelloWorldApi/Default/hello/GET/Resource
  CDKMetadata:
    Type: AWS::CDK::Metadata
    Properties:
      Analytics: v2:deflate64:unique-identifier
    Metadata:
      aws:cdk:path: CdkHelloWorldStack/CDKMetadata/Default
    Condition: CDKMetadataAvailable
Outputs:
  HelloWorldApiEndpointunique-identifier:
    Value:
      Fn::Join:
        - ""
        - - https://
          - Ref: HelloWorldApiunique-identifier
```

```
          - .execute-api.
          - Ref: AWS::Region
          - "."
          - Ref: AWS::URLSuffix
          - /
          - Ref: HelloWorldApiDeploymentStageprodunique-identifier
          - /
Conditions:
  CDKMetadataAvailable:
    Fn::Or:
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - af-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-east-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-northeast-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-northeast-2
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-southeast-1
          - Fn::Equals:
              - Ref: AWS::Region
              - ap-southeast-2
          - Fn::Equals:
              - Ref: AWS::Region
              - ca-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - cn-north-1
          - Fn::Equals:
              - Ref: AWS::Region
              - cn-northwest-1
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
```

```
              - eu-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-north-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-1
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-2
          - Fn::Equals:
              - Ref: AWS::Region
              - eu-west-3
          - Fn::Equals:
              - Ref: AWS::Region
              - il-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - me-central-1
          - Fn::Equals:
              - Ref: AWS::Region
              - me-south-1
          - Fn::Equals:
              - Ref: AWS::Region
              - sa-east-1
      - Fn::Or:
          - Fn::Equals:
              - Ref: AWS::Region
              - us-east-1
          - Fn::Equals:
              - Ref: AWS::Region
              - us-east-2
          - Fn::Equals:
              - Ref: AWS::Region
              - us-west-1
          - Fn::Equals:
              - Ref: AWS::Region
              - us-west-2
Parameters:
  BootstrapVersion:
    Type: AWS::SSM::Parameter::Value<String>
```

```
    Default: /cdk-bootstrap/hnb659fds/version
    Description: Version of the CDK Bootstrap resources in this environment,
 automatically retrieved from SSM Parameter Store. [cdk:skip]
Rules:
  CheckBootstrapVersion:
    Assertions:
      - Assert:
          Fn::Not:
            - Fn::Contains:
                - - "1"
                  - "2"
                  - "3"
                  - "4"
                  - "5"
                - Ref: BootstrapVersion
        AssertDescription: CDK bootstrap stack version 6 required. Please run 'cdk
 bootstrap' with a recent version of the CDK CLI.
```

By using L2 constructs, you define a few properties to configure your resources and use helper methods to integrate them together. The AWS CDK configures the majority of your AWS CloudFormation resources and properties required to provision your application.

Step 5: Deploy your application

In this step, you use the AWS CDK CLI `cdk deploy` command to deploy your application. The AWS CDK works with the AWS CloudFormation service to provision your resources.

> **Important**
>
> You must perform a one-time bootstrapping of your AWS environment before deployment. For instructions, see Bootstrap your environment for use with the AWS CDK.

From the root of your project, run the following. Confirm changes if prompted:

```
$ cdk deploy

#  Synthesis time: 2.44s

...
```

```
Do you wish to deploy these changes (y/n)? <y>
```

When deployment completes, the AWS CDK CLI will output your endpoint URL. Copy this URL for the next step. The following is an example:

```
...
#  HelloWorldStack

#  Deployment time: 45.37s

Outputs:
HelloWorldStack.HelloWorldApiEndpointunique-identifier = https://<api-id>.execute-api.<region>.amazonaws.com/prod/
Stack ARN:
arn:aws:cloudformation:region:account-id:stack/HelloWorldStack/unique-identifier
...
```

Step 6: Interact with your application

In this step, you initiate a GET request to your API endpoint and receive your Lambda function response.

Locate your endpoint URL from the previous step and add the `/hello` path. Then, using your browser or command prompt, send a GET request to your endpoint. The following is an example:

```
$ curl https://<api-id>.execute-api.<region>.amazonaws.com/prod/hello
{"message":"Hello World!"}%
```

Congratulations, you have successfully created, deployed, and interacted with your application using the AWS CDK!

Step 7: Delete your application

In this step, you use the AWS CDK CLI to delete your application from the AWS Cloud.

To delete your application, run `cdk destroy`. When prompted, confirm your request to delete the application:

```
$ cdk destroy
Are you sure you want to delete: CdkHelloWorldStack (y/n)? y
```

```
CdkHelloWorldStack: destroying... [1/1]
...
 #  CdkHelloWorldStack: destroyed
```

Troubleshooting

Error: {"message": "Internal server error"}%

When invoking the deployed Lambda function, you receive this error. This error could occur for multiple reasons.

To troubleshoot further

Use the AWS CLI to invoke your Lambda function.

1. Modify your stack file to capture the output value of your deployed Lambda function name. The following is an example:

```
...

class CdkHelloWorldStack extends Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Define the Lambda function resource
    // ...

    new CfnOutput(this, 'HelloWorldFunctionName', {
      value: helloWorldFunction.functionName,
      description: 'JavaScript Lambda function'
    });

    // Define the API Gateway resource
    // ...
  }
}
```

2. Deploy your application again. The AWS CDK CLI will output the value of your deployed Lambda function name:

```
$ cdk deploy

#  Synthesis time: 0.29s
```

```
...
 #  CdkHelloWorldStack

#  Deployment time: 20.36s

Outputs:
...
CdkHelloWorldStack.HelloWorldFunctionName = CdkHelloWorldStack-
HelloWorldFunctionunique-identifier
...
```

3. Use the AWS CLI to invoke your Lambda function in the AWS Cloud and output the response to a text file:

```
$ aws lambda invoke --function-name CdkHelloWorldStack-HelloWorldFunctionunique-
identifier output.txt
```

4. Check `output.txt` to see your results.

   Possible cause: API Gateway resource is defined incorrectly in your stack file

   If `output.txt` shows a successful Lambda function response, the issue could be with how you defined your API Gateway REST API. The AWS CLI invokes your Lambda directly, not through your endpoint. Check your code to ensure it matches this tutorial. Then, deploy again.

   Possible cause: Lambda resource is defined incorrectly in your stack file

   If `output.txt` returns an error, the issue could be with how you defined your Lambda function. Check your code to ensure it matches this tutorial. Then deploy again.

Example: Create a CDK app with multiple stacks

You can create an AWS Cloud Development Kit (AWS CDK) application containing multiple stacks. When you deploy the AWS CDK app, each stack becomes its own AWS CloudFormation template. You can also synthesize and deploy each stack individually using the AWS CDK CLI `cdk deploy` command.

In this example, we cover the following:

- How to extend the `Stack` class to accept new properties or arguments.
- How to use properties to determine which resources the stack contains and their configuration.

- How to instantiate multiple stacks from this class.

The example in this topic uses a Boolean property, named `encryptBucket` (Python: `encrypt_bucket`). It indicates whether an Amazon S3 bucket should be encrypted. If so, the stack enables encryption using a key managed by AWS Key Management Service (AWS KMS). The app creates two instances of this stack, one with encryption and one without.

Prerequisites

This example assumes that all getting started steps have been completed.

Create a CDK project

First, we create a CDK project using the CDK CLI:

Example

TypeScript

```
mkdir multistack
cd multistack
cdk init --language=typescript
```

JavaScript

```
mkdir multistack
cd multistack
cdk init --language=javascript
```

Python

```
mkdir multistack
cd multistack
cdk init --language=python
source .venv/bin/activate # On Windows, run '.\venv\Scripts\activate' instead
pip install -r requirements.txt
```

Java

```
mkdir multistack
```

```
cd multistack
cdk init --language=java
```

You can import the resulting Maven project into your Java IDE.

C#

```
mkdir multistack
cd multistack
cdk init --language=csharp
```

You can open the file `src/Pipeline.sln` in Visual Studio.

Add an optional parameter

The `props` argument of the `Stack` constructor fulfills the interface `StackProps`. In this example, we want the stack to accept an additional property to tell us whether to encrypt the Amazon S3 bucket. To do this, we create an interface or class that includes the property. This allows the compiler to make sure that the property has a Boolean value and enables auto-completion for it in your IDE.

We open our *stack file* in our IDE or editor and add the new interface, class, or argument. New lines are highlighted in bold:

Example

TypeScript

File: `lib/multistack-stack.ts`

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

interface MultiStackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}

export class MultistackStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: MultiStackProps) {
    super(scope, id, props);
```

```
    // The code that defines our stack goes here
  }
}
```

