

Deployment Pipeline Reference Architecture

September 2022

AWS Professional Services

Copyright © Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Table of contents

1. Overview	3
Architecture	4
Business Outcomes	5
Definitions	5
2. Application Pipeline	6
Architecture	6
Reference Implementations	17
Additional Sources	81

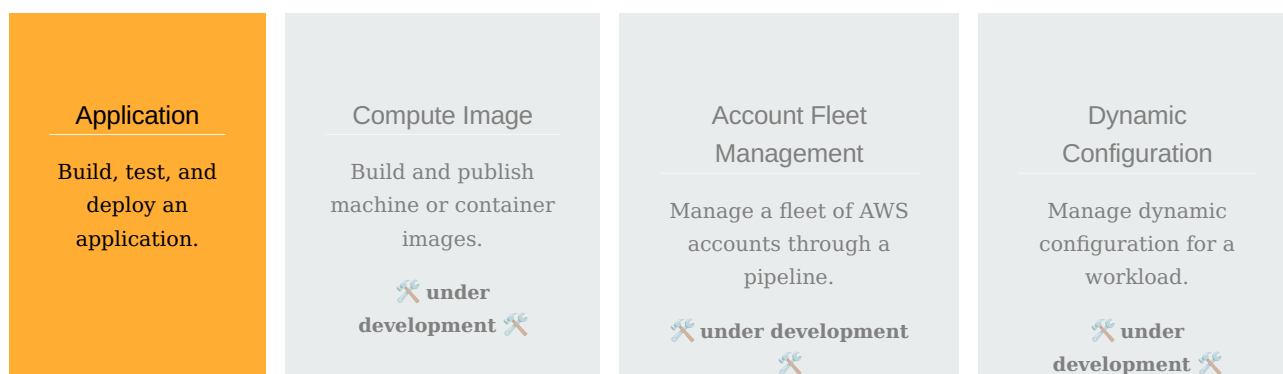
1. Overview

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.

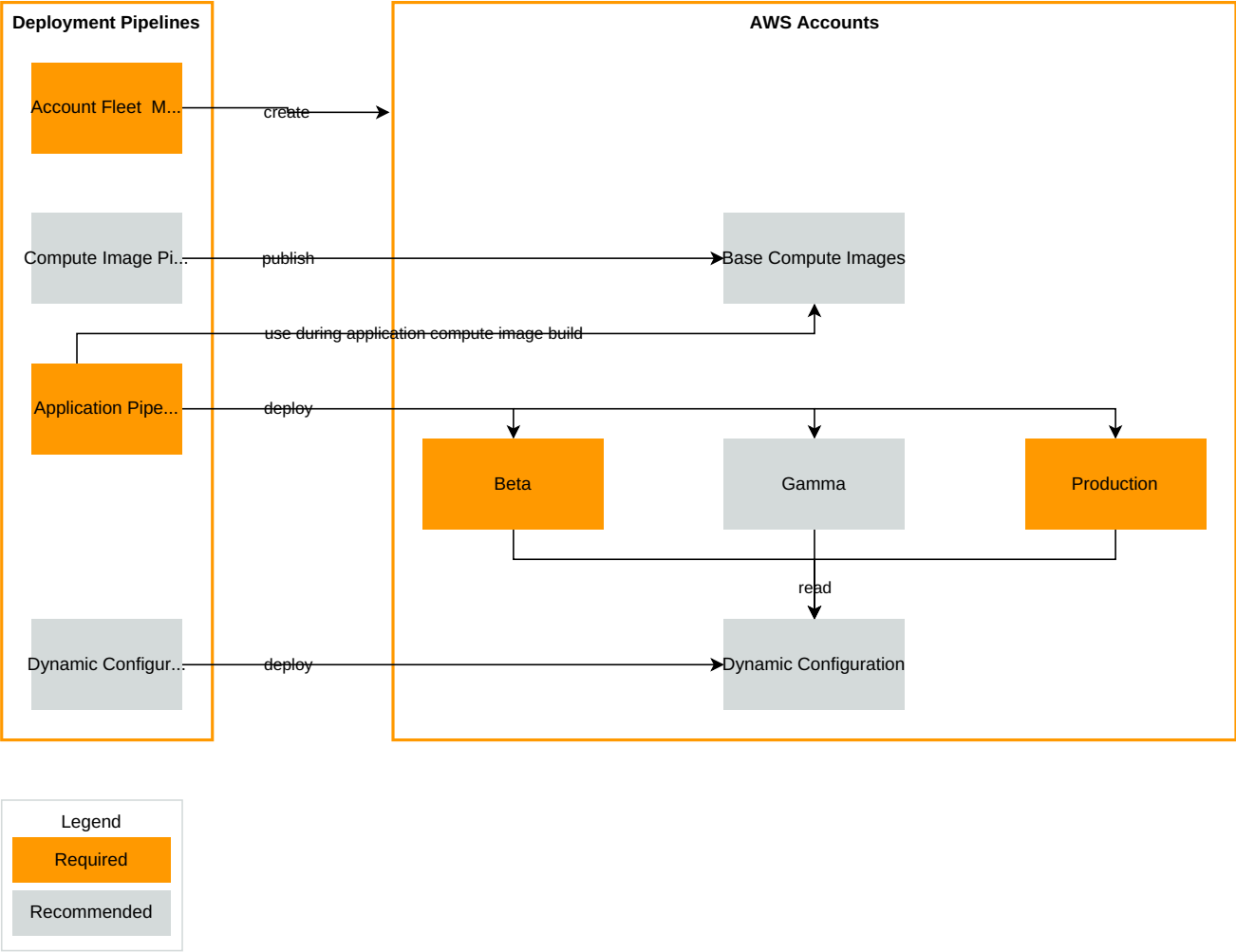
Customers and third-party vendors can use the DPRA to create implementations - reference or otherwise - using their own set of services and tools. We have included reference implementations that use AWS and third-party tools. When an AWS service/tool is available, we list it; when there are no AWS services/tools, 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:



Architecture

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



Text is not SVG - cannot display

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

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

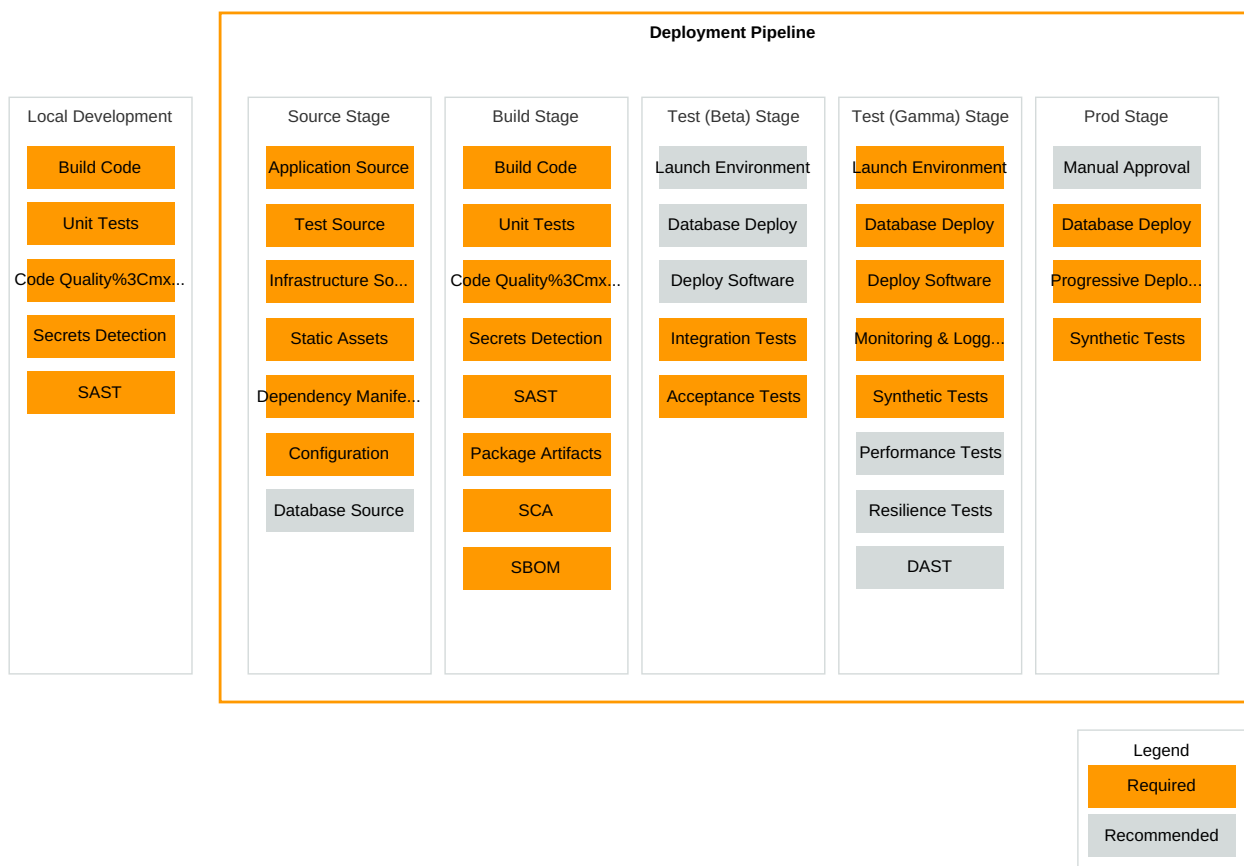
An **environment** is an isolated target for deploying and testing a workload and its dependencies. Environments can be created for validating changes, achieving data compliance, or for improving resiliency. Example environments include creating separate AWS accounts for each developer, creating separate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

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 regularly 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.



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 performed 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 undesirable 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

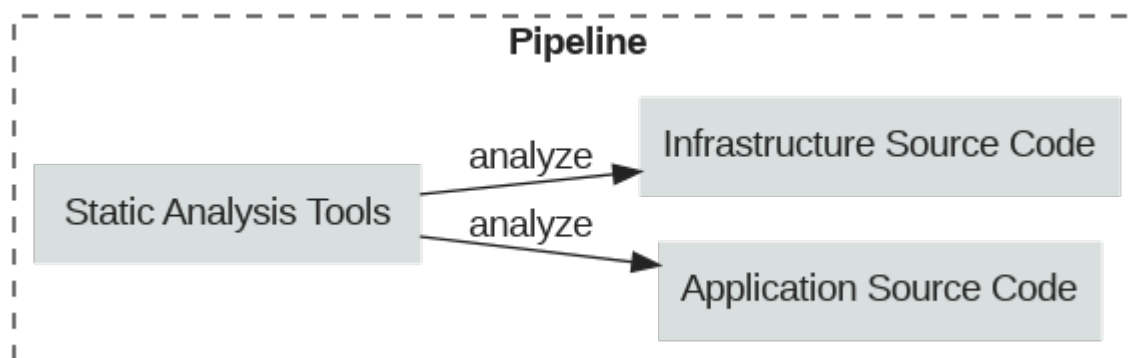
Convert code into artifacts that can be promoted through environments. Most builds complete in seconds. Examples include but are not limited to Maven and tsc.

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.

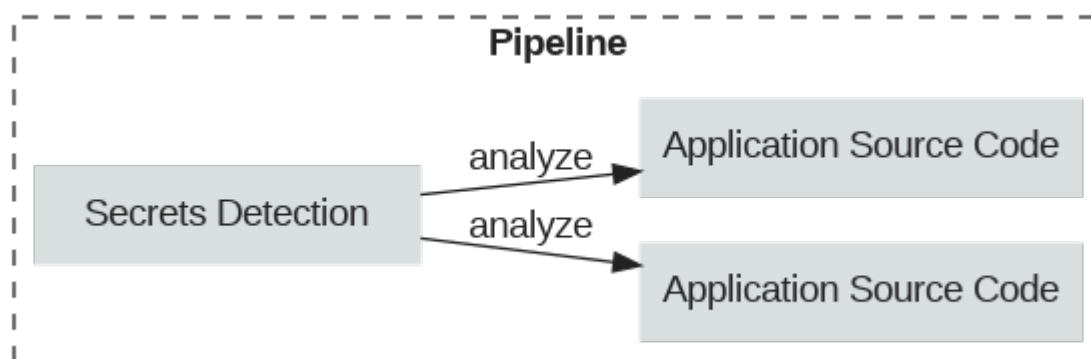
Code Quality

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.



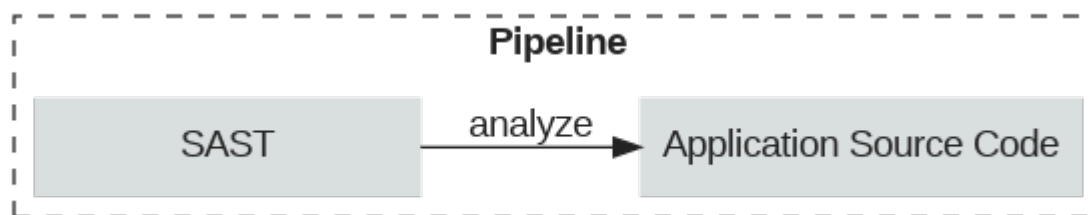
Secrets Detection

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)

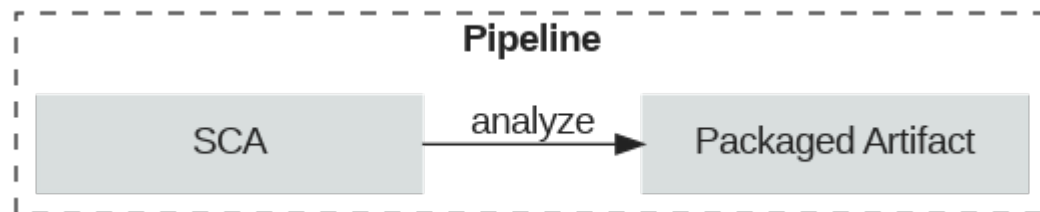
While the *Build Code* action will package most of the relevant artifacts, there may be additional steps to automate for packaging the code artifacts. Artifacts should only be built and packaged once and then deployed to various environments to validate the artifact. Artifacts should never be rebuilt 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)

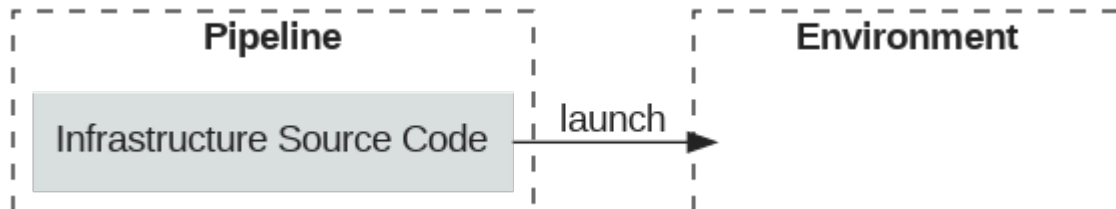
Generate a software bill of materials (SBOM) report detailing all the dependencies used. Examples of SBOM formats include SPDX and CycloneDX.

