1. Accelerator Developer Guide

This document is a reference document. Instead of reading through it in linear order, you can use it to look up specific issues as needed.

It is important to read the Operations Guide before reading this document. If you're interested in actively contributing to the project, you should also review the Governance and Contributing Guide.

2. Table of Contents

- 1. Accelerator Developer Guide
- 2. Table of Contents
- 3. Technology Stack
  - 3.1. TypeScript and NodeJS
    - 3.1.1. pnpm
    - 3.1.2. prettier
    - 3.1.3. eslint
  - 3.2. CloudFormation
  - 3.3. CDK
- 4. Development
  - 4.1. Project Structure
  - 4.2. Installer Stack
  - 4.3. Initial Setup Stack
    - 4.3.1. CodeBuild and Prebuilt Docker Image
    - 4.3.2. Passing Data to Phase Steps and Phase Stacks
  - 4.4. Phase Steps and Phase Stacks
  - 4.5. Store outputs to SSM Parameter Store
    - 4.5.1. Phases and Deployments
    - 4.5.2. Passing Outputs between Phases
    - 4.5.3. Decoupling Configuration from Constructs
  - 4.6. Libraries & Tools
    - 4.6.1. CDK Assume Role Plugin
    - 4.6.2. CDK API
    - 4.6.3. AWS SDK Wrappers
    - 4.6.4. Configuration File Parsing
      - 4.6.4.1. `AcceleratorNameTagger`
      - 4.6.4.2. `AcceleratorStack`
      - 4.6.4.3. Name Generator
      - 4.6.4.4. `AccountStacks`
      - 4.6.4.5. `Vpc` and `ImportedVpc`
      - 4.6.4.6. `Limiter`
    - 4.6.5. Creating Stack Outputs
      - 4.6.5.1. Adding Tags to Shared Resources in Destination Account
    - 4.6.6. Custom Resources
      - 4.6.6.1. Externalizing `aws-sdk`
      - 4.6.6.2. cfn-response
      - 4.6.6.3. cfn-tags
      - 4.6.6.4. webpack-base
  - 4.7. Workarounds
    - 4.7.1. Stacks with Same Name in Different Regions
  - 4.8. Local Development
    - 4.8.1. Installer Stack
    - 4.8.2. Initial Setup Stack
    - 4.8.3. Phase Stacks

- 4.9. Testing
  - 4.9.1. Validating Immutable Property Changes and Logical ID Changes
  - 4.9.2. Upgrade CDK
- 5. Best Practices
  - 5.1. TypeScript and NodeJS
    - 5.1.1. Handle Unhandled Promises
  - 5.2. CloudFormation
    - 5.2.1. Cross-Account/Region References
    - 5.2.2. Resource Names and Logical IDs
    - 5.2.3. Changing Logical IDs
    - 5.2.4. Changing (Immutable) Properties
  - 5.3. CDK
    - 5.3.1. Logical IDs
    - 5.3.2. Moving Resources between Nested Stacks
    - 5.3.3. L1 vs. L2 Constructs
    - 5.3.4. CDK Code Dependency on Lambda Function Code
    - 5.3.5. Custom Resource
    - 5.3.6. Escape Hatches
      - 5.3.6.1. AutoScaling Group Metadata
      - 5.3.6.2. Secret `SecretValue`
- 6. Contributing Guidelines
  - 6.1. How-to
  - 6.2. Adding New Functionality?
  - 6.3. Create a CDK Lambda Function with Lambda Runtime Code
  - 6.4. Create a Custom Resource
  - 6.5. Run All Unit Tests
  - 6.6. Accept Unit Test Snapshot Changes
  - 6.7. Validate Code with Prettier
  - 6.8. Format Code with Prettier
  - 6.9. Validate Code with `tslint`

3. Technology Stack

We use TypeScript, NodeJS, CDK and CloudFormation. You can find some more information in the sections below.

3.1. TypeScript and NodeJS

In the following sections we describe the tools and libraries used along with TypeScript.

3.1.1. pnpm

We use the `pnpm` package manager along with `pnpm workspaces` to manage all the packages in this monorepo.

https://pnpm.js.org

https://pnpm.js.org/en/workspaces

The binary `pnpx` runs binaries that belong to `pnpm` packages in the workspace.

https://pnpm.js.org/en/pnpx-cli

3.1.2. prettier

We use `prettier` to format code in this repository. A GitHub action makes sure that all the code in a pull requests adheres to the configured `prettier` rules. See Github Actions.

3.1.3. eslint

We use `eslint` as a static analysis tool that checks our TypeScript code. A GitHub action makes sure that all the code in a pull requests adheres to the configured `eslint` rules. See Github Actions.

3.2. CloudFormation

CloudFormation deploys both the Accelerator stacks and resources and the deployed stacks and resources. See Operations Guide: System Overview for the distinction between Accelerator resources and deployed resources.

3.3. CDK

AWS CDK defines the cloud resources in a familiar programming language. While AWS CDK supports TypeScript, JavaScript, Python, Java, and C#/.Net, the contributions should be made in Typescript, as outlined in the Accelerator Development First Principles.

Developers can use programming languages to define reusable cloud components known as Constructs. You compose these together into Stacks and Apps. Learn more at https://docs.aws.amazon.com/cdk/latest/guide/home.html

4. Development

There are different types of projects in this monorepo.

1. Projects containing CDK code that compiles to CloudFormation templates and deploy to AWS using the CDK toolkit;
2. Projects containing runtime code that is used by the CDK code to deploy Lambda functions;
3. Projects containing reusable code; both for use by the CDK code and/or runtime code.

The CDK code either deploys Accelerator-management resources or Accelerator-managed resources. See the Operations Guide for the distinction between Accelerator-management and Accelerator-managed resources.

The only language used in the project is TypeScript and exceptionally JavaScript. We do not write CloudFormation templates, only CDK code.

When we want to enable functionality in a managed account we try to

1. use native CloudFormation/CDK resource to enable the functionality;
2. create a custom resource to enable the functionality; or
3. lastly create a new step in the `Initial Setup` state machine to enable the functionality.

4.1. Project Structure

The folder structure of the project is as follows:

- `src/installer/cdk`: See Installer Stack;
- `src/core/cdk`: See Initial Setup Stack;
- `src/core/runtime` See Initial Setup Stack and Phase Steps and Phase Stacks;
- `src/deployments/runtime` See Phase Steps and Phase Stacks;
- `src/deployments/cdk`: See Phase Steps and Phase Stacks;
- `src/lib/accelerator-cdk`: See Libraries & Tools;
- `src/lib/cdk-constructs`: See Libraries & Tools;
- `src/lib/cdk-plugin-assume-role`: See CDK Assume Role Plugin.
- `src/lib/common-config`: See Libraries & Tools;
- `src/lib/common-outputs`: See Libraries & Tools;
- `src/lib/common-types`: See Libraries & Tools;
- `src/lib/common`: See Libraries & Tools;
- `src/lib/custom-resources/**/cdk`: See Custom Resources;
- `src/lib/custom-resources/**/runtime`: See Custom Resources;

4.2. Installer Stack

Read the Operations Guide first before reading this section. This section is a technical addition to the section in the Operations Guide.