JavaScript

File: `lib/multistack-stack.js`

JavaScript doesn't have an interface feature; we don't need to add any code.

```
const cdk = require('aws-cdk-stack');

class MultistackStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // The code that defines our stack goes here
  }
}

module.exports = { MultistackStack }
```

Python

File: `multistack/multistack_stack.py`

Python does not have an interface feature, so we'll extend our stack to accept the new property by adding a keyword argument.

```
import aws_cdk as cdk
from constructs import Construct

class MultistackStack(cdk.Stack):

    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # The code that defines our stack goes here
```

Java

File: `src/main/java/com/myorg/MultistackStack.java`

It's more complicated than we really want to get into to extend a props type in Java. Instead, write the stack's constructor to accept an optional Boolean parameter. Because `props` is an optional argument, we'll write an additional constructor that lets you skip it. It will default to `false`.

```
package com.myorg;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;

import software.amazon.awscdk.services.s3.Bucket;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id, boolean
 encryptBucket) {
        this(scope, id, null, encryptBucket);
    }

    public MultistackStack(final Construct scope, final String id) {
        this(scope, id, null, false);
    }

    public MultistackStack(final Construct scope, final String id, final StackProps
 props,
            final boolean encryptBucket) {
        super(scope, id, props);

        // The code that defines our stack goes here
    }
}
```

C#

File: `src/Multistack/MultistackStack.cs`

```
using Amazon.CDK;
```

```
using constructs;

namespace Multistack
{


    public class MultiStackProps : StackProps
    {
        public bool? EncryptBucket { get; set; }
    }


    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, MultiStackProps props) :
 base(scope, id, props)
        {
            // The code that defines our stack goes here
        }
    }
}
```

The new property is optional. If `encryptBucket` (Python: `encrypt_bucket`) is not present, its value is `undefined`, or the local equivalent. The bucket will be unencrypted by default.

Define the stack class

Next, we define our stack class, using our new property. New code is highlighted in bold:

Example

TypeScript

File: `lib/multistack-stack.ts`

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from constructs;
import * as s3 from 'aws-cdk-lib/aws-s3';

interface MultistackProps extends cdk.StackProps {
  encryptBucket?: boolean;
}
```

```
export class MultistackStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: MultistackProps) {
    super(scope, id, props);

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
    // Encrypted bucket uses KMS-managed keys (SSE-KMS).
    if (props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY});
    }
  }
}
```

JavaScript

File: `lib/multistack-stack.js`

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class MultistackStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    // Add a Boolean property "encryptBucket" to the stack constructor.
    // If true, creates an encrypted bucket. Otherwise, the bucket is unencrypted.
    // Encrypted bucket uses KMS-managed keys (SSE-KMS).
    if ( props && props.encryptBucket) {
      new s3.Bucket(this, "MyGroovyBucket", {
        encryption: s3.BucketEncryption.KMS_MANAGED,
        removalPolicy: cdk.RemovalPolicy.DESTROY
      });
    } else {
      new s3.Bucket(this, "MyGroovyBucket", {
        removalPolicy: cdk.RemovalPolicy.DESTROY});
    }
  }
```

```
}

module.exports = { MultistackStack }
```

Python

File: `multistack/multistack_stack.py`

```
import aws_cdk as cdk
from constructs import Construct
from aws_cdk import aws_s3 as s3

class MultistackStack(cdk.Stack):

    # The Stack class doesn't know about our encrypt_bucket parameter,
    # so accept it separately and pass along any other keyword arguments.
    def __init__(self, scope: Construct, id: str, *, encrypt_bucket=False,
                 **kwargs) -> None:
        super().__init__(scope, id, **kwargs)

        # Add a Boolean property "encryptBucket" to the stack constructor.
        # If true, creates an encrypted bucket. Otherwise, the bucket is
 unencrypted.
        # Encrypted bucket uses KMS-managed keys (SSE-KMS).
        if encrypt_bucket:
            s3.Bucket(self, "MyGroovyBucket",
                      encryption=s3.BucketEncryption.KMS_MANAGED,
                      removal_policy=cdk.RemovalPolicy.DESTROY)
        else:
            s3.Bucket(self, "MyGroovyBucket",
                     removal_policy=cdk.RemovalPolicy.DESTROY)
```

Java

File: `src/main/java/com/myorg/MultistackStack.java`

```
package com.myorg;

import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.constructs.Construct;
import software.amazon.awscdk.RemovalPolicy;

```

```
import software.amazon.awscdk.services.s3.Bucket;
import software.amazon.awscdk.services.s3.BucketEncryption;

public class MultistackStack extends Stack {
    // additional constructors to allow props and/or encryptBucket to be omitted
    public MultistackStack(final Construct scope, final String id,
            boolean encryptBucket) {
        this(scope, id, null, encryptBucket);
    }

    public MultistackStack(final Construct scope, final String id) {
        this(scope, id, null, false);
    }

    // main constructor
    public MultistackStack(final Construct scope, final String id,
            final StackProps props, final boolean encryptBucket) {
        super(scope, id, props);

        // Add a Boolean property "encryptBucket" to the stack constructor.
        // If true, creates an encrypted bucket. Otherwise, the bucket is
        // unencrypted. Encrypted bucket uses KMS-managed keys (SSE-KMS).
        if (encryptBucket) {
            Bucket.Builder.create(this, "MyGroovyBucket")
                    .encryption(BucketEncryption.KMS_MANAGED)
                    .removalPolicy(RemovalPolicy.DESTROY).build();
        } else {
            Bucket.Builder.create(this, "MyGroovyBucket")
                    .removalPolicy(RemovalPolicy.DESTROY).build();
        }
    }
}
```

C#

File: `src/Multistack/MultistackStack.cs`

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

namespace Multistack
{

    public class MultiStackProps : StackProps
```

```
    {
        public bool? EncryptBucket { get; set; }
    }

    public class MultistackStack : Stack
    {
        public MultistackStack(Construct scope, string id, IMultiStackProps props =
 null) : base(scope, id, props)
        {
            // Add a Boolean property "EncryptBucket" to the stack constructor.
            // If true, creates an encrypted bucket. Otherwise, the bucket is
 unencrypted.
            // Encrypted bucket uses KMS-managed keys (SSE-KMS).
            if (props?.EncryptBucket ?? false)
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
                {
                    Encryption = BucketEncryption.KMS_MANAGED,
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
            else
            {
                new Bucket(this, "MyGroovyBucket", new BucketProps
                {
                    RemovalPolicy = RemovalPolicy.DESTROY
                });
            }
        }
    }
}
```

Create two stack instances

In our *application file*, we add the code to instantiate two separate stacks. We delete the existing `MultistackStack` definition and define our two stacks. New code is highlight in bold:

Example

TypeScript

File: `bin/multistack.ts`

```
#!/usr/bin/env node
import 'source-map-support/register';
import * as cdk from 'aws-cdk-lib';
import { MultistackStack } from '../lib/multistack-stack';

const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});

app.synth();
```

JavaScript

File: `bin/multistack.js`

```
#!/usr/bin/env node
const cdk = require('aws-cdk-lib');
const { MultistackStack } = require('../lib/multistack-stack');

const app = new cdk.App();

new MultistackStack(app, "MyWestCdkStack", {
    env: {region: "us-west-1"},
    encryptBucket: false
});

new MultistackStack(app, "MyEastCdkStack", {
    env: {region: "us-east-1"},
    encryptBucket: true
});

app.synth();
```

Python

File: `./app.py`

```
#!/usr/bin/env python3

import aws_cdk as cdk

from multistack.multistack_stack import MultistackStack

app = cdk.App()
MultistackStack(app, "MyWestCdkStack",
                env=cdk.Environment(region="us-west-1"),
                encrypt_bucket=False)

MultistackStack(app, "MyEastCdkStack",
                env=cdk.Environment(region="us-east-1"),
                encrypt_bucket=True)

app.synth()
```

Java

File: `src/main/java/com/myorg/MultistackApp.java`

```
package com.myorg;

import software.amazon.awscdk.App;
import software.amazon.awscdk.Environment;
import software.amazon.awscdk.StackProps;

public class MultistackApp {
    public static void main(final String argv[]) {
        App app = new App();

        new MultistackStack(app, "MyWestCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .region("us-west-1")
                        .build())
                .build(), false);

        new MultistackStack(app, "MyEastCdkStack", StackProps.builder()
                .env(Environment.builder()
                        .region("us-east-1")
```

```
                        .build())
                .build(), true);

        app.synth();
    }
}
```

C#

File: src/Multistack/Program.cs

```
using Amazon.CDK;

namespace Multistack
{
    class Program
    {
        static void Main(string[] args)
        {
            var app = new App();

            new MultistackStack(app, "MyWestCdkStack", new MultiStackProps
            {
                Env = new Environment { Region = "us-west-1" },
                EncryptBucket = false
            });

            new MultistackStack(app, "MyEastCdkStack", new MultiStackProps
            {
                Env = new Environment { Region = "us-east-1" },
                EncryptBucket = true
            });

            app.Synth();
        }
    }
}
```

This code uses the new `encryptBucket` (Python: `encrypt_bucket`) property on the `MultistackStack` class to instantiate the following:

- One stack with an encrypted Amazon S3 bucket in the `us-east-1` AWS Region.

- One stack with an unencrypted Amazon S3 bucket in the `us-west-1` AWS Region.

Synthesize and deploy the stack

Next, we can deploy stacks from the app. First, we synthesize an AWS CloudFormation template for `MyEastCdkStack`. This is the stack in `us-east-1` with the encrypted Amazon S3 bucket.

