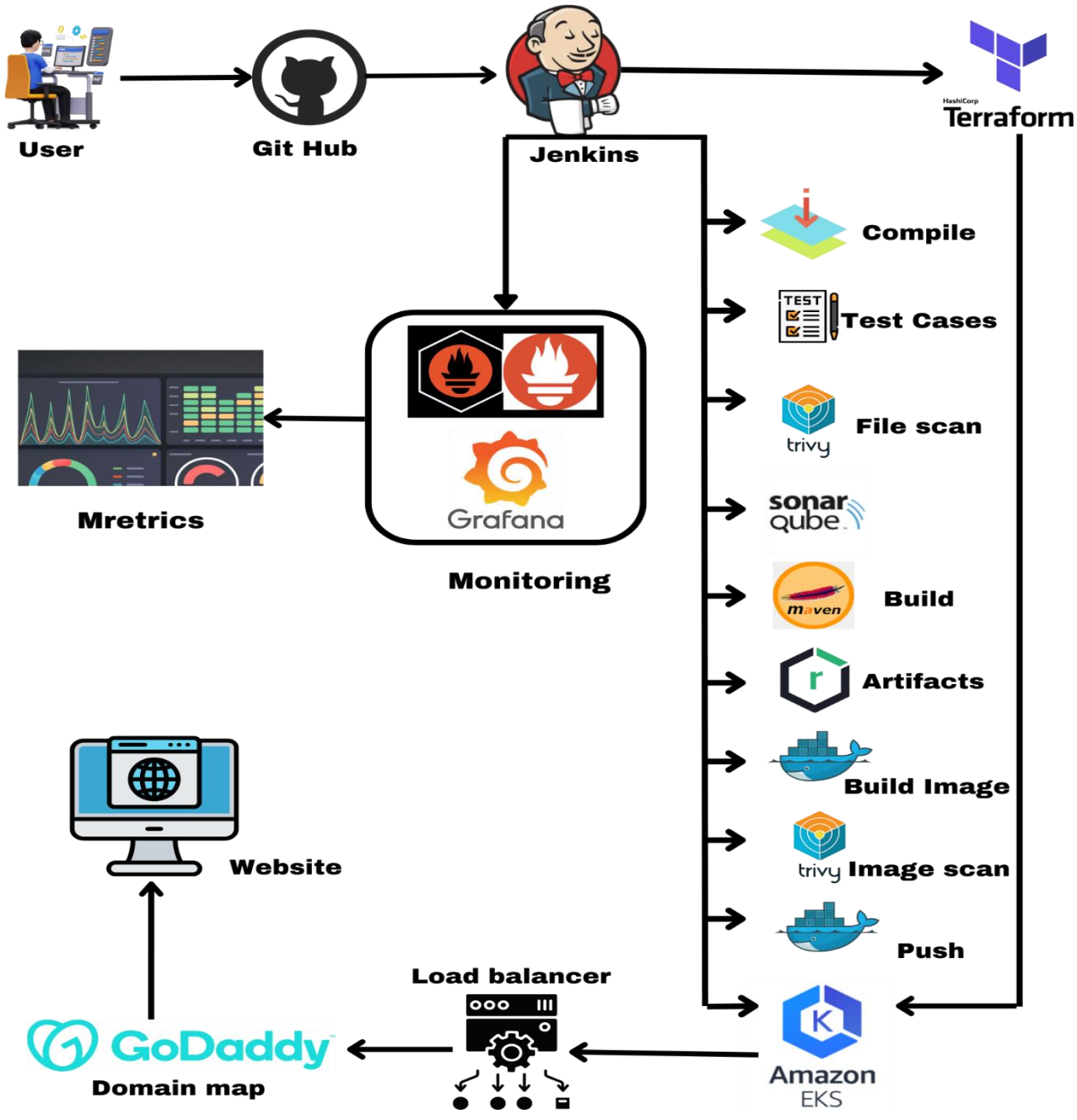
DevOps Shack

Production Level CICD Pipeline Project



Introduction

In the modern software development landscape, Continuous Integration and Continuous Deployment (CI/CD) pipelines are critical for ensuring that code changes are automatically built, tested, and deployed to production environments in a consistent and reliable manner. This document provides a comprehensive guide to setting up a robust CI/CD pipeline using various tools hosted on AWS EC2 instances. The process will cover everything from setting up the necessary infrastructure to deploying an application on an Amazon EKS (Elastic Kubernetes Service) cluster, assigning a custom domain, and monitoring the application to ensure its stability and performance.

