Deployment Pipeline Reference Architecture

September 2022

AWS Global Services Security

Table of contents

1. Overview

A deployment pipeline is the key architectural construct for performing Continuous Integration, Delivery, and Deployment. Pipelines consist of a series of stages like source, build, test, or deploy. Stages consist of automated tasks in the software delivery lifecycle. There are different types of deployment pipelines for different use cases.

The Deployment Pipeline Reference Architecture (DPRA) for AWS workloads describes the stages and actions for different types of pipelines that exist in modern systems. The DPRA also describes the practices teams employ to increase the velocity, stability, and security of software systems through the use of deployment pipelines. For a higher-level perspective, see Clare Liguori’s article in the Amazon Builder’s Library titled Automating safe, hands-off deployments.

Customers and third-party vendors can use the DPRA to create implementations - reference or otherwise - using their own set of services and tools. We have included reference implementations that use AWS and third-party tools. When an AWS service/tool is available, we list it; when there are no AWS services/tools, we list third-party tools. There are many third-party tools that can run on AWS so the ones we chose should only be seen as examples for illustrative purposes. Choose the best tool that meets the needs of your organization.

The DPRA covers the following deployment pipelines in detail:

Application

Build, test, and deploy an application.

Compute Image

Build and publish machine or container images.

🛠 **under development** 🛠

Account Fleet Management

Manage a fleet of AWS accounts through a pipeline.

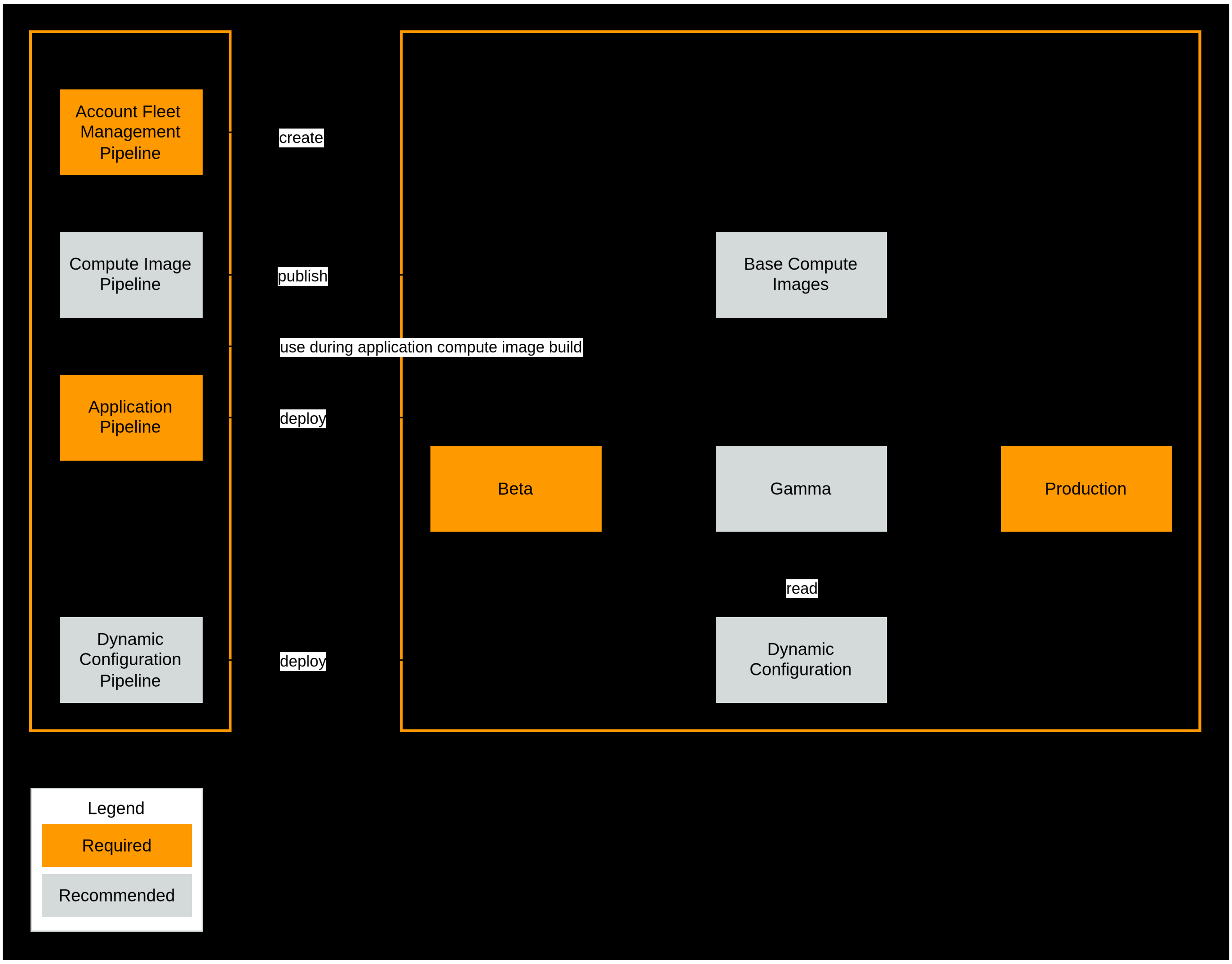
🛠 **under development** 🛠

Dynamic Configuration

Manage dynamic configuration for a workload.

Architecture

A typical solution uses multiple or all of the pipelines in combination as follows:



Business Outcomes

Modern deployment pipelines create the following business outcomes:

- **Automation** - Everything necessary to build, test, deploy, and run an application should be defined as code - code for pipelines, accounts, networking, infrastructure, applications/services, configuration, data, security, compliance, governance, auditing, and documentation – any aspect inside and outside software systems.
- **Consistency** - The source code should only be built and packaged once. The packaged artifact should then be staged in a registry with appropriate metadata and ready for deployment to any environment. Build artifacts only once and then promote them through the pipeline. The output of the pipeline should be versioned and able to be traced back to the source it was built from and from the business requirements that defined it.
- **Small Batches** - The pipeline should be constructed in such a way as to encourage the delivery of software early and often. This is accomplished by removing toil from the software delivery process through automation and fast feedback. Likewise, the pipeline should discourage the use of long-lived branches and encourage trunk-based development. Developers should be able to merge their changes to the trunk and deploy through the pipeline daily.
- **Orchestration** - As part of a deployment pipeline, every merged code change has a fully-automated build, test, publish, deploy, and release process run across all environments. Each stage automatically transitions to the next stage of the pipeline upon success, or stops on failure. In some circumstances human approvals are necessary while organizations mature their automation practices. These approvals most often show up when automation is unable to assess the risk or specific context for approval. If used, human approvals should be used before production deployments only and should be reduced to a button-click interface that triggers an automated pipeline process to continue. A single pipeline should orchestrate the deployment to all environments rather than creating pipelines for each environment.
- **Fast Feedback** - Automatically notify engineers as soon as possible of build, test, quality, and security errors from deployment pipelines through the most effective means such as chat or email.
- **Always Deployable** - When an error occurs in the mainline of a deployment pipeline, the top priority is to fix the build and ensure deployment obtains and remains in a healthy state before introducing any further changes. The pipeline should be the authoritative source for deciding if and when changes are ready to be released into production.
- **Measured** - Provide real-time metrics for code quality, speed (deployment frequency and deployment lead time), security (security control automation %, mean time to resolve security errors), and reliability (change failures and time to restore service). View metrics through a real-time dashboard. When instrumentation is not yet possible, create a Likert-based questionnaire to determine these metrics across teams.

Definitions

Component

A **component** is 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.

(source: AWS Well-Architected Framework definitions)

Workload

A **workload** is a set of components that together deliver business value. A workload is usually the level of detail that business and technology leaders communicate about. Examples of workloads are marketing websites, e-commerce websites, the back-ends for a mobile app, analytic platforms, etc. Workloads vary in levels of architectural complexity, from static websites to architectures with multiple data stores and many components.

(source: AWS Well-Architected Framework)

Environment

An **environment** is an isolated target for deploying and testing a workload and its dependencies. Environments can be created for validating changes, achieving data compliance, or for improving resiliency. Example environments include creating separate

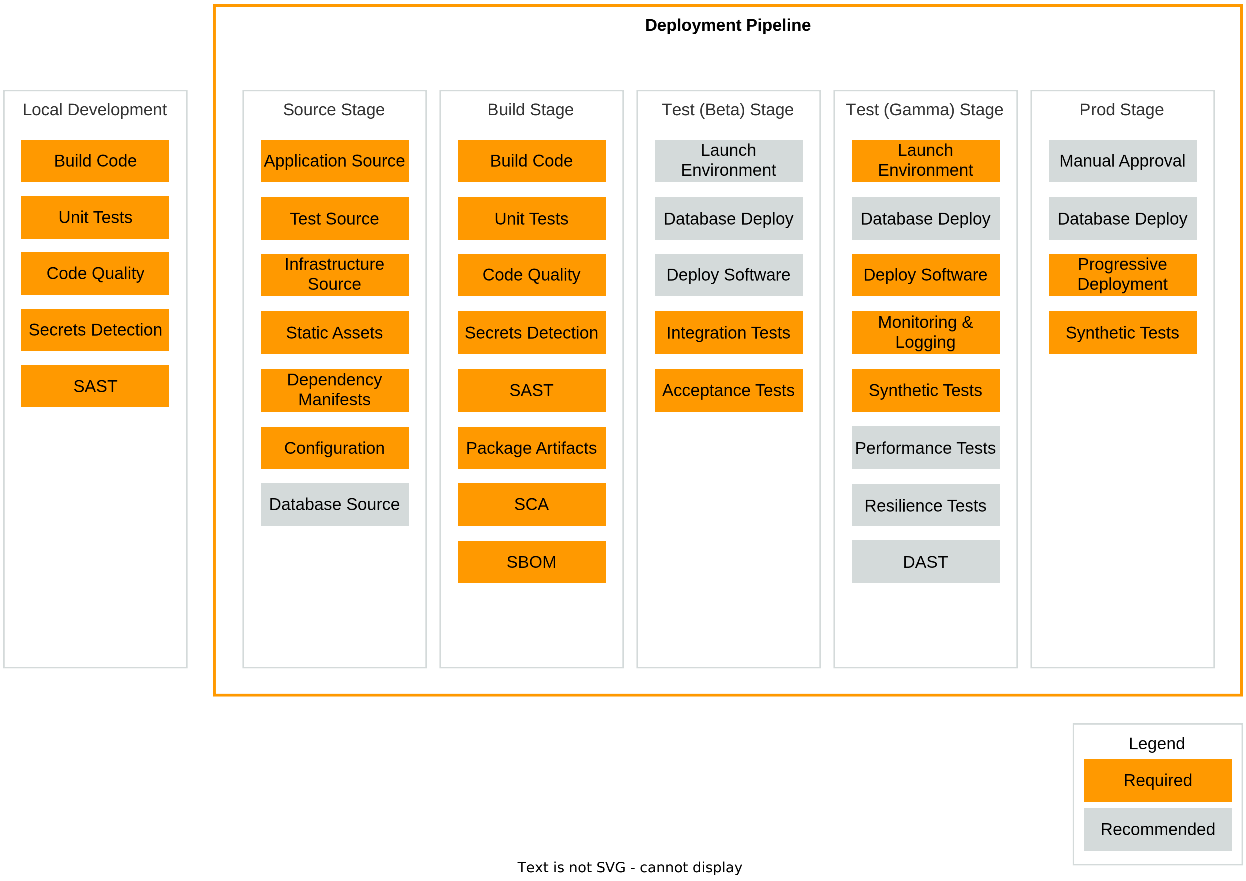
AWS accounts for each developer, creating separate AWS accounts for staging and production, and using multiple regions for production traffic. Best practice is for each environment to run in a separate AWS account.

2. Application Pipeline

Architecture

The term "Application" is used synonymously with the term "Component" as defined by the Well-Architected Framework and other DPRA pipelines. Applications are the most common use case for a deployment pipeline. This pipeline type will take source code files, tests, static analysis, database deployment, configuration, and other code to perform build, test, deploy, and release processes. The pipeline launches an environment from the compute image artifacts generated in the compute image pipeline. Automated tests are run on the environment(s) as part of the deployment pipeline.