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

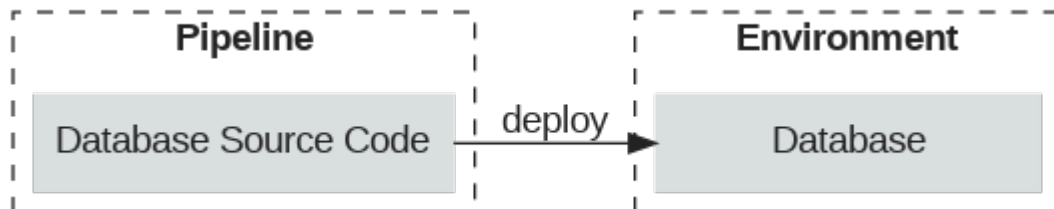
Use a compute image from an image repository (e.g., AMI or a container repo) and launch an environment from the image using *Infrastructure Source Code*. The beta image is generally not accessible to public customers and is only used for internal software validation. The beta environment should be in 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

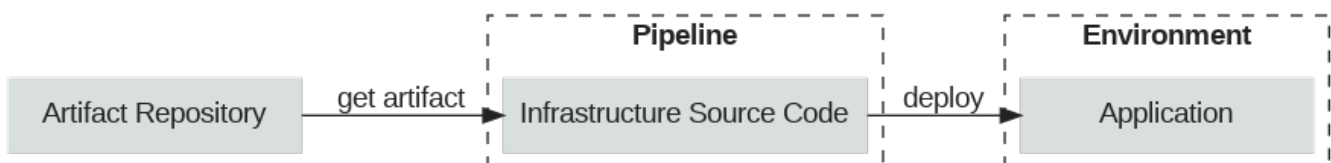
Apply changes to the beta database using the *Database Source Code*. Changes should be made in a manner that ensures rollback safety. Best practice is to connect to the beta database through cross-account IAM roles and IAM database authentication for RDS rather than long lived database credentials. If 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 architecture but should be addressed during *Database Deploy* in preparation of subsequent testing activity.



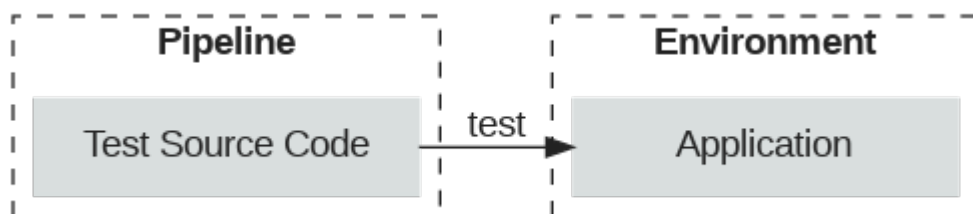
Deploy Software

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.



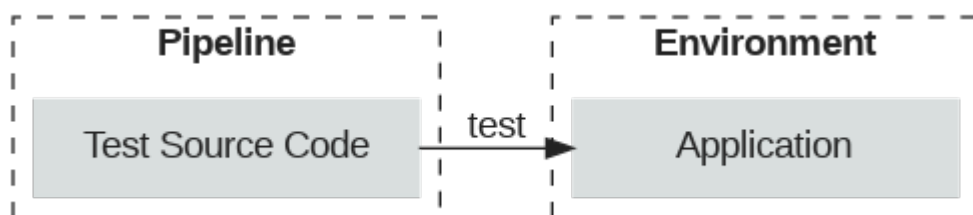
Integration Tests

Run automated tests that verify if the application satisfies 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 tests 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.

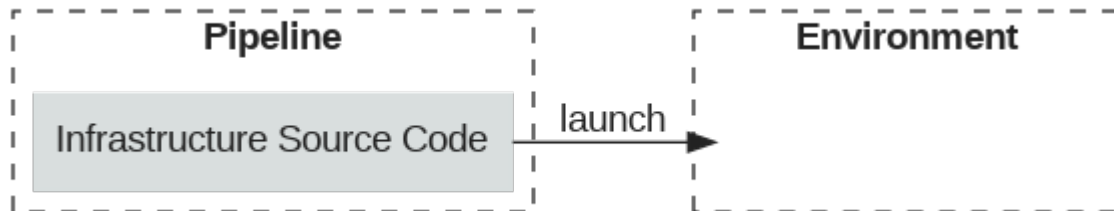


Test (Gamma)

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.

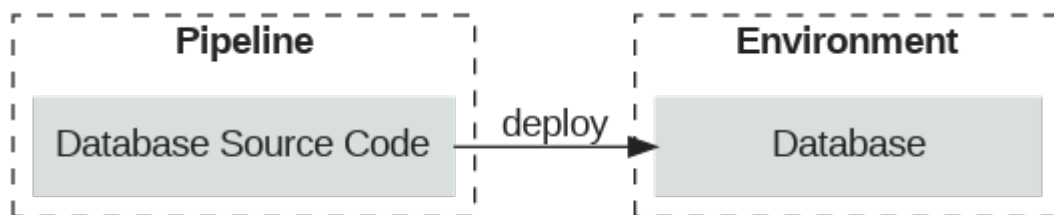
Launch Environment

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.



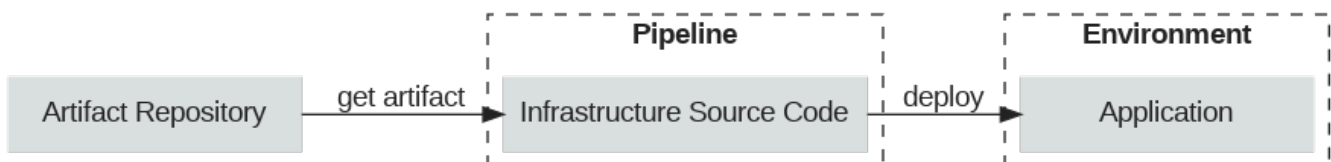
Database Deploy

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.



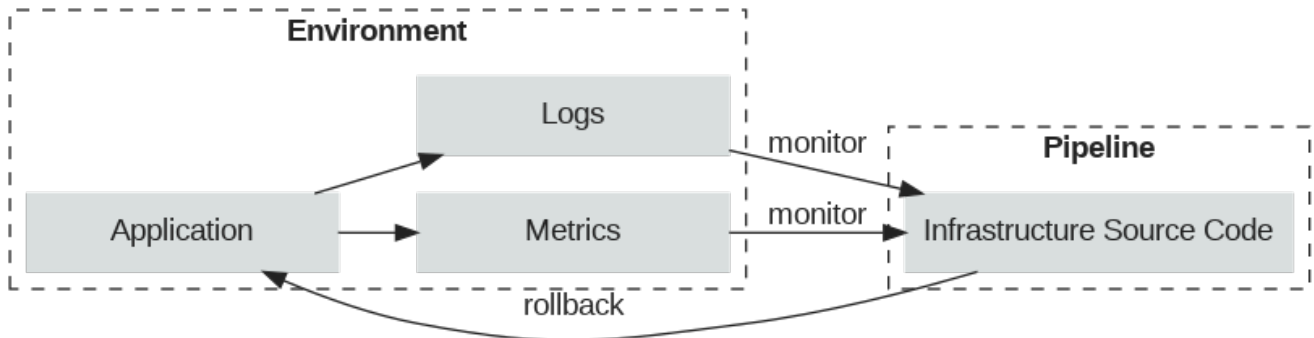
Deploy Software

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.



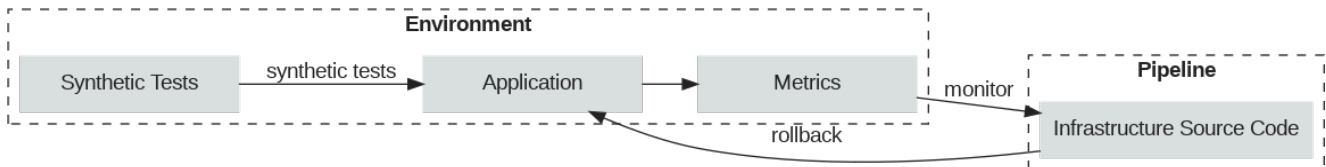
Application Monitoring & Logging

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.



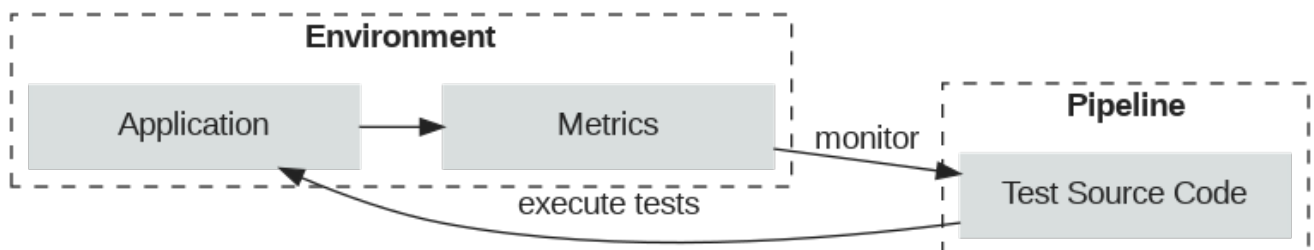
Synthetic Tests

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.



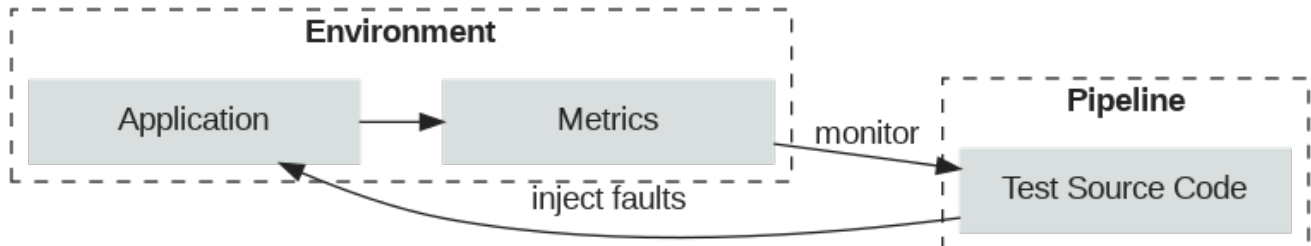
Performance Tests

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 degradation. 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.

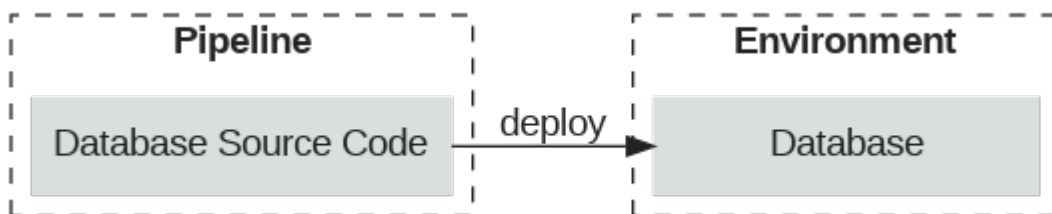
Prod

Manual Approval

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

Database Deploy

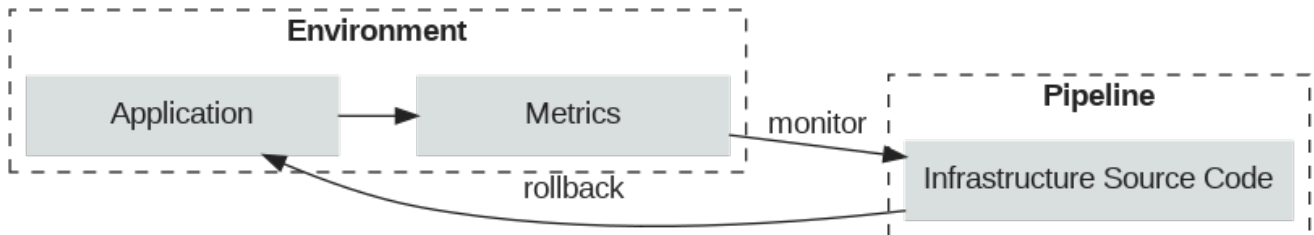
Apply changes to the production database using the *Database Source Code*. Changes should be made in a manner that ensures rollback safety. Best practice is to connect to the production database through cross-account IAM roles and IAM database authentication for RDS rather than long lived database credentials. 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

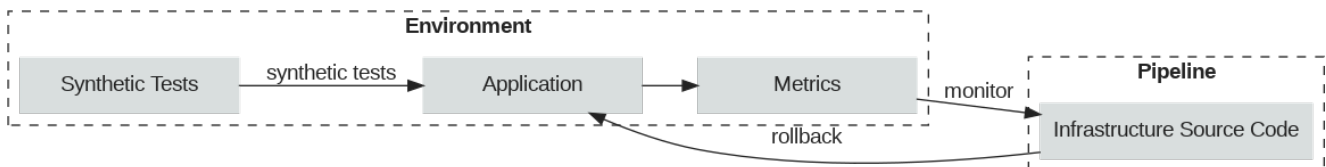
Deployments should be made progressively in waves to limit the impact of failures. A common approach is to deploy changes to a subset of AWS regions and allow sufficient bake time to monitor performance and behavior before proceeding with additional waves of AWS regions.