As stated in the Operations Guide, the `Installer` stack is responsible for installing the `Initial Setup` stack. It is an Accelerator-management resource. The main resource in the `Installer` stack is the `PBMMAccel-Installer` CodePipeline. The CodePipeline uses this GitHub repository as source action and runs CDK in a CodeBuild step to deploy the `Initial Setup` stack.

```
new codebuild.PipelineProject(stack, 'InstallerProject', {
  buildSpec: codebuild.BuildSpec.fromObject({
    version: '0.2',
    phases: {
      install: {
        'runtime-versions': {
          nodejs: 12,
        },
        // The flag '--unsafe-perm' is necessary to run pnpm scripts in Docker
        commands: ['npm install --global pnpm', 'pnpm install --unsafe-perm --frozen-lockfile'],
      },
      build: {
        commands: [
          'cd src/core/cdk',
          // Bootstrap the environment for use by CDK
          'pnpx cdk bootstrap --require-approval never',
          // Deploy the Initial Setup stack
          'pnpx cdk deploy --require-approval never',
        ],
      },
    },
  }),
});
```

When the CodePipeline finishes deploying the `Initial Setup` stack, it starts a Lambda function that starts the execution of the `Initial Setup` stack's main state machine.

The `Initial Setup` stack deployment receives environment variables from the CodePipeline's CodeBuild step. The most notable environment variables are:

- `ACCELERATOR_STATE_MACHINE_NAME`: The `Initial Setup` will use this name for the main state machine. So it is the `Installer` stack that decides the name of the main state machine. This way we can confidently start the main state machine of the `Initial Setup` stack from the CodePipeline;
- `ENABLE_PREBUILT_PROJECT`: See Prebuilt Docker Image.

4.3. Initial Setup Stack

Read Operations Guide first before reading this section. This section is a technical addition to the section in the Operations Guide.

As stated in the Operations Guide, the `Initial Setup` stack consists of a state machine, named `PBMMAccel-MainStateMachine_s` which executes steps to create the Accelerator-managed stacks and resources in the managed accounts. It is an Accelerator-management resource.

The `Initial Setup` stack is defined in the `src/core/cdk` folder.

The `Initial Setup` stack is similar to the `Installer` stack, as in that it runs a CodeBuild project to deploy others stacks using CDK. In case of the `Initial Setup` stack

- we use a AWS Step Functions State Machine to run steps instead of using a CodePipeline;
- we deploy multiple stacks, called `Phase` stacks, in Accelerator-managed accounts. These `Phase` stacks contain Accelerator-managed resources.

In order to install these `Phase` stacks in Accelerator-managed accounts, we need access to those accounts. We create a stack set in the Organization Management (root) account that has instances in all Accelerator-managed accounts. This stack set contains what we call the `PipelineRole`.

The code for the steps in the state machine is in `src/core/runtime`. All the steps are in different files but are compiled into a single file. We used to compile all the steps separately but we would hit a limit in the amount of parameters in the generated CloudFormation template. Each step would have its own CDK asset that would introduce three new parameters. We quickly reached the limit of 60 parameters in a CloudFormation template and decided to compile the steps into a single file and use it across all different Lambda functions.

4.3.1. CodeBuild and Prebuilt Docker Image

The CodeBuild project that deploys the different `Phase` stacks is constructed using the `CdkDeployProject` or `PrebuiltCdkDeployProject` based on the value of the environment variable `ENABLE_PREBUILT_PROJECT`.

The first, `CdkDeployProject` constructs a CodeBuild project that copies this whole Github repository as a ZIP file to S3 using CDK S3 assets. This ZIP file is then used as source for the CodeBuild project. When the CodeBuild project executes, it runs `pnpm recursive install` which in turn will run all `prepare` scripts in all `package.json` files in the project -- as described in section CDK Code Dependency on Lambda Function Code.

After installing the dependencies, the CodeBuild project deploys the `Phase` stacks.

```
cd src/deployments/cdk
sh codebuild-deploy.sh
```

We have more than 50 workspace projects in the monorepo with a `prepare` script, so the `pnpm recursive install` step can take some time. Also, the CodeBuild project will run for each deployed `Phase` stack in each Accelerator-managed account.

This is where the `PrebuiltCdkDeployProject` CodeBuild project comes in. The `PrebuiltCdkDeployProject` contains a Docker image that contains the whole project in the `/app` directory and has all the dependencies already installed.

```
FROM node:12-alpine3.11
# Install the package manager
RUN npm install --global pnpm
RUN mkdir /app
WORKDIR /app
# Copy over the project root to the /app directory
ADD . /app/
# Install the dependencies
RUN pnpm install --unsafe-perm --frozen-lockfile
```

When this CodeBuild project executes, it uses the Docker image as base -- the dependencies are already installed -- and runs the same commands as the `CdkDeployProject` to deploy the `Phase` stacks.

4.3.2. Passing Data to Phase Steps and Phase Stacks

Some steps in the state machine write data to Amazon DynamoDB. This data is necessary to deploy the `Phase` stacks later on. At one time this data was written to Secrets Manager and/or S3, these mechanisms were deemed ineffective due to object size limitations or consistency challenges and were all eventually migrated to DynamoDB.

- `Load Accounts` step: This step finds the Accelerator-managed accounts in AWS Organizations and stores the account key -- the key of the account in `mandatory-account-configs` or `workload-account-configs` object in the Accelerator config -- and account ID and other useful information in the `PBMMAccel-Parameters` table, `accounts/#` key and `accounts-items-count` key;
- `Load Organizations` step: More or less the same as the `Load Accounts` step but for organizational units in AWS Organizations and stores the values in the `PBMMAccel-Parameters` table, `organizations` key;
- `Load Limits` step: This step requests limit increases for Accelerator-managed accounts and stores the current limits in the the `PBMMAccel-Parameters` table, `limits` key.
- `Store Phase X Output`: This step loads stack outputs from all existing `Phase` stacks and stores the outputs in the DynamoDB table `PBMMAccel-Outputs`.

Other data is passed through environment variables:

- `ACCELERATOR_NAME`: The name of the Accelerator;
- `ACCELERATOR_PREFIX`: The prefix for all named Accelerator-managed resources;
- `ACCELERATOR_EXECUTION_ROLE_NAME`: The name of the execution role in the Accelerator-managed accounts. This is the `PipelineRole` we created using stack sets.