This pipeline encourages trunk based development in which developers frequently avoid long-lived branches and regulary commit their changes to the trunk. Therefore this pipeline only executes for commits to the trunk. Every commit to the trunk has a change to go to production if all steps of the pipeline complete successfully.



Deployment Pipeline

Local Development
Build Code
Unit Tests
Code Quality
Secrets Detection
SAST

Source Stage
Application Source
Test Source
Infrastructure Source
Static Assets
Dependency Manifests
Configuration
Database Source

Build Stage
Build Code
Unit Tests
Code Quality
Secrets Detection
SAST
Package Artifacts
SCA
SBOM

Test (Beta) Stage
Launch Environment
Database Deploy
Deploy Software
Integration Tests
Acceptance Tests

Test (Gamma) Stage
Launch Environment
Database Deploy
Deploy Software
Monitoring & Logging
Synthetic Tests
Performance Tests
Resilience Tests
DAST

Prod Stage
Manual Approval
Database Deploy
Progressive Deployment
Synthetic Tests

Legend
Required
Recommended

Text is not SVG - cannot display

The expected outcome of this pipeline is to be able to safely release software changes to customers within a couple hours. Deployment pipelines should publish the following metrics:

- `Lead time` – the average amount of time it takes for a single commit to get all the way into production.
- `Deploy frequency` – the number of production deployments within a given time period.
- `Mean time between failure (MTBF)` – the average amount of time between the start of a successful pipeline and the start of a failed pipeline.
- `Mean time to recover (MTTR)` – the average amount of time between the start of a failed pipeline and the start of the next successful pipeline.

Each stage below will include a required and recommended actions. The actions will include guidance on what steps out to be perfomed in each action. References will be made to real-life examples of tools to help better define the actions involved in each stage. The use of these examples is not an endorsement of any specific tool.

Local Development

Developers need fast-feedback for potential issues with their code. Automation should run in their developer workspace to give them feedback before the deployment pipeline runs.

Pre-Commit Hooks

Pre-Commit hooks are scripts that are executed on the developer's workstation when they try to create a new commit. These hooks have an opportunity to inspect the state of the code before the commit occurs and abort the commit if tests fail. An example of pre-commit hooks are Git hooks. Examples of tools to configure and store pre-commit hooks as code include but are not limited to husky and pre-commit.

IDE Plugins

Warn developers of potential issues with their source code in their IDE using plugins and extensions including but not limited to Visual Studio Code - Python Extension and IntelliJ IDEA - JavaScript linters.

Source

The source stage pulls in various types of code from a distributed version control system.

Application Source Code

Code that is compiled, transpiled or interpreted for the purpose of delivering business capabilities through applications and/or services.

Test Source Code

Code that verifies the expected functionality of the *Application Source Code* and the *Infrastructure Source Code*. This includes source code for unit, integration, end-to-end, capacity, chaos, and synthetic testing. All *Test Source Code* is **required** to be stored in the same repository as the app to allow tests to be created and updated on the same lifecycle as the *Application Source Code*.

Infrastructure Source Code

Code that defines the infrastructure necessary to run the *Application Source Code*. Examples of infrastructure source code include but are not limited to AWS Cloud Development Kit, AWS CloudFormation and HashiCorp Terraform. All *Infrastructure Source Code* is **required** to be stored in the same repository as the app to allow infrastructure to be created and updated on the same lifecycle as the *Application Source Code*.

Static Assets

Assets used by the *Application Source Code* such as html, css, and images.

Dependency Manifests

References to third-party code that is used by the *Application Source Code*. This could be libraries created by the same team, a separate team within the same organization, or from an external entity.

Static Configuration

Files (e.g. JSON, XML, YAML or HCL) used to configure the behavior of the *Application Source Code*. Any configuration that is environment specific should *not* be included in *Application Source Code*. Environment specific configuration should be defined in the *Infrastructure Source Code* and injected into the application at runtime through a mechanism such as environment variables.

Database Source Code

Code that defines the schema and reference data of the database used by the *Application Source Code*. Examples of database source code include but are not limited to Liquibase. If the *Application Source Code* uses a private database that no other application accesses, then the database source code is **required** to be stored in the same repository as the *Application Source Code*. This allows the *Application Source Code* and *Database Source Code* to be updated on the same lifecycle. However, if the database is shared by multiple applications then the *Database Source Code* should be maintained in a separate repository and managed by separate pipeline. It should be noted that this is undesireable as it introduces coupling between applications.

All the above source code is versioned and securely accessed through role based access control with source code repositories including but not limited to AWS CodeCommit, GitHub, GitLab, and Bitbucket.

Build

All actions run in this stage are also run on developer's local environments prior to code commit and peer review. Actions in this stage should all run in less than 10 minutes so that developers can take action on fast feedback before moving on to their next task. If it’s taking more time, consider decoupling the system to reduce dependencies, optimizing the process, using more efficient tooling, or moving some of the actions to latter stages. Each of the actions below are defined and run in code.

Build Code

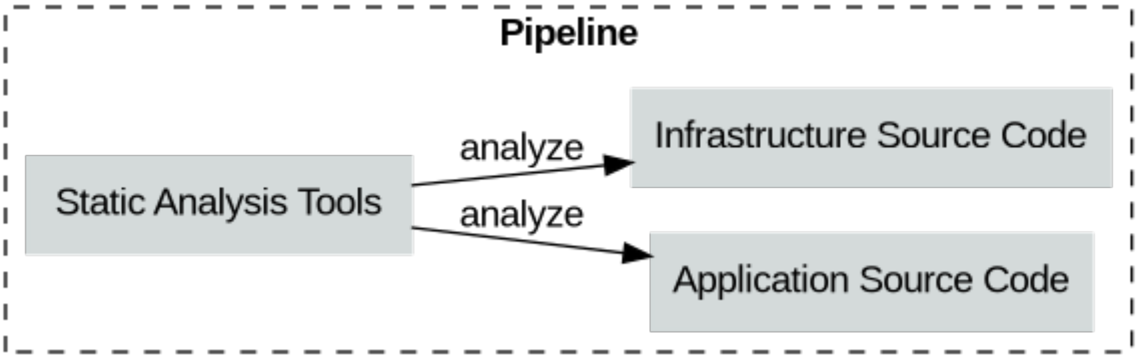
Convert code into artifacts that can be promoted through environments. Most builds complete in seconds. Examples include but are not limited to Maven and tsc.

Unit Tests

Run the test code to verify that individual functions and methods of classes, components or modules of the *Application Source Code* are performing according to expectations. These tests are fast-running tests with zero dependencies on external systems returning results in seconds. Examples of unit testing frameworks include but are not limited to JUnit, Jest, and pytest. Test results should be published somewhere such as AWS CodeBuild Test Reports.

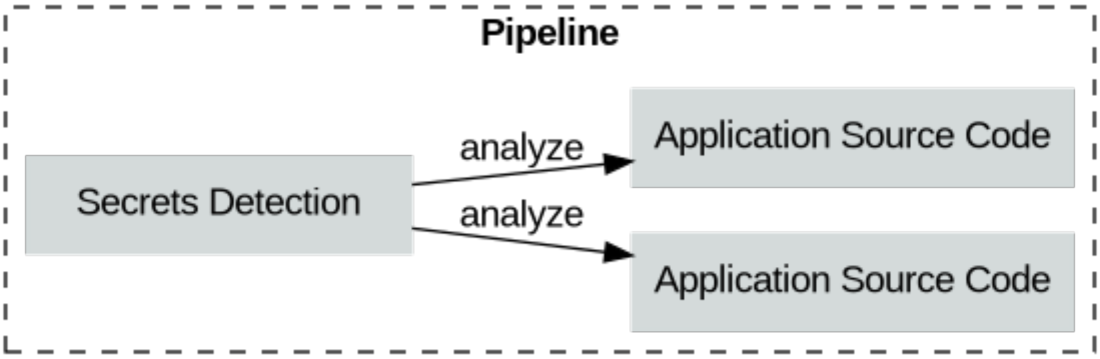
Code Quality

Run various automated static analysis tools that generate reports on code quality, coding standards, security, code coverage, and other aspects according to the team and/or organization’s best practices. AWS recommends that teams fail the build when important practices are violated (e.g., a security violation is discovered in the code). These checks usually run in seconds. Examples of tools to measure code quality include but are not limited to Amazon CodeGuru, SonarQube, black, and ESLint.



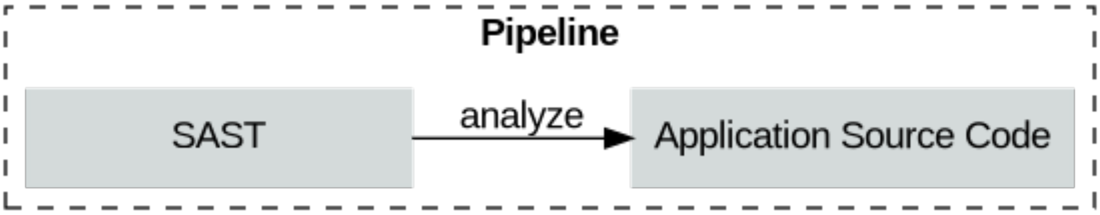
Secrets Detection

Identify secrets such as usernames, passwords, and access keys in code. When discovering secrets, the build should fail immediately. Examples of secret detection tools include but are not limited to GitGuardian and gitleaks.



Static Application Security Testing (SAST)

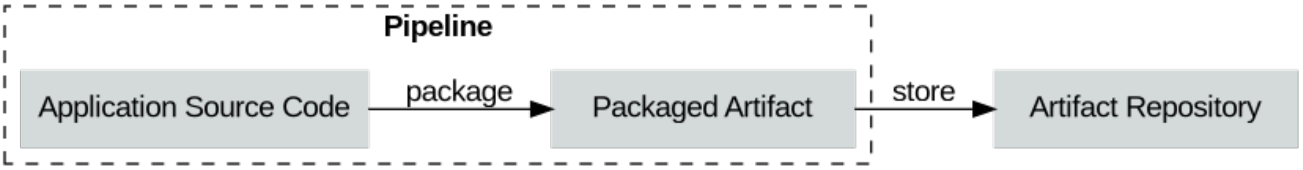
Analyze code for application security violations such as XML External Entity Processing, SQL Injection, and Cross Site Scripting. Any findings that exceed the configured threshold will immediately fail the build and stop any forward progress in the pipeline. Examples of tools to perform static application security testing include but are not limited to Amazon CodeGuru, SonarQube, and Checkmarx.



Package and Store Artifact(s)

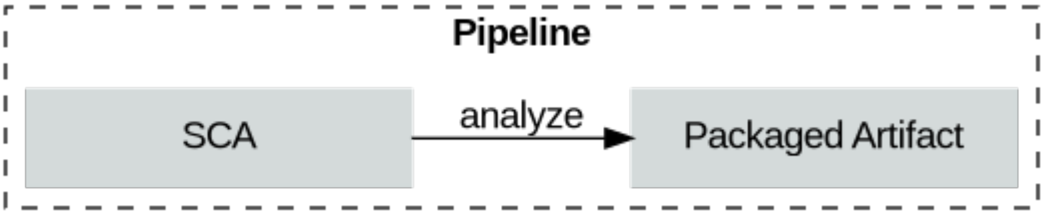
While the *Build Code* action will package most of the relevant artifacts, there may be additional steps to automate for packaging the code artifacts. Artifacts should only be built and packaged once and then deployed to various environments to validate the artifact. Artifacts should never be rebuilt during subsequent deploy stages. Once packaged, automation is run in this action to store the artifacts in an artifact repository for future deployments. Examples of artifact repositories include but are not limited to AWS CodeArtifact, Amazon ECR, Nexus, and JFrog Artifactory.

Packages should be signed with a digital-signature to allow deployment processes to confirm the code being deployed is from a trusted publisher and has not been altered. AWS Signer can be used to cryptographically sign code for AWS Lambda applications and AWS-supported IoT devices.



