CI/CD Pipeline Documentation

This document provides a comprehensive overview of the Continuous Integration and Continuous Deployment (CI/CD) pipeline implemented using GitHub Actions and Terraform. It details the workflow's structure, components, and functionalities of the pipeline.

Table of Contents

1. Overview

The CI/CD pipeline automates the processes of building, testing, and deploying the application. It ensures that code changes are consistently integrated and deployed to staging and production environments, enhancing development efficiency and reliability.

2. Workflow Structure

The pipeline is defined in the `.github/workflows/ci-cd-pipeline.yml` file and comprises multiple jobs executed on specified triggers.

2.1 Triggers

The workflow is triggered on a push event to the `master` branch:

```
on:
  push:
    branches: [ "master" ]
```

This configuration ensures that the pipeline runs whenever changes are pushed to the `master` branch.

2.2 Jobs and Steps

The workflow consists of the following jobs:□

- **Deploy to Staging (`deploy-staging`)**
- **Deploy to Production (`deploy-prod`)**

Each job contains a series of steps executed sequentially.□

Deploy to Staging (`deploy-staging`)

This job deploys the application to the staging environment. Key steps include:□

1. **Checkout Repository**: Retrieves the latest code from the repository.□
2. **Setup Node.js**: Configures the Node.js environment.□
3. **Install Dependencies and Build**: Installs necessary packages and builds the application.□
4. **Setup Terraform**: Configures Terraform for infrastructure management.□
5. **Terraform Init & Validate (Staging)**: Initializes and validates Terraform configurations for the staging environment.□
6. **Terraform Plan (Staging)**: Creates an execution plan for infrastructure changes in staging.□
7. **Upload Staging Plan Artifact**: Uploads the Terraform plan for review.□
8. **Terraform Apply (Staging)**: Applies the Terraform plan to provision resources in the staging environment.□

Deploy to Production (`deploy-prod`)

This job deploys the application to the production environment and depends on the successful completion of the `deploy-staging` job. Key steps mirror those in the staging deployment, adjusted for the production environment.□

3. Terraform Integration

Terraform is used to manage infrastructure as code within this pipeline.□

3.1 Archiving Lambda Functions

The pipeline utilizes Terraform's `archive_file` data source to package Lambda function code into `.zip` archives. This is essential for deploying the functions to AWS Lambda.□

Example Configuration:

```
locals {
  build_folder = "${path.root}/../dist/lambdas"
}

data "archive_file" "create_item_zip" {
  type        = "zip"
  source_file = "${local.build_folder}/createItem.js"
  output_path = "${path.module}/zips/createItem.zip"
}
```

In this configuration:

- `type`: Specifies the archive format (`zip`).
- `source_file`: Points to the JavaScript file to be archived.
- `output_path`: Defines the destination path for the created archive.

3.2 Considerations for `archive_file` Data Source

The `archive_file` data source generates archives during the `terraform plan` phase. This behavior has implications in CI/CD pipelines:

- **Plan and Apply Separation**: In workflows where `terraform plan` and `terraform apply` are executed in separate steps or jobs, the archive created during the plan phase may not be available during the apply phase. To address this, consider:
  - Creating the archive as part of the build process before running Terraform commands.
  - Ensuring that the archive is available in the expected location during both plan and apply phases.

For more details, refer to the Terraform documentation on the `archive_file` data source. citeturn0search2

4. Environments

The pipeline defines two deployment environments:

- **Staging**: Used for testing and validation before production deployment.
- **Production**: The live environment where the application is accessible to end-users.

Each environment is configured in GitHub with required manual approvals to ensure controlled deployments.

5. Artifacts

Artifacts such as Terraform plans are uploaded during the workflow to facilitate review and approval processes. This ensures that infrastructure changes are transparent and can be audited before application.

6. Security Considerations

- **Secrets Management**: Sensitive information, such as AWS credentials, should be stored securely using GitHub Secrets and referenced in the workflow.
- **Manual Approvals**: Implementing required reviews for production deployments adds a layer of security and oversight.

7. Best Practices

- **Modular Terraform Code**: Organize Terraform configurations into modules for reusability and clarity.