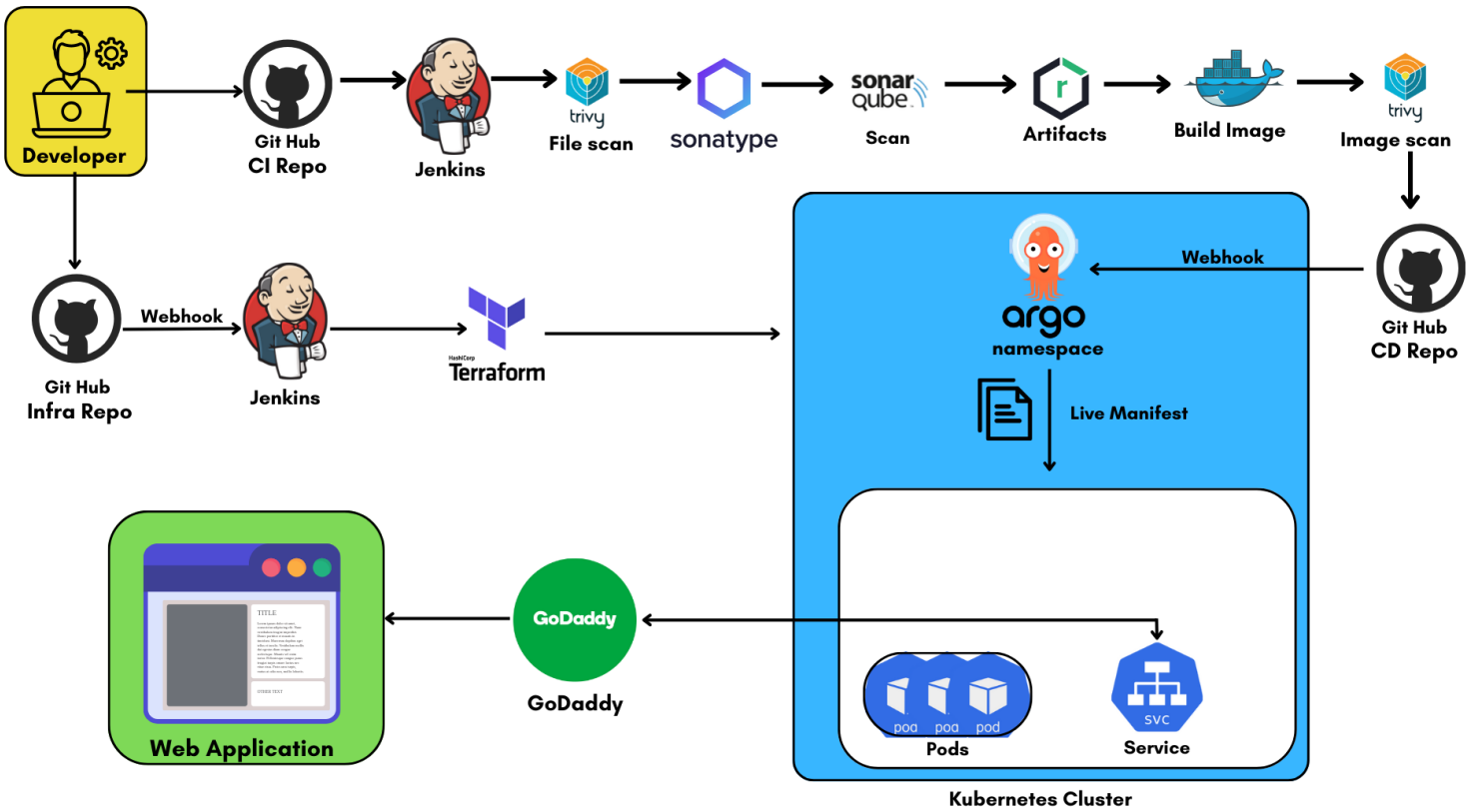
DevOps Shack

GitOps Project: Full Detailed Documentation

Batch-7 of DevSecOps & Cloud DevOps Starts 2nd November | Enrol Now

Overview:

This document outlines the implementation of GitOps, focusing on automating the deployment of infrastructure and applications using Git as a single source of truth. We will walk through the process, principles of GitOps, and step-by-step implementation with code examples.



