Virtual Lab Experiment – 6

CI/CD Pipeline to Deploy Dockerized Application on Kubernetes using Jenkins, GitHub, Maven, Terraform, and Ansible

🎯 Objective:

To build and deploy a Dockerized Java web application to a Kubernetes cluster on AWS using a fully automated CI/CD pipeline with Jenkins, Maven, GitHub, Terraform, and Ansible.

🛠️ Tools & Technologies Used

- Terraform (IaC)
- Ansible (Configuration Management)
- Docker (Containerization)
- Jenkins (CI/CD)
- GitHub (Source Control)
- Maven (Build Automation)
- Kubernetes (Orchestration)
- AWS (Cloud Infrastructure)

📋 Step-by-Step Procedure

🔹 STEP 1: Clone Java Application from GitHub

1. Use a sample Maven-based Java web app (e.g., Spring Boot).
2. Fork or clone it from GitHub:

   git clone https://github.com/your-username/sample-java-app.git

---

🔹 STEP 2: Provision Infrastructure Using Terraform

1. **Install Terraform**:

   sudo apt-get install terraform

2. **Create Terraform scripts** to:
   - Provision an EC2 instance for Jenkins
   - Create an EKS (Elastic Kubernetes Service) cluster
   - Set up IAM roles, security groups, and networking (VPC, Subnets)

3. **Initialize & Apply Terraform**:

terraform init

terraform plan

terraform apply

```
ubuntu@ip-172-31-37-221:~/sample-java-app/terraform-cicd$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v6.15.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

```
ubuntu@ip-172-31-37-221:~/sample-java-app/terraform-cicd$ terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_eks_cluster.cicd_cluster will be created
  + resource "aws_eks_cluster" "cicd_cluster" {
      + arn                         = (known after apply)
      + bootstrap_self_managed_addons = true
      + certificate_authority       = (known after apply)
      + cluster_id                  = (known after apply)
      + created_at                  = (known after apply)
      + deletion_protection         = (known after apply)
      + endpoint                    = (known after apply)
      + id                          = (known after apply)
      + identity                    = (known after apply)
      + name                        = "cicd-cluster"
      + platform_version            = (known after apply)
      + region                      = "eu-north-1"
      + role_arn                    = (known after apply)
      + status                      = (known after apply)
      + tags_all                    = (known after apply)
      + version                     = (known after apply)

      + access_config (known after apply)

      + compute_config (known after apply)

      + kubernetes_network_config (known after apply)
```

```
ubuntu@ip-172-31-37-221:~/sample-java-app/terraform-cicd$ terraform apply -auto-approve

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_eks_cluster.cicd_cluster will be created
  + resource "aws_eks_cluster" "cicd_cluster" {
      + arn                         = (known after apply)
      + bootstrap_self_managed_addons = true
      + certificate_authority       = (known after apply)
      + cluster_id                  = (known after apply)
      + created_at                  = (known after apply)
      + deletion_protection         = (known after apply)
      + endpoint                    = (known after apply)
      + id                          = (known after apply)
      + identity                    = (known after apply)
      + name                        = "cicd-cluster"
      + platform_version            = (known after apply)
      + region                      = "eu-north-1"
      + role_arn                    = (known after apply)
      + status                      = (known after apply)
      + tags_all                    = (known after apply)
      + version                     = (known after apply)