4.4. Phase Steps and Phase Stacks

Read Operations Guide first before reading this section. This section is a technical addition to the *Deploy Phase X* sections in the Operations Guide.

The `Phase` stacks contain the Accelerator-managed resources. The reason the deployment of Accelerator-managed resources is split into different phases is because there cannot be cross account/region references between CloudFormation stacks. See Cross-Account/Region References.

The `Phase` stacks are deployed by a CodeBuild project in the `Initial Setup` stack as stated in the previous paragraphs. The CodeBuild project executes the `codebuild-deploy.sh` script. See `initial-setup.ts`.

The `codebuild-deploy.sh` script executes the `cdk.ts` file.

The `cdk.ts` file is meant as a replacement for the `cdk` CLI command. To deploy a phase stack you would **not** run `pnpx cdk deploy` but `cdk.sh --phase 1`. See CDK API for more information why we use the CDK API instead of using the CDK CLI.

The `cdk.ts` command parses command line arguments and creates all the `cdk.App` for all accounts and regions for the given `--phase`. When you pass the `--region` or `--account-key` command, all the `cdk.App` for all accounts and regions will still be created, except that only the `cdk.App`s matching the parameters will be deployed. This behavior could be optimized in the future. See Stacks with Same Name in Different Regions for more information why we're creating multiple `cdk.App`s.

4.5. Store outputs to SSM Parameter Store

Customers need the CloudFormation outputs of resources that are created by the accelerator in order to deploy their own resources in AWS. eg. vpcId in shared-network account to create an ec2 instance, etc.

This step loads the stack outputs from our DynamoDB Table `PBMMAccel-Outputs` and stores as key value pairs in SSM Parameter Store in each account.

Example values are

- /PBMMAccel/network/vpc/1/name => Endpoint
- /PBMMAccel/network/vpc/1/id => vpc-XXXXXXXXXX

`PBMMAccel-Outputs-Utils` DynamoDB Table is used extensively to maintain same index irrespective of configuration changes.

This allows customers to reliably build Infrastructure as Code (IaC) which depends on accelerator deployed objects like VPC's, security groups, subnets, ELB's, KMS keys, IAM users and policies. Rather than making the parameters dependent on object names, we used an indexing scheme, which we maintain and don't update as a customers configuration changes. We have attempted to keep the index values consistent across accounts (based on the config file), such that when code is propoted through the SDLC cycle from Dev to Test to Prod, the input parameters to the IaC scripts do not need to be updated, the App subnet, for example, will have the same index value in all accounts.

4.5.1. Phases and Deployments

The `cdk.ts` file calls the `deploy` method in the `apps/app.ts`. This `deploy` method loads the Accelerator configuration, accounts, organizations from DynamoDB; loads the stack outputs from Amazon DynamoDB; and loads required environment variables.

```
/**
 * Input to the `deploy` method of a phase.
 */
export interface PhaseInput {
  // The config.json file
  acceleratorConfig: AcceleratorConfig;
  // Auxiliary class to construct stacks
  accountStacks: AccountStacks;
  // The list of accounts, their key in the configuration file and their ID
  accounts: Account[];
  // The parsed environment variables
  context: Context;
  // The list of stack outputs from previous phases
  outputs: StackOutput[];
  // Auxiliary class to manage limits
  limiter: Limiter;
}
```

It is important to note that no configuration is hard-coded. The CloudFormation templates are generated by CDK and the CDK constructs are created according to the configuration file. Changes to the configuration will change the CDK construct tree and that will result in a different CloudFormation template that is deployed.

The different phases are defined in `apps/phase-x.ts`. Historically we created all CDK constructs in the `phase-x.ts` files. After a while the `phase-x.ts` files started to get too big and we moved to separating the logic into separate deployments. Every logical component has a separate folder in the `deployments` folder. Every `deployment` consists of so-called steps. Separate steps are put in loaded in phases.

For example, take the `deployments/defaults` deployment. The deployment consists of two steps, i.e. `step-1.ts` and `step-2.ts`. `deployments/defaults/step-1.ts` is created in `apps/phase-0.ts` and `deployments/defaults/step-2.ts` is created in `apps/phase-1.ts`. You can find more details about what happens in each phase in the Operations Guide.

`apps/phase-0.ts`

```
export async function deploy({ acceleratorConfig, accountStacks, accounts, context }: PhaseInput) {
  // Create defaults, e.g. S3 buckets, EBS encryption keys
  const defaultsResult = await defaults.step1({
    acceleratorPrefix: context.acceleratorPrefix,
    accountStacks,
    accounts,
    config: acceleratorConfig,
  });
```

`apps/phase-1.ts`

```
export async function deploy({ acceleratorConfig, accountStacks, accounts, outputs }: PhaseInput) {
  // Find the central bucket in the outputs
  const centralBucket = CentralBucketOutput.getBucket({
    accountStacks,
    config: acceleratorConfig,
    outputs,
  });

  // Find the log bucket in the outputs
  const logBucket = LogBucketOutput.getBucket({
    accountStacks,
    config: acceleratorConfig,
    outputs,
  });

  // Find the account buckets in the outputs
  const accountBuckets = await defaults.step2({
    accounts,
    accountStacks,
    centralLogBucket: logBucket,
    config: acceleratorConfig,
  });
}
```

4.5.2. Passing Outputs between Phases

The CodeBuild step that is responsible for deploying a `Phase` stack runs in the Organization Management (root) account. We wrote a CDK plugin that allows the CDK deploy step to assume a role in the Accelerator-managed account and create the CloudFormation `Phase` stack in the managed account. See CDK Assume Role Plugin.

After a `Phase-X` is deployed in all Accelerator-managed accounts, a step in the `Initial Setup` state machine collects all the `Phase-X` stack outputs in all Accelerator-managed accounts and regions and stores theses outputs in DynamoDB.

Then the next `Phase-X+1` deploys using the outputs from the previous `Phase-X` stacks.

See Creating Stack Outputs for helper constructs to create outputs.

4.5.3. Decoupling Configuration from Constructs

At the start of the project we created constructs that had tight coupling to the Accelerator config structure. The properties to instantiate a construct would sometimes have a reference to an Accelerator-specific interface. An example of this is the `Vpc` construct in `src/deployments/cdk/common/vpc.ts`.

Later on in the project we started decoupling the Accelerator config from the construct properties. Good examples are in `src/lib/cdk-constructs/`.

Decoupling the configuration from the constructs improves reusability and robustness of the codebase.

4.6. Libraries & Tools

4.6.1. CDK Assume Role Plugin

At the time of writing, CDK does not support cross-account deployments of stacks. It is possible however to write a CDK plugin and implement your own credential loader for cross-account deployment.

We wrote a CDK plugin that can assume a role into another account. In our case, the Organization Management (root) account will assume the `PipelineRole` in an Accelerator-managed account to deploy stacks.