1. Introduction to GitOps

What is GitOps?

GitOps is a methodology for automating infrastructure and application deployment using Git as a single source of truth. It allows teams to manage infrastructure using the same tools they use for application development, leading to more efficient, transparent, and reliable deployment workflows.

Key points:

- Automates the deployment of infrastructure and applications.
- Uses Git repositories for configuration management.
- Applies the Infrastructure as Code (IaC) approach.

GitOps Components:

- **Infrastructure Automation**: Automates the deployment of infrastructure such as Kubernetes clusters (e.g., Amazon EKS, virtual machines).
- **Application Automation**: Automates application deployment (e.g., Docker containers, Kubernetes Pods).
- **Git as Single Source of Truth**: All configurations (both infrastructure and application) are stored in Git repositories.

2. GitOps Principles

GitOps operates based on four main principles:

1. Git as a Single Source of Truth: All infrastructure and application configuration details are stored in Git repositories. This includes:

- Infrastructure specifications (e.g., EKS cluster, virtual machines).
- Application deployment configurations (e.g., number of Pods, Docker registry details).

2. Declarative Desired State: Configurations follow the **declarative** approach, where the desired final state is described without specifying exact steps to achieve it. Examples include:

- Kubernetes cluster should have 3 worker nodes with 30GB storage each.
- Application should run with 3 Pods, each pulling a Docker image from a specific registry.

3. Automated Reconciliation: Whenever there is a change in the Git repository (e.g., updating the number of worker nodes from 3 to 4), the changes are automatically reflected in the deployment. This ensures that the system always matches the desired state stored in the repository.

4. Pull-Based Deployment: GitOps tools like **Argo CD** continuously monitor Git repositories and automatically pull changes to deploy them. Argo CD ensures that the state of the infrastructure and applications in the cluster aligns with the state defined in Git.

3. GitOps vs Traditional CI/CD

Traditional Approach:

- Uses CI/CD tools like Jenkins for building and deploying applications.
- CI involves building the application, running tests, and creating Docker images.
- CD involves deploying the application to Kubernetes, which requires credentials and connections between Jenkins and the Kubernetes cluster.

GitOps Approach:

- CI remains the same: building the application, running tests, and creating Docker images.
- In CD, instead of connecting Jenkins to Kubernetes, changes are pushed to the Git repository, and tools like Argo CD pull those changes and automatically deploy them.

- No need to manage Kubernetes credentials in Jenkins, improving security.

4. Implementation Workflow

Step 1: Setup EKS Cluster Using Terraform

1. **Create a Virtual Machine on AWS**
   Launch an EC2 instance on AWS to serve as your working environment.
2. **SSH into the VM**
   Use the following command to access the VM:

```
ssh -i "your-key.pem" ubuntu@your-vm-ip
```

3. **Install Terraform**
   Install Terraform using the following command:

```
sudo snap install terraform --classic
```

4. **Install AWS CLI**
   Download and install AWS CLI on the VM:

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"

sudo apt install unzip

unzip awscliv2.zip

sudo ./aws/install

aws configure
```

5. **Install Kubectl**
   Install kubectl:

```
curl -o kubectl https://amazon-eks.s3.us-west-2.amazonaws.com/1.19.6/2021-01-05/bin/linux/amd64/kubectl

chmod +x ./kubectl

sudo mv ./kubectl /usr/local/bin

kubectl version --short --client
```

6. **Install EKSCTL**
   Install eksctl for EKS cluster management:

```
curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp

sudo mv /tmp/eksctl /usr/local/bin

eksctl version
```

7. **Save the Script**
   Save all commands in a file (e.g., ctl.sh) and make it executable:

```
chmod +x ctl.sh
```

8. **Create Terraform Files**
   Create the following files for your Terraform setup:
   - main.tf
   - output.tf
   - variables.tf

9. **Run Terraform Commands**
   Initialize and apply Terraform configurations:

```
terraform init

terraform plan

terraform apply -auto-approve
```

10. **Create EKS Cluster**
    Use eksctl to create the EKS cluster:

```
eksctl create cluster --name=EKS-1 \
                      --region=ap-south-1 \
                      --zones=ap-south-1a,ap-south-1b \
                      --without-nodegroup
```

11. **Enable OpenID Connect**

    Associate IAM OIDC provider:

```
eksctl utils associate-iam-oidc-provider \
 --region ap-south-1 \
 --cluster EKS-1 \
 --approve
```

12. **Create Node Group**

    Create a managed node group for your cluster:

```
eksctl create nodegroup --cluster=EKS-1 \
        --region=ap-south-1 \
        --name=node2 \
        --node-type=t3.medium \
        --nodes=3 \
        --nodes-min=2 \
        --nodes-max=4 \
        --node-volume-size=20 \
        --ssh-access \
        --ssh-public-key=DevOps \
        --managed \
        --asg-access \
        --external-dns-access \
        --full-ecr-access \
        --appmesh-access \
        --alb-ingress-access
```

Note: Replace DevOps with your SSH key name.

Step 2: Launch Virtual Machine for Jenkins, SonarQube, and Nexus

1. **EC2 Instance Requirements:**
    - **Instance Type:** t2.large
    - **vCPUs:** 2
    - **Memory:** 8 GB
    - **Network Performance:** Moderate
    - **AMI:** Ubuntu Server 20.04 LTS
2. **Security Groups:**
    - Configure security groups to allow access on necessary ports.
3. **SSH into the VM:**

```
ssh -i "your-key.pem" ubuntu@your-vm-ip
```

Step 3: Install Jenkins on Ubuntu

1. **Install Jenkins:** Create a script (install_jenkins.sh) to automate Jenkins installation:

```
#!/bin/bash

sudo apt install openjdk-17-jre-headless -y

sudo wget -O /usr/share/keyrings/jenkins-keyring.asc
https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key

echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc]
https://pkg.jenkins.io/debian-stable binary/ | sudo tee
/etc/apt/sources.list.d/jenkins.list > /dev/null

sudo apt-get update

sudo apt-get install jenkins -y
```

Run the script:

```
chmod +x install_jenkins.sh

./install_jenkins.sh
```

2. **Install Docker:** Create a script (install_docker.sh) to automate Docker installation:

```
#!/bin/bash

sudo apt-get update

sudo apt-get install -y ca-certificates curl

sudo install -m 0755 -d /etc/apt/keyrings

sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc

sudo chmod a+r /etc/apt/keyrings/docker.asc

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Run the script:

```
chmod +x install_docker.sh

./install_docker.sh
```

Step 4: Setup Nexus

1. **Run Nexus Docker Container:**

```
docker run -d --name nexus -p 8081:8081 sonatype/nexus3:latest
```

   - Access Nexus at http://your-vm-ip:8081.

2. **Get Nexus Initial Password:**

```
docker exec -it <container_ID> /bin/bash
cd sonatype-work/nexus3
cat admin.password
```

Step 5: Install SonarQube

1. **Run SonarQube Docker Container:**

```
docker run -d --name sonar -p 9000:9000 sonarqube:lts-community
```

   - Access SonarQube at http://your-vm-ip:9000.

Step 6: Install Jenkins Plugins

1. **Install Required Plugins:**
   - SonarQube Scanner
   - Config File Provider
   - Maven Integration
   - Kubernetes (and associated plugins)
   - Docker Pipeline
   - Pipeline Stage View
2. **Create Docker and GitHub Credentials:**
   - **Docker Credentials:**

```
Manage Jenkins > Manage Credentials > (global) > Add Credentials
Kind: Username with password
ID: docker-cred
```

   - **GitHub Credentials:**

Manage Jenkins > Manage Credentials > (global) > Add Credentials

Kind: Secret text

ID: git-cred

Step 7: Setup RBAC for Jenkins

1. **Create Service Account (svc.yaml):**

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins
  namespace: webapps
```

Apply the configuration:

bash

Copy code

