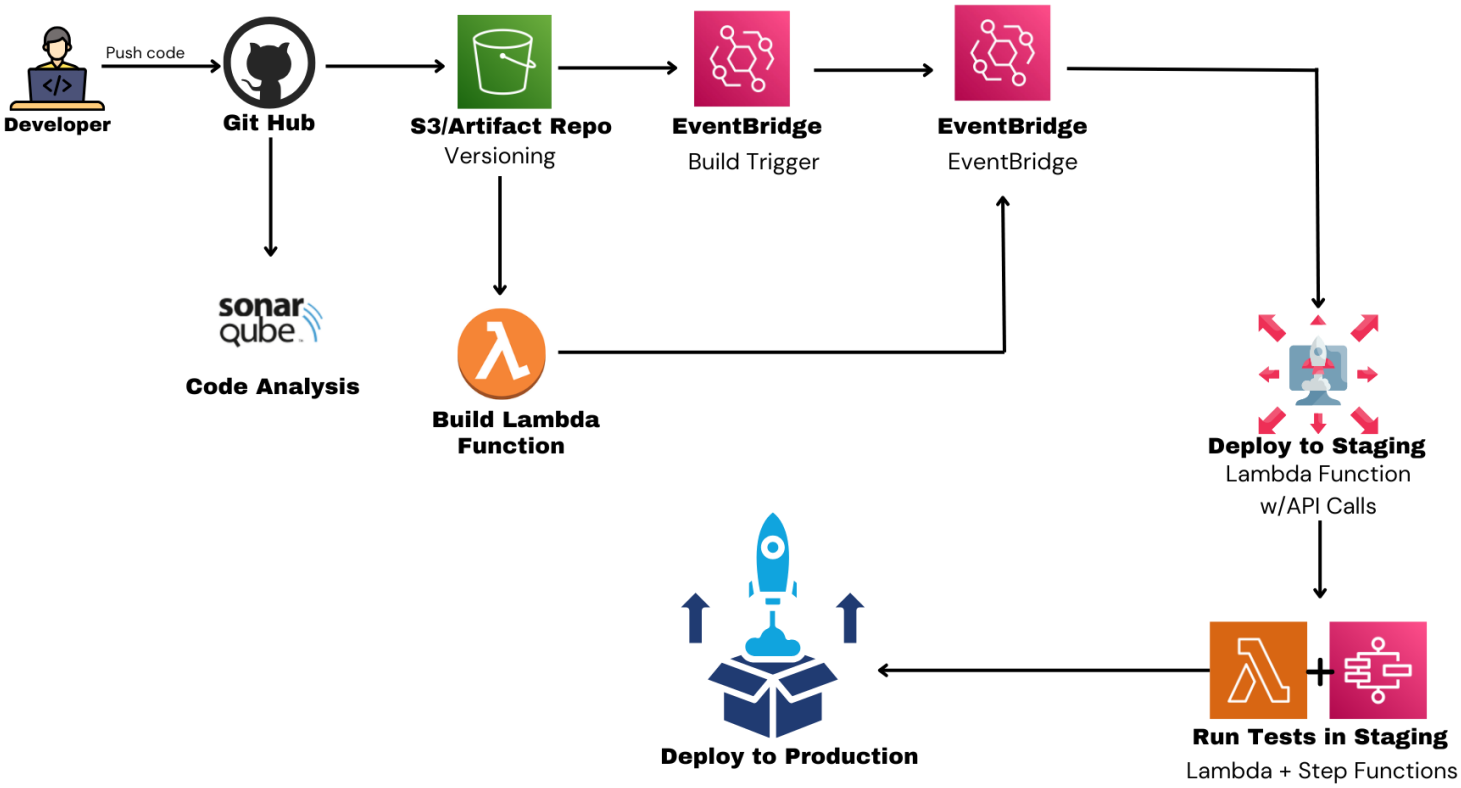
DevOps Shack

Serverless CI/CD Pipeline: A Complete Guide

1.1 What are Serverless CI/CD Pipelines?

Traditional CI/CD pipelines rely on dedicated servers or virtual machines to automate tasks such as building, testing, and deploying code. In contrast, **serverless CI/CD pipelines** utilize serverless compute services, like AWS Lambda, which execute tasks only when triggered, eliminating the need to maintain or provision servers.



This approach is ideal for modern, event-driven workflows where tasks occur sporadically, such as during code pushes.

1.2 Benefits

1. **Cost Efficiency**:
    - You only pay for the actual compute time used by serverless functions.
    - Idle time is eliminated compared to traditional server-based pipelines.
2. **Scalability**:
    - AWS Lambda can handle thousands of concurrent executions automatically.
    - This ensures your pipeline scales with your team's activity.
3. **Reduced Maintenance**:
    - No servers or operating systems to manage.
    - Focus entirely on your pipeline logic.
4. **Fast Iterations**:
    - Rapid deployment and execution of functions, speeding up development cycles.