Software Composition Analysis (SCA)

Run software composition analysis (SCA) tools to find vulnerabilities to package repositories related to open source use, licensing, and security vulnerabilities. SCA tools also launch workflows to fix these vulnerabilities. Any findings that exceed the configured threshold will immediately fail the build and stop any forward progress in the pipeline. These tools also require a software bill of materials (SBOM) exist. Example SCA tools include but are not limited to Dependabot, Snyk, and Blackduck.



Software Bill of Materials (SBOM)

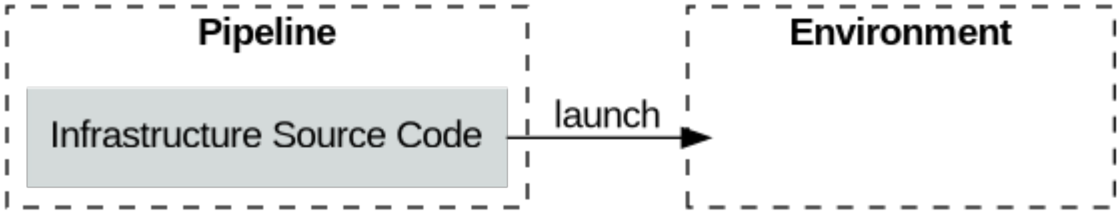
Generate a software bill of materials (SBOM) report detailing all the dependencies used. Examples of SBOM formats include SPDX and CycloneDX

Test (Beta)

Testing is performed in a beta environment to validate that the latest code is functioning as expected. This validation is done by first deploying the code and then running integration and end-to-end tests against the deployment. Beta environments will have dependencies on the applications and services from other teams in their gamma environments. All actions performed in this stage should complete within 30 minutes to provide fast-feedback.

Launch Environment

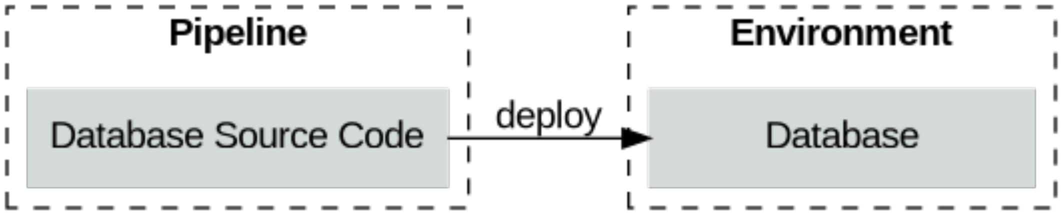
Use a compute image from an image repository (e.g., AMI or a container repo) and launch an environment from the image using *Infrastructure Source Code*. The beta image is generally not accessible to public customers and is only used for internal software validation. The beta environment should be in a different AWS Account from the tools used to run the deployment pipeline. Access to the beta environment should be handled via cross-account IAM roles rather than long lived credentials from IAM users. Example tools for defining infrastructure code include but are not limited to AWS Cloud Development Kit, AWS CloudFormation and HashiCorp Terraform.



Database Deploy

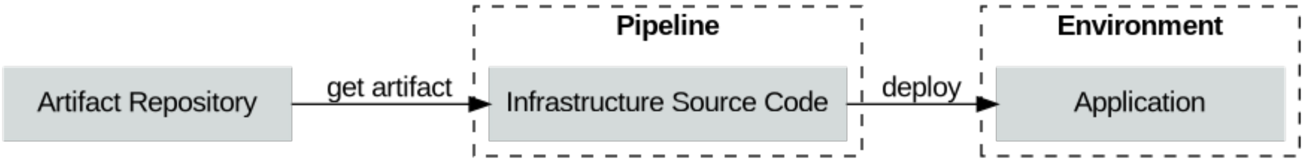
Apply changes to the beta database using the *Database Source Code*. Changes should be made in a manner that ensures rollback safety. Best practice is to connect to the beta database through cross-account IAM roles and IAM database authentication for RDS rather than long lived database credentials. If database credentials must be used, then they should be loaded from a secret manager such as AWS Secrets Manager. Changes to the database should be incremental, only applying the changes since the prior deployment. Examples of tools that apply incremental database changes include but are not limited to Liquibase, VS Database Project, and Flyway.

Test data management is beyond the scope of this reference architecuture but should be addressed during `Database Deploy` in preparation of subsequent testing activity.



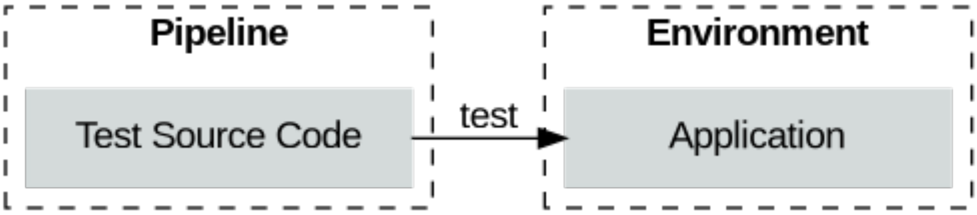
Deploy Software

Deploy software to the beta environment. Software is not deployed from source but rather the artifact that was packaged and stored in the *Build Stage* will be used for the deployment. Software to be deployed should include digital signatures to verify that the software came from a trusted source and that no changes were made to the software. Software deployments should be performed through *Infrastructure Source Code*. Access to the beta environment should be handled via cross-account IAM roles rather than long lived credentials from IAM users. Examples of tools to deploy software include but are not limited to AWS CodeDeploy, Octopus Deploy, and Spinnaker.



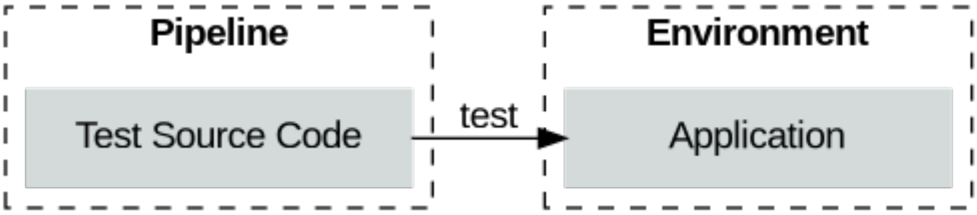
Integration Tests

Run automated tests that verify if the application satisifes business requirements. These tests require the application to be running in the beta environment. Integration tests may come in the form of behavior-driven tests, automated acceptance tests, or automated tests linked to requirements and/or stories in a tracking system. Test results should be published somewhere such as AWS CodeBuild Test Reports. Examples of tools to define integration tests include but are not limited to Cucumber, vRest, and SoapUI.



Acceptance Tests

Run automated testing from the users’ perspective in the beta environment. These tests verify the user workflow, including when performed through a UI. These test are the slowest to run and hardest to maintain and therefore it is recommended to only have a few end-to-end tests that cover the most important application workflows. Test results should be published somewhere such as AWS CodeBuild Test Reports. Examples of tools to define end-to-end tests include but are not limited to Cypress, Selenium, and Telerik Test Studio.

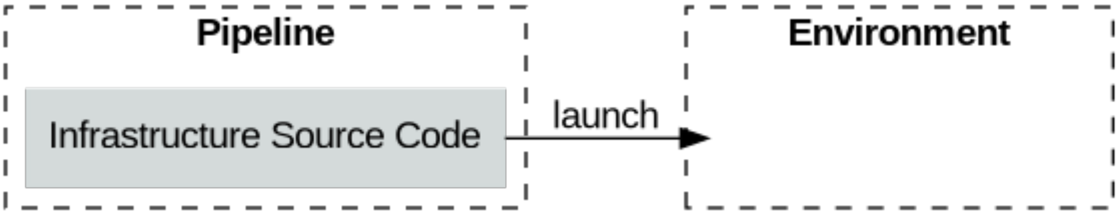


Test (Gamma)

Testing is performed in a gamma environment to validate that the latest code can be safely deployed to production. The environment is as production-like as possible including configuration, monitoring, and traffic. Additionally, the environment should match the same regions that the production environment uses. The gamma environment is used by other team's beta environments and therefore must maintain acceptable service levels to avoid impacting other team productivity. All actions performed in this stage should complete within 30 minutes to provide fast-feedback.

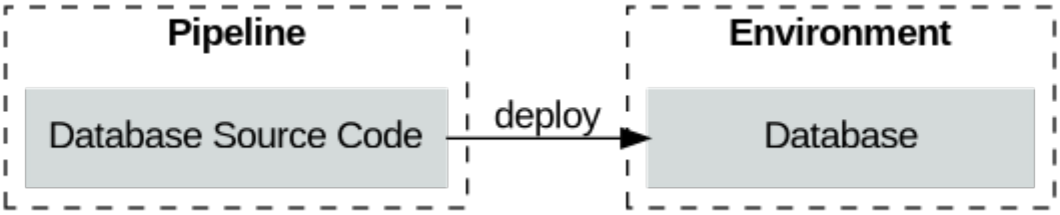
Launch Environment

Use the compute image from an image repository (e.g., AMI or a container repo) and launch an environment from the image using *Infrastructure Source Code*. The gamma environment should be in a different AWS Account from the tools used to run the deployment pipeline. Access to the gamma environment should be handled via cross-account IAM roles rather than long lived credentials from IAM users. Example tools for defining infrastructure code include but are not limited to AWS Cloud Development Kit, AWS CloudFormation and HashiCorp Terraform.



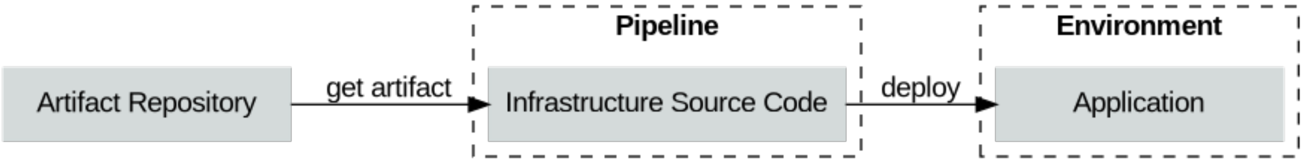
Database Deploy

Apply changes to the gamma database using the *Database Source Code*. Changes should be made in a manner that ensures rollback safety. Best practice is to connect to the gamma database through cross-account IAM roles and IAM database authentication for RDS rather than long lived database credentials. If database credentials must be used, then they should be loaded from a secret manager such as AWS Secrets Manager. Changes to the database should be incremental, only applying the changes since the prior deployment. Examples of tools that apply incremental database changes include but are not limited to Liquibase, VS Database Project, and Flyway.



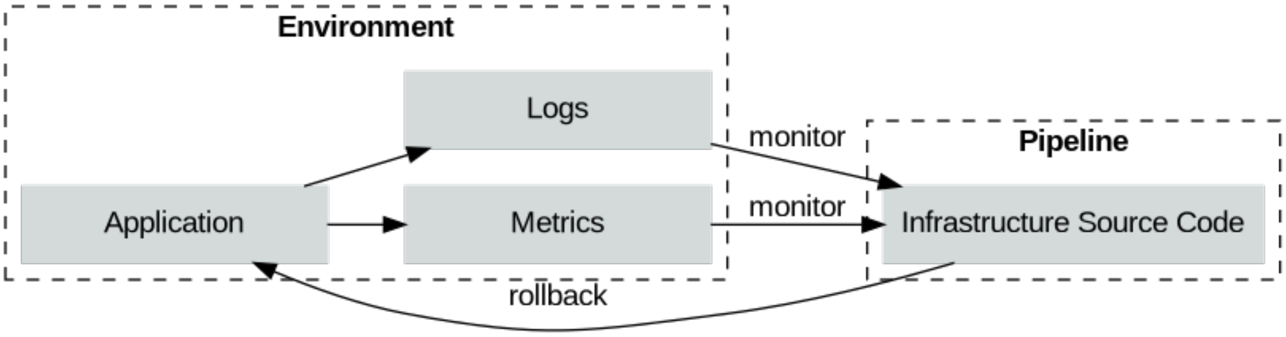
Deploy Software

Deploy software to the gamma environment. Software is not deployed from source but rather the artifact that was packaged and stored in the *Build Stage* will be used for the deployment. Software to be deployed should include digital signatures to verify that the software came from a trusted source and that no changes were made to the software. Software deployments should be performed through *Infrastructure Source Code*. Access to the gamma environment should be handled via cross-account IAM roles rather than long lived credentials from IAM users. Examples of tools to deploy software include but are not limited to AWS CodeDeploy, Octopus Deploy, and Spinnaker.



