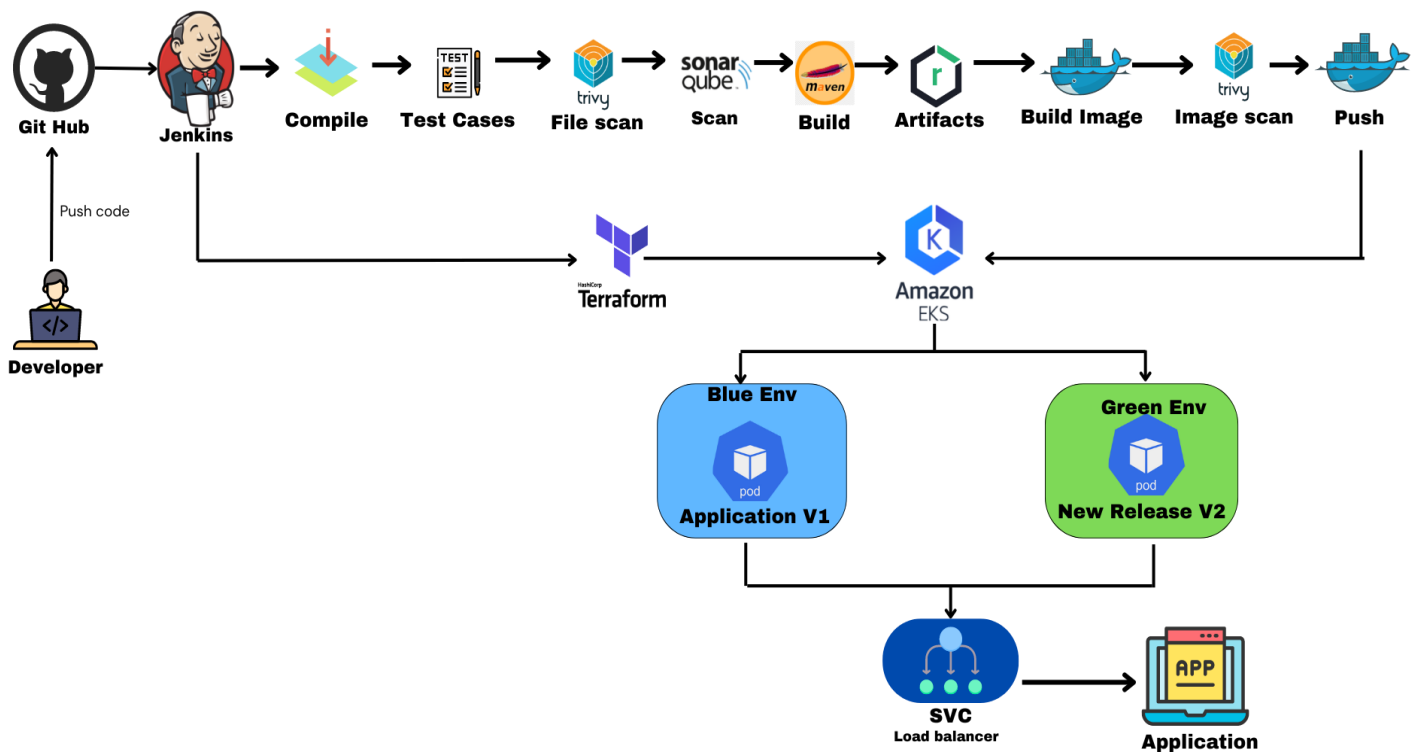




DevOps Shack

Production Level Blue-Green Deployment CICD Blue-Green Deployment

Production Level Blue-Green Deployment CICD Blue-Green Deployment is a strategy designed to reduce downtime and minimize risks during software updates. In this approach, two identical environments are maintained: one is the "Blue" environment, which serves the live production traffic, while the other is the "Green" environment, which hosts the new release. When deploying a new version, the release is deployed to the Green environment, which is completely separate from the current production system. After



thorough testing and validation, the production traffic is routed to the Green environment. If any issues arise, the system can quickly revert back to the Blue environment, allowing for instant rollback and minimal disruption.

Why Use a Blue-Green Environment?