4.6.2. CDK API

We use the internal CDK API to deploy the `Phase` stacks instead of the CDK CLI for the following reasons:

- It allows us to deploy multiple stacks in parallel;
- Disable stack termination before destroying a stack;
- Delete a stack after it initially failed to create;
- Deploy multiple apps at the same time -- see Stacks with Same Name in Different Regions.

The helper class `CdkToolkit` in `toolkit.ts` wraps around the CDK API.

The risk of using the CDK API directly is that the CDK API can change at any time. There is no stable API yet. When upgrading the CDK version, the `CdkToolkit` wrapper might need to be adapted.

4.6.3. AWS SDK Wrappers

You can find `aws-sdk` wrappers in the `src/lib/common/src/aws` folder. Most of the classes and functions just wrap around `aws-sdk` classes and implement promises and exponential backoff to retryable errors. Other classes, like `Organizations` have additional functionality such as listing all the organizational units in an organization in the function `listOrganizationalUnits`.

Please use the `aws-sdk` wrappers throughout the project or write an additional wrapper when necessary.

4.6.4. Configuration File Parsing

The configuration file is defined and validated using the `io-ts` library. See `src/lib/common-config/src/index.ts`. In case any changes need to be made to the configuration file parsing, this is the place to be.

We wrap a class around the `AcceleratorConfig` type that contains additional helper functions. You can add your own additional helper functions.

4.6.4.1. `AcceleratorNameTagger`

`AcceleratorNameTagger` is a CDK aspect that sets the name tag on specific resources based on the construct ID of the resource.

The following example illustrates its purpose.

```
const stack = new cdk.Stack();
new ec2.CfnVpc(stack, 'SharedNetwork', {});
Aspects.of(stack).add(new AcceleratorNameTagger());
```

The example above synthesizes to the following CloudFormation template.

```
Resources:
  SharedNetworkAB7JKF7:
    Properties:
      Tags:
        - Key: Name
          Value: SharedNetwork_vpc
```

4.6.4.2. `AcceleratorStack`

`AcceleratorStack` is a class that extends `cdk.Stack` and adds the `Accelerator` tag to all resources in the stack. It also applies the aspect `AcceleratorNameTagger`.

It is also used by the `accelerator-name-generator.ts` functions to find the name of the `Accelerator`.

4.6.4.3. Name Generator

The `accelerator-name-generator.ts` file contains methods that create names for resources that are optionally prefixed with the Accelerator name, and optionally suffixed with a hash based on the path of the resource, the account ID and region of the stack.

The functions should be used to create pseudo-random names for IAM roles, KMS keys, key pairs and log groups.

4.6.4.4. `AccountStacks`

`AccountStacks` is a class that manages the creation of an `AcceleratorStack` based on a given account key and region. If an account with the given account key cannot be found in the accounts object -- which is loaded by `apps/app.ts` then no stack will be created. This class is used extensively throughout the phases and deployment steps.

```
export async function step1(props: CertificatesStep1Props) {
  const { accountStacks, centralBucket: centralBucket, config } = props;

  for (const { accountKey, certificates } of config.getCertificateConfigs()) {
    if (certificates.length === 0) {
      continue;
    }

    const accountStack = accountStacks.tryGetOrCreateAccountStack(accountKey);
    if (!accountStack) {
      console.warn(`Cannot find account stack ${accountKey}`);
      continue;
    }

    for (const certificate of certificates) {
      createCertificate({
        centralBucket,
        certificate,
        scope: accountStack,
      });
    }
  }
}
```

4.6.4.5. `Vpc` and `ImportedVpc`

`Vpc` is an interface in the `src/lib/cdk-constructs/src/vpc/vpc.ts` file that attempts to define an interface for a VPC. The goal of the interface is to be implemented by a `cdk.Construct` that implements the interface. This CDK issue provides more background [https://github.com/aws/aws-cdk/issues/5927].

Another goal of the interface is to provide an interface on top of imported VPC outputs. This is what the `ImportedVpc` class implements. The class loads outputs from VPC in a previous phase and implements the `Vpc` interface on top of those outputs.

4.6.4.6. `Limiter`

So far we haven't talked about limits yet. There is a step in the `Initial Setup` state machine that requests limit increases according to the desired limits in the configuration file. The step saves the current limits to the `limits` key in the DynamoDB table `PBMMAccel-Parameters`. The `apps/app.ts` file loads the limits and passes them as an input to the phase deployment.

The `Limiter` class helps keeps track of resource we create and prevents exceeding these limits.

```
for (const { ouKey, accountKey, vpcConfig, deployments } of acceleratorConfig.getVpcConfigs()) {
  if (!limiter.create(accountKey, Limit.VpcPerRegion, region)) {
    console.log(`Skipping VPC "${vpcConfig.name}" deployment.`);
    console.log(`Reached maximum VPCs per region for account "${accountKey}" and region "${region}"`);
    continue;
  }

  createVpc({ ouKey, accountKey, vpcConfig });
}
```

> *Action Item:* This functionality could be redesigned to scan all the constructs in a `cdk.App` and remove resource that are exceeding any limits.

4.6.5. Creating Stack Outputs

Initially we would create stack outputs like this:

```
new cdk.CfnOutput(stack, 'BucketOutput', {
  value: bucket.bucketArn,
});
```

But then we'd get a lot of outputs in a stack. We started some outputs together using JSON. This allowed us to store structured data inside the stack outputs.

```
new JsonOutputValue(stack, 'Output', {
  type: 'FirewallInstanceOutput',
  value: {
    instanceId: instance.instanceId,
    name: firewallConfig.name,
    az,
  },
});
```

Using the solution above, we'd not have type checking when reading or writing outputs. That's what the class `StructuredOutputValue` has a solution for. It uses the `io-ts` library to serialize and deserialize structured types.

```
export const FirewallInstanceOutput = t.interface(
  {
    id: t.string,
    name: t.string,
    az: t.string,
  },
  'FirewallInstanceOutput',
);

export type FirewallInstanceOutput = t.TypeOf<typeof FirewallInstanceOutput>;

new StructuredOutputValue<FirewallInstanceOutput>(stack, 'Output', {
  type: FirewallInstanceOutput,
  value: {
    instanceId: instance.instanceId,
    name: firewallConfig.name,
    az,
  },
});
```

And we can even improve on this a bit more.

```
export const CfnFirewallInstanceOutput = createCfnStructuredOutput(FirewallInstanceOutput);

new CfnFirewallInstanceOutput(stack, 'Output', {
  vpcId: vpc.ref,
  vpcName: vpcConfig.name,
});
```

```
export const FirewallInstanceOutputFinder = createStructuredOutputFinder(FirewallInstanceOutput, () => ({}

// Create an OutputFinder
const firewallInstances = FirewallInstanceOutputFinder.findAll({
  outputs,
  accountKey,
});

// Example usage of the OutputFinder
const firewallInstance = firewallInstances.find(i => i.name === target.name && i.az === target.az);
```

Generally you would place the output type definition inside `src/lib/common-outputs` along with the output finder. Then in the deployment folder in `src/deployments/cdk/deployments` you would create an `output.ts` file where you would define the CDK output type with `createCfnStructuredOutput`. You would not define the CDK output type in `src/lib/common-outputs` since that project is also used by runtime code that does not need to know about CDK and CloudFormation.