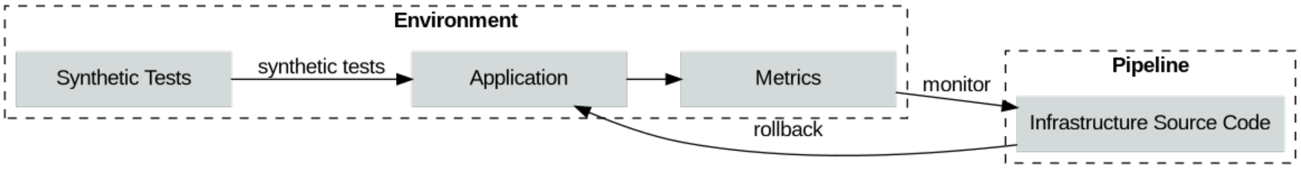
Application Monitoring & Logging

Monitor deployments across regions and fail when threshold breached. The thresholds for metric alarms should be defined in the *Infrastructure Source Code* and deployed along with the rest of the infrastructure in an environment. Ideally, deployments should be automatically failed and rolled back when error thresholds are breached. Examples of automated rollback include AWS CloudFormation monitor & rollback, AWS CodeDeploy rollback and Flagger.



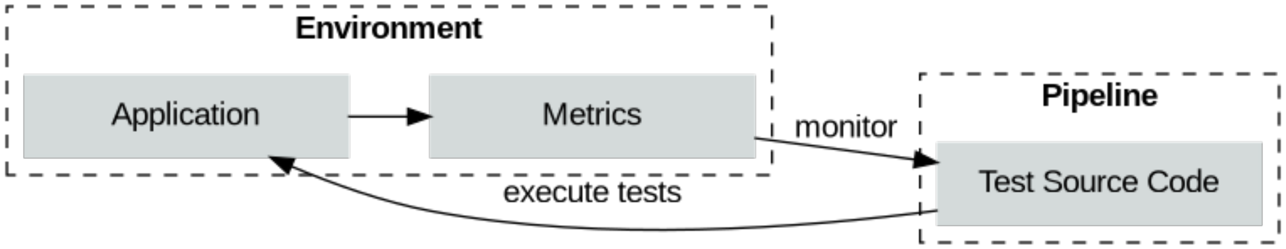
Synthetic Tests

Tests that run continuously in the background in a given environment to generate traffic and verify the system is healthy. These tests serve two purposes: 1/ Ensure there is always adequate traffic in the environment to trigger alarms if a deployment is unhealthy 2/ Test specific workflows and assert that the system is functioning correctly. Examples of tools that can be used for synthetic tests include but are not limited to Amazon CloudWatch Synthetics,Dynatrace Synthetic Monitoring, and Datadog Synthetic Monitoring.



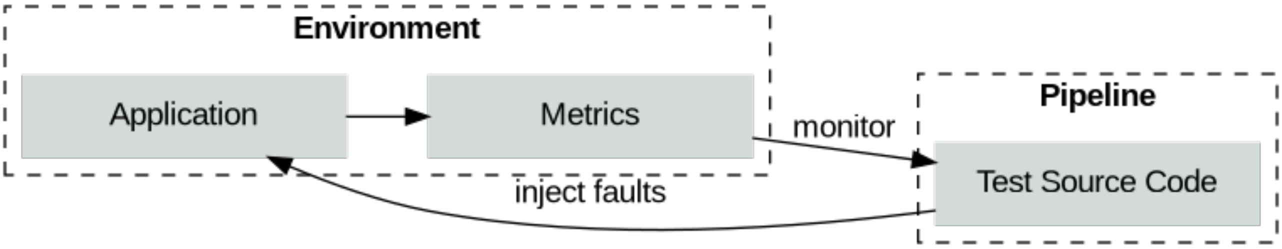
Performance Tests

Run longer-running automated capacity tests against environments that simulate production capacity. Measure metrics such as the transaction success rates, response time and throughput. Determine if application meets performance requirements and compare metrics to past performance to look for performance degredation. Examples of tools that can be used for performance tests include but are not limited to JMeter, Locust, and Gatling.



Resilience Tests

Inject failures into environments to identify areas of the application that are susceptible to failure. Tests are defined as code and applied to the environment while the system is under load. The success rate, response time and throughput are measured during the periods when the failures are injected and compared to periods without the failures. Any significant deviation should fail the pipeline. Examples of tools that can be used for chaos/resilience testing include but are not limited to AWS Fault Injection Simulator, Gremlin, and ChaosToolkit.



Dynamic Application Security Testing (DAST)

Perform testing of web applications and APIs by running automated scans against it to identify vulnerabilities through techniques such as cross-site scripting (XSS) and SQL injection(SQLi). Examples of tools that can be used for dynamic application security testing include but are not limited to OWASP ZAP, StackHawk, and AppScan. See AWS Guidance on Penetration Testing for info on penetration testing in an AWS environment.

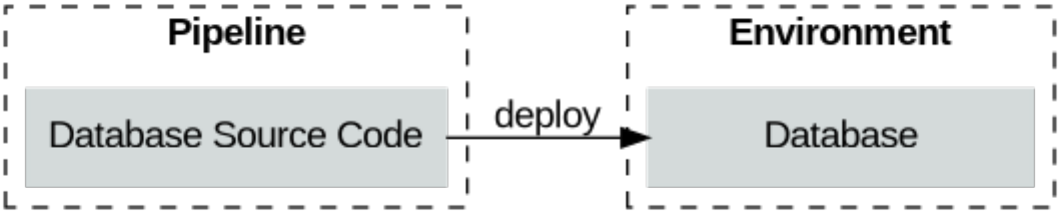
Prod

Manual Approval

As part of an automated workflow, obtain authorized human approval before deploying to the production environment.

Database Deploy

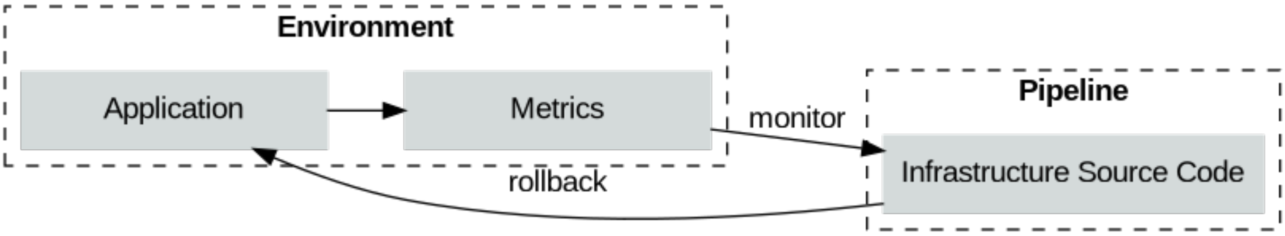
Apply changes to the production database using the *Database Source Code*. Changes should be made in a manner that ensures rollback safety. Best practice is to connect to the production database through cross-account IAM roles and IAM database authentication for RDS rather than long lived database credentials. If database credentials must be used, then they should be loaded from a secret manager such as AWS Secrets Manager. Changes to the database should be incremental, only applying the changes since the prior deployment. Examples of tools that apply incremental database changes include but are not limited to Liquibase, VS Database Project, and Flyway.



Progressive Deployment

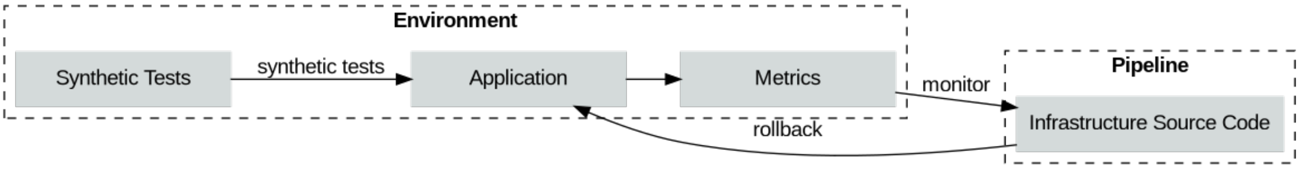
Deployments should be made progressively in waves to limit the impact of failures. A common approach is to deploy changes to a subset of AWS regions and allow sufficient bake time to monitor performance and behavior before proceeding with additional waves of AWS regions.

Software should be deployed using one of progressive deployment involving controlled rollout of a change through techniques such as canary deployments, feature flags, and traffic shifting. Software deployments should be performed through *Infrastructure Source Code*. Access to the production environment should be handled via cross-account IAM roles rather than long lived credentials from IAM users. Examples of tools to deploy software include but are not limited to AWS CodeDeploy. Ideally, deployments should be automatically failed and rolled back when error thresholds are breached. Examples of automated rollback include AWS CloudFormation monitor & rollback, AWS CodeDeploy rollback and Flagger.



Synthetic Tests

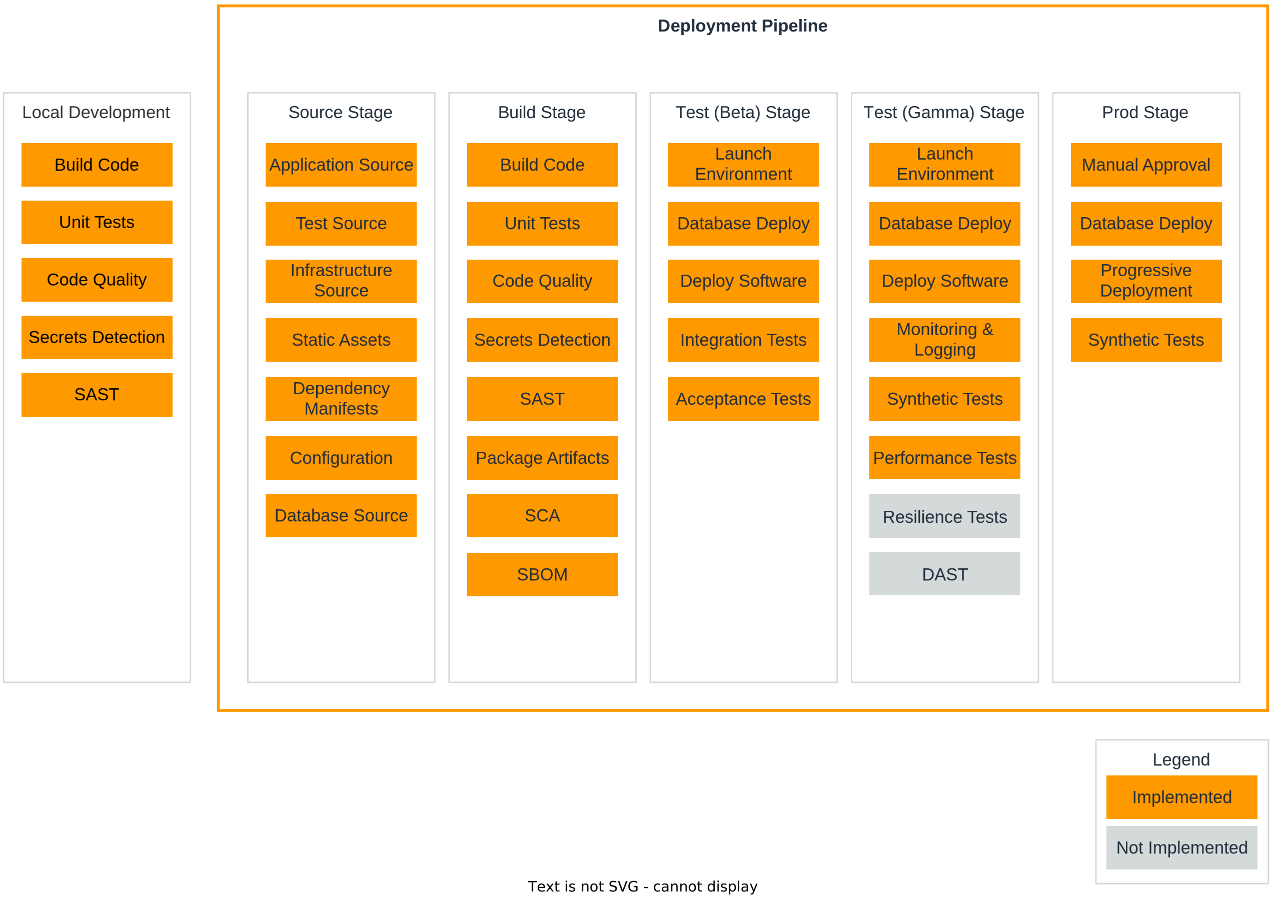
Tests that run continuously in the background in a given environment to generate traffic and verify the system is healthy. These tests serve two purposes: 1/ Ensure there is always adequate traffic in the environment to trigger alarms if a deployment is unhealthy 2/ Test specific workflows and assert that the system is functioning correctly. Examples of tools that can be used for synthetic tests include but are not limited to Amazon CloudWatch Synthetics,Dynatrace Synthetic Monitoring, and Datadog Synthetic Monitoring.