1. **Zero Downtime Deployments:** Blue-Green Deployment ensures that there is no downtime when releasing new features or updates. Users can continue to interact with the system seamlessly while the new release is deployed and tested in the Green environment.
2. **Risk Mitigation:** If a release contains bugs or misconfigurations, switching back to the previous Blue environment is immediate. This ensures that any potential risk or failure introduced by the new version does not affect the users.
3. **Safe and Controlled Rollbacks:** Rollbacks are often challenging in traditional deployment strategies. In a Blue-Green environment, reverting back to the previous version is as simple as switching traffic back to the Blue environment, making the process faster and safer.
4. **Quick Validation and Testing:** The Green environment allows for testing the new release under real-world conditions before it goes live. It can include smoke tests, user acceptance testing (UAT), and load testing without affecting the current production system.
5. **Improved Deployment Flexibility:** Teams have more flexibility to release new features or updates at any time. Since the two environments are independent, deployments can happen in a staged and controlled manner without rushing through updates.
6. **Efficient Use of Resources:** Blue-Green environments encourage efficient infrastructure management by allowing teams to switch between environments without rebuilding or reconfiguring systems every time.

Project Overview: Blue-Green Deployment using Jenkins and Kubernetes on EKS with Terraform

This project outlines the setup and execution of a **Blue-Green Deployment** strategy for an application running on a **Kubernetes** cluster (EKS) using **Terraform** for provisioning. It also includes the configuration of **Jenkins**, **SonarQube**, **Nexus**, and Kubernetes **RBAC** (Role-Based Access Control), followed by the creation of a pipeline in Jenkins to manage the Blue-Green deployment.

Steps Overview:

1. Set up EKS Cluster using Terraform:

- We will use Terraform to provision an Amazon Elastic Kubernetes Service (EKS) cluster on AWS.
- Terraform will handle the infrastructure as code (IaC), setting up the cluster, configuring worker nodes, networking, and other necessary AWS resources.
- This allows us to version-control the infrastructure and easily modify or scale the cluster as needed.

2. Set up Jenkins, SonarQube, and Nexus:

- **Jenkins:** A CI/CD tool that will manage the build, test, and deployment pipelines. Jenkins will be installed on an EC2 instance or on a Kubernetes pod inside the EKS cluster.
- **SonarQube:** A code quality management tool that will integrate with Jenkins to scan the code for bugs, security vulnerabilities, and code smells.
- **Nexus:** A repository manager that will act as the central hub for storing and managing build artifacts (Docker images, Maven dependencies, etc.).
- Jenkins will be configured to use both SonarQube and Nexus in the pipeline, ensuring code quality and artifact management.

3. Set up RBAC for Kubernetes:

- **Role-Based Access Control (RBAC)** will be configured in Kubernetes to manage permissions and access control.
- We will define **roles**, **role bindings**, and **service accounts** that will control which users and systems can perform specific actions in the cluster.
- This ensures secure access to the Kubernetes API and resources for Jenkins, developers, and other services.

4. Create a Pipeline Job in Jenkins for Blue-Green Deployment:

- Once the environment is ready, we will create a **Jenkins pipeline** that automates the Blue-Green deployment strategy.
- The pipeline will:
 - Build the application and create a Docker image.
 - Run the code quality checks using SonarQube.
 - Push the image to Nexus.
 - Deploy the new release to the **Green** environment (a separate Kubernetes deployment).
 - Test the new version in the Green environment (e.g., smoke testing).
 - Switch traffic from the **Blue** environment (existing version) to the **Green** environment using a load balancer.
 - Provide an option for instant rollback to the Blue environment if issues arise.

Key Tools Involved:

- **Terraform**: For provisioning the EKS cluster.
- **Jenkins**: For automating the CI/CD pipeline.
- **SonarQube**: For static code analysis and code quality checks.
- **Nexus**: For artifact storage and management.
- **Kubernetes**: To manage the application deployment using a Blue-Green deployment strategy.

- **AWS EKS:** The managed Kubernetes service on AWS, which hosts the Kubernetes cluster.