      + access_config (known after apply)
```

🔹 STEP 3: Configure Jenkins Server Using Ansible

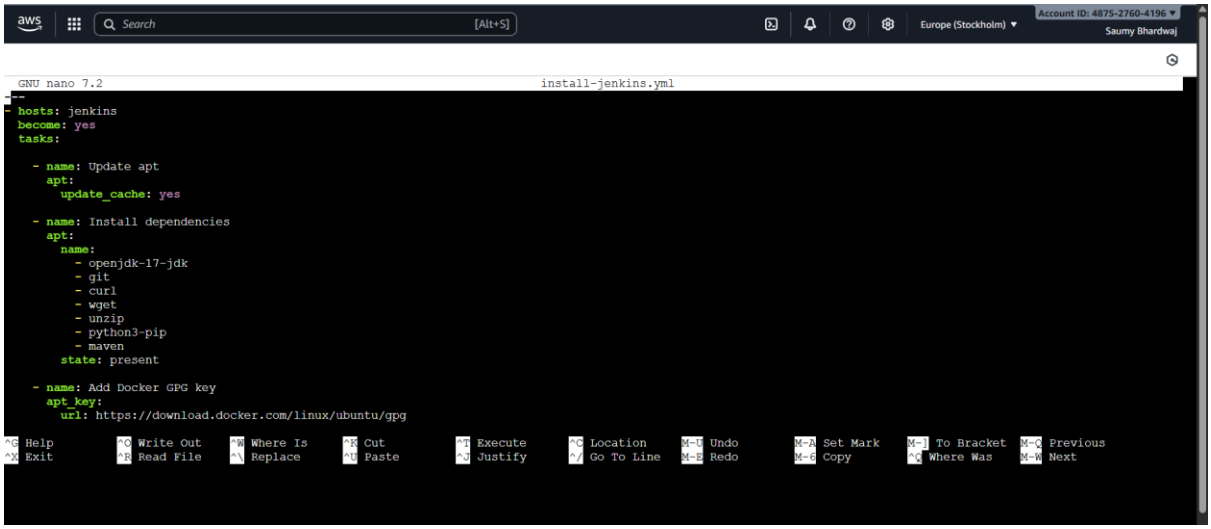
1. **Install Ansible**:

sudo apt install ansible

2. **Create Inventory File** with Jenkins EC2 IP.

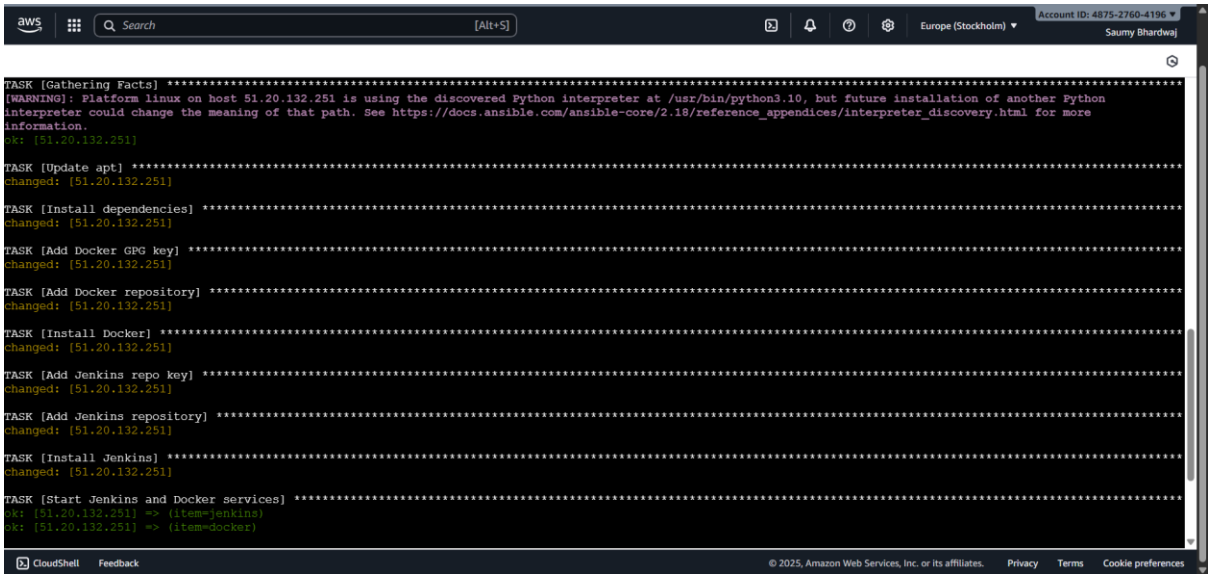
```
ubuntu@ip-172-31-37-221:~/sample-java-app/ansible$ ansible -i inventory.ini jenkins -m ping
[WARNING]: Platform linux on host 51.20.132.251 is using the discovered Python interpreter at /usr/bin/python3.10, but future installation of another Python