Reference Implementations

AWS CDK Pipeline

This presents a reference implementation of the Application Pipeline reference architecture. The pipeline is built with AWS CodePipeline and uses AWS CodeBuild for building the software and performing testing tasks. All the infrastructure for this reference implementation is defined with AWS Cloud Development Kit. The pipelines are defined using the CDK Pipelines L3 constructs. The source code for this reference implementation is available in GitHub for running in your own local account.

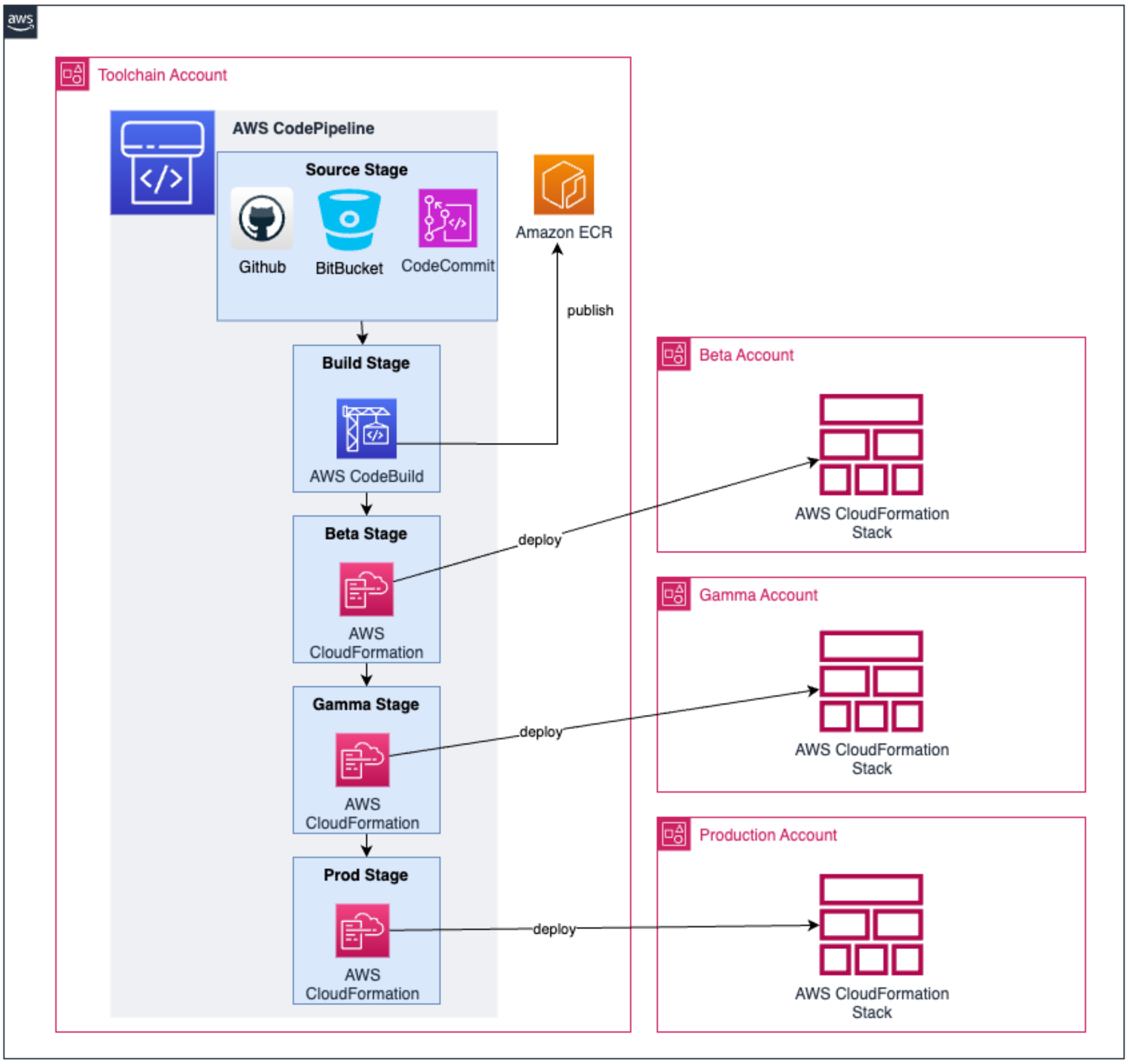


Text is not SVG - cannot display

Disclaimer

This reference implementation is intended to serve as an example of how to accomplish the guidance in the reference architecture using CDK Pipelines. The reference implementation has intentionally bypassed the following AWS Well-Architected best practices to make it accessible by a wider range of customers. Be sure to address these before using parts of this code for any workloads in your own environment:

TLS on HTTP endpoint - the listener for the sample application uses HTTP instead of HTTPS to avoid having to create new ACM certificates and Route53 hosted zones. This should be replaced in your account with an `HTTPS` listener.



Local Development

Developers need fast-feedback for potential issues with their code. Automation should run in their developer workspace to give them feedback before the deployment pipeline runs.

Pre-Commit Hooks

Pre-Commit hooks are scripts that are executed on the developer's workstation when they try to create a new commit. These hooks have an opportunity to inspect the state of the code before the commit occurs and abort the commit if tests fail. An example of pre-commit hooks are Git hooks. Examples of tools to configure and store pre-commit hooks as code include but are not limited to husky and pre-commit.

The following `.pre-commit-config.yaml` is added to the repository that will build the code with Maven, run unit tests with JUnit, check for code quality with Checkstyle, run static application security testing with PMD and check for secrets in the code with gitleaks.

```yaml
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v2.3.0
  hooks:
  -   id: check-yaml
  -   id: check-json
  -   id: trailing-whitespace
- repo: https://github.com/pre-commit/mirrors-eslint
  rev: v8.23.0
  hooks:
  -   id: eslint
- repo: https://github.com/ejba/pre-commit-maven
  rev: v0.3.3
  hooks:
  -   id: maven-test
- repo: https://github.com/zricethezav/gitleaks
  rev: v8.12.0
  hooks:
    - id: gitleaks
```

Source

Application Source Code

The application source code can be found in the src/main/java directory. It is intended to serve only as a reference and should be replaced by your own application source code.

This reference implementation includes a Spring Boot application that exposes a REST API and uses a database for persistence. The API is implemented in `FruitController.java`:

```
public class FruitController {
    /**
     * JPA repository for fruits.
     */
    private final FruitRepository repository;

    /**
     * Logic to map between entities and DTOs
     */
    private final FruitMapper mapper;

    FruitController(final FruitRepository r, final FruitMapper m) {
        this.repository = r;
        this.mapper = m;
    }

    @GetMapping("/api/fruits")
    List<FruitDTO> all() {
        return repository.findAll()
                .stream()
                .map(mapper::toDto)
                .collect(Collectors.toList());
    }

    @PostMapping("/api/fruits")
    FruitDTO newFruit(@RequestBody final FruitDTO fruit) {
        return mapper.toDto(repository.save(mapper.toEntity(fruit)));
    }

    @GetMapping("/api/fruits/{id}")
    FruitDTO one(@PathVariable final Long id) {
        return repository.findById(id)
                .map(mapper::toDto)
                .orElseThrow(() -> new FruitNotFoundException(id));
    }

    @PutMapping("/api/fruits/{id}")
    FruitDTO replaceFruit(
            @RequestBody final FruitDTO newFruit,
            @PathVariable final Long id) {
        newFruit.setId(id);
        return mapper.toDto(repository.save(mapper.toEntity(newFruit)));
    }

    @DeleteMapping("/api/fruits/{id}")
    void deleteFruit(@PathVariable final Long id) {
        repository.deleteById(id);
    }
}
```

The application source code is stored in AWS CodeCommit repository that is created and initialized from the CDK application in the `CodeCommitSource` construct:

```
import {prompts, prompt} from "prompts";
import {CodeConnectionsClient, CreateConnectionCommand, ProviderType, GetConnectionCommand} from "@aws-sdk/client-codeconnections";
import * as child from "child_process";


async function main() {
    console.log("This script will help you create a connection to an external version control system");
    try {
        const source = await promptSourceType();
        console.log(source)
        let cmd = 'npx cdk deploy --profile toolchain --all --require-approval never';
        if (source != 'codecommit') {
            const repoParameters = (await promptExternalSourceParamters(source));
            const command = await setupCodeConnection(source, repoParameters);
            console.log("The connection is created to connect the external version control system");
            await checkForCodeConnectionAvailable(repoParameters, 30, 10, command.ConnectionArn);
            repoParameters["connectionArn"]=command.ConnectionArn
            repoParameters["providerType"]=source
            await updateCdkJson(repoParameters)
            cmd=cmd.concat(` -c owner=${repoParameters.owner} -c repositoryName=${repoParameters.repositoryName} -c trunckBranchname=$
{repoParameters.trunkBranchName} -c connectionArn=${command.ConnectionArn} -c providerType=${source}`)
        }
        else {
            console.log("No external paramters required, proceeding with codecommit");
        }
        console.log(cmd)
        const commandParts = cmd.split(/\s+/);
        const resp = child.spawnSync(commandParts[0], commandParts.slice(1),{ stdio: 'inherit' });
        console.log(resp);
        if (resp.status !== 0) {
            throw new Error(`${resp.status} - Error`);
        }
    }catch (e) {
        console.error(e);
        process.exit(1);
    }
}

async function promptSourceType() {
  const source = prompts.select({
      type: 'select',
      name: 'source',
      message: 'Select pipeline source',
      choices: [
          { title: 'GitHub', value: ProviderType.GITHUB.toString() },
          { title: 'BitBucket', value: ProviderType.BITBUCKET.toString() },
          { title: 'Github Enterprise Server', value: ProviderType.GITHUB_ENTERPRISE_SERVER.toString() },
          { title: 'CodeCommit', value: 'codecommit' },
      ],
  }) as unknown as string;
  return source;
}

async function promptExternalSourceParamters(source :string) {
  const externalSourcePrompts= [
    {
        type: 'text',
        name:'profile',
        message: `Enter your AWS CLI profile name or ( Press ENTER to choose toolchain as default profile )`,
        default: 'toolchain'
    },
    {
        type: 'text',
        name: 'owner',
        message: `Enter ${source} owner`,
    },
    {
        type: 'text',
        name: 'repositoryName',
        message: `Enter ${source} repository name`,
    },
    {
        type: 'text',
        name: 'trunkBranchName',
        message: `Enter ${source} trunk branch name`,
    }
  ]

  const response= prompt(externalSourcePrompts);
  return response
}

async function setupCodeConnection( source: String, repoParameters: any){
    const client = new CodeConnectionsClient({ region: process.env.AWS_REGION ,profile:repoParameters.profile });
    let input = {
        ProviderType: source as ProviderType,
        ConnectionName: `dpri-${source}-${repoParameters.owner}`,
    };
    const command = new CreateConnectionCommand(input);
    const response = await client.send(command);
    return response
}

async function checkForCodeConnectionAvailable(
    repoParameters: any,
    maxRetries: number = 30,
    delay: number = 10,
    connectionArn:any

): Promise<boolean> {
    console.log("\n Waiting for connection to become available ...")
    console.log ("\n Please complete the authorization in the console when prompted")

    const client = new CodeConnectionsClient({ region: process.env.AWS_REGION ,profile:repoParameters.profile });

    for ( let attempt = 0; attempt < maxRetries; attempt ++ ){
        try{
            const getConnectionCommand = new GetConnectionCommand({
                ConnectionArn: connectionArn
            });
            const response = await client.send(getConnectionCommand);
            const status = response.Connection?.ConnectionStatus;

            if (response.Connection?.ConnectionStatus === "AVAILABLE") {
                console.log("\n Connection is available");
                return true;
            }
            else if (response.Connection?.ConnectionStatus == "ERROR"){
                console.log("\n Connection is not available yet. Retrying ...")
                await new Promise(resolve => setTimeout(resolve, delay * 1000));

            }
            console.log(`\n⏳ Current status: ${status} (Attempt ${attempt + 1}/${maxRetries})`);
            await new Promise(resolve => setTimeout(resolve, delay * 1000));

        }catch(error){
            console.error(`Error checking connection status: ${error}`);
            return false;
        }
    }
    throw new Error(`Connection not available after ${maxRetries} attempts`);
```

```
}

// create function to read cdk.json and add update context values as per prompt response
function updateCdkJson(repoParameters: any) {
    const fs = require('fs');
    const cdkJson = JSON.parse(fs.readFileSync('cdk.json', 'utf8'));
    cdkJson.context.owner = repoParameters.owner;
    cdkJson.context.repositoryName = repoParameters.repositoryName;
    cdkJson.context.trunkBranchName = repoParameters.trunkBranchName;
    cdkJson.context.connectionArn = repoParameters.connectionArn;
    cdkJson.context.providerType = repoParameters.providerType;
    fs.writeFileSync('cdk.json', JSON.stringify(cdkJson, null, 2));

}

main().catch(console.error);
```