```
$ cdk synth MyEastCdkStack
```

To deploy this stack to our AWS environment, we can issue one of the following commands. The first command uses our default AWS profile to obtain the credentials to deploy the stack. The second uses a profile that we specify. For `PROFILE_NAME`, we can substitute the name of an AWS CLI profile that contains appropriate credentials for deploying to the `us-east-1` AWS Region.

```
$ cdk deploy MyEastCdkStack
```

```
$ cdk deploy MyEastCdkStack --profile=<PROFILE_NAME>
```

Clean up

To avoid charges for resources that we deployed, we destroy the stack using the following command:

```
cdk destroy MyEastCdkStack
```

The destroy operation fails if there is anything stored in the stack's bucket. There shouldn't be, since we only created the bucket. If we did put something in the bucket, we must delete the bucket contents before destroying the stack. We can use the AWS Management Console or the AWS CLI to delete the bucket contents.

Example: Create an AWS Fargate service using the AWS CDK

In this example, we show you how to create an AWS Fargate service running on an Amazon Elastic Container Service (Amazon ECS) cluster that's fronted by an internet-facing Application Load Balancer from an image on Amazon ECR.

Amazon ECS is a highly scalable, fast, container management service that makes it easy to run, stop, and manage Docker containers on a cluster. You can host your cluster on serverless infrastructure that's managed by Amazon ECS by launching your services or tasks using the Fargate launch type. For more control, you can host your tasks on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances that you manage by using the Amazon EC2 launch type.

In this example, we launch some services using the Fargate launch type. If you've used the AWS Management Console to create a Fargate service, you know that there are many steps to follow to accomplish that task. AWS has several tutorials and documentation topics that walk you through creating a Fargate service, including:

- How to Deploy Docker Containers - AWS
- Setting Up with Amazon ECS
- Getting Started with Amazon ECS Using Fargate

This example creates a similar Fargate service using the AWS CDK.

The Amazon ECS construct used in this example helps you use AWS services by providing the following benefits:

- Automatically configures a load balancer.
- Automatically opens a security group for load balancers. This enables load balancers to communicate with instances without having to explicitly create a security group.
- Automatically orders dependency between the service and the load balancer attaching to a target group, where the AWS CDK enforces the correct order of creating the listener before an instance is created.
- Automatically configures user data on automatically scaling groups. This creates the correct configuration to associate a cluster to AMIs.
- Validates parameter combinations early. This exposes AWS CloudFormation issues earlier, thus saving deployment time. For example, depending on the task, it's easy to improperly configure the memory settings. Previously, we would not encounter an error until we deployed our app. But now the AWS CDK can detect a misconfiguration and emit an error when we synthesize our app.
- Automatically adds permissions for Amazon Elastic Container Registry (Amazon ECR) if we use an image from Amazon ECR.
- Automatically scales. The AWS CDK supplies a method so we can auto scale instances when we use an Amazon EC2 cluster. This happens automatically when we use an instance in a Fargate cluster.

In addition, the AWS CDK prevents an instance from being deleted when automatic scaling tries to stop an instance, but either a task is running or is scheduled on that instance.

Previously, we had to create a Lambda function to have this functionality.

- Provides asset support, so that we can deploy a source from our machine to Amazon ECS in one step. Previously, to use an application source, we had to perform several manual steps, such as uploading to Amazon ECR and creating a Docker image.

> **Important**
>
> The `ApplicationLoadBalancedFargateService` constructs we'll be using includes numerous AWS components, some of which have non-trivial costs if left provisioned in our AWS account, even if we don't use them. Be sure to clean up (`cdk destroy`) if you follow along with this example.

Create a CDK project

We start by creating a CDK project. This is a directory that stores our AWS CDK code, including our CDK app.

Example

TypeScript

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language typescript
```

JavaScript

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language javascript
```

Python

```
mkdir MyEcsConstruct
```

```
cd MyEcsConstruct
cdk init --language python
source .venv/bin/activate # On Windows, run '.\venv\Scripts\activate' instead
pip install -r requirements.txt
```

Java

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language java
```

We may now import the Maven project into our IDE.

C#

```
mkdir MyEcsConstruct
cd MyEcsConstruct
cdk init --language csharp
```

We may now open `src/MyEcsConstruct.sln` in Visual Studio.

Next, we run the app and confirm that it creates an empty stack.

```
cdk synth
```

Create a Fargate service

There are two different ways that we can run our container tasks with Amazon ECS:

- Use the `Fargate` launch type, where Amazon ECS manages the physical machines that oour containers are running on for us.
- Use the `EC2` launch type, where we do the managing, such as specifying automatic scaling.

For this example, we'll create a Fargate service running on an Amazon ECS cluster, fronted by an internet-facing Application Load Balancer.

We add the following AWS Construct Library module imports to our *stack file*:

Example

TypeScript

File: `lib/my_ecs_construct-stack.ts`

```
import * as ec2 from "aws-cdk-lib/aws-ec2";
import * as ecs from "aws-cdk-lib/aws-ecs";
import * as ecs_patterns from "aws-cdk-lib/aws-ecs-patterns";
```

JavaScript

File: `lib/my_ecs_construct-stack.js`

```
const ec2 = require("aws-cdk-lib/aws-ec2");
const ecs = require("aws-cdk-lib/aws-ecs");
const ecs_patterns = require("aws-cdk-lib/aws-ecs-patterns");
```

Python

File: `my_ecs_construct/my_ecs_construct_stack.py`

```
from aws_cdk import (aws_ec2 as ec2, aws_ecs as ecs,
                     aws_ecs_patterns as ecs_patterns)
```

Java

File: `src/main/java/com/myorg/MyEcsConstructStack.java`

```
import software.amazon.awscdk.services.ec2.*;
import software.amazon.awscdk.services.ecs.*;
import software.amazon.awscdk.services.ecs.patterns.*;
```

C#

File: `src/MyEcsConstruct/MyEcsConstructStack.cs`

```
using Amazon.CDK.AWS.EC2;
using Amazon.CDK.AWS.ECS;
using Amazon.CDK.AWS.ECS.Patterns;
```

Within our stack, we add the following code:

Example

TypeScript

```
    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is true
    });
```

JavaScript

```
    const vpc = new ec2.Vpc(this, "MyVpc", {
      maxAzs: 3 // Default is all AZs in region
    });

    const cluster = new ecs.Cluster(this, "MyCluster", {
      vpc: vpc
    });

    // Create a load-balanced Fargate service and make it public
    new ecs_patterns.ApplicationLoadBalancedFargateService(this, "MyFargateService",
 {
      cluster: cluster, // Required
      cpu: 512, // Default is 256
      desiredCount: 6, // Default is 1
```

```
      taskImageOptions: { image: ecs.ContainerImage.fromRegistry("amazon/amazon-ecs-
sample") },
      memoryLimitMiB: 2048, // Default is 512
      publicLoadBalancer: true // Default is true
    });
```

Python

```
        vpc = ec2.Vpc(self, "MyVpc", max_azs=3)     # default is all AZs in region

        cluster = ecs.Cluster(self, "MyCluster", vpc=vpc)

        ecs_patterns.ApplicationLoadBalancedFargateService(self, "MyFargateService",
            cluster=cluster,            # Required
            cpu=512,                    # Default is 256
            desired_count=6,            # Default is 1
            task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(
                image=ecs.ContainerImage.from_registry("amazon/amazon-ecs-sample")),
            memory_limit_mib=2048,      # Default is 512
            public_load_balancer=True)  # Default is True
```