interpreter could change the meaning of that path. See https://docs.ansible.com/ansible-core/2.18/reference_appendices/interpreter_discovery.html for more
information.
51.20.132.251 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3.10"
    },
    "changed": false,
    "ping": "pong"
}
```

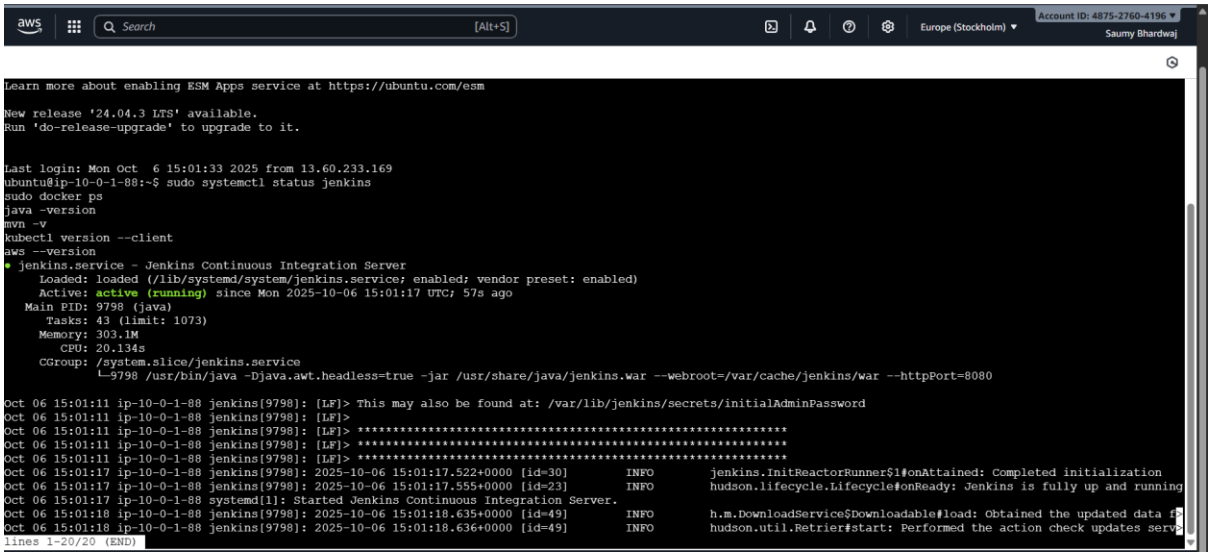
3. **Write Ansible Playbook** to:
    - Install Java, Jenkins, Docker, kubectl, and Maven
    - Start and enable Jenkins service
    - Add Jenkins user to Docker group



4. **Run Playbook**:

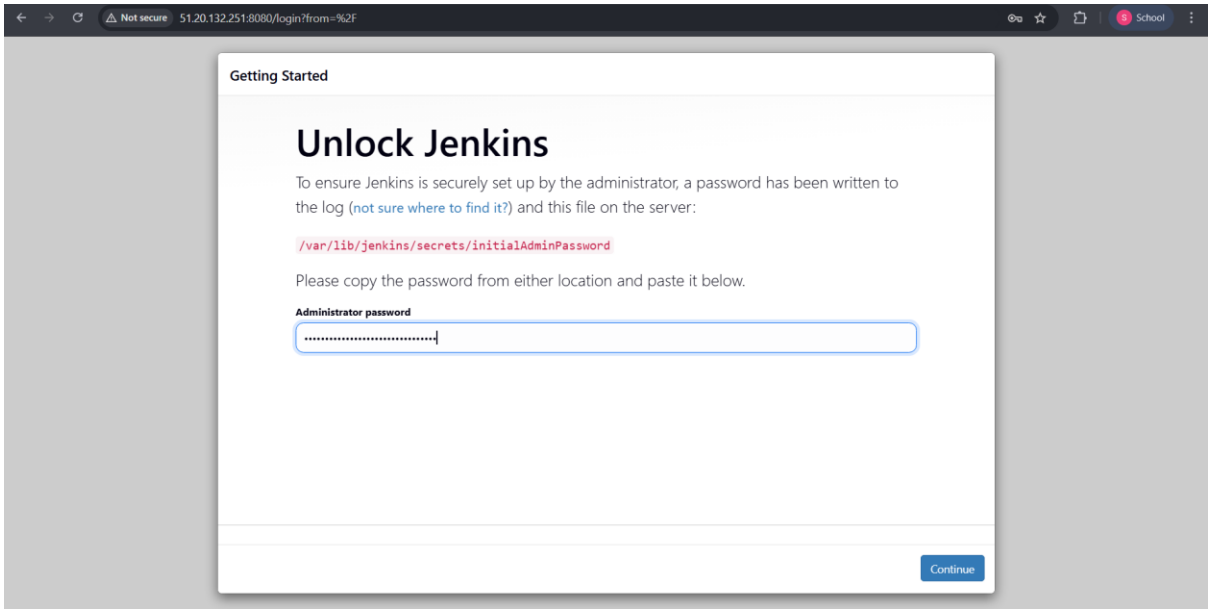