Test Source Code

The test source code can be found in the src/test/java directory. It is intended to serve only as a reference and should be replaced by your own test source code.

The reference implementation includes source code for unit, integration and end-to-end testing. Unit and integration tests can be found in `src/test/java`. For example, `FruitControllerWithoutClassificationTest.java` performs unit tests of each API path with the JUnit testing library:

```
public void shouldReturnList() throws Exception {
  when(repository.findAll()).thenReturn(Arrays.asList(new Fruit("Mango", FruitClassification.pome), new Fruit("Dragonfruit", FruitClassification.berry)));

  this.mockMvc.perform(get("/api/fruits")).andDo(print()).andExpect(status().isOk())
      .andExpect(content().json("[{\"name\": \"Mango\"}, {\"name\": \"Dragonfruit\"}]"));
}
```

Acceptance tests are preformed with SoapUI and are defined in `fruit-api-soapui-project.xml`. They are executed by Maven using plugins in `pom.xml`.

Infrastructure Source Code

The infrastructure source code can be found in the infrastructure directory. It is intended to serve as a reference but much of the code can also be reused in your own CDK applications.

Infrastructure source code defines both the deployment of the pipeline and the deployment of the application are stored in `infrastructure/` folder and uses AWS Cloud Development Kit.

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

const image = new AssetImage('.', { target: 'build' });

const appName = Stack.of(this)
  .stackName.toLowerCase()
  .replace(`-${Stack.of(this).region}-`, '-');
const vpc = new ec2.Vpc(this, 'Vpc', {
  maxAzs: 3,
  natGateways: props?.natGateways,
});
new FlowLog(this, 'VpcFlowLog', {
  resourceType: FlowLogResourceType.fromVpc(vpc),
});

const dbName = 'fruits';
const dbSecret = new DatabaseSecret(this, 'AuroraSecret', {
  username: 'fruitapi',
  secretName: `${appName}-DB`,
});

const db = new DatabaseCluster(this, 'Database', {
  engine: DatabaseClusterEngine.auroraMysql({
    version: AuroraMysqlEngineVersion.VER_3_07_1,
  }),
  credentials: Credentials.fromSecret(dbSecret),
  writer: ClusterInstance.serverlessV2('writer'),
  defaultDatabaseName: dbName,
  serverlessV2MaxCapacity: 2,
  serverlessV2MinCapacity: 0.5,
  vpc,
  clusterIdentifier: appName,
  storageEncrypted: true,
});

const cluster = new ecs.Cluster(this, 'Cluster', {
  vpc,
  containerInsights: true,
  clusterName: appName,
});
const appLogGroup = new LogGroup(this, 'AppLogGroup', {
  retention: RetentionDays.ONE_WEEK,
  logGroupName: `/aws/ecs/service/${appName}`,
  removalPolicy: RemovalPolicy.DESTROY,
});
let deploymentConfig: IEcsDeploymentConfig | undefined = undefined;
if (props?.deploymentConfigName) {
  deploymentConfig = EcsDeploymentConfig.fromEcsDeploymentConfigName(
    this,
    'DeploymentConfig',
    props.deploymentConfigName,
  );
}
const appConfigEnabled =
  props?.appConfigRoleArn !== undefined &&
  props.appConfigRoleArn.length > 0;
const service = new ApplicationLoadBalancedCodeDeployedFargateService(
  this,
  'Api',
  {
    cluster,
    capacityProviderStrategies: [
      {
        capacityProvider: 'FARGATE_SPOT',
        weight: 1,
      },
    ],
    minHealthyPercent: 50,
    maxHealthyPercent: 200,
    desiredCount: 3,
    cpu: 512,
    memoryLimitMiB: 1024,
    taskImageOptions: {
      image,
      containerName: 'api',
      containerPort: 8080,
      family: appName,
      logDriver: AwsLogDriver.awsLogs({
        logGroup: appLogGroup,
        streamPrefix: 'service',
      }),
      secrets: {
        SPRING_DATASOURCE_USERNAME: Secret.fromSecretsManager(
          dbSecret,
          'username',
        ),
        SPRING_DATASOURCE_PASSWORD: Secret.fromSecretsManager(
          dbSecret,
          'password',
        ),
      },
      environment: {
        SPRING_DATASOURCE_URL: `jdbc:mysql://${db.clusterEndpoint.hostname}:${db.clusterEndpoint.port}/${dbName}`,
        APPCONFIG_AGENT_APPLICATION:
          this.node.tryGetContext('workloadName'),
        APPCONFIG_AGENT_ENVIRONMENT:
          this.node.tryGetContext('environmentName'),
        APPCONFIG_AGENT_ENABLED: appConfigEnabled.toString(),
      },
    },
    deregistrationDelay: Duration.seconds(5),
    responseTimeAlarmThreshold: Duration.seconds(3),
    targetHealthCheck: {
      healthyThresholdCount: 2,
      unhealthyThresholdCount: 2,
      interval: Duration.seconds(60),
      path: '/actuator/health',
    },
    deploymentConfig,
    terminationWaitTime: Duration.minutes(5),
    apiCanaryTimeout: Duration.seconds(5),
    apiTestSteps: [
      {
        name: 'getAll',
        path: '/api/fruits',
        jmesPath: 'length(@)',
        expectedValue: 5,
      },
    ],
  },
);

if (appConfigEnabled) {
  service.taskDefinition.addContainer('appconfig-agent', {
    image: ecs.ContainerImage.fromRegistry(
      'public.ecr.aws/aws-appconfig/aws-appconfig-agent:2.x',
    ),
    essential: false,
    logging: AwsLogDriver.awsLogs({
      logGroup: appLogGroup,
      streamPrefix: 'service',
    }),
    environment: {
      SERVICE_REGION: this.region,
      ROLE_ARN: props!.appConfigRoleArn!,
      ROLE_SESSION_NAME: appName,
      LOG_LEVEL: 'info',
    },
    portMappings: [{ containerPort: 2772 }],
  });

  service.taskDefinition.addToTaskRolePolicy(
    new PolicyStatement({
      actions: ['sts:AssumeRole'],
      resources: [props!.appConfigRoleArn!],
    }),
  );
}

service.service.connections.allowTo(
  db,
  ec2.Port.tcp(db.clusterEndpoint.port),
);

this.apiUrl = new CfnOutput(this, 'endpointUrl', {
  value: `http://${service.listener.loadBalancer.loadBalancerDnsName}`,
});
```

Notice that the infrastructure code is written in Typescript which is different from the Application Source Code (Java). This was done intentionally to demonstrate that CDK allows defining infrastructure code in whatever language is most appropriate for the team that owns the use of CDK in the organization.

Static Assets

There are no static assets used by the sample application.

Dependency Manifests

All third-party dependencies used by the sample application are define in the `pom.xml`:

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
        <groupId>org.liquibase</groupId>
        <artifactId>liquibase-core</artifactId>
    </dependency>
</dependencies>
```

Static Configuration

Static configuration for the application is defined in `src/main/resources/application.yml`:

```yaml
spring:
  application:
    name: fruit-api
  main:
    banner-mode: "off"
  jackson:
    default-property-inclusion: non_null


springdoc:
  swagger-ui:
    path: /swagger-ui

appconfig-agent:
  environment: alpha
  log-level-from:
    configuration: operations
```

Database Source Code

The database source code can be found in the src/main/resources/db directory. It is intended to serve only as a reference and should be replaced by your own database source code.

The code that manages the schema and initial data for the application is defined using Liquibase in `src/main/resources/db/changelog/db.changelog-master.yml`:

```
databaseChangeLog:
   - changeSet:
       id: "1"
       author: AWS
       changes:
       - createTable:
           tableName: fruit
           columns:
           - column:
               name: id
               type: bigint
               autoIncrement: true
               constraints:
                   primaryKey:  true
                   nullable:  false
           - column:
               name: name
               type: varchar(250)

       - insert:
           tableName: fruit
           columns:
           - column:
               name: name
               value: Apple

       - insert:
           tableName: fruit
           columns:
           - column:
               name: name
               value: Orange

       - insert:
           tableName: fruit
           columns:
           - column:
               name: name
               value: Banana

       - insert:
           tableName: fruit
           columns:
           - column:
               name: name
               value: Cherry

       - insert:
           tableName: fruit
           columns:
           - column:
               name: name
               value: Grape

   - changeSet:
       id: "2"
       author: AWS
       changes:
       - addColumn:
           tableName: fruit
           columns:
           - column:
               name: classification
               type: varchar(250)
               constraints:
                 nullable: true

       - update:
           tableName: fruit
           columns:
           - column:
               name: classification
               value: pome
           where: name='Apple'

       - update:
           tableName: fruit
           columns:
           - column:
               name: classification
               value: berry
           where: name='Orange'

       - update:
           tableName: fruit
           columns:
           - column:
               name: classification
               value: berry
           where: name='Banana'

       - update:
           tableName: fruit
           columns:
           - column:
               name: classification
               value: drupe
           where: name='Cherry'

       - update:
           tableName: fruit
           columns:
           - column:
               name: classification
               value: berry
           where: name='Grape'
```

Build

Actions in this stage all run in less than 10 minutes so that developers can take action on fast feedback before moving on to their next task. Each of the actions below are defined as code with AWS Cloud Development Kit.