2. Architecture Overview

The serverless CI/CD pipeline architecture includes the following steps:

1. **Code Commit**: Developers push code to a GitHub repository.
2. **Trigger**: AWS EventBridge listens for repository changes and invokes a Lambda function.
3. **Code Analysis**: A Lambda function runs static analysis using tools like SonarQube.
4. **Build**: A Lambda function compiles the code and stores the artifacts in an S3 bucket.
5. **Deploy to Staging**: Another Lambda function deploys the build to a staging environment.
6. **Testing**: Lambda functions or AWS Step Functions orchestrate automated tests.
7. **Deploy to Production**: Successful tests trigger a deployment to the production environment.

3. Tools and Services Used

- **GitHub**: For version control and code management.
- **AWS Lambda**: Executes each step of the pipeline (build, deploy, etc.).
- **AWS EventBridge**: Triggers Lambda functions based on GitHub events.
- **AWS S3**: Stores build artifacts and logs.
- **SonarQube**: Performs code quality analysis.
- **AWS Step Functions**: Orchestrates complex workflows like automated testing.
- **AWS CodeBuild**: (Optional) Handles resource-intensive build steps.

4. Step-by-Step Implementation

4.1 Prerequisites

Before starting, ensure you have the following:

1. **AWS Account**: With permissions to create and manage Lambda, S3, EventBridge, and other services.
2. **GitHub Repository**: A repository with some sample code (e.g., a Python app).
3. **AWS CLI Installed**: For easier resource management.
4. **SonarQube Server**: Hosted locally or in the cloud for code analysis.

4.2 Setting Up the Repository

Step 1: Create a GitHub Repository

1. Log in to GitHub and create a new repository.
2. Clone the repository to your local system:

```
git clone https://github.com/your-username/your-repo.git
cd your-repo
```

Step 2: Add Sample Code

Create a simple Python application with the following structure:

```
/src
  ├── app.py          # Application code
  ├── requirements.txt # Dependencies
/tests
  ├── test_app.py     # Unit tests
```

Sample app.py:

```
def hello_world():
    return "Hello, Serverless CI/CD!"
```

Sample test_app.py:

```
import unittest
from src.app import hello_world

class TestApp(unittest.TestCase):
    def test_hello_world(self):
        self.assertEqual(hello_world(), "Hello, Serverless CI/CD!")
```

4.3 Creating and Configuring AWS Services

Step 1: Create S3 Buckets

1. Go to the AWS S3 console and create two buckets:
    - my-ci-artifacts (for build artifacts).
    - my-ci-logs (for logs).
2. Enable versioning and lifecycle policies for these buckets to manage storage effectively.

Step 2: Set Up EventBridge

1. Navigate to the **EventBridge Console**.
2. Create a new rule:
    - **Name**: GitHubPushTrigger
    - **Event Source**: Custom (webhook integration with GitHub).
3. Configure the rule to invoke a Lambda function.

4.4 Pipeline Steps

Step 1: Lambda for Code Analysis

This function clones the repository and sends the code to SonarQube for analysis.

Code:

```
import subprocess

def lambda_handler(event, context):
    # Clone the GitHub repository
    repo_url = event['repo_url']
    subprocess.run(["git", "clone", repo_url, "/tmp/repo"])

    # Run SonarQube analysis
```

```
    subprocess.run(["sonar-scanner", "-Dsonar.projectKey=my_project", "-
Dsonar.sources=/tmp/repo"])
    return {"status": "Code analysis complete"}
```

Step 2: Lambda for Building Artifacts
Code:

```
import boto3
import subprocess

s3_client = boto3.client('s3')

def lambda_handler(event, context):
    # Package the code
    subprocess.run(["zip", "-r", "/tmp/build.zip", "/tmp/repo"])

    # Upload to S3
    s3_client.upload_file("/tmp/build.zip", "my-ci-artifacts", "build.zip")
    return {"status": "Build uploaded"}
```

Step 3: Lambda for Deployment
Code:

```
import boto3

def lambda_handler(event, context):
    ecs_client = boto3.client('ecs')
    response = ecs_client.update_service(
        cluster='my-cluster',
        service='my-service',
        taskDefinition='my-task:2'
    )
    return {"status": "Deployment complete"}
```

Project Structure

```
/serverless-cicd
  ├── pipeline.py        # Main Lambda function handlers
  ├── eventbridge_rule.json # EventBridge rule for GitHub webhook
  ├── requirements.txt    # Dependencies for Lambda functions
  ├── deploy.sh          # Deployment script using AWS CLI
  └── README.md          # Documentation
```