4.6.5.1. Adding Tags to Shared Resources in Destination Account

There is another special type of output, `AddTagsToResourcesOutput`. It can be used to attach tags to resources that are shared into another account.

```
new AddTagsToResourcesOutput(this, 'OutputSharedResourcesSubnets', {
  dependencies: sharedSubnets.map(o => o.subnet),
  produceResources: () =>
    sharedSubnets.map(o => ({
      resourceId: o.subnet.ref,
      resourceType: 'subnet',
      sourceAccountId: o.sourceAccountId,
      targetAccountIds: o.targetAccountIds,
      tags: o.subnet.tags.renderTags(),
    })),
});
```

This will add the outputs to the stack in the account that is initiating the resource share.

Next, the state machine step `Add Tags to Shared Resources` looks for all those outputs. The step will assume the `PipelineRole` in the `targetAccountIds` and attach the given tags to the shared resource.

4.6.6. Custom Resources

There are different ways to create a custom resource using CDK. See the Custom Resource section for more information.

All custom resources have a `README.md` that demonstrates their usage.

4.6.6.1. Externalizing `aws-sdk`

Some custom resources set the `aws-sdk` as external dependency and some do not.

Example of setting `aws-sdk` as external dependency.

`src/lib/custom-resources/cdk-kms-grant/runtime/package.json`

```
{
  "externals": ["aws-lambda", "aws-sdk"],
  "dependencies": {
    "aws-lambda": "1.0.6",
    "aws-sdk": "2.631.0"
  }
}
```

Example of setting `aws-sdk` as embedded dependency.

`src/lib/custom-resources/cdk-guardduty-enable-admin/runtime/package.json`

```
{
  "externals": ["aws-lambda"],
  "dependencies": {
    "aws-lambda": "1.0.6",
    "aws-sdk": "2.711.0"
  }
}
```

Setting the `aws-sdk` library as external is sometimes necessary when a newer `aws-sdk` version is necessary for the Lambda runtime code. At the time of writing the NodeJS 12 runtime uses `aws-sdk` version `2.631.0`

For example the method `AWS.GuardDuty.enableOrganizationAdminAccount` was only introduced in `aws-sdk` version `2.660`. That means that Webpack has to embed the `aws-sdk` version specified in `package.json` into the compiled JavaScript file. This can be achieved by removing `aws-sdk` from the `external` array.

`src/lib/custom-resources/cdk-kms-grant/runtime/package.json`

4.6.6.2. cfn-response

This library helps you send a custom resource response to CloudFormation.

`src/lib/custom-resources/cdk-kms-grant/runtime/src/index.ts`

```
export const handler = errorHandler(onEvent);

async function onEvent(event: CloudFormationCustomResourceEvent) {
  console.log(`Creating KMS grant...`);
  console.log(JSON.stringify(event, null, 2));

  // eslint-disable-next-line default-case
  switch (event.RequestType) {
    case 'Create':
      return onCreate(event);
    case 'Update':
      return onUpdate(event);
    case 'Delete':
      return onDelete(event);
  }
}
```

4.6.6.3. cfn-tags

This library helps you send attaching tags to resource created in a custom resource.

4.6.6.4. webpack-base

This library defines the base Webpack template to compile custom resource runtime code.

`src/lib/custom-resources/cdk-kms-grant/runtime/package.json`

```
{
  "name": "@aws-accelerator/custom-resource-kms-grant-runtime",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "prepare": "webpack-cli --config webpack.config.ts"
  },
  "source": "src/index.ts",
  "main": "dist/index.js",
  "types": "dist/index.d.ts",
  "externals": ["aws-lambda", "aws-sdk"],
  "devDependencies": {
    "@aws-accelerator/custom-resource-runtime-webpack-base": "workspace:^0.0.1",
    "@types/aws-lambda": "8.10.46",
    "@types/node": "12.12.6",
    "ts-loader": "7.0.5",
    "typescript": "3.8.3",
    "webpack": "4.42.1",
    "webpack-cli": "3.3.11"
  },
  "dependencies": {
    "@aws-accelerator/custom-resource-runtime-cfn-response": "workspace:^0.0.1",
    "aws-lambda": "1.0.6",
    "aws-sdk": "2.668.0"
  }
}
```

`src/lib/custom-resources/cdk-ec2-image-finder/runtime/webpack.config.ts`

```
import { webpackConfigurationForPackage } from '@aws-accelerator/custom-resource-runtime-webpack-base';
import pkg from './package.json';

export default webpackConfigurationForPackage(pkg);
```

4.7. Workarounds

4.7.1. Stacks with Same Name in Different Regions

The reason we're creating a `cdk.App` per account and per region and per phase is because stack names across environments might overlap, and at the time of writing, the CDK CLI does not handle stacks with the same name well. For example, when there is a stack `Phase1` in `us-east-1` and another stack `Phase1` in `ca-central-1`, the stacks will both be synthesized by CDK to the `cdk.out/Phase1.template.json` file and one stack will overwrite another's output. Using multiple `cdk.Apps` overcomes this issues as a different `outdir` can be set on each `cdk.App`. These `cdk.Apps` are managed by the `AccountStacks` abstraction.

4.8. Local Development

4.8.1. Installer Stack

Use CDK to synthesize the CloudFormation template.

```
cd src/installer/cdk
pnpx cdk synth
```

The installer template file is now in `cdk.out/AcceleratorInstaller.template.json`. This file can be used to install the installer stack.

You can also deploy the installer stack directly from the command line but then you'd have to pass some stack parameters. See CDK documentation: Deploying with parameters.

```
cd accelerator/installer
pnpx cdk deploy --parameters GithubBranch=main --parameters ConfigS3Bucket=pbmmaccel-myconfigbucket
```

4.8.2. Initial Setup Stack

There is a script called `cdk.sh` in `src/core/cdk` that allows you to deploy the Initial Setup stack.

The script sets the required environment variables and makes sure all workspace projects are built before deploying the CDK stack.

4.8.3. Phase Stacks

There is a script called `cdk.sh` in `src/deployments/cdk` that allows you to deploy a phase stack straight from the command-line without having to deploy the Initial Setup stack first.

The script enables development mode which means that accounts, organizations, configuration, limits and outputs will be loaded from the local environment instead of loading the values from DynamoDB. The local files that need to be available in the `src/deployments/cdk` folder are the following.