Step 1 .Setup EKS Cluster Using terraform

Create a Virtual Machine on AWS

SSH into the VM and Run the command to install Terraform

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

AWSCLI

Download AWS CLI on 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
```

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
```

EKSCTL

```
curl --silent --location
```

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

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

```
eksctl version
```

Save all the script in a file, for example, ctl.sh, and make it executable using:

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

Create Terraform files

1.main.tf

2.output.tf

3.variable.tf

Terraform Code -> [Click here](#)

Run the command

```
terraform init
```

```
terraform plan
```

```
terraform apply -auto approve
```

Create EKS Cluster

```
eksctl create cluster --name=EKS-1 \
```

```
--region=ap-south-1 \
```

```
--zones=ap-south-1a,ap-south-1b \
```

```
--without-nodegroup
```

Open ID Connect

```
eksctl utils associate-iam-oidc-provider \
```

```
--region ap-south-1 \
```

```
--cluster EKS-1 \
```

```
--approve
```

Create node Group

```
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
```

Make sure to change the name of ssh-public-Key with your SSH key.

Step 2. Launch Virtual Machine for Jenkins, Sonarqube and Nexus

Here is a detailed list of the basic requirements and setup for the EC2 instance i have used for running Jenkins, including the specifics of the instance type, AMI, and security groups.

EC2 Instance Requirements and Setup:

1. Instance Type

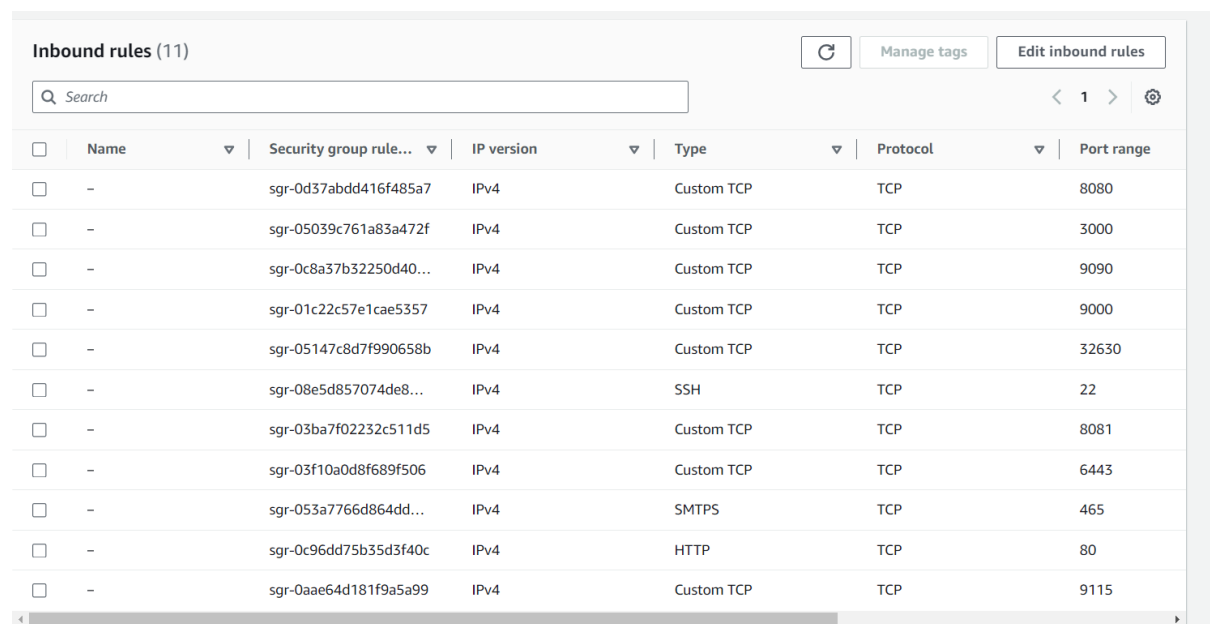
- Instance Type: `t2.large`
- vCPUs: 2
- Memory: 8 GB
- Network Performance: Moderate

2. Amazon Machine Image (AMI)

- AMI: Ubuntu Server 20.04 LTS (Focal Fossa)

3. Security Groups

Security groups act as a virtual firewall for your instance to control inbound and outbound traffic.