```
kubectl apply -f svc.yaml
```

2. **Create Role (role.yaml):**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: app-role
  namespace: webapps
rules:
  - apiGroups: ["", "apps", "extensions", "rbac.authorization.k8s.io"]
    resources: ["pods", "services", "deployments"]
    verbs: ["get", "list", "create", "update", "delete"]
```

Apply the configuration:

```
kubectl apply -f role.yaml
```

3. **Bind Role to Service Account (bind.yaml):**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-rolebinding
  namespace: webapps
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: app-role
subjects:
  - kind: ServiceAccount
    name: jenkins
    namespace: webapps
```

Apply the configuration:

```
kubectl apply -f bind.yaml
```

4. **Create Service Account Token (secret.yaml):**

```
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: mysecretname
```

```
  annotations:
    kubernetes.io/service-account.name: jenkins
```

Retrieve the token:

```
kubectl describe secret mysecretname -n webapps
```

5. **Create ClusterRole and ClusterRoleBinding:**

```
kubectl create clusterrole jenkins-admin --verb=* --resource=pods

kubectl create clusterrolebinding jenkins-admin-binding --clusterrole=jenkins-admin --serviceaccount=webapps:jenkins
```

Step 7 : Pipeline

```
pipeline {
    agent any

    // Parameters allow passing custom values to the pipeline
    parameters {
        string(name: 'DOCKER_TAG', defaultValue: 'latest', description: 'Tag for the Docker image')
    }

    // Define the tools to be used, Maven in this case
    tools {
        maven 'maven3'
    }

    // Define environment variables, like the SonarQube scanner home
    environment {
        SCANNER_HOME = tool 'sonar-scanner'
    }

    stages {

        // Stage to clean the workspace
        stage('Clean Workspace') {
            steps {
                cleanWs()  // Clean up previous files
            }
        }

        // Stage to checkout the code from Git
        stage('Git Checkout') {
            steps {
                git branch: 'main', credentialsId: 'git-cred', url: 'https://github.com/jaiswaladi246/Multi-Tier-
BankApp-CI.git'
            }
        }

        // Compile the code
        stage('Compile') {
            steps {
                sh "mvn compile"
            }
        }

        // Run unit tests
```

```
        stage('Test') {
            steps {
                sh "mvn test"
            }
        }

        // Perform a filesystem security scan with Trivy
        stage('Trivy FS Scan') {
            steps {
                sh "trivy fs --format table -o fs.html ."  // Scan filesystem and save report as HTML
            }
        }

        // Perform static code analysis using SonarQube
        stage('SonarQube Analysis') {
            steps {
                withSonarQubeEnv('sonar') {
                    sh ''' $SCANNER_HOME/bin/sonar-scanner -Dsonar.projectName=BoardGame -
Dsonar.projectKey=BoardGame \
                        -Dsonar.java.binaries=. '''
                }
            }
        }

        // Wait for the SonarQube Quality Gate result
        stage('Quality Gate') {
            steps {
                script {
                  waitForQualityGate abortPipeline: false, credentialsId: 'sonar-token'
                }
            }
        }

        // Build the Maven project
        stage('Build') {
            steps {
                sh "mvn package -DskipTests=true"  // Build the package without running tests
            }
        }

        // Publish the package to Nexus Repository
        stage('Publish To Nexus') {
            steps {
                withMaven(globalMavenSettingsConfig: 'global-settings', jdk: 'jdk17', maven: 'maven3',
traceability: true) {
                    sh "mvn deploy"  // Deploy the artifact to Nexus
                }
            }
```

```
    }

    // Build and push the Docker image
    stage('Docker') {
        steps {
            script {
                withDockerRegistry(credentialsId: 'docker-cred') {
                    // Build Docker image with the provided tag and push to Docker registry
                    sh "docker build -t adijaiswal/bankapp:${params.DOCKER_TAG} ."
                    sh "docker push adijaiswal/bankapp:${params.DOCKER_TAG}"
                }
            }
        }
    }