1. `accounts.json` based on `accelerator/accounts` (-Parameters table)

```json
[
  {
    "key": "shared-network",
    "id": "000000000001",
    "arn": "arn:aws:organizations::000000000000:account/o-0123456789/000000000001",
    "name": "myacct-pbmm-shared-network",
    "email": "myacct+pbmm-mandatory-shared-network@example.com",
    "ou": "core"
  },
  {
    "key": "operations",
    "id": "000000000002",
    "arn": "arn:aws:organizations::000000000000:account/o-0123456789/000000000002",
    "name": "myacct-pbmm-operations",
    "email": "myacct+pbmm-mandatory-operations@example.com",
    "ou": "core"
  }
]
```

2. `organizations.json` based on `accelerator/organizations` (-Parameters table)

```
[
  {
    "ouId": "ou-0000-00000000",
    "ouArn": "arn:aws:organizations::000000000000:ou/o-0123456789/ou-0000-00000000",
    "ouName": "core",
    "ouPath": "core"
  },
  {
    "ouId": "ou-0000-00000001",
    "ouArn": "arn:aws:organizations::000000000000:ou/o-0123456789/ou-0000-00000001",
    "ouName": "prod",
    "ouPath": "prod"
  }
]
```

3. `limits.json` based on `accelerator/limits` (-Parameters table)

```
[
  {
    "accountKey": "shared-network",
    "limitKey": "Amazon VPC/VPCs per Region",
    "serviceCode": "vpc",
    "quotaCode": "L-F678F1CE",
    "value": 15,
    "region": "ca-central-1"
  },
  {
    "accountKey": "shared-network",
    "limitKey": "Amazon VPC/Interface VPC endpoints per VPC",
    "serviceCode": "vpc",
    "quotaCode": "L-29B6F2EB",
    "value": 50,
    "region": "ca-central-1"
  }
]
```

4. `outputs.json` based on the -Outputs table

```
[
  {
    "accountKey": "shared-network",
    "outputKey": "DefaultBucketOutputC7CE5936",
    "outputValue": "{\"type\":\"AccountBucket\",\"value\":{\"bucketArn\":\"arn:aws:s3:::pbmmaccel-sharedne
  }
]
```

5. `context.json` that contains the default values for values that are otherwise passed as environment variables.

```
{
  "acceleratorName": "PBMM",
  "acceleratorPrefix": "PBMMAccel-",
  "acceleratorExecutionRoleName": "PBMMAccel-PipelineRole",
  "defaultRegion": "ca-central-1"
}
```

6. `config.json` that contains the Accelerator configuration.

The script also sets the default execution role to allow CDK to assume a role in subaccounts to deploy the phase stacks.

Now that you have all the required local files you can deploy the phase stacks using `cdk.sh`.

```
cd src/deployments/cdk
./cdk.sh deploy --phase 1                             # deploy all phase 1 stacks
./cdk.sh deploy --phase 1 --parallel                  # deploy all phase 1 stacks in parallel
./cdk.sh deploy --phase 1 --account shared-network    # deploy phase 1 stacks for account shared-network i
./cdk.sh deploy --phase 1 --region ca-central-1       # deploy phase 1 stacks for region ca-central-1 for
./cdk.sh deploy --phase 1 --account shared-network --region ca-central-1 # deploy phase 1 stacks for accou
```

Other CDK commands are also available.

```
cd src/deployments/cdk
./cdk.sh bootstrap --phase 1
./cdk.sh synth --phase 1
```

4.9. Testing

We use `jest` for unit testing. There are no integration tests but this could be set-up by configuring the `Installer` CodePipeline to have a webhook on the repository and deploying changes automatically.

To run unit tests locally you can run the following command in the monorepo.

```
pnpx recursive run test -- --pass-with-no-tests --silent
```

See CDK's documentation on Testing constructs for more information on how to tests CDK constructs.

4.9.1. Validating Immutable Property Changes and Logical ID Changes

The most important unit test in this project is one that validates that logical IDs and immutable properties do not change unexpectedly. To avoid the issues described in section Resource Names and Logical IDs, Changing Logical IDs and Changing (Immutable) Properties.

This test can be found in the `src/deployments/cdk/test/apps/unsupported-changes.spec.ts` file. It synthesizes the `Phase` stacks using mocked outputs and uses `jest` snapshots to compare against future changes.

The test will fail when changing immutable properties or changing logical IDs of existing resources. In case the changes are expected then the snapshots will need to be updated. You can update the snapshots by running the following command.

```
pnpx run test -- -u
```

See Accept Unit Test Snapshot Changes.

4.9.2. Upgrade CDK

There's a test in the file `src/deployments/cdk/test/apps/unsupported-changes.spec.ts` that is currently commented out. The test takes a snapshot of the whole `Phase` stack and compares the snapshot to changes in the code.

```
test('templates should stay exactly the same', () => {
  for (const [stackName, resources] of Object.entries(stackResources)) {
    // Compare the relevant properties to the snapshot
    expect(resources).toMatchSnapshot(stackName);
  }
});
```

Before upgrading CDK we uncomment this test. We run the test to update all the snapshots. Then we update all CDK versions and run the test again to compare the snapshots with the code using the new CDK version. If the test passes, then the upgrade should be stable.

Action Item: Automate this process.

5. Best Practices

5.1. TypeScript and NodeJS

5.1.1. Handle Unhandled Promises

Entry point TypeScript files -- files that start execution instead of just defining methods and classes -- should have the following code snippet at the start of the file.

```
process.on('unhandledRejection', (reason, _) => {
  console.error(reason);
  process.exit(1);
});
```

This prevents unhandled promise rejection errors by NodeJS. Please read https://medium.com/dailyjs/how-to-prevent-your-node-js-process-from-crashing-5d40247b8ab2 for more information.

5.2. CloudFormation

5.2.1. Cross-Account/Region References

When managing multiple AWS accounts, the Accelerator may need permissions to modify resources in the managed accounts. For example, a transit gateway could be created in a shared network account and it need to be shared to the perimeter account to create a VPN connection.

In a single-account environment we would could just:

1. create a single stack and use `!Ref` to refer to the transit gateway;
2. or deploy two stacks
   - one stack that contains the transit gateway and creates a CloudFormation exported output that contains the transit gateway ID;
   - another stack that imports the exported output value from the previous stack and uses it to create a VPN connection.

In a multi-account environment this is not possible and we had to find a way to share outputs across accounts and regions.

See Passing Outputs Between Phases.

5.2.2. Resource Names and Logical IDs

Some resources, like `AWS::S3::Bucket`, can have an explicit name. Setting an explicit name can introduce some possible issues.