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

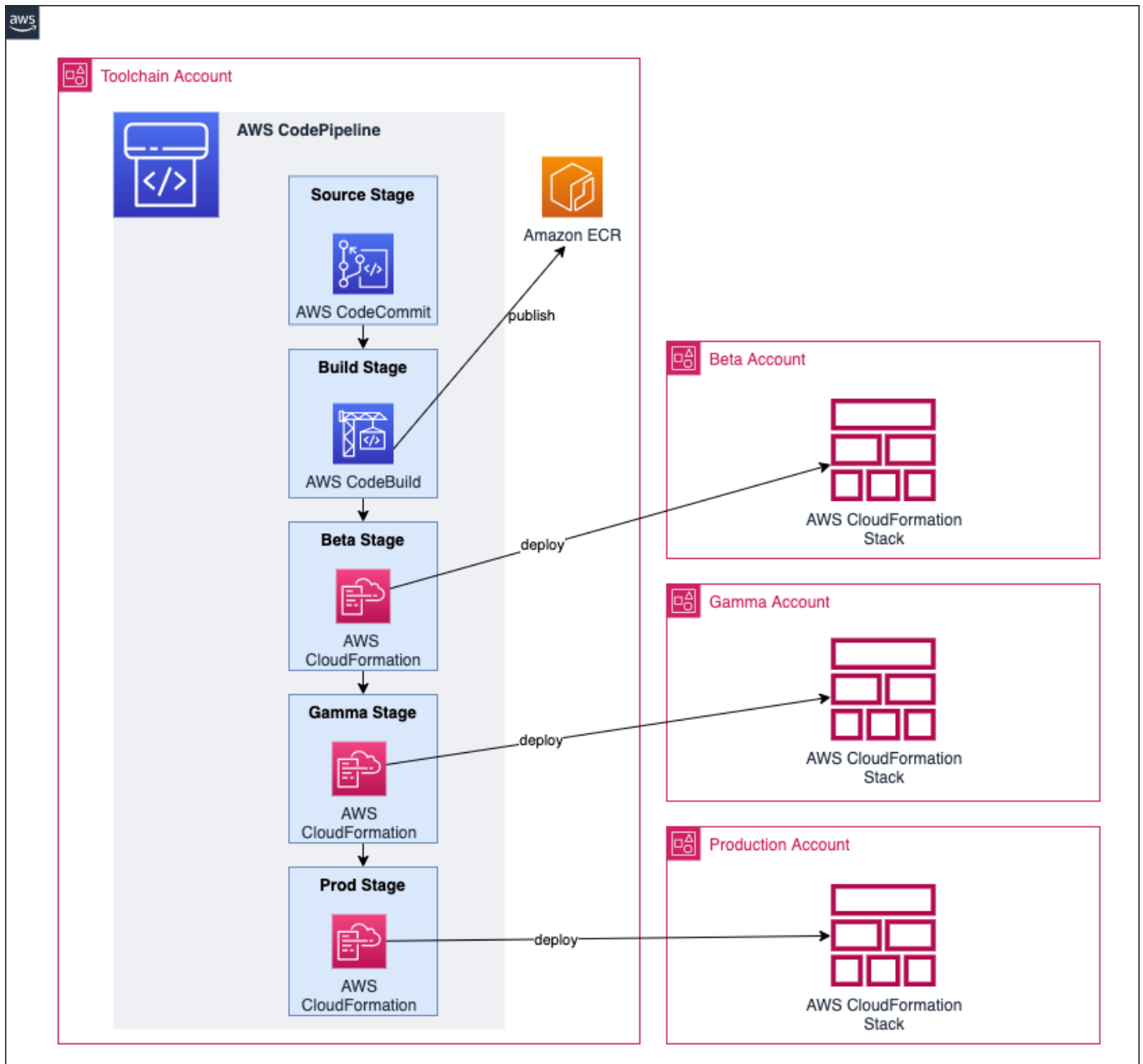
This presents a reference implementation of the Application Pipeline reference architecture. The pipeline is built with AWS CodePipeline and uses AWS CodeBuild for building the software and performing testing tasks. All the infrastructure for this reference implementation is defined with AWS Cloud Development 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.



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 not followed 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:

- ❗ **cdk bootstrap with AdministratorAccess** - the default policy used for `cdk bootstrap` is `AdministratorAccess` but should be replaced with a more appropriate policy with least privilege in your account.
- ❗ **LS 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.

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

    FruitController(final FruitRepository r) {
        this.repository = r;
    }

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

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

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

    @PutMapping("/api/fruits/{id}")
    FruitDTO replaceFruit(
        @RequestBody final FruitDTO newFruit,
        @PathVariable final Long id) {
        Fruit entity = repository.findById(id)
            .map(fruit -> {
                fruit.setName(newFruit.getName());
                return repository.save(fruit);
            })
            .orElseGet(() -> {
                newFruit.setId(id);
                return repository.save(convertToEntity(newFruit));
            });
        return convertToDto(entity);
    }

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

    FruitDTO convertToDto(final Fruit fruit) {
        FruitDTO dto = new FruitDTO();
        dto.setId(fruit.getId());
        dto.setName(fruit.getName());
        return dto;
    }

    Fruit convertToEntity(final FruitDTO fruit) {
        Fruit entity = new Fruit();
        entity.setId(fruit.getId());
        entity.setName(fruit.getName());
        return entity;
    }
}
```

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

```
super(scope, id);
this.trunkBranchName = props?.trunkBranchName || 'main';
let gitignore = fs.readFileSync('.gitignore').toString().split(/\r?\n/);
gitignore.push('.git/');

// Allow canary code to package properly
// see: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Synthetics_Canaries_WritingCanary_Nodejs.html#CloudWatch_Synthetics_Canaries_package
gitignore = gitignore.filter(g => g !== 'node_modules/');
gitignore.push('/node_modules/');

const codeAsset = new Asset(this, 'SourceAsset', {
    path: '.',
    ignoreMode: IgnoreMode.GIT,
    exclude: gitignore,
});
```

```

this.repository = new Repository(this, 'CodeCommitRepo', {
  repositoryName: props.repositoryName,
  code: Code.fromAsset(codeAsset, this.trunkBranchName),
});

if (props.associateCodeGuru !== false) {
  new CfnRepositoryAssociation(this, 'CfnRepositoryAssociation', {
    name: this.repository.repositoryName,
    type: 'CodeCommit',
  });
}
this.codePipelineSource = CodePipelineSource.codeCommit(this.repository, this.trunkBranchName);

```

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, `FruitControllerTest.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"), new Fruit("Dragonfruit")));

    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 ServerlessCluster(this, 'AuroraCluster', {
  engine: DatabaseClusterEngine.AURORA_MYSQL,
  vpc,
  credentials: Credentials.fromSecret(dbSecret),
  defaultDatabaseName: dbName,
  deletionProtection: false,
  clusterIdentifier: appName,
});

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/${appName}`,
  removalPolicy: RemovalPolicy.DESTROY,
});
let deploymentConfig: IEcsDeploymentConfig | undefined = undefined;
if (props?.deploymentConfigName) {
  deploymentConfig = EcsDeploymentConfig.fromEcsDeploymentConfigName(this, 'DeploymentConfig', props.deploymentConfigName);
}
const service = new ApplicationLoadBalancedCodeDeployedFargateService(this, 'Api', {
  cluster,
  capacityProviderStrategies: [
    {
      capacityProvider: 'FARGATE_SPOT',
      weight: 1,
    },
  ],
  minHealthyPercent: 50,
  maxHealthyPercent: 200,
  desiredCount: 3,
  cpu: 512,
  memoryLimitMiB: 1024,
  taskImageOptions: {
    image,
    containerName: 'api',
    containerPort: 8080,
    family: appName,
    logDriver: AwsLogDriver.awsLogs({
      logGroup: appLogGroup,
      streamPrefix: 'service',
    }),
  },
  secrets: {
    SPRING_DATASOURCE_USERNAME: Secret.fromSecretsManager(dbSecret, 'username'),
    SPRING_DATASOURCE_PASSWORD: Secret.fromSecretsManager(dbSecret, 'password'),
  },
  environment: {
    SPRING_DATASOURCE_URL: `jdbc:mysql://${db.clusterEndpoint.hostname}:${db.clusterEndpoint.port}/${dbName}`,
  },
  deregistrationDelay: Duration.seconds(5),
  responseTimeAlarmThreshold: Duration.seconds(3),
  healthCheck: {
    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,
  }],
});
```

```
service.service.connections.allowTo(db, 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` :

```
<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` :

```
name: Fruit API
spring:
  main:
    banner-mode: "off"

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


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
```

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.

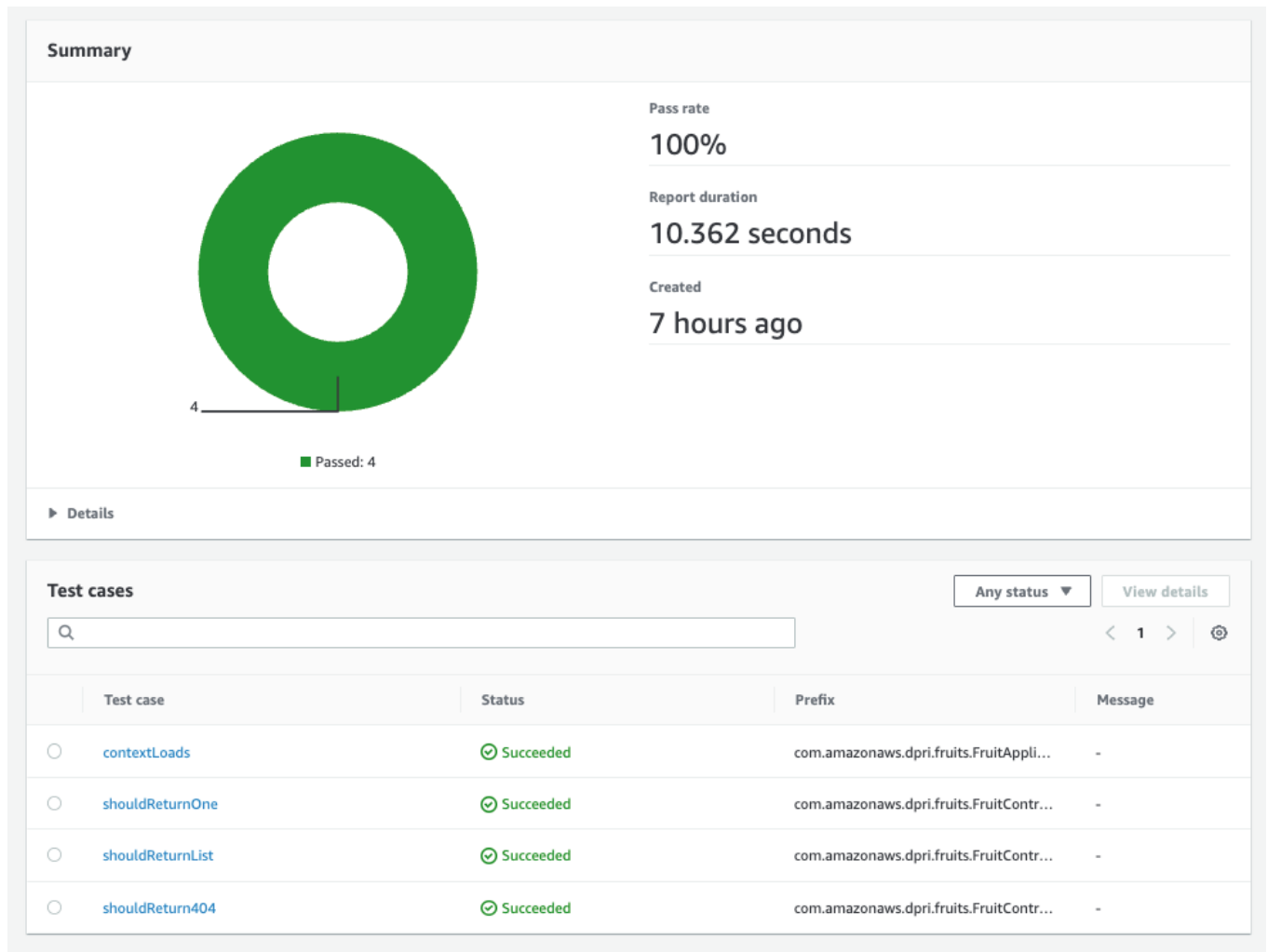
Build Code

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.