Inbound rules (11)							Refresh	Manage tags	Edit inbound rules
Search							< 1 >	Settings	
<input type="checkbox"/>	Name	Security group rule...	IP version	Type	Protocol	Port range			
<input type="checkbox"/>	-	sgr-0d37abdd416f485a7	IPv4	Custom TCP	TCP	8080			
<input type="checkbox"/>	-	sgr-05039c761a83a472f	IPv4	Custom TCP	TCP	3000			
<input type="checkbox"/>	-	sgr-0c8a37b32250d40...	IPv4	Custom TCP	TCP	9090			
<input type="checkbox"/>	-	sgr-01c22c57e1cae5357	IPv4	Custom TCP	TCP	9000			
<input type="checkbox"/>	-	sgr-05147c8d7f990658b	IPv4	Custom TCP	TCP	32630			
<input type="checkbox"/>	-	sgr-08e5d857074de8...	IPv4	SSH	TCP	22			
<input type="checkbox"/>	-	sgr-03ba7f02232c511d5	IPv4	Custom TCP	TCP	8081			
<input type="checkbox"/>	-	sgr-03f10a0d8f689f506	IPv4	Custom TCP	TCP	6443			
<input type="checkbox"/>	-	sgr-053a7766d864dd...	IPv4	SMTPS	TCP	465			
<input type="checkbox"/>	-	sgr-0c96dd75b35d3f40c	IPv4	HTTP	TCP	80			
<input type="checkbox"/>	-	sgr-0aae64d181f9a5a99	IPv4	Custom TCP	TCP	9115			

After Launching your Virtual machine ,SSH into the Server.

Step3. Installing Jenkins on Ubuntu

Execute these commands on Jenkins Server

```
#!/bin/bash
# Install OpenJDK 17 JRE Headless
sudo apt install openjdk-17-jre-headless -y
# Download Jenkins GPG key
sudo wget -O /usr/share/keyrings/jenkins-keyring.asc \
https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key
# Add Jenkins repository to package manager sources
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
# Update package manager repositories
sudo apt-get update
# Install Jenkins
sudo apt-get install jenkins -y
```

Save this script in a file, for example, `install_jenkins.sh`, and make it executable using:

```
chmod +x install_jenkins.sh
```

Then, you can run the script using:

```
./install_jenkins.sh
```

This script will automate the installation process of OpenJDK 17 JRE Headless and Jenkins.

Install docker for future use

Execute these commands on Jenkins, SonarQube and Nexus Servers

```
#!/bin/bash
# Update package manager repositories
sudo apt-get update
# Install necessary dependencies
sudo apt-get install -y ca-certificates curl
# Create directory for Docker GPG key
sudo install -m 0755 -d /etc/apt/keyrings
# Download Docker's GPG key
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
# Ensure proper permissions for the key
sudo chmod a+r /etc/apt/keyrings/docker.asc
# Add Docker repository to Apt sources
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
# Update package manager repositories
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-
plugin
docker-compose-plugin
```

Save this script in a file, for example, `install_docker.sh`, and make it executable using:

```
chmod +x install_docker.sh
```

Then, you can run the script using:

```
./install_docker.sh
```

Setup Nexus

Execute these commands on Nexues VM

```
#!/bin/bash
# Update package manager repositories
sudo apt-get update
# Install necessary dependencies
sudo apt-get install -y ca-certificates curl
# Create directory for Docker GPG key
sudo install -m 0755 -d /etc/apt/keyrings
# Download Docker's GPG key
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
# Ensure proper permissions for the key
sudo chmod a+r /etc/apt/keyrings/docker.asc
# Add Docker repository to Apt sources
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
# Update package manager repositories
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-
plugin
docker-compose-plugin
```

Save this script in a file, for example, `install_docker.sh`, and make it executable using:

```
chmod +x install_docker.sh
```

Then, you can run the script using:

```
./install_docker.sh
```

Create Nexus using docker container

To create a Docker container running Nexus 3 and exposing it on port 8081, you can

use the following command:

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

This command does the following:

- -d: Detaches the container and runs it in the background.
- --name nexus: Specifies the name of the container as "nexus".
- -p 8081:8081: Maps port 8081 on the host to port 8081 on the container, allowing access to Nexus through port 8081.
- sonatype/nexus3:latest: Specifies the Docker image to use for the container, in this case, the latest version of Nexus 3 from the Sonatype repository.

