Multi Cloud DevOps Project Documentation

Table of Contents

Introduction

Overview

The MultiCloudDevOpsProject is a comprehensive DevOps initiative aimed at streamlining the deployment and management of an application on cloud environment. This project integrates best practices in infrastructure provisioning, configuration management, containerization, continuous integration, and automated deployment to achieve a robust and scalable DevOps pipeline.

Purpose

The primary purpose of this project is to demonstrate the implementation of a modern DevOps workflow, emphasizing portability and flexibility across different cloud providers. By adopting industry-standard tools and practices, we aim to showcase the seamless orchestration of cloud resources, efficient code integration, and deployment automation.

Goals

1. Cloud-Agnostic Deployment: Develop a deployment pipeline that can seamlessly deploy the application across various cloud providers, ensuring flexibility and avoiding vendor lock-in.

2. Infrastructure as Code (IaC): Utilize Terraform for infrastructure provisioning to enforce version-controlled, repeatable, and consistent infrastructure changes.

3. Configuration Management: Leverage Ansible playbooks for the configuration of EC2 instances, ensuring the reproducibility and maintainability of the application environment.

4. Containerization: Implement Docker containerization to encapsulate the application and its dependencies, promoting consistency and portability across different environments.

5. Continuous Integration: Establish a Jenkins-based CI/CD pipeline to automate the build, test, and deployment processes, ensuring rapid and reliable software delivery.

6. Automated Deployment Pipeline: Develop a Jenkins pipeline (Jenkinsfile) to orchestrate the complete deployment lifecycle, from code checkout to deployment on OpenShift, incorporating unit testing and SonarQube analysis.

7. Monitoring and Logging: Set up centralized logging on OpenShift for efficient monitoring of container logs and integrate AWS CloudWatch for monitoring AWS resources.

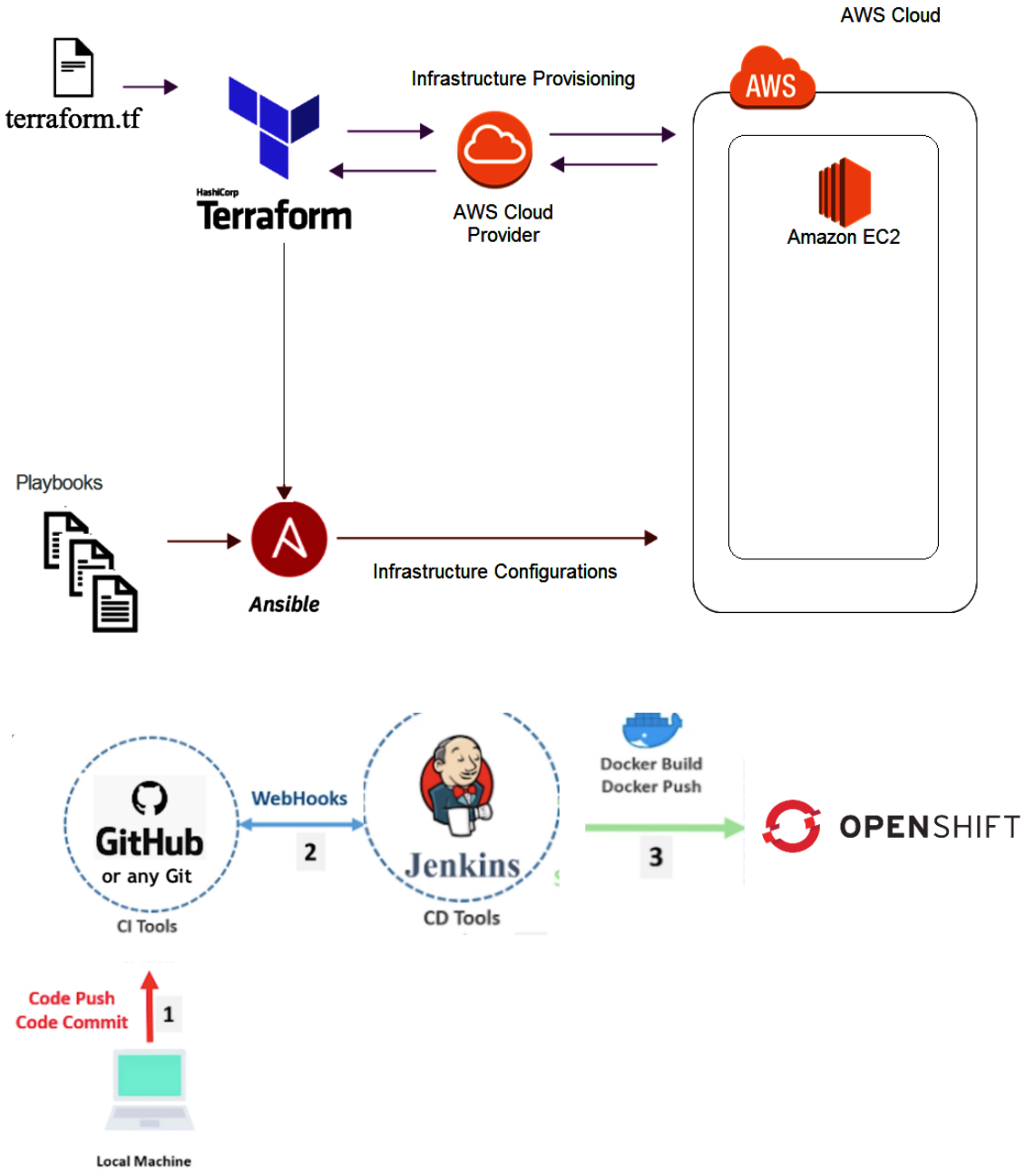
8. AWS Integration: Demonstrate the integration of AWS services, utilizing S3 as the Terraform Backend state and integrating CloudWatch for monitoring.