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: source.codePipelineSource,
  reviewRequired: false,
  filter: CodeGuruReviewFilter.defaultCodeSecurityFilter(),
});
const codeGuruQuality = new CodeGuruReviewCheck('CodeGuruQuality', {
  source: source.codePipelineSource,
  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 than 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.

```

const appName = 'fruit-api';
app = new App({ context: { appName } });
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}/AuroraCluster/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' },
], 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',
  },
], true);

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

```

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

**Disable transition****UpdatePipeline**

Succeeded

Pipeline execution ID: [c3ae9e0a-99f1-44ae-a5cf-e77268d8aad8](#)**SelfMutate**

AWS CodeBuild

Succeeded - 6 hours ago[Details](#)

5e576ebc fruit-api: init

**Disable transition****Assets**

Succeeded

Pipeline execution ID: [c3ae9e0a-99f1-44ae-a5cf-e77268d8aad8](#)**DockerAsset1**

AWS CodeBuild

Succeeded - 6 hours ago[Details](#)

Software Composition Analysis (SCA)

Trivy is used to scan the source for vulnerabilities in its dependencies. The `pom.xml` and `Dockerfile` files are scanned for configuration issues or vulnerabilities in any dependencies. The scanning is accomplished by a CDK construct that creates a CodeBuild job to run `trivy`:

```
import { TrivyScan } from './trivy-scan';

...

const trivyScan = new TrivyScan('TrivyScan', {
  source: source.codePipelineSource,
  severity: ['CRITICAL', 'HIGH'],
  checks: ['vuln', 'config', 'secret'],
});
```

Trivy is also used within the `Dockerfile` to scan the image after it is built. The `docker build` will fail if Trivy finds any vulnerabilities in the final image:

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:17-al2022-jdk as build
USER nobody
WORKDIR /app
COPY target/fruit-api.jar /app
ENTRYPOINT ["java", "-jar", "/app/fruit-api.jar"]

# Use multi-stage builds to scan newly created image with Trivy. This second stage 'vulnscan'
# isn't published to Amazon ECR and is never run. It is only used to run the Trivy scan
# against the newly created image in the 'build' stage.
#
# This stage must run as root so Trivy can scan all files in the image, not just
# those accessible by the nobody user. The user is switched back to 'nobody' at
# the end to ensure that even if this image is used for something it is done
# without the 'root' user.

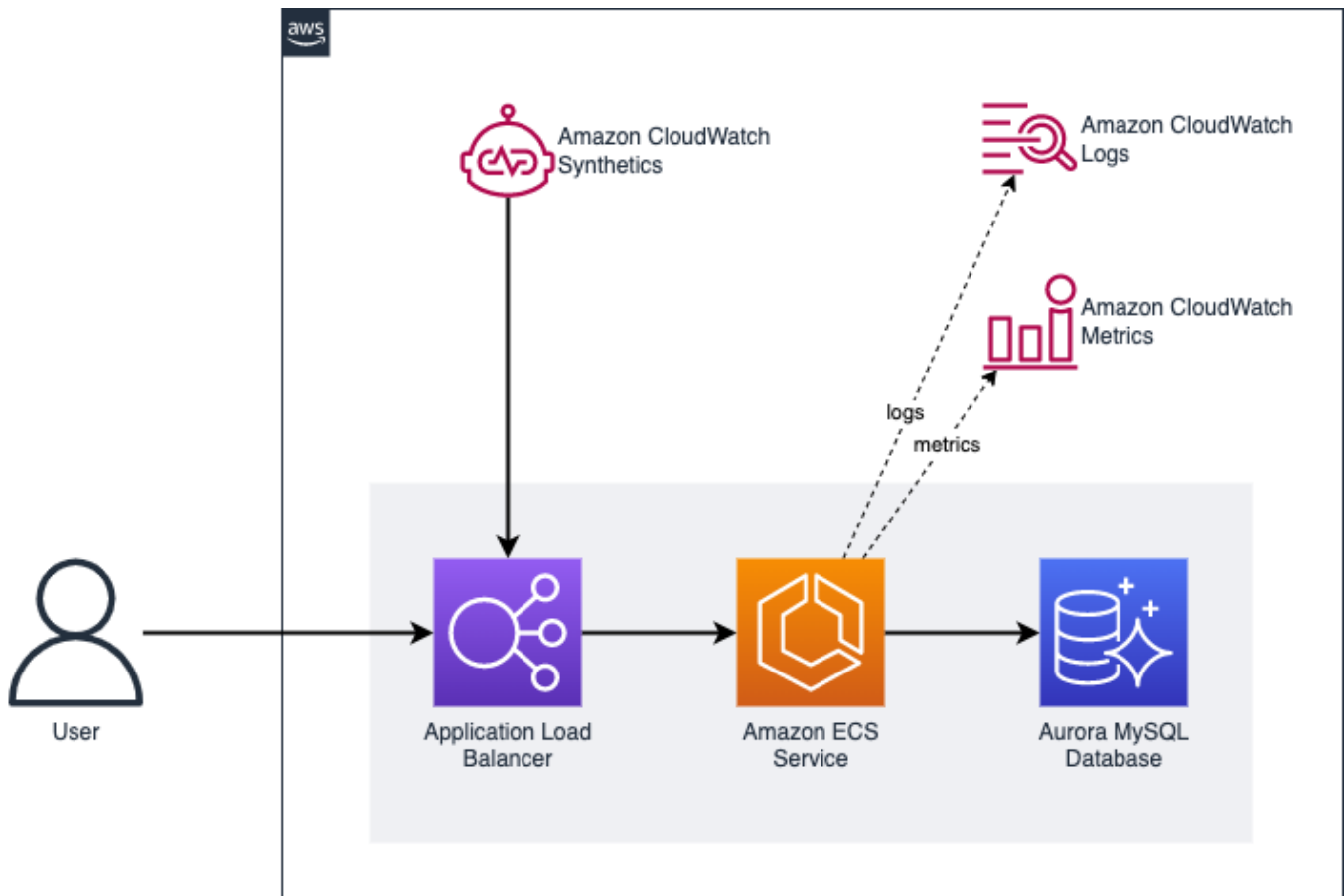
FROM build AS vulnscan
USER root
COPY --from=aquasec/trivy:latest /usr/local/bin/trivy /usr/local/bin/trivy
RUN trivy filesystem --exit-code 1 --no-progress --ignore-unfixed -s HIGH,CRITICAL /
USER nobody
```

Software Bill of Materials (SBOM)

Trivy generates an SBOM in the form of a CycloneDX JSON report. The SBOM is saved as a CodePipeline asset. Trivy supports additional SBOM formats such as SPDX, and SARIF.

Test (Beta)

Launch Environment



The infrastructure for each environment is defined in 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 ServerlessCluster(this, 'AuroraCluster', {
  engine: DatabaseClusterEngine.AURORA_MYSQL,
  vpc,
  credentials: Credentials.fromSecret(dbSecret),
  defaultDatabaseName: dbName,
  deletionProtection: false,
  clusterIdentifier: appName,
});

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 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}`,
    },
},
deregistrationDelay: Duration.seconds(5),
responseTimeAlarmThreshold: Duration.seconds(3),
healthCheck: {
    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,
}],
});

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

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

```

The `DeploymentStack` construct is then instantiated for each environment:

```

export const Beta: EnvironmentConfig = {
    name: 'Beta',
    account: accounts.beta,
    waves: [
        ['us-west-2'],
    ],
};

...

new PipelineEnvironment(pipeline, Beta, (deployment, stage) => {
    stage.addPost(
        new SoapUITest('E2E Test', {
            source: source.codePipelineSource,
            endpoint: deployment.apiUrl,
            cacheBucket,
        }),
    );
});

```

Database Deploy

Spring Boot is configured to run Liquibase on startup. This reads the configuration in `src/main/resources/db/changelog/db.changelog-master.yml` to define the tables and initial data for the database:

```
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
```

Deploy Software

The *Launch Environment* action above creates a new Amazon ECS Task Definition for the new docker image and then updates the Amazon ECS Service to use the new Task Definition.

Integration Tests

Integration tests are preformed during the *Build Source* action. They are defined with SoapUI in `fruit-api-soapui-project.xml`. They are executed by Maven in the `integration-test` phase using plugins in `pom.xml`. Spring Boot is configure to start a local instance of the application with an H2 database during the `pre-integration-test` phase and then to terminate on the `post-integration-test` phase. The results of the unit tests are uploaded to AWS Code Build Test Reports to track over time.

```
<plugins>
  <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <executions>
      <execution>
        <id>pre-integration-test</id>
        <goals>
          <goal>start</goal>
        </goals>
      </execution>
      <execution>
        <id>post-integration-test</id>
        <goals>
          <goal>stop</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  <plugin>
    <groupId>com.smartbear.soapui</groupId>
    <artifactId>soapui-maven-plugin</artifactId>
    <version>5.7.0</version>
    <configuration>
      <junitReport>true</junitReport>
      <outputFolder>target/soapui-reports</outputFolder>
      <endpoint>${soapui.endpoint}</endpoint>
    </configuration>
    <executions>
      <execution>
        <phase>integration-test</phase>
        <goals>
          <goal>test</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
```

Acceptance Tests

Acceptance tests are performed after the *Launch Environment* and *Deploy Software* actions:

Beta Pending
Pipeline execution ID: [c3ae9e0a-99f1-44ae-a5cf-e77268d8aad8](#)

fruit-api.Prepare
AWS CloudFormation [Details](#)
 Succeeded - 6 hours ago
[Details](#)

↓

fruit-api.Deploy
AWS CloudFormation [Details](#)
 Succeeded - 6 hours ago
[Details](#)

↓

E2E_Test AWS CodeBuild Succeeded - 6 hours ago Details	PromoteFromBeta Manual approval Waiting for approval - Review
--	---

5e576ebc fruit-api: init

The tests are defined with SoapUI in `fruit-api-soapui-project.xml`. They are executed by Maven with the endpoint overridden to the URL from the CloudFormation output. A CDK construct called `SoapUITest` was created to create the CodeBuild Project to run SoapUI.

```
const stepProps = {
  envFromCfnOutputs: {
    ENDPOINT: props.endpoint,
  },
  input: props.source,
  commands: [],
  buildEnvironment: {
    buildImage: LinuxBuildImage.STANDARD_6_0,
  },
  partialBuildSpec: BuildSpec.fromObject({
    env: {
      variables: {
```

```

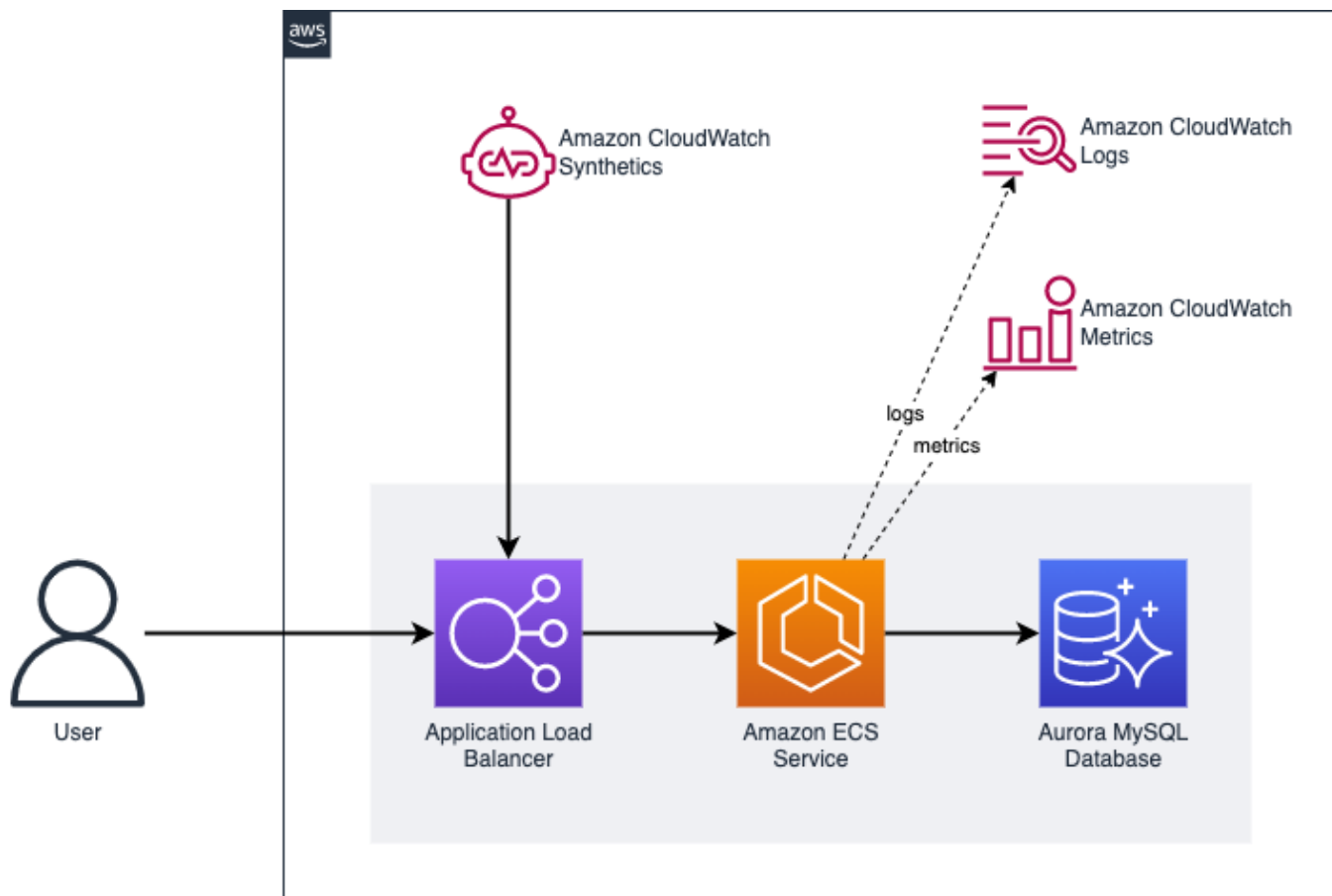
    MAVEN_OPTS: props.mavenOpts || '-XX:+TieredCompilation -XX:TieredStopAtLevel=1',
    MAVEN_ARGS: props.mavenArgs || '--batch-mode --no-transfer-progress',
  },
  phases: {
    install: {
      'runtime-versions': {
        java: (props.javaRuntime || 'corretto17'),
      },
    },
    build: {
      commands: ['mvn ${MAVEN_ARGS} soapui:test -Dsoapui.endpoint=${ENDPOINT}'],
    },
  },
  cache: props.cacheBucket ? {
    paths: ['/root/.m2/**/*'],
  } : undefined,
  reports: {
    e2e: {
      'files': ['target/soapui-reports/*.xml'],
      'file-format': 'JUNITXML',
    },
  },
  version: '0.2',
}),
cache: props.cacheBucket ? Cache.bucket(props.cacheBucket) : undefined,
};
super(id, stepProps);

```

The results of the unit tests are uploaded to AWS Code Build Test Reports to track over time.