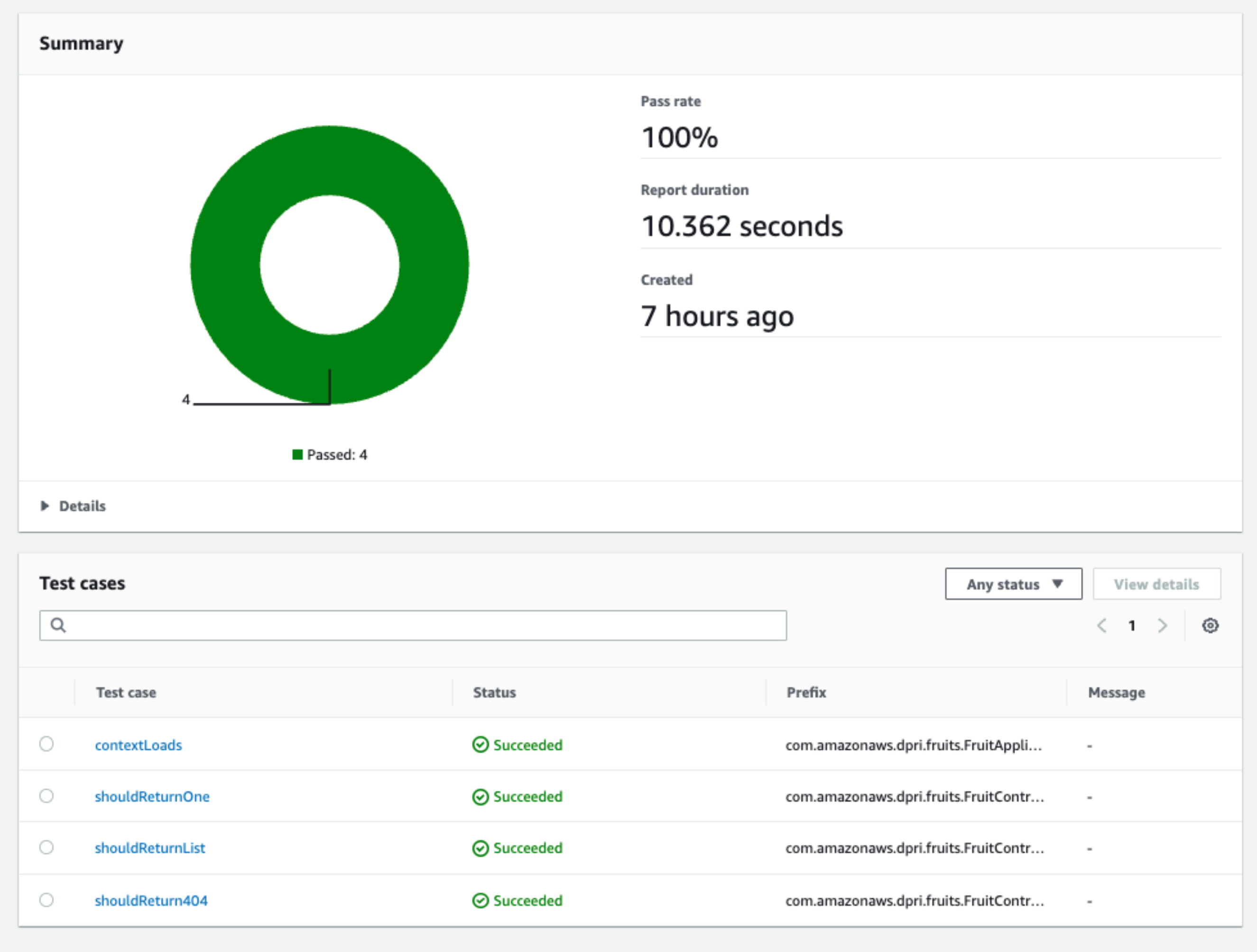
Build Code

The Java source code is compiled, unit tested and packaged by Maven. A step is added to the pipeline through a CDK construct called `MavenBuild`:

```
const stepProps = {
  input: props.source,
  commands: [],
  buildEnvironment: {
    buildImage: LinuxBuildImage.STANDARD_6_0,
  },
  partialBuildSpec: BuildSpec.fromObject({
    env: {
      variables: {
        MAVEN_OPTS: props.mavenOpts || '-XX:+TieredCompilation -XX:TieredStopAtLevel=1',
        MAVEN_ARGS: props.mavenArgs || '--batch-mode --no-transfer-progress',
      },
    },
    phases: {
      install: {
        'runtime-versions': {
          java: (props.javaRuntime || 'corretto17'),
        },
      },
      build: {
        commands: [`mvn \${MAVEN_ARGS} clean ${props.mavenGoal || 'verify'}`],
      },
    },
    cache: props.cacheBucket ? {
      paths: ['/root/.m2/**/*'],
    } : undefined,
    reports: {
      unit: {
        'files': ['target/surefire-reports/*.xml'],
        'file-format': 'JUNITXML',
      },
      integration: {
        'files': ['target/soapui-reports/*.xml'],
        'file-format': 'JUNITXML',
      },
    },
    version: '0.2',
  }),
  cache: props.cacheBucket ? Cache.bucket(props.cacheBucket) : undefined,
  primaryOutputDirectory: '.',
};
super(id, stepProps);
```

Unit Tests

The unit tests are run by Maven at the same time the `Build Code` action occurs. The results of the unit tests are uploaded to AWS Code Build Test Reports to track over time.



Summary

Pass rate
100%

Report duration
10.362 seconds

Created
7 hours ago

Test cases

	Test case	Status	Prefix	Message
○	contextLoads	✓ Succeeded	com.amazonaws.dpri.fruits.FruitAppli...	-
○	shouldReturnOne	✓ Succeeded	com.amazonaws.dpri.fruits.FruitContr...	-
○	shouldReturnList	✓ Succeeded	com.amazonaws.dpri.fruits.FruitContr...	-
○	shouldReturn404	✓ Succeeded	com.amazonaws.dpri.fruits.FruitContr...	-

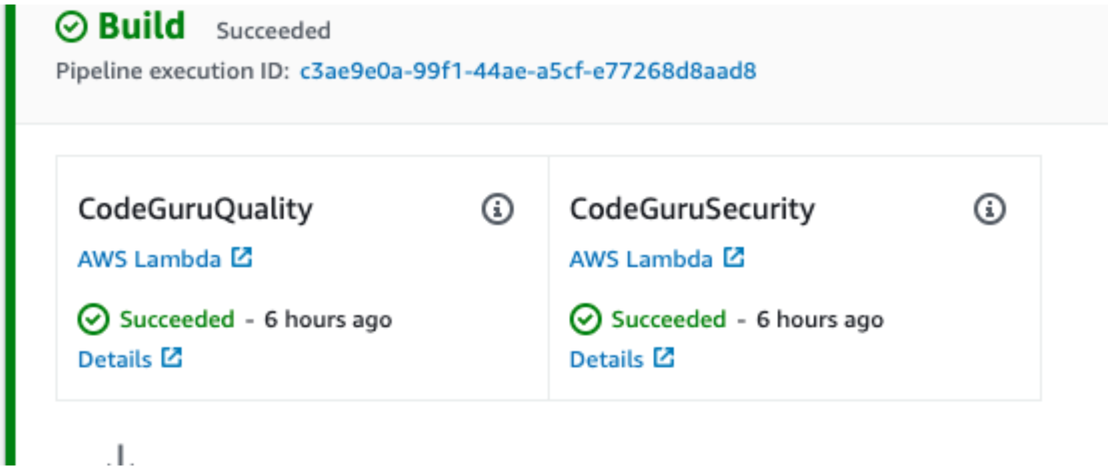
Code Quality

A CDK construct was created to require that Amazon CodeGuru performed a review on the most recent changes and that the recommendations don't exceed the severity thresholds. If no review was found or if the severity thresholds were exceeded, the pipeline fails. The construct is added to the pipeline with:

```
import { CodeGuruReviewCheck, CodeGuruReviewFilter } from './codeguru-review-check';

…

    const codeGuruSecurity = new CodeGuruReviewCheck('CodeGuruSecurity', {
      source: pipelineSource,
      reviewRequired: false,
      filter: CodeGuruReviewFilter.defaultCodeSecurityFilter(),
    });
    const codeGuruQuality = new CodeGuruReviewCheck('CodeGuruQuality', {
      source: pipelineSource,
      reviewRequired: false,
      filter: CodeGuruReviewFilter.defaultCodeQualityFilter(),
    });
```

The `Filter` attribute can be customized to control what categories of recommendations are considered and what the thresholds are:

```
export enum CodeGuruReviewRecommendationCategory {
    AWS_BEST_PRACTICES = 'AWSBestPractices',
    AWS_CLOUDFORMATION_ISSUES = 'AWSCloudFormationIssues',
    CODE_INCONSISTENCIES = 'CodeInconsistencies',
    CODE_MAINTENANCE_ISSUES = 'CodeMaintenanceIssues',
    CONCURRENCY_ISSUES = 'ConcurrencyIssues',
    DUPLICATE_CODE = 'DuplicateCode',
    INPUT_VALIDATIONS = 'InputValidations',
    JAVA_BEST_PRACTICES = 'JavaBestPractices',
    PYTHON_BEST_PRACTICES = 'PythonBestPractices',
    RESOURCE_LEAKS = 'ResourceLeaks',
    SECURITY_ISSUES = 'SecurityIssues',
}
export class CodeGuruReviewFilter {
    // Limit which recommendation categories to include
    recommendationCategories!: CodeGuruReviewRecommendationCategory[];

    // Fail if more that this # of lines of code were suppressed aws-codeguru-reviewer.yml
    maxSuppressedLinesOfCodeCount?: number;

    // Fail if more than this # of CRITICAL recommendations were found
    maxCriticalRecommendations?: number;

    // Fail if more than this # of HIGH recommendations were found
    maxHighRecommendations?: number;

    // Fail if more than this # of MEDIUM recommendations were found
    maxMediumRecommendations?: number;

    // Fail if more than this # of INFO recommendations were found
    maxInfoRecommendations?: number;

    // Fail if more than this # of LOW recommendations were found
    maxLowRecommendations?: number;
}
```