1. pipeline.py (Main Code)

```
import boto3
import subprocess
import os

# Initialize AWS clients
s3_client = boto3.client('s3')
ecs_client = boto3.client('ecs')

# S3 Bucket Names
ARTIFACTS_BUCKET = "my-ci-artifacts"

# Lambda Function: Code Analysis with SonarQube
def code_analysis_handler(event, context):
    repo_url = event.get('repo_url', 'https://github.com/user/repo.git')

    # Clone the repository
    subprocess.run(["git", "clone", repo_url, "/tmp/repo"], check=True)

    # Run SonarQube analysis
    sonar_project_key = "my_project"
    subprocess.run(
        ["sonar-scanner",
         f"-Dsonar.projectKey={sonar_project_key}",
         "-Dsonar.sources=/tmp/repo"],
        check=True
    )
    return {"status": "Code analysis complete"}
```

```
# Lambda Function: Build Artifacts
def build_handler(event, context):
    # Package the code
    build_path = "/tmp/build.zip"
    subprocess.run(["zip", "-r", build_path, "/tmp/repo"], check=True)

    # Upload build to S3
    s3_client.upload_file(build_path, ARTIFACTS_BUCKET, "build.zip")
    return {"status": "Build uploaded to S3"}

# Lambda Function: Deploy to ECS
def deploy_handler(event, context):
    cluster_name = "my-cluster"
    service_name = "my-service"
    task_definition = "my-task:2"  # Update with your ECS task definition

    # Update ECS service
    response = ecs_client.update_service(
        cluster=cluster_name,
        service=service_name,
        taskDefinition=task_definition
    )
    return {"status": "Deployment complete", "response": response}
```

2. eventbridge_rule.json

Configure the EventBridge rule to trigger Lambda functions upon GitHub push events.

```
{
  "EventPattern": {
    "source": ["aws.codecommit"],
    "detail-type": ["CodeCommit Repository State Change"],
    "detail": {
      "event": ["referenceUpdated"]
    }
  },
  "State": "ENABLED",
  "Targets": [
```

```
  {
   "Id": "CodeAnalysisLambda",
   "Arn": "arn:aws:lambda:region:account-id:function:CodeAnalysisFunction"
  }
 ]
}
```

3. requirements.txt

Dependencies for Lambda functions:

```
boto3
```

If SonarQube scanning tools are required, prepackage binaries for the sonar-scanner utility.

4. deploy.sh (Deployment Script)

This script automates the deployment of all resources using the AWS CLI.

```
#!/bin/bash

# Variables
REGION="us-east-1"
ARTIFACTS_BUCKET="my-ci-artifacts"
CODE_ANALYSIS_LAMBDA="CodeAnalysisFunction"
BUILD_LAMBDA="BuildFunction"
DEPLOY_LAMBDA="DeployFunction"

# Create S3 bucket
aws s3api create-bucket --bucket $ARTIFACTS_BUCKET --region $REGION --create-bucket-configuration LocationConstraint=$REGION

# Package and deploy Lambda functions
zip pipeline.zip pipeline.py
aws lambda create-function --function-name $CODE_ANALYSIS_LAMBDA \
 --runtime python3.9 --role <YOUR_IAM_ROLE_ARN> --handler pipeline.code_analysis_handler \
 --timeout 120 --zip-file fileb://pipeline.zip --region $REGION

aws lambda create-function --function-name $BUILD_LAMBDA \
```

```
 --runtime python3.9 --role <YOUR_IAM_ROLE_ARN> --handler
pipeline.build_handler \
 --timeout 120 --zip-file fileb://pipeline.zip --region $REGION

aws lambda create-function --function-name $DEPLOY_LAMBDA \
 --runtime python3.9 --role <YOUR_IAM_ROLE_ARN> --handler
pipeline.deploy_handler \
 --timeout 120 --zip-file fileb://pipeline.zip --region $REGION

# Create EventBridge rule and attach to Lambda
aws events put-rule --name GitHubPushTrigger --event-pattern
file://eventbridge_rule.json --region $REGION
aws lambda add-permission --function-name $CODE_ANALYSIS_LAMBDA --action
lambda:InvokeFunction --statement-id GitHubEvent --principal
events.amazonaws.com --region $REGION
aws events put-targets --rule GitHubPushTrigger --targets
"Id"="1","Arn"="arn:aws:lambda:$REGION:<account-
id>:function:$CODE_ANALYSIS_LAMBDA"

echo "Deployment complete!"
```

5. Testing the Pipeline

1. Commit changes to GitHub.
2. Confirm that EventBridge triggers the pipeline.
3. Check Lambda logs in CloudWatch to verify execution.

6. Cost Optimization Tips

- Use S3 lifecycle policies to clean up old artifacts.
- Optimize Lambda memory and timeout settings.
- Enable detailed billing to track costs.

7. Conclusion

Serverless CI/CD pipelines provide an efficient way to automate deployments without the burden of maintaining servers. By leveraging AWS Lambda and associated services, you can build scalable, cost-effective pipelines for any application.