After running this command, Nexus will be accessible on your host machine at <http://IP:8081>.

Get Nexus initial password

Your provided commands are correct for accessing the Nexus password stored in the

container. Here's a breakdown of the steps:

1. **Get Container ID:** You need to find out the ID of the Nexus container. You can do this by running:

```
docker ps
```

This command lists all running containers along with their IDs, among other information.

2. **Access Container's Bash Shell:** Once you have the container ID, you can execute the docker exec command to access the container's bash shell:

```
docker exec -it <container_ID> /bin/bash
```

Replace **<container_ID>** with the actual ID of the Nexus container.

3. **Navigate to Nexus Directory:** Inside the container's bash shell, navigate to the directory where Nexus stores its configuration:

```
cd sonatype-work/nexus3
```

4. **View Admin Password:** Finally, you can view the admin password by displaying the contents of the admin.password file:

```
cat admin.password
```

5. **Exit the Container Shell:** Once you have retrieved the password, you can exit the container's bash shell:

```
exit
```

This process allows you to access the Nexus admin password stored within the container.

SetUp SonarQube

Execute these commands on SonarQube VM

```
#!/bin/bash
# Update package manager repositories
sudo apt-get update
# Install necessary dependencies
sudo apt-get install -y ca-certificates curl
# Create directory for Docker GPG key
sudo install -m 0755 -d /etc/apt/keyrings
# Download Docker's GPG key
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
/etc/apt/keyrings/docker.asc
# Ensure proper permissions for the key
sudo chmod a+r /etc/apt/keyrings/docker.asc
# Add Docker repository to Apt sources
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
# Update package manager repositories
sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-buildx-
plugin
docker-compose-plugin
```

Save this script in a file, for example, `install_docker.sh`, and make it executable using:

```
chmod +x install_docker.sh
```

Then, you can run the script using:

```
./install_docker.sh
```

Create Sonarqube Docker container

To run SonarQube in a Docker container with the provided command, you can follow

these steps:

1. Open your terminal or command prompt.
2. Run the following command:

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

This command will download the sonarqube:lts-community Docker image from Docker

Hub if it's not already available locally. Then, it will create a container named "sonar"

from this image, running it in detached mode (-d flag) and mapping port 9000 on the

host machine to port 9000 in the container (-p 9000:9000 flag).

3. Access SonarQube by opening a web browser and navigating to <http://VmIP:9000>.

This will start the SonarQube server, and you should be able to access it using the

provided URL. If you're running Docker on a remote server or a different port, replace localhost with the appropriate hostname or IP address and adjust the port accordingly.

Step 4 .Install and Configure Plugins on Jenkins

Plugins:

1. SonarQube Scanner
2. Config file provider
3. Maven Integration
4. Pipeline maven integration.
5. Kubernetes
6. Kubernetes Client API
7. Kubernetes Credentials
8. Kubernetes CLI
9. Docker
10. Docker Pipeline
11. Pipeline Stage View
12. Eclipse Temurin Installer

You need to create credentials for Docker and GitHub access.

Create Docker Credentials:

- Go to Manage Jenkins > Manage Credentials > (global) > Add Credentials.
- Choose Username with password as the kind.
- ID: docker-cred
- Username: Your Docker Hub username.
- Password: Your Docker Hub password.
- Click OK.

Create GitHub Credentials:

- Go to Manage Jenkins > Manage Credentials > (global) > Add Credentials.
- Choose Secret text as the kind.
- ID: git-cred
- Secret: Your GitHub Personal Access Token.
- Click OK.

Step 5. Setup RBAC

Create Service Account, Role & Assign that role, And create a secret for Service Account and generate a Token. We will Deploy our Application on the main branch .