The pipeline will incorporate several industry-standard tools:

AWS : Creating virtual machines .

Jenkins for automating the build, test, and deployment processes.

SonarQube for static code analysis to ensure code quality.

Trivy file scan to scan files and vulnerability scanning for Docker images.

Nexus Repository Manager for managing artifacts.

Terraform as infrastructure as code to create EKS Cluster.

Docker: Containerization for consistency and portability.

Kubernetes: Container orchestration for deployment.

Prometheus and Grafana for monitoring the pipeline and application performance.

By following this guide, you'll be able to set up a fully functional CI/CD pipeline that supports continuous delivery and helps maintain high standards for code quality and application performance.

Step 1 : Set Up GitHub Repository and Push Local Code

Create a New GitHub Repository:

1. Login to GitHub:

- Visit GitHub and log in to your account.

2. Create a New Repository:

- Click on the "+" icon in the top-right corner and select "New repository."

- Repository Name: Enter a name for your repository (e.g., `my-ci-cd-project`).

- Description: (Optional) Add a brief description of your project.

- Visibility: Choose between Public or Private.

- Initialize Repository:

- You can choose to add a README file, `.gitignore` file, and select a license, or leave these options unchecked if you're pushing an existing project.

3. Click on "Create repository."

2.2 Push Existing Local Code to GitHub:

1. Initialize Git in Your Local Project:

- Open your terminal or command prompt.

- Navigate to your project directory:

```
cd /path/to/your/project
```

- Initialize Git:

```
git init
```

2. Add Remote Repository:

- Add your GitHub repository as a remote:

```
git remote add origin https://github.com/your-username/my-ci-cd-project.git
```

- Replace `your-username` and `my-ci-cd-project` with your GitHub username and repository name.

3. Add and Commit Your Code:

- Stage all changes:

```
git add .
```

- Commit the changes:

```
git commit -m "Initial commit"
```

4. Push to GitHub:

- Push your code to the `main` branch:

```
git push -u origin main
```

Repository Link

Repository URL Add your GitHub repository link here:

https://github.com/jaiswaladi246/FullStack-Blogging-App.git

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

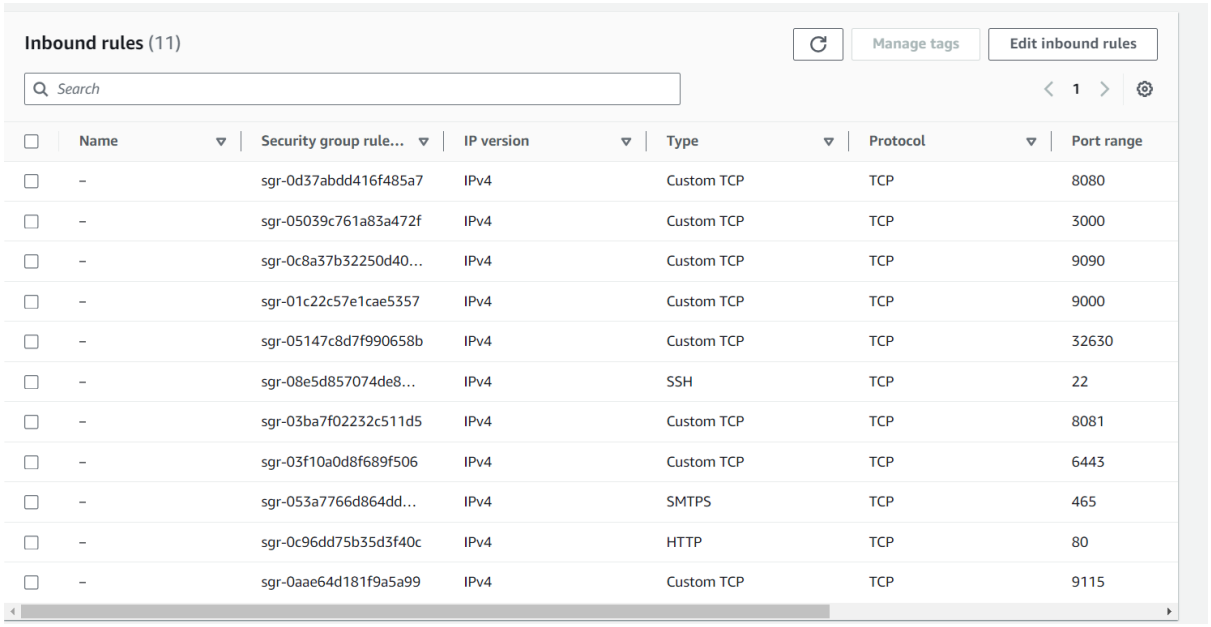
  - Network Performance: Moderate

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.



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 Configur 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

Plugin Configuration –> Check Video

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 .Creating Pipeline

Create a new Pipeline job

```
pipeline {
   agent any

   tools {
      maven 'maven3'
   }

   environment {
      SCANNER_HOME= tool 'sonar-scanner'
   }

   stages {
      stage('Git Checkout') {
         steps {
            git branch: 'main', url: 'https://github.com/jaiswaladi246/Task-
Master-Pro.git'
         }
      }

      stage('Compile') {
         steps {
          sh  'mvn compile'
         }
      }

      stage('Unit-Test') {
         steps {
          sh 'mvn test'
         }
      }

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

```
        stage('Sonar Analysis') {
            steps {
                withSonarQubeEnv('sonar') {
                    sh ''' $SCANNER_HOME/bin/sonar-scanner -
Dsonar.projectName=taskmaster -Dsonar.projectKey=taskmaster \
                    -Dsonar.java.binaries=target '''
                }
            }
        }

        stage('Build Application') {
            steps {
             sh 'mvn package'
            }
        }

        stage('Publish Artifact') {
            steps {
             withMaven(globalMavenSettingsConfig: 'settings-maven', jdk: '',
maven: 'maven3', mavenSettingsConfig: '', traceability: true) {
                    sh 'mvn deploy'
                }
            }
        }

        stage('Docker Build & Tag') {
            steps {
                script {
             withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {
                    sh 'docker build -t adijaiswal/taskmaster:latest .'
                }
            }
        }}

        stage('Trivy Image Scan') {
            steps {
             sh 'trivy image --format table -o image.html
adijaiswal/taskmaster:latest'
            }
        }
```

```
      stage('Docker Push') {
         steps {
            script {
          withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {
                sh 'docker push adijaiswal/taskmaster:latest'
            }
         }
      }}

      stage('K8 Deploy') {
         steps {
          withKubeConfig(caCertificate: '', clusterName: ' devopsshack-cluster',
contextName: '', credentialsId: 'k8-token', namespace: 'webapps',
restrictKubeConfigAccess: false, serverUrl:
'https://1DC375532F6FB38A39069BFC0460C894.gr7.ap-south-
1.eks.amazonaws.com') {
                sh 'kubectl apply -f deployment-service.yml'
                sleep 30
            }
         }
      }

      stage('Verify K8 Deployment') {
         steps {
          withKubeConfig(caCertificate: '', clusterName: ' devopsshack-cluster',
contextName: '', credentialsId: 'k8-token', namespace: 'webapps',
restrictKubeConfigAccess: false, serverUrl:
'https://1DC375532F6FB38A39069BFC0460C894.gr7.ap-south-
1.eks.amazonaws.com') {
                sh 'kubectl get pods -n webapps'
                sh 'kubectl get svc -n webapps'