Test (Gamma)

Launch Environment



The infrastructure for each environment is defined in 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 ServerlessCluster(this, 'AuroraCluster', {
  engine: DatabaseClusterEngine.AURORA_MYSQL,
  vpc,
  credentials: Credentials.fromSecret(dbSecret),
  defaultDatabaseName: dbName,
  deletionProtection: false,
  clusterIdentifier: appName,
});

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 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}/${db.dbName}`,
    },
},
deregistrationDelay: Duration.seconds(5),
responseTimeAlarmThreshold: Duration.seconds(3),
healthCheck: {
    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,
}],
});

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

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

```

The `DeploymentStack` construct is then instantiated for each environment:

```

export const Gamma: EnvironmentConfig = {
    name: 'Gamma',
    account: accounts.gamma,
    waves: [
        ['us-west-2', 'us-east-1'],
    ],
};

...

new PipelineEnvironment(pipeline, Gamma, (deployment, stage) => {
    stage.addPost(
        new JMeterTest('Performance Test', {
            source: source.codePipelineSource,
            endpoint: deployment.apiUrl,
            threads: 300,
            duration: 300,
            throughput: 6000,
            cacheBucket,
        }),
    );
}, wave => {
    wave.addPost(
        new ManualApprovalStep('PromoteToProd'),
    );
});

```

Database Deploy

Spring Boot is configured to run Liquibase on startup. This reads the configuration in `src/main/resources/db/changelog/db.changelog-master.yml` to define the tables and initial data for the database:

```
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
```

Deploy Software

The *Launch Environment* action above creates a new Amazon ECS Task Definition for the new docker image and then updates the Amazon ECS Service to use the new Task Definition.

Application Monitoring & Logging

Amazon ECS uses Amazon CloudWatch Metrics and Amazon CloudWatch Logs for observability by default.

Synthetic Tests

Amazon CloudWatch Synthetics is used to continuously deliver traffic to the application and assert that requests are successful and responses are received within a given threshold. The canary is defined via CDK using the `@cdklabs/cdk-ecs-codedeploy` construct:

```
const service = new ApplicationLoadBalancedCodeDeployedFargateService(this, 'Api', {
  ...

  apiCanaryTimeout: Duration.seconds(5),
  apiTestSteps: [{
    name: 'getAll',
    path: '/api/fruits',
    jmesPath: 'length(@)',
    expectedValue: 5,
  }],
});
```

Performance Tests

Apache JMeter is used to run performance tests against the deployed application. The tests are stored in `src/test/jmeter` and added to the pipeline via CDK:

```
import { JMeterTest } from './jmeter-test';

...

new JMeterTest('Performance Test', {
  source: source.codePipelineSource,
  endpoint: deployment.apiUrl,
  threads: 300,
  duration: 300,
  throughput: 6000,
  cacheBucket,
});
```

Resilience Tests

Not Implemented

Dynamic Application Security Testing (DAST)

Not Implemented

Prod**Manual Approval**

A manual approval step is added to the end of the `Gamma` stage. The step is added at the end to keep the environment in a stable state while manual testing is performed. Once the step is approved, the pipeline continues execution to the next stage.

```
new PipelineEnvironment(pipeline, Gamma, (deployment, stage) => {
  stage.addPost(
    new JMeterTest('Performance Test', {
      source: source.codePipelineSource,
      endpoint: deployment.apiUrl,
      threads: 300,
      duration: 300,
      throughput: 6000,
      cacheBucket,
    }),
    new ManualApprovalStep('PromoteFromGamma'),
  );
});
```

When a manual approval step is used, IAM permissions should be used to restrict which principals can approve actions and stages to enforce least privilege.

```
{
  "Effect": "Allow",
  "Action": [
    "codepipeline:PutApprovalResult"
  ],
  "Resource": "arn:aws:codepipeline:us-east-2:80398EXAMPLE:MyFirstPipeline/MyApprovalStage/MyApprovalAction"
}
```

Database Deploy

Spring Boot is configured to run Liquibase on startup. This reads the configuration in `src/main/resources/db/changelog/db.changelog-master.yml` to define the tables and initial data for the database:

```
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

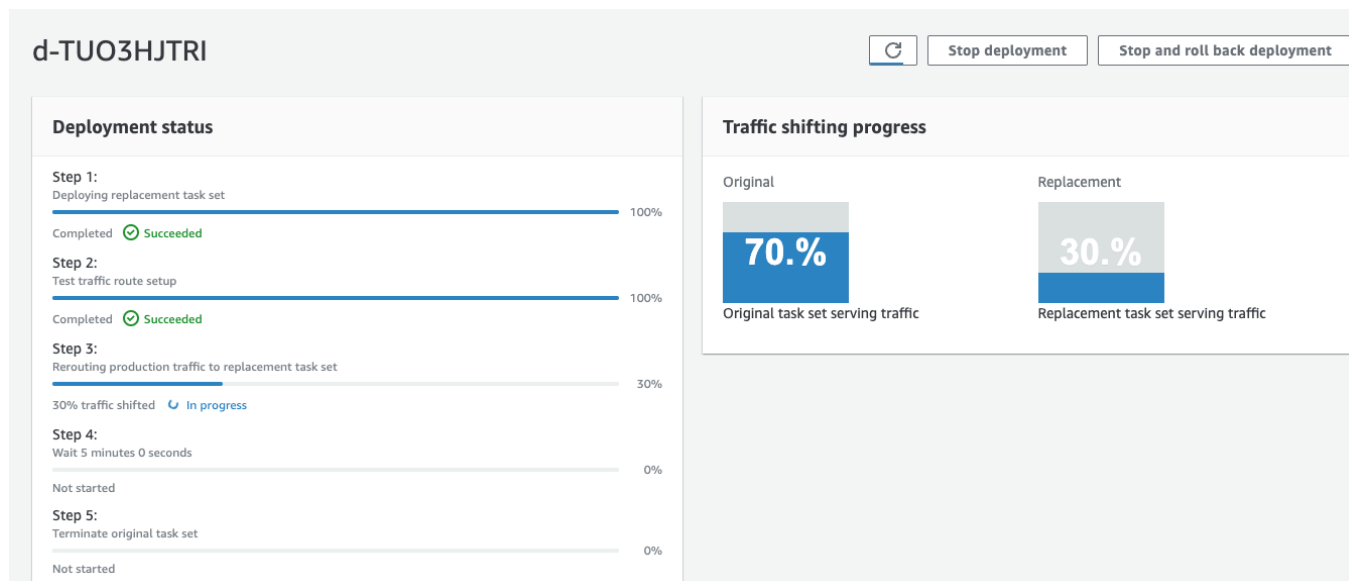
  - insert:
    tableName: fruit
    columns:
    - column:
      name: name
      value: Banana

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

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

Progressive Deployment

Progressive deployment is implemented with AWS CodeDeploy for ECS. CodeDeploy performs a linear blue/green by deploying the new task definition as a new task with a separate target group and then shifting 10% of traffic every minute until all traffic is shifted. A CloudWatch alarm is monitored by CodeDeploy and an automatic rollback is triggered if the alarm exceeds the threshold.



Implementation of this type deployment presents challenges due to the following limitations:

- [aws/aws-cdk #19163](#) - CDK Pipelines aren't intended to be used with CodeDeploy actions.
- [AWS CloudFormation User Guide](#) - The use of `AWS::CodeDeploy::BlueGreen` hooks and `AWS::CodeDeployBlueGreen` restricts the types of changes that can be made. Additionally, you can't use auto-rollback capabilities of CodeDeploy.
- [aws/aws-cdk #5170](#) - CDK doesn't support defining CloudFormation rollback triggers. This rules out CloudFormation based blue/green deployments.

The solution was to use the `@cdklabs/cdk-ecs-codedeploy` construct from the Construct Hub which addresses [aws/aws-cdk #1559](#) - Lack of support for Blue/Green ECS Deployment in CDK.

```
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}`,
  },
},
{
  deregistrationDelay: Duration.seconds(5),
  responseTimeAlarmThreshold: Duration.seconds(3),
  healthCheck: {
    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,
}],
});

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

Deployments are made incrementally across regions using the CDK Pipeline - Wave construct. Each wave contains a list of regions to deploy to in parallel. One wave must fully complete before the next wave starts. The diagram below shows how each wave deploys to 2 regions at a time.

Prod-0 SucceededPipeline execution ID: [c7074e15-5228-40ed-b157-7c70a8b4fabcd](#)


Prod-us-west-2.fruit-api.Prepare

 [AWS CloudFormation](#) Succeeded - 14 hours ago[Details](#)

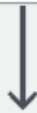
Prod-us-east-1.fruit-api.Prepare

 [AWS CloudFormation](#) Succeeded - 14 hours ago[Details](#)

Prod-us-west-2.fruit-api.Deploy

 [AWS CloudFormation](#) Succeeded - 13 hours ago[Details](#)

Prod-us-east-1.fruit-api.Deploy

 [AWS CloudFormation](#) Succeeded - 13 hours ago[Details](#)[6bc1fd38](#) fruit-api: lint

Disable transition

Prod-1 SucceededPipeline execution ID: [c7074e15-5228-40ed-b157-7c70a8b4fabcd](#)


Prod-eu-central-1.fruit-api.Prepare

 [AWS CloudFormation](#) Succeeded - 13 hours ago[Details](#)

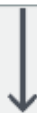
Prod-eu-west-1.fruit-api.Prepare

 [AWS CloudFormation](#) Succeeded - 13 hours ago[Details](#)

Prod-eu-central-1.fruit-api.Deploy

 [AWS CloudFormation](#) Succeeded - 13 hours ago[Details](#)

Prod-eu-west-1.fruit-api.Deploy

 [AWS CloudFormation](#) Succeeded - 13 hours ago[Details](#)[6bc1fd38](#) fruit-api: lint

Disable transition

Prod-2 SucceededPipeline execution ID: [c7074e15-5228-40ed-b157-7c70a8b4fabcd](#)

Prod-ap-south-1.fruit-api.Prepare

 [AWS CloudFormation](#) Succeeded - 13 hours ago[Details](#)

Prod-ap-southeast-2.fruit-api.Prepare

 [AWS CloudFormation](#) Succeeded - 13 hours ago[Details](#)

Prod-ap-south-1.fruit-api.Deploy



Prod-ap-southeast-2.fruit-api.Deploy



Environments are configured in CDK with the list of waves:

```
// BETA environment is 1 wave with 1 region
export const Beta: EnvironmentConfig = {
  name: 'Beta',
  account: accounts.beta,
  waves: [
    ['us-west-2'],
  ],
};

// GAMMA environment is 1 wave with 2 regions
export const Gamma: EnvironmentConfig = {
  name: 'Gamma',
  account: accounts.gamma,
  waves: [
    ['us-west-2', 'us-east-1'],
  ],
};

// PROD environment is 3 wave with 2 regions each wave
export const Prod: EnvironmentConfig = {
  name: 'Prod',
  account: accounts.production,
  waves: [
    ['us-west-2', 'us-east-1'],
    ['eu-central-1', 'eu-west-1'],
    ['ap-south-1', 'ap-southeast-2'],
  ],
};
```

A `PipelineEnvironment` class is responsible for loading the `EnvironmentConfig` into CodePipeline stages:

```
new PipelineEnvironment(pipeline, Beta, (deployment, stage) => {
  stage.addPost(
    new SoapUITest('E2E Test', {
      source: source.codePipelineSource,
      endpoint: deployment.apiUrl,
      cacheBucket,
    }),
  );
});

...

new PipelineEnvironment(pipeline, Gamma, (deployment, stage) => {
  stage.addPost(
    new JMeterTest('Performance Test', {
      source: source.codePipelineSource,
      endpoint: deployment.apiUrl,
      threads: 300,
      duration: 300,
      throughput: 6000,
      cacheBucket,
    }),
  );
}, wave => {
  wave.addPost(
    new ManualApprovalStep('PromoteToProd'),
  );
});

...

class PipelineEnvironment {
  constructor(
    pipeline: CodePipeline,
    environment: EnvironmentConfig,
    stagePostProcessor?: PipelineEnvironmentStageProcessor,
    wavePostProcessor?: PipelineEnvironmentWaveProcessor) {
    if (!environment.account?.accountId) {
      throw new Error(`Missing accountId for environment '${environment.name}'. Do you need to update '.accounts.env'?`);
    }
    for (const [i, regions] of environment.waves.entries()) {
      const wave = pipeline.addWave(`${environment.name}-${i}`);
      for (const region of regions) {
        const deployment = new Deployment(pipeline, environment.name, {
          account: environment.account!.accountId!,
          region,
        });
        const stage = wave.addStage(deployment);
        if (stagePostProcessor) {
          stagePostProcessor(deployment, stage);
        }
      }
      if (wavePostProcessor) {
        wavePostProcessor(wave);
      }
    }
  }
}
```

Synthetic Tests

Amazon CloudWatch Synthetics is used to continuously deliver traffic to the application and assert that requests are successful and responses are received within a given threshold. The canary is defined via CDK using the `@cdklabs/cdk-ecs-codedeploy` construct:

```
const service = new ApplicationLoadBalancedCodeDeployedFargateService(this, 'Api', {
  ...

  apiCanaryTimeout: Duration.seconds(5),
  apiTestSteps: [{
    name: 'getAll',
    path: '/api/fruits',
    jmesPath: 'length(@)',
    expectedValue: 5,
  }],
});
```

Frequently Asked Questions

What operating models does this reference implementation support?

This reference implementation can accommodate any operation model with minor updates:

- **Fully Separated** - Restrict the role that CDK uses for CloudFormation execution to only create resources from approved product portfolios in AWS Service Catalog. Ownership of creating the products in Service Catalog is owned by the **Platform Engineering** team and operational support of Service Catalog is owned by the **Platform Operations** team. The **Platform Engineering** team should publish CDK constructs internally that provision AWS resources through Service Catalog. Update the CDK app in the `infrastructure/` directory to use CDK constructs provided by the `Platform Engineering` team. Use a CODEOWNERS file to require all changes to the `infrastructure/` directory be approved by the **Application Operations** team. Additionally, restrict permissions to the **Manual Approval** action to only allow members of the **Application Operations** to approve.
- **Separated AEO and IEO with Centralized Governance** - Restrict the role that CDK uses for CloudFormation execution to only create resources from approved product portfolios in AWS Service Catalog. Ownership of creating the products in Service Catalog is owned by the **Platform Engineering** team and operational support of Service Catalog is owned by the **Platform Engineering** team. The **Platform Engineering** team should publish CDK constructs internally that provision AWS resources through Service Catalog. Update the CDK app in the `infrastructure/` directory to use CDK constructs provided by the `Platform Engineering` team.
- **Separated AEO and IEO with Decentralized Governance** - The **Platform Engineering** team should publish CDK constructs internally that provision AWS resources in manner that achieve organizational compliance. Update the CDK app in the `infrastructure/` directory to use CDK constructs provided by the `Platform Engineering` team.

Where is manual testing performed in this pipeline?

Ideally, all testing is accomplished through automation in the **Integration Tests** and **Acceptance Tests** actions. If an organization relies on people manually executing tests then these tests would be performed in the **Gamma Stage**. The **Manual Approval** action would be required and the approval would be granted by a Quality Assurance team member once the manual testing completes successfully.

Amazon CodeCatalyst Pipeline

This presents a reference implementation of the Application Pipeline reference architecture. The pipeline is created with Amazon CodeCatalyst for building the software and performing testing tasks. All the infrastructure for this reference implementation is defined with AWS Cloud Development Kit. The source code for this reference implementation is available in GitHub for review.

This reference implementation has been contributed to Amazon CodeCatalyst as blueprint named `devOps deployment pipeline`. You can try out this reference implementation in your own CodeCatalyst space by creating a new project from the blueprint.



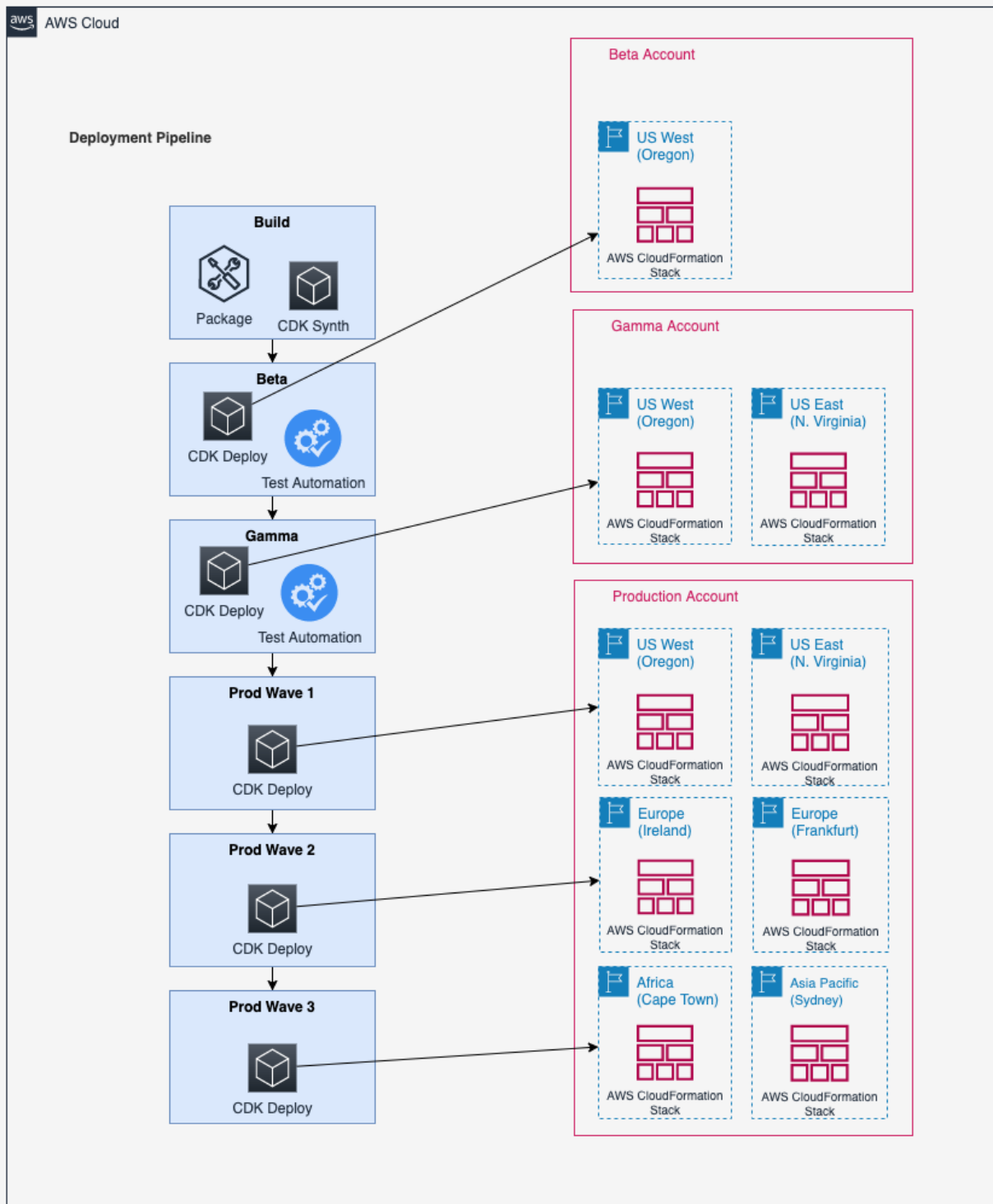
Text is not SVG - cannot display

Disclaimer

This reference implementation is intended to serve as an example of how to accomplish the guidance in the reference architecture using Amazon CodeCatalyst. The reference implementation has intentionally not followed 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:

❗ **cdk bootstrap with AdministratorAccess** - the default policy used for `cdk bootstrap` is `AdministratorAccess` but should be replaced with a more appropriate policy with least privilege in your account.

❗ **LS 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.

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

    FruitController(final FruitRepository r) {
        this.repository = r;
    }

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

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

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

    @PutMapping("/api/fruits/{id}")
    FruitDTO replaceFruit(
        @RequestBody final FruitDTO newFruit,
        @PathVariable final Long id) {
        Fruit entity = repository.findById(id)
            .map(fruit -> {
                fruit.setName(newFruit.getName());
                return repository.save(fruit);
            })
            .orElseGet(() -> {
                newFruit.setId(id);
                return repository.save(convertToEntity(newFruit));
            });
        return convertToDto(entity);
    }

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

    FruitDTO convertToDto(final Fruit fruit) {
        FruitDTO dto = new FruitDTO();
        dto.setId(fruit.getId());
        dto.setName(fruit.getName());
        return dto;
    }

    Fruit convertToEntity(final FruitDTO fruit) {
        Fruit entity = new Fruit();
        entity.setId(fruit.getId());
        entity.setName(fruit.getName());
        return entity;
    }
}
```

The application source code is stored in Amazon CodeCatalyst repository that is created and initialized from the blueprint.

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, `FruitControllerTest.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"), new Fruit("Dragonfruit")));

    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 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 ServerlessCluster(this, 'AuroraCluster', {
  engine: DatabaseClusterEngine.AURORA_MYSQL,
  vpc,
  credentials: Credentials.fromSecret(dbSecret),
  defaultDatabaseName: dbName,
  deletionProtection: false,
  clusterIdentifier: appName,
});

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/${appName}`,
  removalPolicy: RemovalPolicy.DESTROY,
});
let deploymentConfig: IEcsDeploymentConfig | undefined = undefined;
if (props?.deploymentConfigName) {
  deploymentConfig = EcsDeploymentConfig.fromEcsDeploymentConfigName(this, 'DeploymentConfig', props.deploymentConfigName);
}
const service = new ApplicationLoadBalancedCodeDeployedFargateService(this, 'Api', {
  cluster,
  capacityProviderStrategies: [
    {
      capacityProvider: 'FARGATE_SPOT',
      weight: 1,
    },
  ],
  minHealthyPercent: 50,
  maxHealthyPercent: 200,
  desiredCount: 3,
  cpu: 512,
  memoryLimitMiB: 1024,
  taskImageOptions: {
    image,
    containerName: 'api',
    containerPort: 8080,
    family: appName,
    logDriver: AwsLogDriver.awsLogs({
      logGroup: appLogGroup,
      streamPrefix: 'service',
    }),
  },
  secrets: {
    SPRING_DATASOURCE_USERNAME: Secret.fromSecretsManager(dbSecret, 'username'),
    SPRING_DATASOURCE_PASSWORD: Secret.fromSecretsManager(dbSecret, 'password'),
  },
  environment: {
    SPRING_DATASOURCE_URL: `jdbc:mysql://${db.clusterEndpoint.hostname}:${db.clusterEndpoint.port}/${dbName}`,
  },
},
  deregistrationDelay: Duration.seconds(5),
  responseTimeAlarmThreshold: Duration.seconds(3),
  healthCheck: {
    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,
  }],
});
```

```
service.service.connections.allowTo(db, 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` :

```
<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` :

```
name: Fruit API
spring:
  main:
    banner-mode: "off"

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

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
```

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. An action is added to the workflow to build and package the source code:

```
Package:
  Identifier: aws/build@v1
Inputs:
  Sources:
    - WorkflowSource
Outputs:
  AutoDiscoverReports:
    Enabled: true
    ReportNamePrefix: build
    SuccessCriteria:
      PassRate: 100
  Artifacts:
    - Name: package
      Files:
        - "**/*.*"
Configuration:
  Steps:
    - Run: mvn verify --batch-mode --no-transfer-progress
```

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.

▼ Report run overview

Report status: ✔ Succeeded
The pass rate meets the minimum criteria.

Pass rate

100%

Above 100% minimum

Test case summary



3 succeeded 0 failed 0 skipped

Duration (hh:mm:ss)

00:00:01

Results

Raw files

Test suites

Display

Status and duration (hh:mm:ss)

▼ All test suites

✔ com.amazonaws.dpri.fr... 00:00:01

Test cases (3) Hint

Filter test cases by name, status, or message

Name	Status	Message	Duration (ss.mmm)
shouldReturn404	✔ Succeeded		00.127
shouldReturnList	✔ Succeeded		00.024
shouldReturnOne	✔ Succeeded		00.070

Code Quality

Code quality is enforced through the PMD and Checkstyle Maven plugins:

```
<plugin>
  <artifactId>maven-pmd-plugin</artifactId>
  <configuration>
    <printFailingErrors></printFailingErrors>
  </configuration>
  <executions>
    <execution>
      <phase>test</phase>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
<plugin>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <configuration>
    <printFailingErrors></printFailingErrors>
  </configuration>
  <executions>
    <execution>
      <phase>test</phase>
      <goals>
        <goal>check</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Additionally, cdk-nag is run against 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.

```
const appName = 'fruit-api';
app = new App({ context: { appName } });
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}/AuroraCluster/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' },
], 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',
  },
], true);

```

Secrets Detection

Trivy is used to scan the source for secrets. The Trivy GitHub Action is used in the Amazon CodeCatalyst workflow to perform the scan:

```

SCA:
  Identifier: aws/github-actions-runner@v1
  Inputs:
    Sources:
      - WorkflowSource
  Configuration:
    Steps:
      - name: Trivy Vulnerability Scanner
        uses: aquasecurity/trivy-action@master
        with:
          scan-type: fs
          ignore-unfixed: true
          format: cyclonedx
          output: sbom.json
          severity: CRITICAL,HIGH
          security-checks: vuln,config,secret

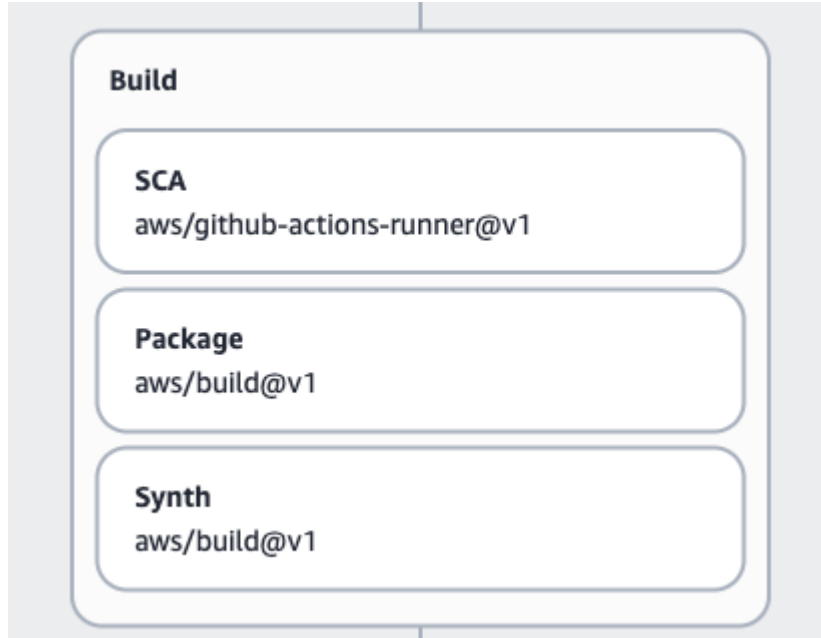
```

Static Application Security Testing (SAST)

The SpotBugs Maven plugin is used along with the Find Security Bugs plugin to identify OWASP Top 10 and CWE vulnerabilities in the application source code.