Create a file : Vim svc.yml

Creating Service Account

```
apiVersion: v1
```

```
kind: ServiceAccount
```

```
metadata:
```

```
  name: jenkins
```

```
  namespace: webapps
```

To run the svc.yml : kubectl apply -f svc.yaml

Similarly create a role.yml file

Create Role

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: Role
```

```
metadata:
```

```
  name: app-role
```

```
  namespace: webapps
```

```
rules:
```

```
  - apiGroups:
```

```
    - ""
```

```
    - apps
```


- autoscaling

- batch

- extensions

- policy

- rbac.authorization.k8s.io

resources:

- pods

- componentstatuses

- configmaps

- daemonsets

- deployments

- events

- endpoints

- horizontalpodautoscalers

- ingress

- jobs

- limitranges

- namespaces

- nodes

- pods

- persistentvolumes

- persistentvolumeclaims

- resourcequotas

- replicaset

- replicationcontrollers

- serviceaccounts

- services

```
verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

To run the role.yaml file: `kubectl apply -f role.yaml`

Create Bind

Similarly create a bind.yml file

Bind the role to service account

```
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:
```

```
- namespace: webapps
```

```
  kind: ServiceAccount
```

```
  name: jenkins
```

To run the bind.yaml file: `kubectl apply -f bind.yaml`

Create Token

Similarly create a secret.yml file

```
apiVersion: v1
```

```
kind: Secret
```

```
type: kubernetes.io/service-account-token
```

```
metadata:
```

```
  name: mysecretname
```

```
  annotations:
```

```
    kubernetes.io/service-account.name: Jenkins
```

Kubernates Secret Docker

```
kubectrl create secret docker-registry regcred \
```

```
--docker-server=https://index.docker.io/v1/ \
```

```
--docker-username=adijaiswal \
```

```
--docker-password=XYZ@123 \
```

```
--namespace=webapps
```

Now Run – `kubectrl describe secret mysecretname -n webapps`

Save the Token.

Step 6. Pipeline

```
pipeline {
  agent any

  parameters {
    choice(name: 'DEPLOY_ENV', choices: ['blue', 'green'], description: 'Choose which environment
to deploy: Blue or Green')
    choice(name: 'DOCKER_TAG', choices: ['blue', 'green'], description: 'Choose the Docker image
tag for the deployment')
    booleanParam(name: 'SWITCH_TRAFFIC', defaultValue: false, description: 'Switch traffic
between Blue and Green')
  }

  environment {
    IMAGE_NAME = "adijaiswal/bankapp"
    TAG = "${params.DOCKER_TAG}" // The image tag now comes from the parameter
    KUBE_NAMESPACE = 'webapps'
    SCANNER_HOME= tool 'sonar-scanner'
  }

  stages {
    stage('Git Checkout') {
      steps {
        git branch: 'main', credentialsId: 'git-cred', url: 'https://github.com/jaiswaladi246/3-Tier-
NodeJS-MySQL-Docker.git'
      }
    }

    stage('SonarQube Analysis') {
      steps {
        withSonarQubeEnv( 'sonar') {
          sh "$SCANNER_HOME/bin/sonar-scanner -Dsonar.projectKey=nodejsmysql -
Dsonar.projectName=nodejsmysql"
        }
      }
    }

    stage('Trivy FS Scan') {
      steps {
        sh "trivy fs --format table -o fs.html ."
      }
    }

    stage('Docker build') {
      steps {
        script {
          withDockerRegistry(credentialsId: 'docker-cred') {
```

```

        sh "docker build -t ${IMAGE_NAME}:${TAG} ."
    }
}
}
}

```

```

stage('Trivy Image Scan') {
    steps {
        sh "trivy image --format table -o image.html ${IMAGE_NAME}:${TAG}"
    }
}

```

```

stage('Docker Push Image') {
    steps {
        script {

```

```

            withDockerRegistry(credentialsId: 'docker-cred') {
                sh "docker push ${IMAGE_NAME}:${TAG}"
            }
        }
    }
}

```

```

stage('Deploy SVC-APP') {
    steps {
        script {
            withKubeConfig(caCertificate: "", clusterName: 'devopsshack-cluster', contextName: "",
credentialsId: 'k8-token', namespace: 'webapps', restrictKubeConfigAccess: false, serverUrl:
'https://46743932FDE6B34C74566F392E30CABA.gr7.ap-south-1.eks.amazonaws.com') {
                sh """ if ! kubectl get svc app -n ${KUBE_NAMESPACE}; then
                    kubectl apply -f app-service.yml -n ${KUBE_NAMESPACE}
                fi
            """
        }
    }
}