            }
         }
      }
   }
         post {
   always {
      script {
```

```
        def jobName = env.JOB_NAME
        def buildNumber = env.BUILD_NUMBER
        def pipelineStatus = currentBuild.result ?: 'UNKNOWN'
        def bannerColor = pipelineStatus.toUpperCase() == 'SUCCESS' ? 'green' :
'red'

        def body = """
          <html>
          <body>
          <div style="border: 4px solid ${bannerColor}; padding: 10px;">
          <h2>${jobName} - Build ${buildNumber}</h2>
          <div style="background-color: ${bannerColor}; padding: 10px;">
          <h3 style="color: white;">Pipeline Status:
${pipelineStatus.toUpperCase()}</h3>
          </div>
          <p>Check the <a href="${BUILD_URL}">console output</a>.</p>
          </div>
          </body>
          </html>
        """

        emailext (
          subject: "${jobName} - Build ${buildNumber} -
${pipelineStatus.toUpperCase()}",
          body: body,
          to: 'jaiswaladi246@gmail.com',
          from: 'jenkins@example.com',
          replyTo: 'jenkins@example.com',
          mimeType: 'text/html',
          attachmentsPattern: 'trivy-image-report.html'
        )
      }
    }
}

}
```

Step 6 .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_$(uname -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.

Continuous Deployment

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
    - replicasets
    - 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

```
kubectl create secret docker-registry regcred \
  --docker-server=https://index.docker.io/v1/ \
  --docker-username=adijaiswal \
  --docker-password=XYZ@123 \
        --namespace=webapps
```

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

Save the Token.

-Create a dummy job in your Jenkins with Pipeline job and go to the **pipeline syntax** and select **With Kubernetes:Configure Kubernetes**

1. **Credentials** – Provide the Token that you have saved .
2. **Kubernates Endpoint API**- You can find it in your AWS EKS cluster.
3. **Cluster name**- Provide any name.
4. **NameSpace** – webapps

Click on Generate Syntax.

You will get pipeline syntax :-

```
withKubeCredentials(kubectlCredentials: [[caCertificate: '', clusterName: 'EKS-1', contextName: '', credentialsId: 'k8-token', namespace: 'webapps', serverUrl: 'https://B7C7C20487B2624AAB0AD54DF1469566.yl4.ap-south-1.eks.amazonaws.com']]) {
//block of code
        }
```

Deployment Script

```
      stage('K8 Deploy') {
         steps {
          withKubeConfig(caCertificate: '', clusterName: ' devopsshack-cluster',
contextName: '', credentialsId: 'k8-token', namespace: 'webapps',
restrictKubeConfigAccess: false, serverUrl:
'https://1DC375532F6FB38A39069BFC0460C894.gr7.ap-south-
1.eks.amazonaws.com') {
                sh 'kubectl apply -f deployment-service.yml'
                sleep 30
            }
         }
      }

      stage('Verify K8 Deployment') {
         steps {
          withKubeConfig(caCertificate: '', clusterName: ' devopsshack-cluster',
contextName: '', credentialsId: 'k8-token', namespace: 'webapps',
restrictKubeConfigAccess: false, serverUrl:
'https://1DC375532F6FB38A39069BFC0460C894.gr7.ap-south-
1.eks.amazonaws.com') {
                sh 'kubectl get pods -n webapps'
                sh 'kubectl get svc -n webapps'

            }
         }
      }
   }
        post {
   always {
      script {
         def jobName = env.JOB_NAME
         def buildNumber = env.BUILD_NUMBER
         def pipelineStatus = currentBuild.result ?: 'UNKNOWN'
         def bannerColor = pipelineStatus.toUpperCase() == 'SUCCESS' ? 'green' :
'red'

         def body = """
            <html>
            <body>
```

```
            <div style="border: 4px solid ${bannerColor}; padding: 10px;">
            <h2>${jobName} - Build ${buildNumber}</h2>
            <div style="background-color: ${bannerColor}; padding: 10px;">
            <h3 style="color: white;">Pipeline Status:
${pipelineStatus.toUpperCase()}</h3>
            </div>
            <p>Check the <a href="${BUILD_URL}">console output</a>.</p>
            </div>
            </body>
            </html>
          """

          emailext (
            subject: "${jobName} - Build ${buildNumber} -
${pipelineStatus.toUpperCase()}",
            body: body,
            to: 'jaiswaladi246@gmail.com',
            from: 'jenkins@example.com',
            replyTo: 'jenkins@example.com',
            mimeType: 'text/html',
            attachmentsPattern: 'trivy-image-report.html'
          )
        }
    }
}