The first issue that could occur goes as follows:

- the named resource has a retention policy to retain the resource after deleting;
- then the named resource is created through a CloudFormation stack;
- next, an error happens while creating or updating the stack and the stack rolls back;

- and finally the named resource is deleted from the stack but has a retention policy to retain, so the resource not be deleted;

Suppose then that the stack creation issue is resolved and we retry to create the named resource through the CloudFormation stack:

- the named resource is created through a CloudFormation stack;
- the named resource will fail to create because a resource with the given name already exists.

The best way to prevent this issue from happening is to not explicitly set a name for the resource and let CloudFormation generate the name.

Another issue could occur when changing the logical ID of the named resource. This is documented in the following section.

5.2.3. Changing Logical IDs

When changing the logical ID of a resource CloudFormation assumes the resource is a new resource since it has a logical ID it does not know yet. When updating a stack, CloudFormation will always prioritize resource creation before deletion.

The following issue could occur when the resource has an explicit name. CloudFormation will try to create the resource anew and will fail since a resource with the given name already exists. Example of resources where this could happen are `AWS::S3::Bucket`, `AWS::SecretManager::Secret`.

5.2.4. Changing (Immutable) Properties

Not only changing logical IDs could cause CloudFormation to replace resources. Changing immutable properties also cause replacement of resources. See Update behaviors of stack resources.

Be especially careful when:

- changing immutable properties for a named resource. Example of a resource is `AWS::Budgets::Budget`, `AWS::ElasticLoadBalancingV2::LoadBalancer`.
- updating network interfaces for an `AWS::EC2::Instance`. Not only will this cause the instance to re-create, it will also fail to attach the network interfaces to the new EC2 instance. CloudFormation creates the new EC2 instance first before deleting the old one. It will try to attach the network interfaces to the new instance, but the network interfaces are still attached to the old instance and CloudFormation will fail.

For some named resources, like `AWS::AutoScaling::LaunchConfiguration` and `AWS::Budgets::Budget`, we append a hash to the name of the resource that is based on its properties. This way when an immutable property is changed, the name will also change, and the resource will be replaced successfully. See for example `src/lib/cdk-constructs/src/autoscaling/launch-configuration.ts` and `src/lib/cdk-constructs/src//billing/budg`

```
export type LaunchConfigurationProps = autoscaling.CfnLaunchConfigurationProps;

/**
 * Wrapper around CfnLaunchConfiguration. The construct adds a hash to the launch configuration name that
 * the launch configuration properties. The hash makes sure the launch configuration gets replaced correct
 * CloudFormation.
 */
export class LaunchConfiguration extends autoscaling.CfnLaunchConfiguration {
  constructor(scope: cdk.Construct, id: string, props: LaunchConfigurationProps) {
    super(scope, id, props);

    if (props.launchConfigurationName) {
      const hash = hashSum({ ...props, path: this.node.path });
      this.launchConfigurationName = `${props.launchConfigurationName}-${hash}`;
    }
  }
}
```

5.3. CDK

CDK makes heavy use of CloudFormation so all best practices that apply to CloudFormation also apply to CDK.

5.3.1. Logical IDs

The logical ID of a CDK component is calculated based on its path in the construct tree. Be careful moving around constructs in the construct tree -- e.g. changing the parent of a construct or nesting a construct in another construct -- as this will change the logical ID of the construct. Then you could end up with the issues described in section Changing Logical IDs and section Changing (Immutable) Properties.

See Logical ID Stability for more information.

5.3.2. Moving Resources between Nested Stacks

In some cases we use nested stacks to overcome the limit of 200 CloudFormation resources per stack.

In the code snippet below you can see how we generate a dynamic amount of nested stack based on the amount of interface endpoints we construct. The `InterfaceEndpoint` construct contains CloudFormation resources so we have to be careful to not exceed the limit of 200 CloudFormation resources per nested stack. That is why we limit the amount of interface endpoints to 30 per nested stack.

```
let endpointCount = 0;
let endpointStackIndex = 0;
let endpointStack;
for (const endpoint of endpointConfig.endpoints) {
  if (!endpointStack || endpointCount >= 30) {
    endpointStack = new NestedStack(accountStack, `Endpoint${endpointStackIndex++}`);
    endpointCount = 0;
  }
  new InterfaceEndpoint(endpointStack, pascalCase(endpoint), {
    serviceName: endpoint,
  });
  endpointCount++;
}
```

We have to be careful here though. Suppose the configuration file contains 40 interface endpoints. The first 30 interface endpoints will be created in the first nested stack; the next 10 interface endpoints will be created in the second nested stack. Suppose now that we remove the first nested endpoint from the configuration file. This will cause the 31st interface endpoint to become the 30th interface endpoint in the list and it will cause the interface endpoint to be moved from the second nested stack to the first nested stack. This will cause the stack updates to fail since CloudFormation will first try to create the interface endpoint in the first nested stack before removing it from the second nested stack. We do currently not support changes to the interface endpoint configuration because of this behavior.

5.3.3. L1 vs. L2 Constructs

See AWS Construct library for an explanation on L1 and L2 constructs.

The L2 constructs for EC2 and VPC do not map well onto the Accelerator-managed resources. For this reason we mostly use L1 CDK constructs -- such as `ec2.CfnVPC`, `ec2.CfnSubnet` -- instead of using L2 CDK constructs -- such as `ec2.Vpc` and `ec2.Subnet`.

5.3.4. CDK Code Dependency on Lambda Function Code

You can read about the distinction between CDK code and runtime code in the introduction of the Development section.

CDK code can depend on runtime code. For example when we want to create a Lambda function using CDK, we need the runtime code to define the Lambda function. We use `npm scripts`, `npm` dependencies and the `NodeJS modules` API to define this dependency between CDK code and runtime code.

First of all, we create a separate folder that contains the workspace and runtime code for our Lambda function. Throughout the project we've called these workspaces `...-lambda` but it could also be named `...-runtime`. See `src/lib/custom-resources/cdk-acm-import-certificate/runtime/package.json`.

This workspace's `package.json` file needs a `prepare` script that compiles the runtime code. See `npm-scripts`.

The `package.json` file also needs a `name` and a `main` entry that points to the compiled code.

`runtime/package.json`

```json
{
  "name": "lambda-fn-runtime",
  "main": "dist/index.js",
  "scripts": {
    "prepare": "webpack-cli --config webpack.config.ts"
  }
}
```

Now when another workspace depends on our Lambda function runtime code workspace, the `prepare` script will run and it will compile the Lambda function runtime code.

Next, we add the dependency to the new workspace to the workspace that contains the CDK code using `pnpm` or by adding it to `package.json`.

`cdk/package.json`

```json
{
  "devDependencies": {
    "lambda-fn-runtime": "workspace:^0.0.1"
  }
}
```

In the CDK code we can now resolve the path to the compiled code using the `NodeJS modules` API. See NodeJS `modules` API.