Java

```
        Vpc vpc = Vpc.Builder.create(this, "MyVpc")
                            .maxAzs(3)  // Default is all AZs in region
                            .build();

        Cluster cluster = Cluster.Builder.create(this, "MyCluster")
                            .vpc(vpc).build();

        // Create a load-balanced Fargate service and make it public
        ApplicationLoadBalancedFargateService.Builder.create(this,
 "MyFargateService")
                    .cluster(cluster)           // Required
                    .cpu(512)                   // Default is 256
                     .desiredCount(6)            // Default is 1
                     .taskImageOptions(
                             ApplicationLoadBalancedTaskImageOptions.builder()
                                     .image(ContainerImage.fromRegistry("amazon/
amazon-ecs-sample"))
                                     .build())
                     .memoryLimitMiB(2048)       // Default is 512
                     .publicLoadBalancer(true)   // Default is true
```

```
                    .build();
```

C#

```
            var vpc = new Vpc(this, "MyVpc", new VpcProps
            {
                MaxAzs = 3 // Default is all AZs in region
            });

            var cluster = new Cluster(this, "MyCluster", new ClusterProps
            {
                Vpc = vpc
            });

            // Create a load-balanced Fargate service and make it public
            new ApplicationLoadBalancedFargateService(this, "MyFargateService",
                new ApplicationLoadBalancedFargateServiceProps
                {
                    Cluster = cluster,          // Required
                    DesiredCount = 6,           // Default is 1
                    TaskImageOptions = new ApplicationLoadBalancedTaskImageOptions
                    {
                        Image = ContainerImage.FromRegistry("amazon/amazon-ecs-
sample")
                    },
                    MemoryLimitMiB = 2048,      // Default is 256
                    PublicLoadBalancer = true   // Default is true
                }
            );
```

Next, we validate our code by running the following to synthesize our stack:

```
cdk synth
```

The stack is hundreds of lines, so we won’t show it here. The stack should contain one default instance, a private subnet and a public subnet for the three Availability Zones, and a security group.

To deploy the stack, we run the following:

```
cdk deploy
```

AWS CloudFormation displays information about the dozens of steps that it takes as it deploys our app.

Once deployment completes, we have successfully created a Fargate powered Amazon ECS service to run a Docker image.

Clean up

As a general maintenance best practice, and to minimize unnecessary costs, we delete our stack when complete:

```
cdk destroy
```

Use other tools with the AWS CDK

You can use other tools with the AWS CDK to improve your development workflows.

AWS Toolkit for Visual Studio Code

The AWS Toolkit for Visual Studio Code is an open source plugin for Visual Studio Code that makes it easier to create, debug, and deploy applications on AWS. The toolkit provides an integrated experience for developing AWS CDK applications. It includes the AWS CDK Explorer feature to list your AWS CDK projects and browse the various components of the CDK application. Install the AWS Toolkit and learn more about using the AWS CDK Explorer.

AWS SAM integration

Use the AWS CDK and the AWS Serverless Application Model (AWS SAM) together to locally build and test serverless applications defined in the CDK. For complete information, see AWS Cloud Development Kit (AWS CDK) in the *AWS Serverless Application Model Developer Guide*. To install the AWS SAM CLI, see Install the AWS SAM CLI.

Security for the AWS Cloud Development Kit (AWS CDK)

Cloud security at Amazon Web Services (AWS) is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations. Security is a shared responsibility between AWS and you. The Shared Responsibility Model describes this as Security of the Cloud and Security in the Cloud.

Security of the Cloud – AWS is responsible for protecting the infrastructure that runs all of the services offered in the AWS Cloud and providing you with services that you can use securely. Our security responsibility is the highest priority at AWS, and the effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS Compliance Programs.

Security in the Cloud – Your responsibility is determined by the AWS service you are using, and other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Identity and access management for the AWS Cloud Development Kit (AWS CDK)

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS resources. IAM is an AWS service that you can use with no additional charge.

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS.

Service user – If you use AWS services to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator.

Service administrator – If you're in charge of AWS resources at your company, you probably have full access to AWS resources. It's your job to determine which AWS services and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS services.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

To access AWS programmatically, AWS provides the AWS CDK, software development kits (SDKs), and a command line interface (CLI) to cryptographically sign your requests using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see Signature Version 4 signing process in the *AWS General Reference*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An IAM group is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It's similar to an IAM user, but isn't associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For

more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Compliance validation for the AWS Cloud Development Kit (AWS CDK)

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

The security and compliance of AWS services is assessed by third-party auditors as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others. AWS provides a frequently updated list of AWS services in scope of specific compliance programs at AWS Services in Scope by Compliance Program.

Third-party audit reports are available for you to download using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

For more information about AWS compliance programs, see AWS Compliance Programs.

Your compliance responsibility when using the AWS CDK to access an AWS service is determined by the sensitivity of your data, your organization's compliance objectives, and applicable laws and regulations. If your use of an AWS service is subject to compliance with standards such as HIPAA, PCI, or FedRAMP, AWS provides resources to help:

- Security and Compliance Quick Start Guides – Deployment guides that discuss architectural considerations and provide steps for deploying security-focused and compliance-focused baseline environments on AWS.
- AWS Compliance Resources – A collection of workbooks and guides that might apply to your industry and location.
- AWS Config – A service that assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – A comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience for the AWS Cloud Development Kit (AWS CDK)

The Amazon Web Services (AWS) global infrastructure is built around AWS Regions and Availability Zones.

AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking.

With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Infrastructure security for the AWS Cloud Development Kit (AWS CDK)

The AWS CDK follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Troubleshooting common AWS CDK issues

This topic describes how to troubleshoot the following issues with the AWS CDK.

- After updating the AWS CDK the AWS CDK Toolkit (CLI) reports a mismatch with the AWS Construct Library
- When deploying my AWS CDK stack I receive a NoSuchBucket error
- When deploying my AWS CDK stack I receive a forbidden: null message
- When synthesizing an AWS CDK stack I get the message --app is required either in command-line in cdk.json or in ~/.cdk.json
- When synthesizing an AWS CDK stack I receive an error because the AWS CloudFormation template contains too many resources
- I specified three (or more) Availability Zones for my Auto Scaling group or VPC but it was only deployed in two
- My S3 bucket DynamoDB table or other resource is not deleted when I issue cdk destroy

After updating the AWS CDK, the AWS CDK Toolkit (CLI) reports a mismatch with the AWS Construct Library

The version of the AWS CDK Toolkit (which provides the `cdk` command) must be at least equal to the version of the main AWS Construct Library module, `aws-cdk-lib`. The Toolkit is intended to be backward compatible. The latest 2.x version of the toolkit can be used with any 1.x or 2.x release of the library. For this reason, we recommend you install this component globally and keep it up to date.

```
npm update -g aws-cdk
```

If you need to work with multiple versions of the AWS CDK Toolkit, install a specific version of the toolkit locally in your project folder.

If you are using TypeScript or JavaScript, your project directory already contains a versioned local copy of the CDK Toolkit.

If you are using another language, use `npm` to install the AWS CDK Toolkit, omitting the `-g` flag and specifying the desired version. For example:

```
npm install aws-cdk@2.0
```

To run a locally installed AWS CDK Toolkit, use the command `npx aws-cdk` instead of only `cdk`. For example:

```
npx aws-cdk deploy MyStack
```

`npx aws-cdk` runs the local version of the AWS CDK Toolkit if one exists. It falls back to the global version when a project doesn't have a local installation. You may find it convenient to set up a shell alias to make sure `cdk` is always invoked this way.

Example

macOS/Linux

```
alias cdk="npx aws-cdk"
```

Windows

```
doskey cdk=npx aws-cdk $*
```

When deploying my AWS CDK stack, I receive a `NoSuchBucket` error

Your AWS environment has not been bootstrapped, and so does not have an Amazon S3 bucket to hold resources during deployment. You can create the staging bucket and other required resources with the following command:

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

To avoid generating unexpected AWS charges, the AWS CDK does not automatically bootstrap any environment. You must explicitly bootstrap each environment into which you will deploy.

By default, the bootstrap resources are created in the Region or Regions that are used by stacks in the current AWS CDK application. Alternatively, they are created in the Region specified in your local AWS profile (set by `aws configure`), using that profile's account. You can specify a different account and Region on the command line as follows. (You must specify the account and Region if you are not in an app's directory.)

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

For more information, see AWS CDK bootstrapping.