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.



Software Composition Analysis (SCA)

Trivy is used to scan the source for vulnerabilities in its dependencies. The `pom.xml` and `Dockerfile` files are scanned for configuration issues or vulnerabilities in any dependencies. The scanning is accomplished by a CDK construct that creates a CodeBuild job to run `trivy`:

```
SCA:
  Identifier: aws/github-actions-runner@v1
  Inputs:
    Sources:
      - WorkflowSource
  Configuration:
    Steps:
      - name: Trivy Vulnerability Scanner
        uses: aquasecurity/trivy-action@master
        with:
          scan-type: fs
          ignore-unfixed: true
          format: cyclonedx
          output: sbom.json
          severity: CRITICAL,HIGH
          security-checks: vuln,config,secret
```

Trivy is also used within the `Dockerfile` to scan the image after it is built. The `docker build` will fail if Trivy finds any vulnerabilities in the final image:

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:17-al2022-jdk as build
USER nobody
WORKDIR /app
COPY target/fruit-api.jar /app
ENTRYPOINT ["java","-jar","/app/fruit-api.jar"]

# Use multi-stage builds to scan newly created image with Trivy. This second stage 'vulnscan'
# isn't published to Amazon ECR and is never run. It is only used to run the Trivy scan
# against the newly created image in the 'build' stage.
#
# This stage must run as root so Trivy can scan all files in the image, not just
# those accessible by the nobody user. The user is switched back to 'nobody' at
# the end to ensure that even if this image is used for something it is done
# without the 'root' user.

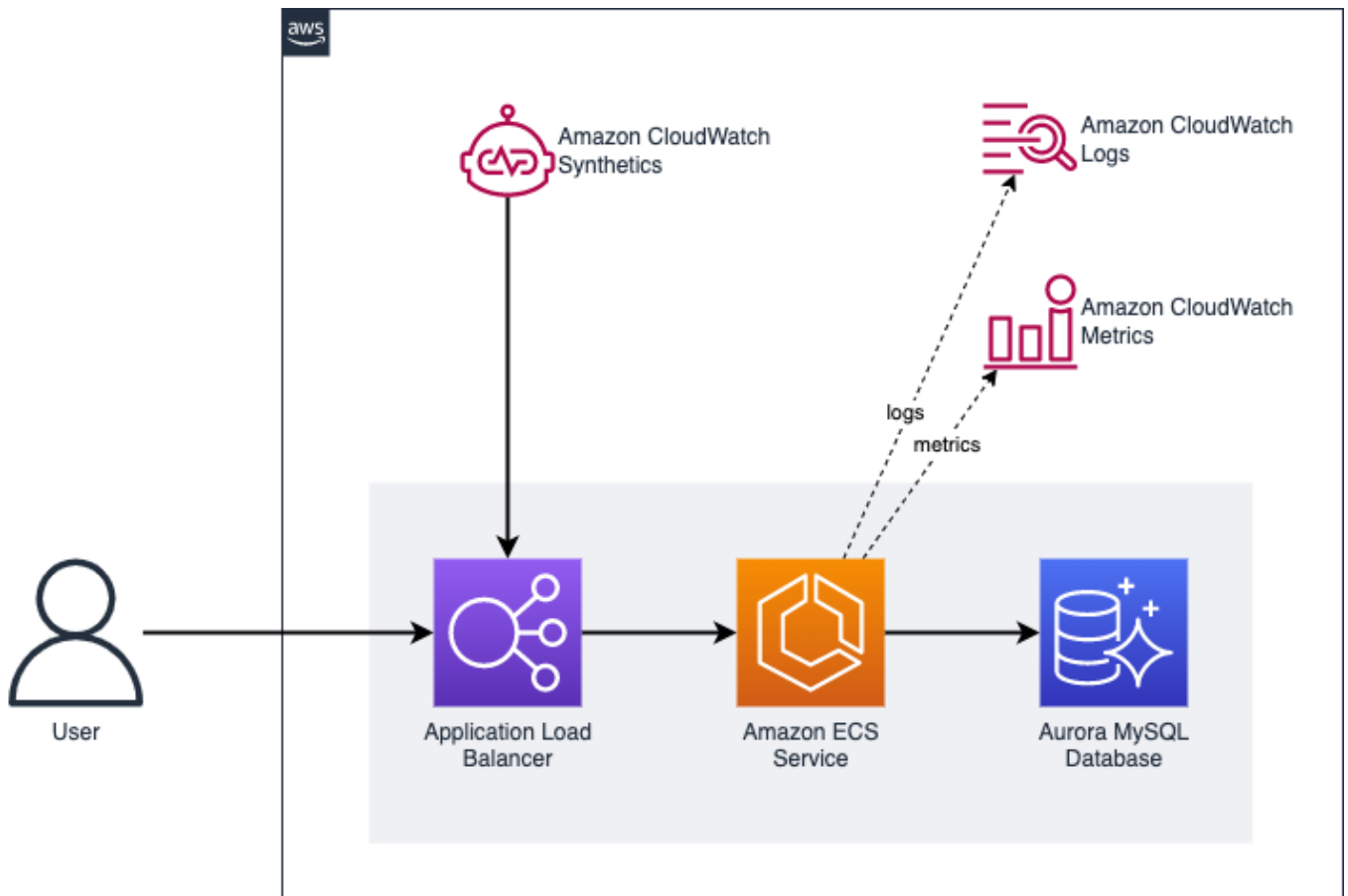
FROM build AS vulnscan
USER root
COPY --from=aquasec/trivy:latest /usr/local/bin/trivy /usr/local/bin/trivy
RUN trivy filesystem --exit-code 1 --no-progress --ignore-unfixed -s HIGH,CRITICAL /
USER nobody
```

Software Bill of Materials (SBOM)

Trivy generates an SBOM in the form of a CycloneDX JSON report. The SBOM is saved as a CodeCatalyst asset. Trivy supports additional SBOM formats such as SPDX, and SARIF.

Test (Beta)

Launch Environment



The infrastructure for each environment is defined in 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 ServerlessCluster(this, 'AuroraCluster', {
  engine: DatabaseClusterEngine.AURORA_MYSQL,
  vpc,
  credentials: Credentials.fromSecret(dbSecret),
  defaultDatabaseName: dbName,
  deletionProtection: false,
  clusterIdentifier: appName,
});

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 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}`,
    },
},
deregistrationDelay: Duration.seconds(5),
responseTimeAlarmThreshold: Duration.seconds(3),
healthCheck: {
    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,
}],
});

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

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

```

The CDK deployment is then performed for each environment:

```

Deploy:
  Identifier: aws/cdk-deploy@v1
  DependsOn:
    - Build
  Inputs:
    Artifacts:
      - package
  Environment:
    Name: Beta
    Connections:
      - Name: beta
        Role: codecatalyst
  Configuration:
    StackName: fruit-api
    Region: us-west-2
    Context: '{"deploymentConfigurationName":"CodeDeployDefault.ECSCanary10Percent5Minutes"}'
    CfnOutputVariables: ['"endpointUrl"']

```

Database Deploy

Spring Boot is configured to run Liquibase on startup. This reads the configuration in `src/main/resources/db/changelog/db.changelog-master.yml` to define the tables and initial data for the database:

```
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
```

Deploy Software

The *Launch Environment* action above creates a new Amazon ECS Task Definition for the new docker image and then updates the Amazon ECS Service to use the new Task Definition.

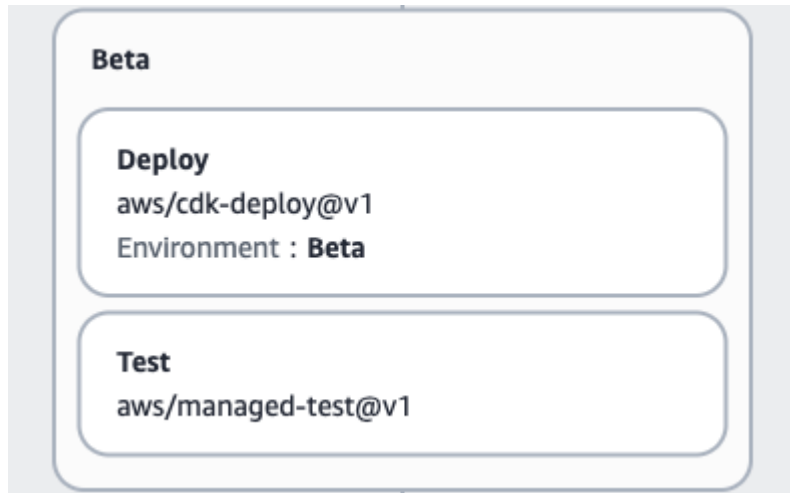
Integration Tests

Integration tests are preformed during the *Build Source* action. They are defined with SoapUI in `fruit-api-soapui-project.xml`. They are executed by Maven in the `integration-test` phase using plugins in `pom.xml`. Spring Boot is configure to start a local instance of the application with an H2 database during the `pre-integration-test` phase and then to terminate on the `post-integration-test` phase. The results of the unit tests are uploaded to Amazon CodeCatalyst to track over time.

```
<plugins>
  <plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <executions>
      <execution>
        <id>pre-integration-test</id>
        <goals>
          <goal>start</goal>
        </goals>
      </execution>
      <execution>
        <id>post-integration-test</id>
        <goals>
          <goal>stop</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  <plugin>
    <groupId>com.smartbear.soapui</groupId>
    <artifactId>soapui-maven-plugin</artifactId>
    <version>5.7.0</version>
    <configuration>
      <junitReport>true</junitReport>
      <outputFolder>target/soapui-reports</outputFolder>
      <endpoint>${soapui.endpoint}</endpoint>
    </configuration>
    <executions>
      <execution>
        <phase>integration-test</phase>
        <goals>
          <goal>test</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
```

Acceptance Tests

Acceptance tests are performed after the *Launch Environment* and *Deploy Software* actions:



The tests are defined with SoapUI in `fruit-api-soapui-project.xml`. They are executed by Maven with the endpoint overridden to the URL from the CloudFormation output. An action is added to the CodeCatalyst workflow to run SoapUI:

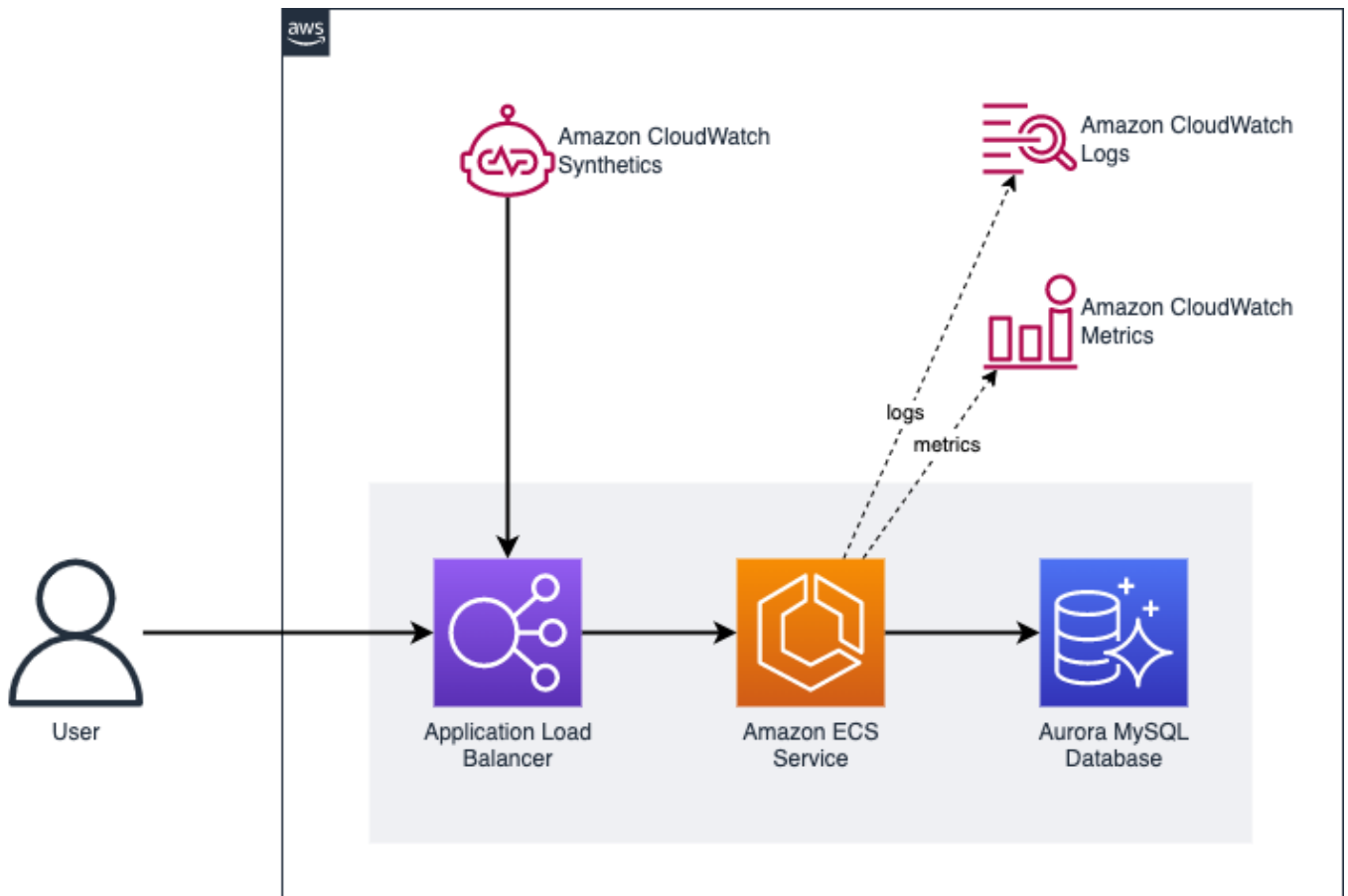
```

Test:
  Identifier: aws/managed-test@v1
  Inputs:
    Artifacts:
      - package
    Variables:
      - Name: endpointUrl
        Value: ${Deploy.endpointUrl}
  Configuration:
    Steps:
      - Run: mvn --batch-mode --no-transfer-progress soapui:test -Dsoapui.endpoint=${endpointUrl}
      - Run: mvn --batch-mode --no-transfer-progress compile jmeter:jmeter jmeter:results -Djmeter.endpoint=${endpointUrl} -Djmeter.threads=300 -
        Djmeter.duration=300 -Djmeter.throughput=6000
    Outputs:
      AutoDiscoverReports:
        Enabled: true
        IncludePaths:
          - target/soapui-reports/*
      ReportNamePrefix: Beta
      SuccessCriteria:
        PassRate: 100
  
```

The results of the unit tests are uploaded to Amazon CodeCatalyst to track over time.