✓ Build Succeeded

Pipeline execution ID: c3ae9e0a-99f1-44ae-a5cf-e77268d8aad8

CodeGuruQuality

AWS Lambda

✓ Succeeded - 6 hours ago

Details

CodeGuruSecurity

AWS Lambda

✓ Succeeded - 6 hours ago

Details

Additionally, cdk-nag is run against both the pipeline stack and the deployment stack to identify any security issues with the resources being created. The pipeline will fail if any are detected. The following code demonstrates how cdk-nag is called as a part of the build stage. The code also demonstrates how to suppress findings.

```
import { App, Aspects } from 'aws-cdk-lib';
import { Annotations, Match, Template } from 'aws-cdk-lib/assertions';
import { SynthesisMessage } from 'aws-cdk-lib/cx-api';
import { AwsSolutionsChecks, NagSuppressions } from 'cdk-nag';
import { DeploymentStack } from '../src/deployment';


function synthesisMessageToString(sm: SynthesisMessage): string {
  return `${sm.entry.data} [${sm.id}]`;
}
expect.addSnapshotSerializer({
  test: (val) => typeof val === 'string' && val.match(/^dummy.dkr.ecr.us-east.1/) !== null,
  serialize: () => '"dummy-ecr-image"',
});
expect.addSnapshotSerializer({
  test: (val) => typeof val === 'string' && val.match(/^[a-f0-9]+\.zip$/) !== null,
  serialize: () => '"code.zip"',
});

describe('cdk-nag', () => {
  let stack: DeploymentStack;
  let app: App;

  beforeAll(() => {
    const appName = 'fruit-api';
    const workloadName = 'food';
    const environmentName = 'unit-test';
    app = new App({ context: { appName, environmentName, workloadName } });
    stack = new DeploymentStack(app, 'TestStack', {
      env: {
        account: 'dummy',
        region: 'us-east-1',
      },
    });
    Aspects.of(stack).add(new AwsSolutionsChecks());

    // Suppress CDK-NAG for TaskDefinition role and ecr:GetAuthorizationToken permission
    NagSuppressions.addResourceSuppressionsByPath(
      stack,
      `/${stack.stackName}/Api/TaskDef/ExecutionRole/DefaultPolicy/Resource`,
      [{ id: 'AwsSolutions-IAM5', reason: 'Allow ecr:GetAuthorizationToken', appliesTo: ['Resource::*'] }],
    );

    // Suppress CDK-NAG for secret rotation
    NagSuppressions.addResourceSuppressionsByPath(
      stack,
      `/${stack.stackName}/AuroraSecret/Resource`,
      [{ id: 'AwsSolutions-SMG4', reason: 'Dont require secret rotation' }],
    );

    // Suppress CDK-NAG for RDS Serverless
    NagSuppressions.addResourceSuppressionsByPath(
      stack,
      `/${stack.stackName}/Database/Resource`,
      [
        { id: 'AwsSolutions-RDS6', reason: 'IAM authentication not supported on Serverless v1' },
        { id: 'AwsSolutions-RDS10', reason: 'Disable delete protection to simplify cleanup of Reference Implementation' },
        { id: 'AwsSolutions-RDS11', reason: 'Custom port not supported on Serverless v1' },
        { id: 'AwsSolutions-RDS14', reason: 'Backtrack not supported on Serverless v1' },
        { id: 'AwsSolutions-RDS16', reason: 'CloudWatch Log Export not supported on Serverless v1' },
      ],
    );

    NagSuppressions.addResourceSuppressionsByPath(stack, [
      `/${stack.stackName}/Api/DeploymentGroup/Deployment/DeploymentProvider/framework-onEvent`,
      `/${stack.stackName}/Api/DeploymentGroup/Deployment/DeploymentProvider/framework-isComplete`,
      `/${stack.stackName}/Api/DeploymentGroup/Deployment/DeploymentProvider/framework-onTimeout`,
      `/${stack.stackName}/Api/DeploymentGroup/Deployment/DeploymentProvider/waiter-state-machine`,
    ], [
      { id: 'AwsSolutions-IAM5', reason: 'Unrelated to construct under test' },
      { id: 'AwsSolutions-L1', reason: 'Unrelated to construct under test' },
      { id: 'AwsSolutions-SF1', reason: 'Unrelated to construct under test' },
      { id: 'AwsSolutions-SF2', reason: 'Unrelated to construct under test' },
    ], true);

    // Ignore findings from access log bucket
    NagSuppressions.addResourceSuppressionsByPath(stack, [
      `/${stack.stackName}/Api/AccessLogBucket`,
    ], [
      { id: 'AwsSolutions-S1', reason: 'Dont need access logs for access log bucket' },
      { id: 'AwsSolutions-IAM5', reason: 'Allow resource:*', appliesTo: ['Resource::*'] },
    ]);

    NagSuppressions.addResourceSuppressionsByPath(stack, [
      `/${stack.stackName}/Api/Canary/ServiceRole`,
    ], [{ id: 'AwsSolutions-IAM5', reason: 'Allow resource:*' }]);

    NagSuppressions.addResourceSuppressionsByPath(stack, [
      `/${stack.stackName}/Api/CanaryArtifactsBucket`,
    ], [{ id: 'AwsSolutions-S1', reason: 'Dont need access logs for canary bucket' }]);

    NagSuppressions.addResourceSuppressionsByPath(stack, [
      `/${stack.stackName}/Api/DeploymentGroup/ServiceRole`,
    ], [
      { id: 'AwsSolutions-IAM4', reason: 'Allow AWSCodeDeployRoleForECS policy', appliesTo: ['Policy::arn:<AWS::Partition>:iam::aws:policy/
AWSCodeDeployRoleForECS'] },
    ]);

    NagSuppressions.addResourceSuppressionsByPath(stack, [
      `/${stack.stackName}/Api/DeploymentGroup/Deployment`,
    ], [
      {
        id: 'AwsSolutions-IAM4',
        reason: 'Allow AWSLambdaBasicExecutionRole policy',
        appliesTo: ['Policy::arn:<AWS::Partition>:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole'],
      },
    ], true);

    NagSuppressions.addResourceSuppressionsByPath(stack, [
      `/${stack.stackName}/Api/TaskDef`,
    ], [
      {
        id: 'AwsSolutions-ECS2',
        reason: 'Allow environment variables for configuration of values that are not confidential',
      },
    ]);

    NagSuppressions.addResourceSuppressionsByPath(stack, [
      `/${stack.stackName}/Api/LB/SecurityGroup`,
    ], [
      {
        id: 'AwsSolutions-EC23',
        reason: 'Allow public inbound access on ELB',
      },
    ]);
  });

  test('Snapshot', () => {
    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });

  test('cdk-nag AwsSolutions Pack errors', () => {
    const errors = Annotations.fromStack(stack).findError(
      '*',
      Match.stringLikeRegexp('AwsSolutions-.*'),
    ).map(synthesisMessageToString);
    expect(errors).toHaveLength(2);
  });

  test('cdk-nag AwsSolutions Pack warnings', () => {
    const warnings = Annotations.fromStack(stack).findWarning(
      '*',
      Match.stringLikeRegexp('AwsSolutions-.*'),
    ).map(synthesisMessageToString);
    expect(warnings).toHaveLength(0);
  });
});

describe('Deployment without AppConfig', () => {
  let stack: DeploymentStack;
  let app: App;

  beforeAll(() => {
    const appName = 'fruit-api';
    const environmentName = 'unit-test';
    app = new App({ context: { appName, environmentName } });
    stack = new DeploymentStack(app, 'TestStack', {
      env: {
        account: 'dummy',
        region: 'us-east-1',
      },
    });
  });

  test('Snapshot', () => {
    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
  test('taskdef', () => {
    const template = Template.fromStack(stack);
    template.hasResourceProperties('AWS::ECS::TaskDefinition', {
      ContainerDefinitions: [
        {
          Environment: [{
            Name: 'SPRING_DATASOURCE_URL',
          }, {
            Name: 'APPCONFIG_AGENT_APPLICATION',
          }, {
            Name: 'APPCONFIG_AGENT_ENVIRONMENT',
            Value: 'unit-test',
          }, {
            Name: 'APPCONFIG_AGENT_ENABLED',
            Value: 'false',
          }],
        },
      ],
    });
  });
});

describe('Deployment with AppConfig', () => {
  let stack: DeploymentStack;
  let app: App;

  beforeAll(() => {
    const appName = 'fruit-api';
    const workloadName = 'food';
    const environmentName = 'unit-test';
    app = new App({ context: { appName, environmentName, workloadName } });
    stack = new DeploymentStack(app, 'TestStack', {
      appConfigRoleArn: 'dummy-role-arn',
      env: {
        account: 'dummy',
        region: 'us-east-1',
      },
    });
  });

  test('Snapshot', () => {
    const template = Template.fromStack(stack);
    expect(template.toJSON()).toMatchSnapshot();
  });
  test('taskdef', () => {
    const template = Template.fromStack(stack);
    template.hasResourceProperties('AWS::ECS::TaskDefinition', {
      ContainerDefinitions: [
        {
          Environment: [{
            Name: 'SPRING_DATASOURCE_URL',
          }, {
            Name: 'APPCONFIG_AGENT_APPLICATION',
            Value: 'food',
          }, {
            Name: 'APPCONFIG_AGENT_ENVIRONMENT',
            Value: 'unit-test',
          }, {
            Name: 'APPCONFIG_AGENT_ENABLED',
            Value: 'true',
          }],
        },
        {
          Environment: [{
            Name: 'SERVICE_REGION',
            Value: 'us-east-1',
          }, {
            Name: 'ROLE_ARN',
            Value: 'dummy-role-arn',
          }, {
            Name: 'ROLE_SESSION_NAME',
          }, {
            Name: 'LOG_LEVEL',
            Value: 'info',
          }],
        },
      ],
    });
  });
});
```

Secrets Detection

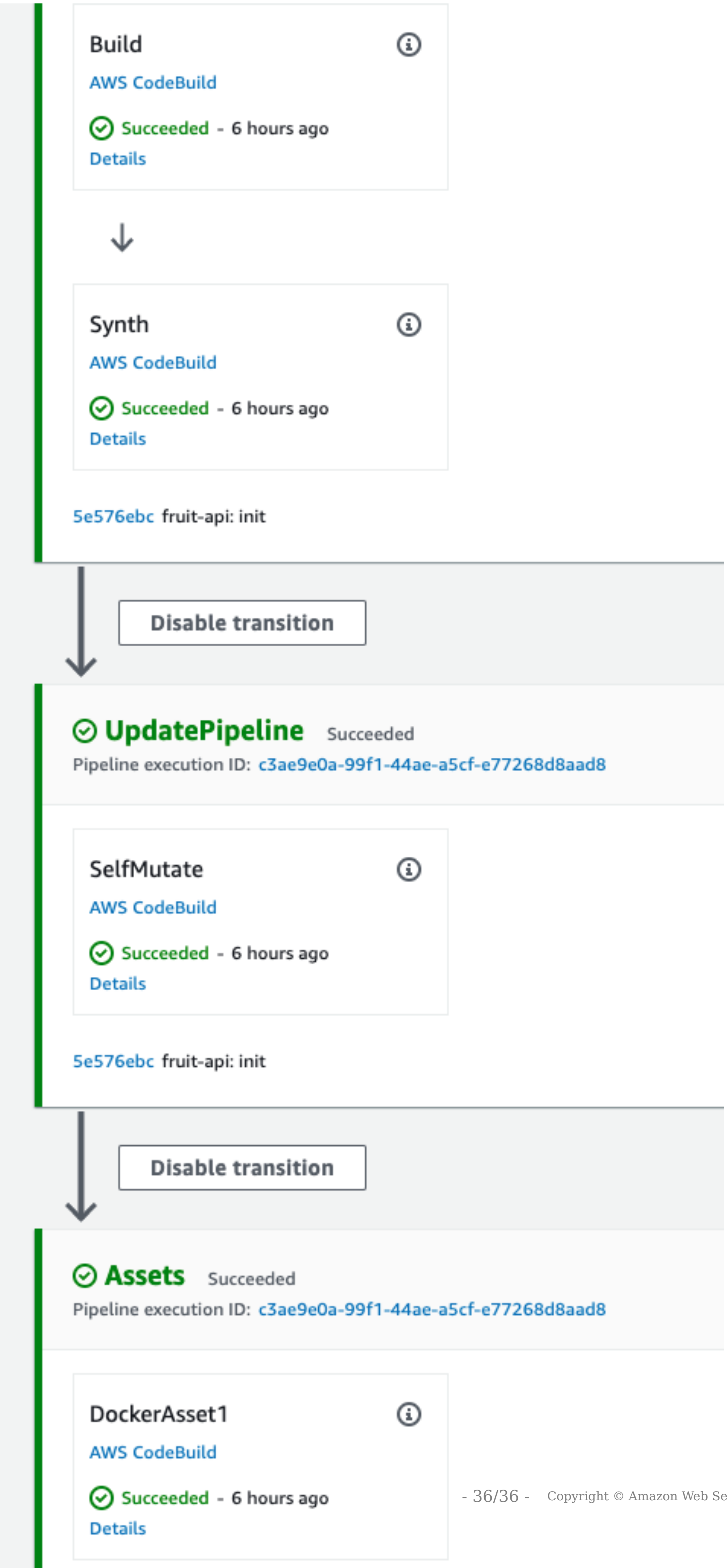
The same CDK construct that was created for *Code Quality* above is also used for secrets detection with Amazon CodeGuru.

Static Application Security Testing (SAST)

The same CDK construct that was created for *Code Quality* above is also used for SAST with Amazon CodeGuru.

Package and Store Artifact(s)

AWS Cloud Development Kit handles the packaging and storing of assets during the `Synth` action and `Assets` stage. The `Synth` action generates the CloudFormation templates to be deployed into the subsequent environments along with staging up the files necessary to create a docker image. The `Assets` stage then performs the docker build step to create a new image and push the image to Amazon ECR repositories in each environment account.



Build

AWS CodeBuild

✓ Succeeded - 6 hours ago

Details

↓

Synth

AWS CodeBuild

✓ Succeeded - 6 hours ago

Details

5e576ebc fruit-api: init

✓ UpdatePipeline Succeeded

Pipeline execution ID: c3ae9e0a-99f1-44ae-a5cf-e77268d8aad8

SelfMutate

AWS CodeBuild

✓ Succeeded - 6 hours ago

Details

5e576ebc fruit-api: init

✓ Assets Succeeded

Pipeline execution ID: c3ae9e0a-99f1-44ae-a5cf-e77268d8aad8

DockerAsset1

AWS CodeBuild

✓ Succeeded - 6 hours ago

Details