When deploying my AWS CDK stack, I receive a `forbidden: null` message

You are deploying a stack that requires bootstrap resources, but are using an IAM role or account that lacks permission to write to it. (The staging bucket is used when deploying stacks that contain assets or that synthesize an AWS CloudFormation template larger than 50K.) Use an account or role that has permission to perform the action `s3:*` against the bucket mentioned in the error message.

When synthesizing an AWS CDK stack, I get the message `--app is required either in command-line, in cdk.json or in ~/.cdk.json`

This message usually means that you aren't in the main directory of your AWS CDK project when you issue `cdk synth`. The file `cdk.json` in this directory, created by the `cdk init` command, contains the command line needed to run (and thereby synthesize) your AWS CDK app. For a TypeScript app, for example, the default `cdk.json` looks something like this:

```
{
  "app": "npx ts-node bin/my-cdk-app.ts"
}
```

We recommend issuing `cdk` commands only in your project's main directory, so the AWS CDK toolkit can find `cdk.json` there and successfully run your app.

If this isn't practical for some reason, the AWS CDK Toolkit looks for the app's command line in two other locations:

- In `cdk.json` in your home directory
- On the `cdk synth` command itself using the `-a` option

For example, you might synthesize a stack from a TypeScript app as follows.

```
cdk synth --app "npx ts-node my-cdk-app.ts" MyStack
```

When synthesizing an AWS CDK stack, I receive an error because the AWS CloudFormation template contains too many resources

The AWS CDK generates and deploys AWS CloudFormation templates. AWS CloudFormation has a hard limit on the number of resources a stack can contain. With the AWS CDK, you can run up against this limit more quickly than you might expect.

> **Note**
>
> The AWS CloudFormation resource limit is 500 at this writing. See AWS CloudFormation quotas for the current resource limit.

The AWS Construct Library's higher-level, intent-based constructs automatically provision any auxiliary resources that are needed for logging, key management, authorization, and other purposes. For example, granting one resource access to another generates any IAM objects needed for the relevant services to communicate.

In our experience, real-world use of intent-based constructs results in 1–5 AWS CloudFormation resources per construct, though this can vary. For serverless applications, 5–8 AWS resources per API endpoint is typical.

Patterns, which represent a higher level of abstraction, let you define even more AWS resources with even less code. The AWS CDK code in Example: Create an AWS Fargate service using the AWS CDK, for example, generates more than 50 AWS CloudFormation resources while defining only three constructs!

Exceeding the AWS CloudFormation resource limit is an error during AWS CloudFormation synthesis. The AWS CDK issues a warning if your stack exceeds 80% of the limit. You can use a different limit by setting the `maxResources` property on your stack, or disable validation by setting `maxResources` to 0.

Tip

You can get an exact count of the resources in your synthesized output using the following utility script. (Since every AWS CDK developer needs Node.js, the script is written in JavaScript.)

```
// rescount.js - count the resources defined in a stack
// invoke with: node rescount.js <path-to-stack-json>
// e.g. node rescount.js cdk.out/MyStack.template.json

import * as fs from 'fs';
const path = process.argv[2];

if (path) fs.readFile(path, 'utf8', function(err, contents) {
  console.log(err ? `${err}` :
  `${Object.keys(JSON.parse(contents).Resources).length} resources defined in
 ${path}`);
}); else console.log("Please specify the path to the stack's output .json
 file");
```

As your stack's resource count approaches the limit, consider re-architecting to reduce the number of resources your stack contains: for example, by combining some Lambda functions, or by breaking your stack into multiple stacks. The CDK supports references between stacks, so you can separate your app's functionality into different stacks in whatever way makes the most sense to you.

Note

AWS CloudFormation experts often suggest the use of nested stacks as a solution to the resource limit. The AWS CDK supports this approach via the NestedStack construct.

I specified three (or more) Availability Zones for my Auto Scaling group or VPC, but it was only deployed in two

To get the number of Availability Zones that you request, specify the account and Region in the stack's `env` property. If you do not specify both, the AWS CDK, by default, synthesizes the stack as environment-agnostic. You can then deploy the stack to a specific Region using AWS CloudFormation. Because some Regions have only two Availability Zones, an environment-agnostic template doesn't use more than two.

> **Note**
>
> In the past, Regions have occasionally launched with only one Availability Zone. Environment-agnostic AWS CDK stacks cannot be deployed to such Regions. At this writing, however, all AWS Regions have at least two AZs.

You can change this behavior by overriding your stack's availabilityZones (Python: `availability_zones`) property to explicitly specify the zones that you want to use.

For more information about specifying a stack's account and region at synthesis time, while retaining the flexibility to deploy to any region, see Environments for the AWS CDK.

My S3 bucket, DynamoDB table, or other resource is not deleted when I issue `cdk destroy`

By default, resources that can contain user data have a `removalPolicy` (Python: `removal_policy`) property of `RETAIN`, and the resource is not deleted when the stack is destroyed. Instead, the resource is orphaned from the stack. You must then delete the resource manually after the stack is destroyed. Until you do, redeploying the stack fails. This is because the name of the new resource being created during deployment conflicts with the name of the orphaned resource.

If you set a resource's removal policy to `DESTROY`, that resource will be deleted when the stack is destroyed.

Example

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as s3 from 'aws-cdk-lib/aws-s3';

export class CdkTestStack extends cdk.Stack {
```

```
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY,
    });
  }
}
```

JavaScript

```
const cdk = require('aws-cdk-lib');
const s3 = require('aws-cdk-lib/aws-s3');

class CdkTestStack extends cdk.Stack {
  constructor(scope, id, props) {
    super(scope, id, props);

    const bucket = new s3.Bucket(this, 'Bucket', {
      removalPolicy: cdk.RemovalPolicy.DESTROY
    });
  }
}

module.exports = { CdkTestStack }
```

Python

```
import aws_cdk as cdk
from constructs import Construct
import aws_cdk.aws_s3 as s3

class CdkTestStack(cdk.stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        bucket = s3.Bucket(self, "Bucket",
            removal_policy=cdk.RemovalPolicy.DESTROY)
```

Java

```
software.amazon.awscdk.*;
```

```
import software.amazon.awscdk.services.s3.*;
import software.constructs;

public class CdkTestStack extends Stack {
    public CdkTestStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public CdkTestStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Bucket.Builder.create(this, "Bucket")
                .removalPolicy(RemovalPolicy.DESTROY).build();
    }
}
```

C#

```
using Amazon.CDK;
using Amazon.CDK.AWS.S3;

public CdkTestStack(Construct scope, string id, IStackProps props) : base(scope, id,
 props)
{
    new Bucket(this, "Bucket", new BucketProps {
        RemovalPolicy = RemovalPolicy.DESTROY
    });
}
```

Note

AWS CloudFormation cannot delete a non-empty Amazon S3 bucket. If you set an Amazon S3 bucket's removal policy to `DESTROY`, and it contains data, attempting to destroy the stack will fail because the bucket cannot be deleted. You can have the AWS CDK delete the objects in the bucket before attempting to destroy it by setting the bucket's `autoDeleteObjects` prop to `true`.

OpenPGP keys for the AWS CDK and jsii

This topic contains current and historical OpenPGP keys for the AWS CDK and jsii.

Current keys

These keys should be used to validate current releases of the AWS CDK and jsii.

AWS CDK OpenPGP key

Key ID:	0x42B9CF2286CD987A
Type:	RSA
Size:	4096/4096
Created:	2022-07-05
Expires:	2026-07-04
User ID:	AWS Cloud Development Kit <aws-cdk@amazon.com>
Key fingerprint:	69B5 2D5B A295 1D11 FA65 413B 42B9 CF22 86CD 987A

Select the "Copy" icon to copy the following OpenPGP key:


```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGLEgOsBEADCoAMwvnszMLybJ+AD9cHhVyX6+rYIUEXYSgVnfkl6Z7qawIwv
wgd/a5fEs9Kiz2XJmfwS9Rxb4d+0+Y1ls1A+gnpw9FMLcZlqkC9KLnS2MqvuXWLB
t3z4kjZaL9fQ+58PoD4gy/M2hDg6gZrYqR3gtJuw8FcFpb/1KlkzRQUM8eAMFxf2
TyfjP0V0tSHwcB+84oushX7fUXVMyc3+OHsCPOe/WBFMIlWgKA+n33JKIQlUUC8f
kCWBAsAFupil0lCveT6mZu5slNRlc1I3iBLjUZ3/MtLygfqAMKwUVXeawtDvRIZe
PrAFc2NyODEhly2JG6K0FW7eIcvBqR3rg8U49t9Y74ELTM0kKnfd+flvq35xWqQC
0zghnk3kDppRTN4zWBgTKiCMxBcsHXGOoGn57t4B9VY9Zy3vkeySigeiwl/Tw9nJ
PE0SRnwEc/HnjTTfX+GTG1aQVE0xSVyZ4m5ymRNCu6+rNH8lKwo5FujlXJ+GXPkp
```