`cdk/src/index.ts`

```typescript
class LambdaFun extends cdk.Construct {
  constructor(scope: cdk.Construct, id: string) {
    super(scope, id);

    // Find the runtime package folder and resolves the `main` entry of `package.json`.
    // In our case this is `node_modules/lambda-fn-runtime/dist/index.js`.
    const runtimeMain = resolve.require('lambda-fn-runtime');

    // Find the directory containing our `index.js` file.
    // In our case this is `node_modules/lambda-fn-runtime/dist`.
    const runtimeDir = path.dirname(lambdaPath);

    new lambda.Function(this, 'Resource', {
      runtime: lambda.Runtime.NODEJS_12_X,
      code: lambda.Code.fromAsset(runtimeDir),
      handler: 'index.handler', // The `handler` function in `index.js`
    });
  }
}
```

You now have a CDK Lambda function that uses the compiled Lambda function runtime code.

Note: The runtime code needs to recompile every time it changes since the `prepare` script only runs when the runtime workspace is installed.

5.3.5. Custom Resource

We create custom resources for functionality that is not supported natively by CloudFormation. We have two types of custom resources in this project:

1. Custom resource that calls an SDK method;
2. Custom resource that needs additional functionality and is backed by a custom Lambda function.

CDK has a helper construct for the first type of custom resources. See CDK `AwsCustomResource` documentation. This helper construct is for example used in the custom resource `ds-log-subscription`.

The second type of custom resources requires a custom Lambda function runtime as described in the previous section. For example `acm-import-certificate` is backed by a custom Lambda function.

Only a single Lambda function is created per custom resource, account and region. This is achieved by creating only a single Lambda function in the construct tree.

`src/lib/custom-resources/custom-resource/cdk/index.ts`

```
class CustomResource extends cdk.Construct {
  constructor(scope: cdk.Construct, id: string, props: CustomResourceProps) {
    super(scope, id);

    new cdk.CustomResource(this, 'Resource', {
      resourceType: 'Custom::CustomResource',
      serviceToken: this.lambdaFunction.functionArn,
    });
  }

  private get lambdaFunction() {
    const constructName = `CustomResourceLambda`;

    const stack = cdk.Stack.of(this);
    const existing = stack.node.tryFindChild(constructName);
    if (existing) {
      return existing as lambda.Function;
    }

    // The package '@aws-accelerator/custom-resources/cdk-custom-resource-runtime' contains the runtime co
    const lambdaPath = require.resolve('@aws-accelerator/custom-resources/cdk-custom-resource-runtime');
    const lambdaDir = path.dirname(lambdaPath);

    return new lambda.Function(stack, constructName, {
      code: lambda.Code.fromAsset(lambdaDir),
    });
  }
}
```

5.3.6. Escape Hatches

Sometimes CDK does not support a property on a resource that CloudFormation does support. You can then override the property using the `addOverride` or `addPropertyOverride` methods on CDK CloudFormation resources. See CDK escape hatches.

5.3.6.1. AutoScaling Group Metadata

An example where we override metadata is when we create a launch configuration.S

```
const launchConfig = new autoscaling.CfnLaunchConfiguration(this, 'LaunchConfig', { ... });

launchConfig.addOverride('Metadata.AWS::CloudFormation::Authentication', {
  S3AccessCreds: {
    type: 'S3',
    roleName,
    buckets: [bucketName],
  },
});

launchConfig.addOverride('Metadata.AWS::CloudFormation::Init', {
  configSets: {
    config: ['setup'],
  },
  setup: {
    files: {
      // Add files here
    },
    services: {
      // Add services here
    },
    commands: {
      // Add commands here
    },
  },
});
```

5.3.6.2. Secret `SecretValue`

Another example is when we want to use `secretsmanager.Secret` and set the secret value.

```
function setSecretValue(secret: secrets.Secret, value: string) {
  const cfnSecret = secret.node.defaultChild as secrets.CfnSecret; // Get the L1 resource that backs this
  cfnSecret.addPropertyOverride('SecretString', value); // Override the property `SecretString` on the L1
  cfnSecret.addPropertyDeletionOverride('GenerateSecretString'); // Delete the property `GenerateSecretStr
}
```

6. Contributing Guidelines

6.1. How-to

6.2. Adding New Functionality?

Before making a change or adding new functionality you have to verify what kind of functionality is being added.

- Is it an Accelerator-management change?
  - Is the change related to the `Installer` stack?
    - Is the change CDK related?
      - Make the change in `src/installer/cdk`.
    - Is the change runtime related?
      - Make the change in `src/installer/cdk/assets`.
  - Is the change related to the `Initial Setup` stack?
    - Is the change CDK related?
      - Make the change in `src/core/cdk`
    - Is the change runtime related?
      - Make the change in `src/core/runtime`
- Is it an Accelerator-managed change?
  - Is the change related to the `Phase` stacks?
    - Is the change CDK related?
      - Make the change in `src/deployments/cdk`
    - Is the change runtime related?
      - Make the change in `src/deployments/runtime`

6.3. Create a CDK Lambda Function with Lambda Runtime Code

See CDK Code Dependency on Lambda Function Code for a short introduction.

6.4. Create a Custom Resource

See Custom Resource and Custom Resources for a short introduction.

1. Create a separate folder that contains the CDK and Lambda function runtime code, e.g. `src/lib/custom-resources/my-c`
2. Create a folder `my-custom-resource` that contains the CDK code;
   1. Create a `package.json` file with a dependency to the `my-custom-resource/runtime` package;
   2. Create a `cdk` folder that contains the source of the CDK code;
3. Create a folder `my-custom-resource/runtime` that contains the runtime code;
   1. Create a `runtime/package.json` file with a `"name"`, `"prepare"` script and a `"main"`;
   2. Create a `runtime/webpack.config.ts` file that compiles TypeScript code to a single JavaScript file;
   3. Create a `runtime/src` folder that contains the source of the Lambda function runtime code;

You can look at the `src/lib/custom-resources/cdk-acm-import-certificate` custom resource as an example.

It is best practice to add tags to any resources that the custom resource creates using the `cfn-tags` library.

6.5. Run All Unit Tests

Run in the root of the project.

```
pnpm recursive run test --no-bail --stream -- --silent
```

6.6. Accept Unit Test Snapshot Changes

Run in `src/deployments/cdk`.

```
pnpm run test -- -u
```

6.7. Validate Code with Prettier

Run in the root of the project.

```
pnpx prettier --check **/*.ts
```

6.8. Format Code with Prettier

Run in the root of the project.

```
pnpx prettier --write **/*.ts
```

6.9. Validate Code with `tslint`

Run in the root of the project.

```
pnpm recursive run lint --stream --no-bail
```

...Return to Accelerator Table of Contents