```

```

stage('Deploy to Kubernetes') {
    steps {
        script {
            def deploymentFile = ""
            if (params.DEPLOY_ENV == 'blue') {
                deploymentFile = 'app-deployment-blue.yml'
            } else {
                deploymentFile = 'app-deployment-green.yml'
            }

```

```

            withKubeConfig(caCertificate: "", clusterName: 'devopsshack-cluster', contextName: "",
credentialsId: 'k8-token', namespace: 'webapps', restrictKubeConfigAccess: false, serverUrl:
'https://46743932FDE6B34C74566F392E30CABA.gr7.ap-south-1.eks.amazonaws.com') {

```

```

sh "kubectl apply -f pv-pvc.yml -n
${KUBE_NAMESPACE}"
sh "kubectl apply -f mysql-ds.yml -n
${KUBE_NAMESPACE}"
sh "kubectl apply -f ${deploymentFile} -n ${KUBE_NAMESPACE}"
}
}
}
}
}

```

```

stage('Switch Traffic') {
  when {
    expression { return params.SWITCH_TRAFFIC }
  }
  steps {
    script {
      def newEnv = params.DEPLOY_ENV

```

```

// Always switch traffic based on DEPLOY_ENV
withKubeConfig(caCertificate: "", clusterName: 'devopsshack-cluster', contextName: "",
credentialsId: 'k8-token', namespace: 'webapps', restrictKubeConfigAccess: false, serverUrl:
'https://46743932FDE6B34C74566F392E30CABA.gr7.ap-south-1.eks.amazonaws.com') {
  sh """
    kubectl patch service app -p "{\"spec\": {\"selector\": {\"app\": \"app\",
\\\"version\\\": \"\" + newEnv + \"\"}}}" -n ${KUBE_NAMESPACE}
    """
  }
  echo "Traffic has been switched to the ${newEnv} environment."
}
}
}
}

```

```

stage('Verify Deployment') {
  steps {
    script {
      def verifyEnv = params.DEPLOY_ENV
      withKubeConfig(caCertificate: "", clusterName: 'devopsshack-cluster', contextName: "",
credentialsId: 'k8-token', namespace: 'webapps', restrictKubeConfigAccess: false, serverUrl:
'https://46743932FDE6B34C74566F392E30CABA.gr7.ap-south-1.eks.amazonaws.com') {
        sh """
          kubectl get pods -l version=${verifyEnv} -n ${KUBE_NAMESPACE}
          kubectl get svc app -n ${KUBE_NAMESPACE}
          """
        }
      }
    }
  }
}

```

}

}

}

Advantages

- **Zero Downtime Deployments:** Seamless traffic switch ensures no user disruption during deployments.
- **Rapid Rollbacks:** Instantly revert to the previous stable version if issues arise.
- **Enhanced Testing:** New releases can be thoroughly tested in a production-like environment before going live.
- **Risk Mitigation:** Isolated environments reduce the risk of impacting live services.
- **Flexibility in Releases:** Allows for controlled and gradual release of updates with minimal risk.

Conclusion

This project successfully demonstrates a robust Blue-Green Deployment strategy using Kubernetes on AWS EKS, provisioned through Terraform. By integrating Jenkins for CI/CD, SonarQube for code quality analysis, and Nexus for artifact management, the deployment pipeline is both automated and secure.

The use of **RBAC** in Kubernetes ensures that access control and permissions are tightly managed, providing a secure and compliant environment for developers and CI/CD pipelines. The implementation of the Blue-Green deployment model minimizes the risks associated with new releases, ensuring that there is zero downtime during deployment and offering a fast rollback option in case of failures.

By leveraging modern DevOps practices and tools, this project exemplifies how to achieve continuous delivery with scalability, security, and reliability. It lays a strong foundation for future enhancements, such as adding automated rollback processes, improved monitoring, and more complex deployment strategies like canary releases.