```
qT+Lx6Ix/Ny7PaoweWxwtZUkLRS4pWUsg0yotZrGyIbS+X3yMEG8WBTFI9hf6HTq
0ryfi5/TsBrdrGKqWB99EC9xYEGgtHp4fKO5X0ynOagVOhf0jSe8t1uyuJPGb2Gc
MQagSys5xMhdG/ZnEY4Cb+JDtH/4jc3tca0+4Z5RQ7kF9IhCncFtrbjJbwARAQAB
tC5BV1MgQ2xvdWQgRGV2ZWxvcG1lbnQgS2l0IDxhd3MtY2RrQGFtYXpvbi5jb20+
iQI/BBMBAgApBQJixIDrAhsvBQkHhM4ABwsJCAcDAgEGFQgCCQoLBBYCAwECHgEC
F4AACgkQQrnPIobNmHo2qg//Zt9p/kN1DevflzxWKouUX0AS7UmUtRYXu5k/EEbu
wkYNHpUr7+lZ+Me5YyjcIpt6UwuG9cW4SvwuxIfXucyKAWiwEbydCQauvnrYDxDa
J6Yr/ntk7Sii6An9re99qic3IsvX+xlUXh+qJ/34ooP/1PHziCMqykvW/DwAIyhx
2qvTXy+9+OlOWSUbhkCnNz5XKb4XQGq73DqalZX1nH4dG6fckZmYRX+dpw2njfTw
ZLdZ7bkrfiL84FI4A21RfSbEU4s4ngiV17lZ9ivilBKTbDv3da7+yc919M7C5N4J
yrlxvtyYNDoqKAD2WYZAnpEbG/shu3f56RyOJd56tXGwl9nKPh+F9y+379XthSwA
xZTURFtjWf7wWHaDZadU0DKi+Oeeszjg2f/VJaGmmS8PIg7q6GiSHHpqHqNvACHm
ZXMw12QFd3qt3xu0JMmE11ZC5VBgblwpkQTrO04Sq1rOpJwXI9ODMS/ZEhAIoYmT
OR7ouknlAx6mj9fwpavWDAAJHLdVUMYBZTXiQYFzDvx51ivvTRWkB1zTJcFdqShY
B37+Jz2jLDNdMrcHk2yfVp/VvfbxKcexg8wEwrrtQUslTUenl5jBZJouoz/wW81s
Y4U1nCPCdTK5/C7JCKzR2gVnCpe6uaxAWkkM2feQhjqJZkTC4cFVgBT+4M6WcT1r
yq4=
=ahbs
-----END PGP PUBLIC KEY BLOCK-----
```


jsii OpenPGP key

Key ID:	0x056C4E15DAE3D8D9
Type:	RSA
Size:	4096/4096
Created:	2022-07-05
Expires:	2026-07-04
User ID:	AWS JSII Team <aws-jsii@amazon.com>
Key fingerprint:	1E07 31D4 57E5 FE87 87E5 530A 056C 4E15 DAE3 D8D9

Select the "Copy" icon to copy the following OpenPGP key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
mQINBGLEgOkBEAD27EPVG9g2mHQ3+M6tF6le+tfhARJ2EV7m7NKIrTdSlCZATLWn
AVLlxG1unW34NlkKZbcbR86gAxRnnAhuEhPuloU/S5wAqPGbRiFl58YjYZDNJw6U
1SSMpE4O1sfjxv9yAbiRihLYtvksyHHZmaDhYner2aK1PdeWu+BKq/tjfm3Yzsd2
uuVEduJ72YoQk/29dEiGOHfT+2kUKxUX+0tJSJ9MGlEf4NtQE4WLzrT6Xqb2SG4+
alIiIVxIEi0XKDn7n8ZLjFwfJwOYxVYLtEUkqFWM8e8vgoc9/nYc+vDXZVED2g3Z
FWrwSnDSXbQpnMa2cLhD4xLpDHUS3i2p7r3dkJQGLo/5JGOopLibrOAbYZ72izhu
H/TuPFogSz0mNFPglrWdnLF04UIjIq420+06V4WQZC9n55Zjcbki/OhnC3B9pAdU
tiy8zg070bWq45dPGf5STkPPn7G8A2zmKefyO51iLIi26ZzW78siB+FvcGRhdg25
39sHJ1cmrTeC+B+k4KeV5sQ/m3UucimrZnk1xdaiVp8mWzRqWb8bB6Rs8K9RMrMV
tFBOKOBAT2QxOQtRGAantVgm193E1T1cmNpD0FKAKkDdPs64rKBEwFiHxccXHbah
eMd1weVwn3AKFD6uAm8ZRMV+dyssfcQxqpo/kfT1XpA6cQeOmGDOcKBfdwARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQIAKQUC
YsSA6QIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEAVsThXa
49jZjU4QANoyqOJUT4gRrXshE3N0mW5Ad4i8Ke09GA62HyvTtfbsA+2nkNVGJpXm
sFMzdaFO95Q65RkLS9vW4nhhjXBEc2XYNCt2AnARudA/41ykjDPwU112z9ZTB9he
y4ItIeNGpHvMWr51fihl0y2nkpODOBeiv44jscLbHyOmZfki1f5fuIu2U2IbUGK3
5FtYyeHcgRHnpYkzLuzK4PfayOywqQPJ7M9DWrHf+v5Cu4ZCZDOIKfzF+ew7MWwc
6KaoWHCYbFpX8jxFppbGsSFOQ8Sl2quoP0TLz9Wsq70Khi6C2P8JI6lm0HRLO+1M
jFbQxNOwAcN3k4HSwunAjXBlmT/6oc1RsdBdpXBaZ2AWseIXwSYZqNXp+5L179uZ
vSiD3DSSUqLJbdQRVOsJi3/87V5QU59byq2dToHveRjtSbVnK0TkTx9ZlgkcpjvM
BwHNqWhratV6af2Upjq2YQ0fdSB42f3pgopInxNJPMvlAb+cCfr0Pfwu7ge7UooQ
WHTxbpCvwtn/HNctMGpWscO02WsWgoYVjnVFay/XphE77pQ9rRUkhMe6VKXfxj/n
OCZJKrydluIIwR8vvONNqO+QwZ1xDEhO7MaSZlOm1AuUZIXFPgaWQkPZHKiiwFA/
QWnL/+shuRtMH2geTjkev198Jgb5HyXFm4SyYtZferQROyliEhik
=BuGv
-----END PGP PUBLIC KEY BLOCK-----
```


Historical keys

These keys may be used to validate releases of the AWS CDK and jsii before 2022-07-05.

> **Important**
>
> New keys are created before the previous ones expire. As a result, at any given moment in time, more than one key may be valid. Keys are used to sign artifacts starting the day they are created, so use the more recently-issued key where keys' validity overlaps.

AWS CDK OpenPGP key (2022-04-07)

Note

This key was not used to sign AWS CDK artifacts after 2022-07-05.

Key ID:	0x015584281F44A3C3
Type:	RSA
Size:	4096/4096
Created:	2022-04-07
Expires:	2026-04-06
User ID:	AWS Cloud Development Kit <aws-cdk@amazon.com>
Key fingerprint:	EAE1 1A24 82B0 AA86 456E 6C67 0155 8428 1F44 A3C3