}
```

Note- This Script is already added in step 5

For better understanding refer video- Click here

Step 7. Monitoring

Prometheus

Links to download Prometheus, Node_Exporter & black Box exporter
https://prometheus.io/download/

Extract and Run Prometheus

After downloading Promethous extract the .tar file

Now Cd into the extracted file and and run

./prometheus &

By default Prometheus runs on Port 9090 and access it using your instance

<IP address>:9090

Similarly download and run Blackbox exporter.

./ backbox_exporter &

Grafana

Links to download Grafana https://grafana.com/grafana/download

OR

Run This code on Monitoring VM to Install Grafana

```
sudo apt-get install -y adduser libfontconfig1 musl
wget https://dl.grafana.com/enterprise/release/grafana-enterprise_10.4.2_amd64.deb
sudo dpkg -i grafana-enterprise_10.4.2_amd64.deb
```

once Installed run

```
sudo /bin/systemctl start Grafana-server
```

by default Grafana runs on port 3000 so access it using instance <IPaddress>:3000

Configure Prometheus

Go inside the Prometheus.yaml file and edit it

```
scrape_configs:
 - job_name: 'blackbox'
   metrics_path: /probe
   params:
    module: [http_2xx]  # Look for a HTTP 200 response.
   static_configs:
    - targets:
      - http://prometheus.io    # Target to probe with http.
      - https://prometheus.io   # Target to probe with https.
      - http://example.com:8080 # Target to probe with http on port 8080.
   relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
```

```
      target_label: instance
    - target_label: __address__
      replacement:<IP address>:9115
```

Replace the IP address with your instance IP address.

After this Restart Prometheus using this command

```
pgrep Prometheus
```

Once you run the above command you will get the Id of Prometheus

then use the id and kill it

```
kill <ID>
```

Add Prometheus as Data sources inside Grafana

Go to Prometheus server > Data Sources

> Prometheus add IPaddress of Prometheus

> Import Dashboard form web .

Conclusion

This CI/CD pipeline project demonstrates the end-to-end process of automating the software development lifecycle, from code integration to deployment and monitoring. By following the steps outlined in this guide, you've successfully:

Set Up a Repository: Established a version-controlled environment where code can be collaboratively managed and tracked.

Configured Necessary Infrastructure: Provisioned AWS EC2 instances and set up essential tools like Jenkins, SonarQube, Nexus, Prometheus, and Grafana to facilitate continuous integration, deployment, and monitoring.

Pushed Local Code to GitHub: Centralized your codebase in a GitHub repository, enabling seamless collaboration and integration with other tools in the pipeline.

Built and Deployed the Application: Leveraged Jenkins to automate the build, test, and deployment processes, ensuring that your application is consistently deployed to a Kubernetes cluster on Amazon EKS.

Monitored Application Performance: Used Prometheus and Grafana to set up a robust monitoring system that tracks the performance and health of your application in real-time.

Assigned a Custom Domain: Mapped your application to a custom domain, making it accessible to users in a production-ready environment.

By integrating these tools and processes, you've created a powerful CI/CD pipeline that enhances code quality, reduces deployment time, and ensures application reliability. This setup not only accelerates the development cycle but also fosters a culture of continuous improvement, where code is regularly integrated, tested, and deployed with minimal manual intervention.

Moving forward, you can extend this pipeline by adding more advanced features, such as automated security testing, blue-green deployments, or canary releases, to further improve the robustness and scalability of your CI/CD processes.