Scope

The scope of the project encompasses the end-to-end DevOps process, starting from the creation of the GitHub repository to the deployment of the application on OpenShift. It includes infrastructure provisioning, configuration management, containerization, continuous integration, and cloud integration across multiple cloud platforms.

This documentation provides a detailed guide on each aspect of the project, facilitating seamless setup, deployment, and understanding of the underlying technologies.

Project Architecture



Project Flow

1. Infrastructure Provisioning with Terraform

- Terraform scripts are used to provision AWS resources, including VPC, Subnets, and Security Groups.

- EC2 instances are created for application deployment.

- Delivered in project repo: “Terraform”

2. Configuration Management with Ansible

- Ansible playbooks are utilized to configure EC2 instances.

- Required packages such as Git, Docker, Java, Jenkins, and SonarQube are installed.

- Necessary environment variables are set up.

- Delivered in project repo: “ansible”

3. Containerization with Docker

- Dockerfile is provided for building the application image.

- Delivered in project repo: Docker”

4. Continuous Integration with Jenkins

Jenkins Job configuration instructions

setting Up Jenkins Job configuration instructions for building Docker image on code commits.

Step 1: Jenkins Installation:

- Download and install Jenkins on the desired server or machine.
- Follow the installation instructions for your operating system.
- Start the Jenkins service and access the Jenkins dashboard through a web browser.

Run the following commands to start jenkins service:

```
# systemctl start jenkins    # Start the Jenkins service
# systemctl enable jenkins   # Enable Jenkins to start on system boot
```

Step 2: Create a New Jenkins Job

- Open Jenkins and navigate to the Dashboard.
- Click on "New Item" to create a new job.
- Choose Pipeline and provide a name.

Step 3: Add DockerHub Credentials in Jenkins:

- To push the Docker image to DockerHub, add DockerHub credentials in Jenkins.
- Navigate to the Credentials section and add DockerHub username and password or token.

Step 4: Install Docker:

- Install Docker on your machine where Jenkins is installed.
- Follow the Docker installation instructions for your operating system.

Run the following commands for docker installation and follow the installation guide provided by docker in any trouble:

```
# yum update
# yum install docker.io
# systemctl start docker
# systemctl enable docker
```

Step 5: Install Docker Pipeline Plugin:

- In Jenkins, go to the "Manage Jenkins" section.

- Select "Manage Plugins" and install the Docker Pipeline plugin.

- This plugin allows you to define Docker-related steps in your Jenkins pipeline.

Step 6: Configure Git Credentials in Jenkins:

- In Jenkins, navigate to the Credentials section.

- Add the necessary credentials (username/password or SSH key) to access your Git repository.

- These credentials will be used by Jenkins to fetch the source code during the build/checkout process.

Step 7: Set Up Build Trigger:

- In the job configuration, go to the "Build Triggers" section.

- Configure triggers to run the job on code commits.

- Webhooks and build triggers are often used together.

- Webhooks serve as a mechanism to notify Jenkins about events,

and build triggers respond to those events by initiating the build process.

Step 8: Write Jenkinsfile (Pipeline Code):

- Define stages and steps in the pipeline,

define the GitHub repository URL.

including the Docker build step.

Use the Docker Pipeline DSL to specify Docker-related actions in your pipeline.

- This a basic format for pipeline code:

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                script {
                    docker.build('your-image-name')
                }
            }
        }
    }
}
```

Step 9: Configure Webhook:

- In your version control system (e.g., GitHub), go to the repository settings.

- Set up a webhook to trigger the Jenkins job on code commits.

- Configure the webhook payload URL to point to the Jenkins server and include the necessary security tokens.

Step 10: Test the Setup:

Make code commits to your repository.

Verify that Jenkins is automatically triggered through the webhook.

Check the Jenkins console logs and build output to ensure the Docker image is successfully built.

5. Automated Deployment Pipeline

- Jenkins pipeline (Jenkinsfile) defines stages for Git Checkout, Build, Unit Test, SonarQube Test, and Deploy on OpenShift.

- Shared Jenkins Library is utilized for modular and reusable pipeline code.

- Delivered in project repo: “Jenkins”

6. Monitoring and Logging

- Centralized logging is set up on OpenShift for container logs.

Instructions to setup centralized logging:

Step 1: Deploy Elasticsearch

Elasticsearch will be deployed to store and index logs.

Instructions:

Run the following commands to create a new project named "logging" and deploy Elasticsearch:

#oc new-project logging

#oc apply -f https://raw.githubusercontent.com/openshift/cluster-logging-operator/main/assets/examples/openshift/elasticsearch/01_elasticsearch_instance.yaml

Step 2: Deploy Fluentd

Fluentd, a log collector, will be deployed to collect logs from application pods and send them to Elasticsearch.

Instructions:

Run the following command to deploy:

#oc apply -f https://raw.githubusercontent.com/openshift/cluster-logging-operator/main/assets/examples/openshift/fluentd/02_fluentd_instance.yaml

Step 3: Deploy Kibana

Kibana, a visualization tool, will be deployed to visualize and query logs.

Instructions:

Run the following command:

#oc apply -f https://raw.githubusercontent.com/openshift/cluster-logging-operator/main/assets/examples/openshift/kibana/03_kibana_instance.yaml

Step 4: Set Up Fluentd Forwarding

Configure Fluentd to forward logs to Elasticsearch.

Instructions:

Edit Fluentd configuration use this command:

#oc edit cm -n openshift-operators fluentd

Add the following block in the config section:

```
- name: @type forward
  @id forward_output
  send_timeout 10s
  recover_wait 5s
  heartbeat_type tcp
  heartbeat_interval 1s
  hard_timeout 60s
  <server>
    name fluentd
    host <FLUENTD-HOST> #Replace <FLUENTD-HOST> with the Fluentd service IP.
    port 24224
  </server>
```

Step 5: View Logs in Kibana

Access Kibana to view and search logs.

Instructions:

Get the Kibana route using the following command:

#oc get route -n openshift-logging kibana -o jsonpath='{.spec.host}'

Visit the URL in your browser and log in using OpenShift cluster credentials.

7. AWS Integration

- S3 Terraform Backend state is used for managing Terraform state.

- CloudWatch is integrated for monitoring AWS resources.

Instructions for AWS integration in Terraform code:

Use S3 as Terraform Backend State

Step 1: Create an S3 Bucket for Terraform State

Instructions:

Log in to the AWS Management Console.

Navigate to the S3 service.

Click "Create bucket" and follow the prompts.

Choose a globally unique bucket name.

Configure settings as needed (permissions, logging, etc.).

Step 2: Configure Terraform Backend

Instructions:

In your Terraform configuration, add the following block to configure the S3 backend:

```
terraform {
  backend "s3" {
    bucket         = "your-unique-s3-bucket-name"
    key            = "path/to/terraform.tfstate"
    region         = "your-aws-region"
    encrypt        = true
    dynamodb_table = "your-lock-table-name"  # Optional for state locking
  }
}
```

Run terraform init to initialize the backend.

Step 3: State Locking (Optional)

Instructions:

If you want to enable state locking, create a DynamoDB table for locking.

Add the dynamodb_table configuration in the Terraform backend block.

Integrate CloudWatch for Monitoring

Step 1: Configure AWS CloudWatch

Instructions:

Log in to the AWS Management Console.

Navigate to the CloudWatch service.

Under "Logs," create a log group for your application logs.

Step 2: Instrument Your Application

Instructions:

Depending on your application type, install an AWS CloudWatch agent or use SDKs to instrument your application.

For EC2 instances, install the CloudWatch agent.

For serverless applications, use the AWS SDK.

Step 3: Add CloudWatch Alarms

Instructions:

In CloudWatch, under "Alarms," create alarms to monitor specific metrics.

For example, set up alarms for CPU utilization, memory usage, or custom application metrics.

Step 4: Integrate CloudWatch Events (Optional)

Instructions:

Use CloudWatch Events to trigger actions based on events.

Create rules to react to specific events and automate responses.

Troubleshooting Guidelines

While implementing and maintaining the MultiCloudDevOpsProject, you may encounter challenges or issues. Below are some common troubleshooting guidelines to help address potential issues:

1. Jenkins Job Failures:

- Check Jenkins Console Logs:
    - Examine the detailed console logs in Jenkins to identify any error messages or issues during the build and deployment stages.
- Verify Docker Installation:
    - Ensure that Docker is correctly installed on the Jenkins server, and the Jenkins user has the necessary permissions to interact with Docker.

2. Ansible Playbook Failures:

- Inspect Ansible Output:
    - Examine the output of Ansible playbooks to identify any failures or errors in the configuration process.

3. Docker Image Build Failures:

- Review Dockerfile:
    - Check the Dockerfile for syntax errors or issues that might be causing the build to fail.
- Validate Docker Installation:
    - Ensure that Docker is installed on the machine where the Docker image is being built.

4. OpenShift Deployment Issues:

- Check OpenShift Logs:
    - Inspect OpenShift logs to identify any issues during the deployment process.
- Verify OpenShift Project Configuration:
    - Ensure that the OpenShift project configuration aligns with the deployment parameters specified in the Jenkinsfile.

Conclusion

In conclusion, the MultiCloudDevOpsProject successfully implemented a robust DevOps pipeline for deploying applications across multiple cloud environments. By integrating tools like Terraform, Ansible, Docker, and Jenkins, we achieved streamlined infrastructure provisioning, configuration management, and continuous integration.