    // Update the image tag in the YAML manifest in another repository
    stage('Update YAML Manifest in Other Repo') {
        steps {
            script {
                withCredentials([gitUsernamePassword(credentialsId: 'git-cred', gitToolName: 'Default')]) {
                    sh '''
                        # Clone the repo containing the Kubernetes deployment manifest
                        git clone https://www.github.com/jaiswaladi246/Multi-Tier-BankApp-CD.git
                        cd Multi-Tier-BankApp-CD

                        # Check if the bankapp-ds.yml file exists
                        ls -l bankapp

                        # Update the Docker image tag in the YAML file
                        sed -i 's|image: adijaiswal/bankapp:.*|image: adijaiswal/bankapp:'${DOCKER_TAG}'|'
$(pwd)/bankapp/bankapp-ds.yml
                    '''

                    // Confirm the change in the updated YAML file
                    sh '''
                        echo "Updated YAML file contents:"
                        cat Multi-Tier-BankApp-CD/bankapp/bankapp-ds.yml
                    '''

                    // Configure Git for the commit
                    sh '''
                        cd Multi-Tier-BankApp-CD
                        git config user.email "office@devopsshack.com"
                        git config user.name "DevOps Shack"
                    '''

                    // Commit and push the changes back to the repository
                    sh '''
```

```
                    cd Multi-Tier-BankApp-CD
                    git add bankapp/bankapp-ds.yml
                    git commit -m "Update image tag to ${DOCKER_TAG}"
                    git push origin main
                '''
            }
        }
    }
  }
}
```

Step 8: Set Up ArgoCD on EKS

ArgoCD is a declarative, GitOps-based continuous delivery tool for Kubernetes. Here's how to set it up on your EKS cluster:

1. Install ArgoCD CLI

First, install the ArgoCD command-line interface (CLI) to interact with ArgoCD from your local environment or the EC2 instance:

Download ArgoCD CLI

```
sudo curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/latest/download/argocd-linux-amd64
```

```
# Make it executable
sudo chmod +x /usr/local/bin/argocd
```

```
# Verify the installation
argocd version
```

2. Install ArgoCD in the Kubernetes Cluster

To deploy ArgoCD in your EKS cluster, follow these steps:

1. **Create a Namespace for ArgoCD:**

```
kubectl create namespace argocd
```

2. **Install ArgoCD in the argocd Namespace:**

Use the following command to install ArgoCD using the official manifests:

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

3. **Verify the ArgoCD Installation:**

You can verify the installation by checking the status of ArgoCD pods:

```
kubectl get pods -n argocd
```

This will show a list of ArgoCD components such as argocd-server, argocd-repo-server, argocd-application-controller, and argocd-dex-server.

3. Expose the ArgoCD Server

By default, ArgoCD is not accessible from outside the cluster. To expose the ArgoCD UI, you can use a LoadBalancer service or a port-forward command.

Option 1: Port-Forwarding (Quick Setup)

1. Run the following command to access the ArgoCD UI via localhost:8080:

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

2. Access ArgoCD's web UI using your browser at:

```
https://localhost:8080
```

Option 2: Expose ArgoCD with LoadBalancer (For Public Access)

1. Edit the ArgoCD argocd-server service to switch from ClusterIP to LoadBalancer:

```
kubectl edit svc argocd-server -n argocd
```

2. In the spec section, change type: ClusterIP to type: LoadBalancer and save the changes. This will make the ArgoCD UI accessible via an external IP address.
3. Retrieve the external IP address:

```
kubectl get svc argocd-server -n argocd
```

Now, you can access ArgoCD UI at:

```
https://<EXTERNAL-IP>
```

4. Access ArgoCD Web UI

1. **Get ArgoCD Admin Password:**

The default password for the admin user is stored as a secret. Retrieve it using the following command:

```
kubectl get secret argocd-initial-admin-secret -n argocd -o jsonpath="{.data.password}" | base64 -d
```

This will return the initial password. You can log in to the ArgoCD UI with the username admin and this password.

2. **Login to ArgoCD via CLI (Optional):**

If you want to use the ArgoCD CLI, log in using the following command:

```
argocd login <ARGOCD_SERVER_IP>
```

Replace <ARGOCD_SERVER_IP> with either localhost:8080 (for port-forward) or the external IP (if you used a LoadBalancer).

Authenticate using:

- Username: admin
- Password: (retrieved from the secret above)

5. Connect a GitHub Repository to ArgoCD

Now that ArgoCD is installed, the next step is to connect your GitHub repository with Kubernetes manifests or Helm charts.

1. **Create a New ArgoCD Application:**

```
argocd app create <app-name> \
--repo <repository-url> \
--path <directory-in-repo> \
--dest-server https://kubernetes.default.svc \
--dest-namespace <k8s-namespace>
```

Replace:

- <app-name> with your application's name.
- <repository-url> with the URL of your GitHub repository.
- <directory-in-repo> with the directory in your repo that contains Kubernetes manifests or Helm charts.
- <k8s-namespace> with the Kubernetes namespace where the app should be deployed.

2. **Sync the Application to Deploy:**

After creating the application, sync it with the following command to deploy your application to Kubernetes:

```
argocd app sync <app-name>
```

3. **Monitor the Application:**

Check the status of the application deployment using:

```
argocd app get <app-name>
```

6. Automate Syncing with GitOps

By default, ArgoCD follows the GitOps model, where changes to your Git repository automatically sync with your Kubernetes cluster.

1. **Enable Auto-Sync:**

You can enable automatic syncing of the application with the following command:

```
argocd app set <app-name> --sync-policy automated
```

This ensures that any new changes in the Git repository are automatically deployed to your Kubernetes cluster without manual intervention.

Step 9 DNS Mapping with GoDaddy Domain

To map a load balancer domain with a GoDaddy domain, you'll need to follow these steps:

1. Get the Load Balancer DNS Name

- If you're using AWS, go to the EC2 Dashboard > Load Balancers.
- Select your load balancer and find the DNS name under the Description tab.
- Copy this DNS name as you'll need it when configuring your GoDaddy domain.

2. Log In to GoDaddy

- Go to the GoDaddy website and log in to your account.

3. Access Domain Settings

- Once logged in, go to the Domains section and select the domain name you want to map to your load balancer.
- Click on the domain name to access its settings.

4. Manage DNS Settings

- Scroll down to the DNS Settings section.
- Click on Manage DNS to open the DNS management page.

5. Add a CNAME Record

- In the DNS management page, under the Records section, click on Add to create a new DNS record.
- Choose CNAME as the type of record.
- In the Host field, enter the subdomain you want to use (e.g., www or app).

- In the Points to field, paste the DNS name of your load balancer that you copied earlier.
- Set the TTL (Time To Live) value (the default is usually fine).
- Click Save to apply the changes.

6. Update the A Record (Optional)

- If you want to map the root domain (e.g., example.com), you may need to update the A Record.
- Instead of a CNAME, add or modify the existing A Record to point to the IP address of your load balancer (if available) or use a service like AWS Route 53 to map the root domain to the load balancer.

7. Wait for DNS Propagation

- DNS changes can take some time to propagate, usually within a few minutes to 24 hours.
- You can use tools like What's My DNS to check if the changes have propagated globally.

8. Test the Setup

- Once DNS propagation is complete, you should be able to access your application using your GoDaddy domain name mapped to the load balancer.

Conclusion

In this project, we successfully set up a complete CI/CD pipeline using various DevOps tools on an AWS EKS cluster. We started by configuring essential services like Jenkins, SonarQube, Nexus, and Prometheus to ensure seamless code integration, quality checks, artifact management, and system monitoring. By setting up an EKS cluster, deploying applications with Kubernetes, and automating the infrastructure with GitOps using ArgoCD, we built a robust and scalable environment for continuous delivery.

The integration of ArgoCD enabled a powerful GitOps-based approach, allowing for declarative management of application deployments through version-controlled manifests. This setup ensures that our cluster is always in sync with our GitHub repository, promoting automation, transparency, and reliability across the entire software delivery lifecycle.

Overall, this project demonstrated the value of modern DevOps practices, enabling rapid and stable deployments while enhancing collaboration and consistency in the development process. The combination of continuous integration, automated testing, deployment, and monitoring has laid a solid foundation for maintaining high-quality production applications with ease and scalability.