Select the "Copy" icon to copy the following OpenPGP key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBGJPLgUBEADtlR5jQtxtBmROQvmWlPOViqqnJNhk0dULc3tXnq8NS/l6X81r
wHk+/CHG5kBunwvM0qaqLFRC6z9NnnNDxEHcTi47n+OAjWyDM6unxxWOPz8Dfaps
Uq/ZWa4by292ZeqRC9Ir2wdrizb69JbRjeshBwlJDAS/qtqCAqBRH/f7Zw7QSD6/
XTxyIy+KOVjZwFPFNHMRQ/NmgUc/Rfxsa0pUjk1YAj/AkvQlwwD8DEnASoBhO0DP
QonZxouLqIpgp4LsGo8TZdQv30ocIj0C9DuYUiUXWlCPlYPgDj6IWf3rgpMQ6nB9
wC9lx4t/L3Zg1HUD52y8aymndmbdHVn90mzlNg4XWyc58rioYrEk57YwbDnea/Kk
Hv4kVHZRfJ4/OFPyqs5ex1X3X6rb07VvA1tfLgPywO9XF2Xws8YWOWcEobaWTcnb
AzyVC6wKya8rEQzxkYJ6UkJlhDB6g6bZwIpsI2zlimG+kSBsyFvE2oRYMS0cXPqU
o+tX0+4TvxEyW3RrUQzQHIpqXrb0X1Q8Z2idPn5dwsipDEa4gsFXtrSXmbB/0Cee
eJVvKWQAsxol3+NE9L/yozq3cz5PWh0SSbmCLRcs78lMJ23MmzbMWV7BWC9DXdY+
TywY5IkDUPjGCKlD8VlrI3TgC222bH6qaua6LYCiTtRtvpDYuJNAlUjhawARAQAB
tC5BV1MgQ2xvdWQgRGV2ZWxvcG1lbnQgS2l0IDxhd3MtY2RrQGFtYXpvbi5jb20+
iQI/BBMBAgApBQJiTy4FAhsvBQkHhM4ABwsJCAcDAgEGFQgCCQoLBBYCAwECHgEC
```

```
F4AACgkQAVWEKB9Eo8NpbxAAiBF0kR/lVw3vuam60mk4l0iGMVsP8Xq6g/buzbEO
2MEB4Ftk04qOnoa+93S0ZiLR9PqxrwsGSp4ADDX3Vtc4uxwzUlKUi1ywEhQ1cwyL
YHQI3Hd75K1J81ozMEu6qJH+yF0TtTDZMeZHtH/XvuIYJW3Lx4o5ZFlsEegFPAgX
YCCpUS+k9qC6M8g2VjcltQJpyjGswsKm6FWaKHW+B9dfjdOHlImB9E2jaknJ8eoY
zb9zHgFANluMzpZ6rYVSiCuXiEgYmazQWCvlPcMOP7nX+1hq1z11LMqeSnfE09gX
H+OYho9cMEJkb1dzx1H9MRpylFIn9tL+2iCp4UPJjnqi6uawWyLZ2tp4G11haqQq
1yAh69u233I8GZKFUySzjHwH5qWGRgBTjrZ6FdcjSS2w/wMkVKuCPkWtdvo/TJrm
msCd1Reye8SEKYqrs0ujTwmlvWmUZm006AdUjo1kWiBKeslTJrWEuG7Yk4pFOoA4
dsaq83gxpOJNVCh6M3y4DLNrvl7dhF95NwTWMROPj2otw7NIjF4/cdzve2+P7YNN
pVAtyCtTJdD3eZbQPVaL3T8cf1VGqt6++pnLGnWJ0+X3TyvfmTohdJvN3TE+tq7A
7cprDX/q9c56HaXdJzVpxEzuf/YC+JuYKeHwsX3QouDhyRg3PsigdZES/02Wr8so
l6U=
=MQI4
-----END PGP PUBLIC KEY BLOCK-----
```


jsii OpenPGP key (2022-04-07)

> **Note**
>
> This key was not used to sign jsii artifacts after 2022-07-05.

Key ID:	0x985F5BC974B79356
Type:	RSA
Size:	4096/4096
Created:	2022-04-07
Expires:	2026-04-06
User ID:	AWS JSII Team <aws-jsii@amazon.com>
Key fingerprint:	35A7 1785 8FA6 282D C5AC CD95 985F 5BC9 74B7 9356

Select the "Copy" icon to copy the following OpenPGP key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
mQINBGJPLewBEADHH4TXup/gOlHrKDZRbj8MvsMTdM6eDteA6/c32UYV/YsK9rDA
jN8Jv/xlfosOebcHrfnFpHF9VTkmjuOpN695XdwMrW/NvlEPISTGEJf21x6ZTQ2r
1xWFYzC3sl3FZmvj9XAXTmygdv+XM3TqsFgZeCaBkZVdiLbQf+FhYrovUlgotb5D
YiCQI3ofV5QTE+141jhO5Pkd3ZIoBG+P826LaT8NXhwS0o1XqVk39DCZNoFshNmR
WFZpkVCTHyv5ZhVey1NWXnD8opO375htGNV4AeSmSIH9YkURD1g5F+2t7RiosKFo
kJrfPmUjhHn8IFpReGc8qmMMZX0WaV3t+VAWfOHGGyrXDfQ4xz1VCot75C2+qypM
+qhwOAOOP0zA7CfI96ULZzSH/j8HuQk3O0DsUCybpMuKEazEMxP3tgGtRerwDaFG
jQvAlK8Rbq3v8buBI6YJuXTwSzJE8KLjleUiTFumE6WP4rsAvlP/5rBvubeMfa3n
NIMm5Rkl36Z+jt3e2Z2ZqWDPpBRta8m7QHccrZhkvqu3YC3Gl6kdnm4Vio3Xfpg2
qtWhIQutQ6DmItewV+weQHas3hl88RPJtSrfWWIIMkpbF7Y4vbX9xcnsYCLlp2Mz
tWbbnU+EWATNSsufml/Kdnu9iEEuLmeovE11I69nwjNOq9P+GJ3r/FUb2wARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQIAKQUC
Yk8t7AIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEJhfW8l0
t5NWo64P/2y7gcMRylLLW/wbrCjton2O4+YRocwQxKm1cBml9FVDUR5967YczNuu
EwEOfH/Pu3UAlrBfKAfxPNhKchLwYiOBNh2Wk5UUxRcldNHTLb5jn5gxCeWNAsl/
Tc46qY+ObdBMdOf2Vu33UCOg83WLbg1bfBoA8Bm1cd0XObtLGucu6O6EBt1dBrKq
9UTcbJfuGivY2Xjy5r4kEiMHBoLKcFrSo2Mm7VtYlE4Mabjyj9+orqUio7qxOl60
aa7Psa6rMvs1Ip9IOrAdG7o5Y29tQpeINH0R1/u47BrlTEAgG63Dfy49w2h/1g0G
c9KPXVuN55OWRIu0hsiySDMk/2ERsF348TU3NURZltnCOxp6pHlbPJIxRVTNa9Cn
f8tbLB3y3HfA80516g+qwNYIYiqksDdV2bz+VbvmCWcO+FellDZli831gyMGa5JJ
rq7d0lEr6nqjcnKiVwItTQXyFYmKTAXweQtVC72g1sd3oZIyqa7T8pvhWpKXxoJV
WP+OPBhGg/JEVC9sguhuv53tzVwayrNwb54JxJsD2nemfhQm1Wyvb2bPTEaJ3mrv
mhPUvXZj/I9rgsEq3L/sm2Xjy09nra4o3oe3bhEL8nOj11wkIodi17VaGP0y+H3s
I5zB5UztS6dy+cH+J7DoRaxzVzq7qtH/ZY2quClt30wwqDHUX1ef
=+iYX
-----END PGP PUBLIC KEY BLOCK-----
```

AWS CDK OpenPGP key (2018-06-19)

Key ID:	0x0566A784E17F3870
Type:	RSA
Size:	4096/4096
Created:	2018-06-19
Expires:	2022-06-18
User ID:	AWS CDK Team <aws-cdk@amazon.com>