Test (Gamma)

Launch Environment



The infrastructure for each environment is defined in 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 ServerlessCluster(this, 'AuroraCluster', {
  engine: DatabaseClusterEngine.AURORA_MYSQL,
  vpc,
  credentials: Credentials.fromSecret(dbSecret),
  defaultDatabaseName: dbName,
  deletionProtection: false,
  clusterIdentifier: appName,
});

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 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}`,
    },
},
deregistrationDelay: Duration.seconds(5),
responseTimeAlarmThreshold: Duration.seconds(3),
healthCheck: {
    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,
}],
});

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

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

```

The CDK deployment is then performed for each environment:

```

Deploy:
  Identifier: aws/cdk-deploy@v1
  DependsOn:
    - Build
  Inputs:
    Artifacts:
      - package
  Environment:
    Name: Beta
    Connections:
      - Name: beta
        Role: codecatalyst
  Configuration:
    StackName: fruit-api
    Region: us-west-2
    Context: '{"deploymentConfigurationName":"CodeDeployDefault.ECSCanary10Percent5Minutes"}'
    CfnOutputVariables: ['"endpointUrl"']

```

Database Deploy

Spring Boot is configured to run Liquibase on startup. This reads the configuration in `src/main/resources/db/changelog/db.changelog-master.yml` to define the tables and initial data for the database:

```
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
```

Deploy Software

The *Launch Environment* action above creates a new Amazon ECS Task Definition for the new docker image and then updates the Amazon ECS Service to use the new Task Definition.

Application Monitoring & Logging

Amazon ECS uses Amazon CloudWatch Metrics and Amazon CloudWatch Logs for observability by default.

Synthetic Tests

Amazon CloudWatch Synthetics is used to continuously deliver traffic to the application and assert that requests are successful and responses are received within a given threshold. The canary is defined via CDK using the `@cdklabs/cdk-ecs-codedeploy` construct:

```
const service = new ApplicationLoadBalancedCodeDeployedFargateService(this, 'Api', {
  ...

  apiCanaryTimeout: Duration.seconds(5),
  apiTestSteps: [{
    name: 'getAll',
    path: '/api/fruits',
    jmesPath: 'length(@)',
    expectedValue: 5,
  }],
});
```

Performance Tests

Apache JMeter is used to run performance tests against the deployed application. The tests are stored in `src/test/jmeter` and added to the CodeCatalyst workflow:

```
Test:
  Identifier: aws/managed-test@v1
  Inputs:
    Artifacts:
      - package
    Variables:
      - Name: endpointUrl
        Value: ${Deploy.endpointUrl}
  Configuration:
    Steps:
      - Run: mvn --batch-mode --no-transfer-progress soapui:test -Dsoapui.endpoint=${endpointUrl}
      - Run: mvn --batch-mode --no-transfer-progress compile jmeter:jmeter jmeter:results -Djmeter.endpoint=${endpointUrl} -Djmeter.threads=300 -
Djmeter.duration=300 -Djmeter.throughput=6000
  Outputs:
    AutoDiscoverReports:
      Enabled: true
      IncludePaths:
        - target/soapui-reports/*
      ReportNamePrefix: Beta
      SuccessCriteria:
        PassRate: 100
```

Resilience Tests

Not Implemented

Dynamic Application Security Testing (DAST)

Not Implemented

Prod

Manual Approval

Not Implemented

Database Deploy

Spring Boot is configured to run Liquibase on startup. This reads the configuration in `src/main/resources/db/changelog/db.changelog-master.yml` to define the tables and initial data for the database:

```
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

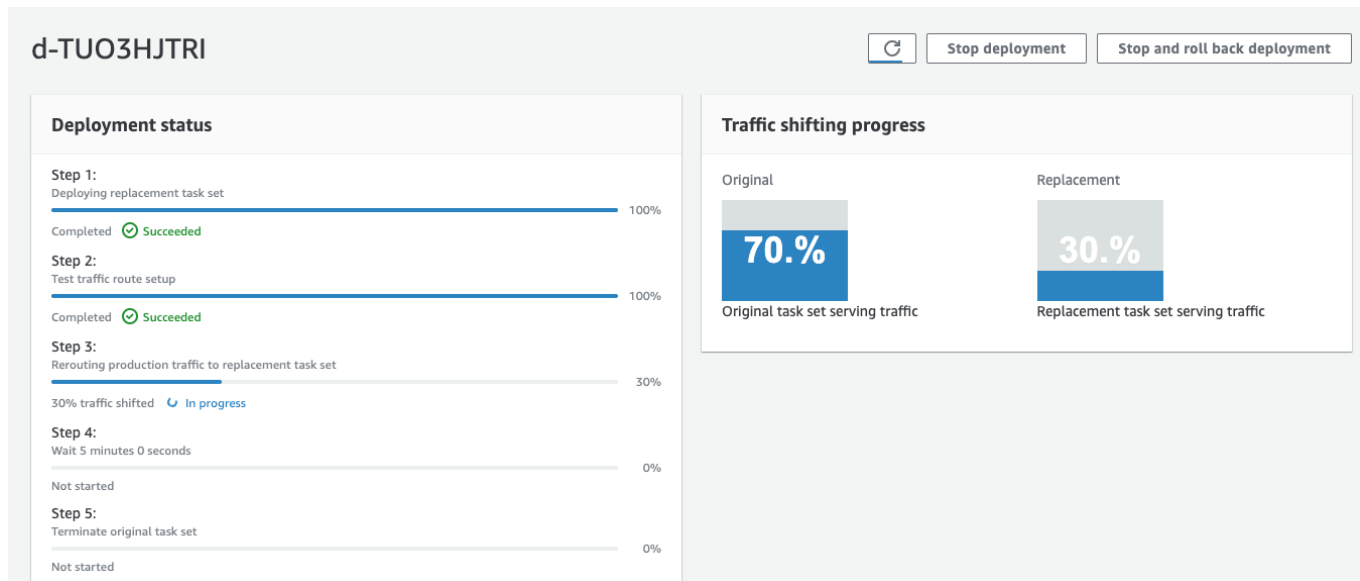
  - insert:
    tableName: fruit
    columns:
    - column:
      name: name
      value: Banana

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

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

Progressive Deployment

Progressive deployment is implemented with AWS CodeDeploy for ECS. CodeDeploy performs a linear blue/green by deploying the new task definition as a new task with a separate target group and then shifting 10% of traffic every minute until all traffic is shifted. A CloudWatch alarm is monitored by CodeDeploy and an automatic rollback is triggered if the alarm exceeds the threshold.



Implementation of this type deployment presents challenges due to the following limitations:

- aws/aws-cdk #19163 - CDK Pipelines aren't intended to be used with CodeDeploy actions.
- AWS CloudFormation User Guide - The use of `AWS::CodeDeploy::BlueGreen` hooks and `AWS::CodeDeployBlueGreen` restricts the types of changes that can be made. Additionally, you can't use auto-rollback capabilities of CodeDeploy.
- aws/aws-cdk #5170 - CDK doesn't support defining CloudFormation rollback triggers. This rules out CloudFormation based blue/green deployments.

The solution was to use the `@cdklabs/cdk-ecs-codedeploy` construct from the Construct Hub which addresses [aws/aws-cdk #1559](#) - Lack of support for Blue/Green ECS Deployment in CDK.

```
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}`,
  },
}, {
  deregistrationDelay: Duration.seconds(5),
  responseTimeAlarmThreshold: Duration.seconds(3),
  healthCheck: {
    healthyThresholdCount: 2,
    unhealthyThresholdCount: 2,
    interval: Duration.seconds(60),
    path: '/actuator/health',
  },
}, deploymentConfig,
```

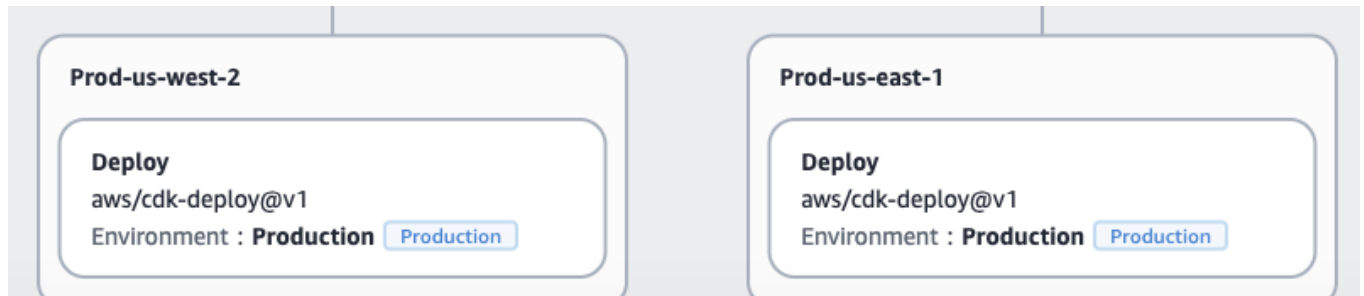
```

    terminationWaitTime: Duration.minutes(5),
    apiCanaryTimeout: Duration.seconds(5),
    apiTestSteps: [{
      name: 'getAll',
      path: '/api/fruits',
      jmesPath: 'length(@)',
      expectedValue: 5,
    }],
  });

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

```

Deployments are made incrementally across regions as waves using the action groups and the CDK deploy action. Each wave contains a list of regions to deploy to in parallel. One wave must fully complete before the next wave starts. The diagram below shows how each wave deploys to 2 regions at a time.



Synthetic Tests

Amazon CloudWatch Synthetics is used to continuously deliver traffic to the application and assert that requests are successful and responses are received within a given threshold. The canary is defined via CDK using the `@cdklabs/cdk-ecs-codedeploy` construct:

```

const service = new ApplicationLoadBalancedCodeDeployedFargateService(this, 'Api', {
  ...

  apiCanaryTimeout: Duration.seconds(5),
  apiTestSteps: [{
    name: 'getAll',
    path: '/api/fruits',
    jmesPath: 'length(@)',
    expectedValue: 5,
  }],
});

```

Frequently Asked Questions

What operating models does this reference implementation support?

This reference implementation can accommodate any operation model with minor updates:

- Fully Separated - Restrict the role that CDK uses for CloudFormation execution to only create resources from approved product portfolios in AWS Service Catalog. Ownership of creating the products in Service Catalog is owned by the **Platform Engineering** team and operational support of Service Catalog is owned by the **Platform Operations** team. The **Platform Engineering** team should publish CDK constructs internally that provision AWS resources through Service Catalog. Update the CDK app in the `infrastructure/` directory to use CDK constructs provided by the `Platform Engineering` team. Use a CODEOWNERS file to require all changes to the `infrastructure/` directory be approved by the **Application Operations** team. Additionally, restrict permissions to the **Manual Approval** action to only allow members of the **Application Operations** to approve.
- Separated AEO and IEO with Centralized Governance - Restrict the role that CDK uses for CloudFormation execution to only create resources from approved product portfolios in AWS Service Catalog. Ownership of creating the products in Service Catalog is owned by the **Platform Engineering** team and operational support of Service Catalog is owned by the **Platform Engineering** team. The **Platform Engineering** team should publish CDK constructs internally that provision AWS resources through Service Catalog. Update the CDK app in the `infrastructure/` directory to use CDK constructs provided by the `Platform Engineering` team.
- Separated AEO and IEO with Decentralized Governance - The **Platform Engineering** team should publish CDK constructs internally that provision AWS resources in manner that achieve organizational compliance. Update the CDK app in the `infrastructure/` directory to use CDK constructs provided by the `Platform Engineering` team.

Additional Sources

The following blog posts align to this reference architecture and will be used to build future reference implementations:

- Building end-to-end AWS DevSecOps CI/CD pipeline with open source SCA, SAST and DAST tools
- Building an end-to-end Kubernetes-based DevSecOps software factory on AWS