Key fingerprint:	E88B E3B6 F0B1 E350 9E36 4F96 0566 A784 E17F 3870

Select the "Copy" icon to copy the following OpenPGP key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFsovE8BEADEFVCHeAVPvoQgsjVu9FPUczxy9P+2zGIT/MLI3/vPLiULQwRy
IN2oxyBNDtcDToNa/fTkW3Ev0NTP4V1h+uBoKDZD/p+dTmSDRfByECMI0sGZ3UsG
Ohhyl2Of44s0sL8gdLtDnqSRLf+ZrfT3gpgUnplW7VitkwLxr78jDpW4QD8p8dZ9
WNm3JgB55jyPgaJKqA1Ln4Vduni/1XkrG42nxrrU71uUdZPvPZ2ELLJa6n0/raG8
jq3le+xQh45gAIs6PGaAgy7jAsfbwkGTBHjjujITAY1DwvQH5iS31OaCM9n4JNpc
xGZeJAVYTLilznf2QtS/a50t+ZOmpq67Ssp2j6qYpiumm0Lo9q3K/R4/yF0FZ8SL
1TuNX0ecXEptiMVUfTiqrLsANg18EPtLZZOYW+ZkbcVytkDpiqj7bMwA7mI7zGCJ
1gjaTbcEmOmVdQYS1G6ZptwbTtvrgA6AfnZxX1HUxLRQ7tT/wvRtABfbQKAh85Ff
a3U9W4oC3c1MP5IyhNV1Wo8Zm0flZiZc0iZnojTtSG6UbcxNNL4Q8e08FWjhungj
yxSsIBnQO1Aeo1N4BbzlI+n9iaXVDUN7Kz1QEyS4PNpjvUyrUiQ+a9C5sRA7WP+x
IEOaBBGpoAXB3oLsdTNO6AcwcDd9+r2NlXlhWC4/uH2YHQUIegPqHmPWxwARAQAB
tCFBV1MgQ0RLIFRlYW0gPGF3cy1jZGtAYW1hem9uLmNvbT6JAj8EEwEIACkFAlso
vE8CGy8FCQeEzgAHCwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRAFZqeE4X84
cLGxD/0XHnhoR2xvz38GM8HQlwlZy9W1wVhQKmNDQUavw8Zx7+iRR3m7nq3xM7Qq
BDbcbKSg1lVLSBQ6H2V6vRpysOhkPSH1nN2dO8DtvSKIPcxK48+1x7lmO+ksSs/+
oo1UvOmTDaRzOitYh3kOGXHHXk/l11GtF2FGQzYssX5iM4PHcjBsK1unThs56IMh
OJeZezEYzBaskTu/ytRJ236bPP2kZIEXfzAvhmTytuXWUXEftxOxc6fIAcYiKTha
aofG7WyR+Fvb1j5gNLcbY552QMxa23NZd5cSZH7468WEW1SGJ3AdLA7k5xvsPPOC
2YvQFD+vUOZ1JJuu6B5rHkiEMhRTLklkvqXEShTxuXiCp7iTOo6TBCmrWAT4eQr7
htLmqlXrgKi8qPkWmRdXXG+MQBzI/UyZq2q8KC6cx2md1PhANmeefhiM7FZZfeNM
WLonWfh8gVCsNH5h8WJ9fxsQCADd3Xxx3NelS2zDYBPRoaqZEEBbgUP6LnWFprA2
EkSlc/RoDqZCpBGgcoy1FFWvV/ZLgNU6OTQlYH6oYOWiylSJnaTDyurrktsxJI6d
4gdsFb6tqwTGecuUPvvZaEuvhWExLxAebhu780FdAPXgVTX+YCLI2zf+dWQvkFQf
80RE7ayn7BsiaLzFBVux/zz/WgvudsZX18r8tDiVQBL51ORmqw==
=0wuQ
-----END PGP PUBLIC KEY BLOCK-----
```

jsii OpenPGP key (2018-08-06)

Key ID:	0x1C7ACE4CB2A1B93A
Type:	RSA

Size:	4096/4096
Created:	2018-08-06
Expires:	2022-08-05
User ID:	AWS JSII Team <aws-jsii@amazon.com>
Key fingerprint:	85EF 6522 4CE2 1E8C 72DB 28EC 1C7A CE4C B2A1 B93A

Select the "Copy" icon to copy the following OpenPGP key:


```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFtoSs0BEAD6WweLD0B26h0F7Jo9iR6tVQ4PgQBK1Va5H/eP+A2Iqw79UyxZ
WNzHYhzQ5MjYYI1SgcPavXy5/LV1N8HJ7QzyKszybnLYpNTLPYArWE8ZM9ZmjvIR
p1GzwnVBGQfoOlxyeutE9T5ZkAn45dTS5jlno4unji4gHjnwXKf2nP1APU2CZfdK
8vDpLOgj9LeeGlerYNbx+7xtY/I+csFIQvK09FPLSNMJQLlkBhY0r6Rt9ZQG+653
tJn+AUjyM237w0UIX1IqyYc5IONXu8HklPGu0NYuX9AY/63Ak2Cyfj0w/PZlvueQ
noQNM3j0nkOEsTOEXCyaLQw9iBKpxvLnm5RjMSODDCkj8c9uu0LHr7J4EOtgt2S1
pem7Y/c/N+/Z+Ksg9fP8fVTfYwRPvdI1x2sCiRDfLoQSG9tdrN5VwPFi4sGV04sI
x7Al8Vf/OBjAGZrDaJgM/gVvb9SKAQUA6t3ofeP14gDrS0eYodEXZ+lamnxFglxF
Sn8NRC4JFNmkXSUaTNGUdFf//F0D69PRNT8CnFfmniGj0CphN5037PCA2LC/Buq2
3+K6mTPkCcCHYPC/SwItp/xIDAQsGuDc1i1SfDYXrjsK7uOuwC5jLA9X6wZ/jgXQ
4umRRJBAV1aW8b1+yfaYYCO2AfXXO6caObv8IvH7Pc4leC2DoqylD3KklQARAQAB
tCNBV1MgSlNJSSBUZWFtIDxhd3MtanNpaUBhbWF6b24uY29tPokCPwQTAQgAKQUC
W2hKzQIbLwUJB4TOAAcLCQgHAwIBBhUIAgkKCwQWAgMBAh4BAheAAAoJEBx6zkyy
obk6B34P/iNb5QjKyhT0glZiq1wK7tuDDRpR6fC/sp6Jd/GhaNjO4BzlDbUPSjW5
950VT+qwaHXbIma/QVP7EIRztfwWy7m8eOodjpiu7JyJprhwG9nocXiNsLADcMoH
BvabkDRWXWIWSurq2wbcFMlTVwxjHPIQs6kt2oojpzP985CDS/KTzyjow6/gfMim
DLdhSSbDUM34STEgew79L2sQzL7cvM/N59k+AGyEMHZDXHkEw/Bge5Ovz50YOnsp
lisH4BzPRIw7uWqPlkVPzJKwMuo2WvMjDfgbYLbyjfvs5mqDxT2GTwAx/rd2taU6
iSqP0QmLM54BtTVVdoVXZSmJyTmXAAGlITq8ECZ/coUW9K2pUSgVuWyu63lktFP6
MyCQYRmXPh9aSd4+ielteXM9Y39snlyLgEJBhMxioZXVO2oszwluPuhPoAp4ekwj
/umVsBf6As6PoAchg7Qzr+lRZGmV9YTJOgDn2Z7jf/7tOes0g/mdiXTQMSGtp/Fp
ggnifTBx3iXkrQhqHlwtam8XTHGHy3MvX17ZslNuB8Pjh+07hhCxv0VUVZPUHJqJ
ZsLa398LMteQ8UMxwJ3t06jwDWAd7mbr2tatIilLHtWWBFoCwBh1XLe/03ENCpDp
njZ7OsBsBK2nVVcN0H2v5ey0T1yE93o6r7xOwCwBiVp5skTCRUob
=2Tag
-----END PGP PUBLIC KEY BLOCK-----
```

AWS CDK Developer Guide history

See Releases for information about AWS CDK releases. The AWS CDK is updated approximately once a week. Maintenance versions may be released between weekly releases to address critical issues. Each release includes a matched AWS CDK Toolkit (CDK CLI), AWS Construct Library, and API Reference. Updates to this Guide generally do not synchronize with AWS CDK releases.

Note

The table below represents significant documentation milestones. We fix errors and improve content on an ongoing basis.

Change	Description	Date
Document bootstrap template versions v26 and v27	For more information, see Bootstrap template version history.	March 27, 2025
Add documentation for building and deploying container image assets	When you build container image assets with the AWS CDK, Docker is utilized by default to perform these actions. If you want to use a different container management tool, you can replace Docker. For more information, see Build and deploy container image assets in CDK apps.	March 21, 2025
Add documentation for CDK Migrate feature	Use the AWS CDK CLI `cdk migrate` command to migrate deployed AWS resources, deployed AWS CloudFormation stacks, and local AWS CloudForm	February 2, 2024

	ation templates to AWS CDK. For more information, see Migrate to AWS CDK.	
IAM best practices updates	Updated guide to align with the IAM best practices . For more information, see Security best practices in IAM.	March 23, 2023
Document `cdk.json`	Add documentation of `cdk.json` configuration values.	April 20, 2022
Dependency management	Add topic on managing dependencies with the AWS CDK.	April 7, 2022
Remove double-braces from Java examples	Replace this anti-pattern with Java 9 `Map.of` throughout.	March 9, 2022
AWS CDK v2 release	Version 2 of the AWS CDK Developer Guide is released. See Releases for informati on about AWS CDK releases. The AWS CDK is updated approximately once a week. Maintenance versions may be released between weekly releases to address critical issues. Each release includes a matched AWS CDK Toolkit (CDK CLI), AWS Construct Library, and API Reference . Updates to this Guide generally do not synchronize with AWS CDK releases.	December 4, 2021