ansible-playbook install-jenkins.yml -i inventory



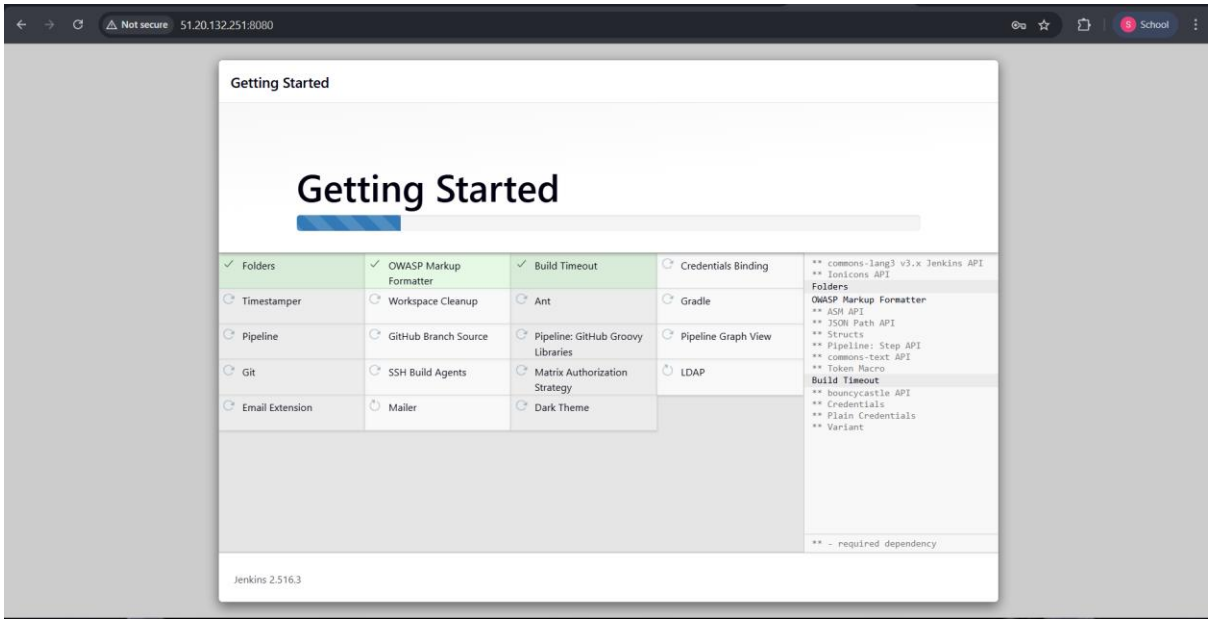


STEP 4: Configure Jenkins for CI/CD

1. Access Jenkins via browser: http://<jenkins-ec2-ip>:8080



2. Install required plugins:
    - GitHub, Maven Integration, Docker, Kubernetes CLI



3. Create Jenkins Credentials for:
    - GitHub
    - Docker Hub
    - AWS IAM Access
4. Create a **new Jenkins Pipeline Job**.

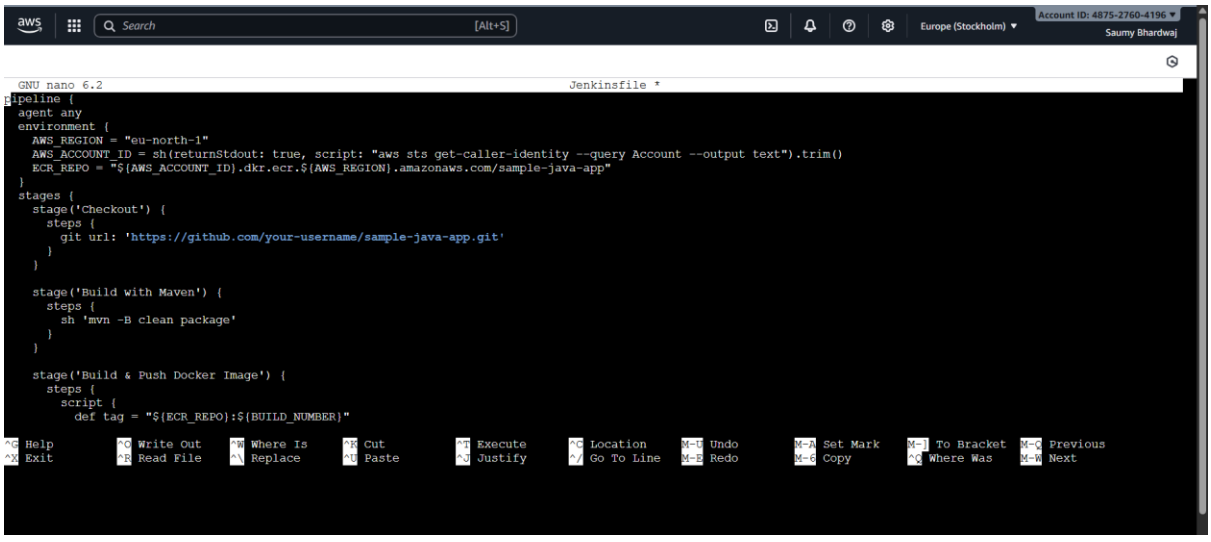
🔹 **STEP 5: Write Jenkinsfile for CI/CD Pipeline**

Jenkinsfile example:

groovy

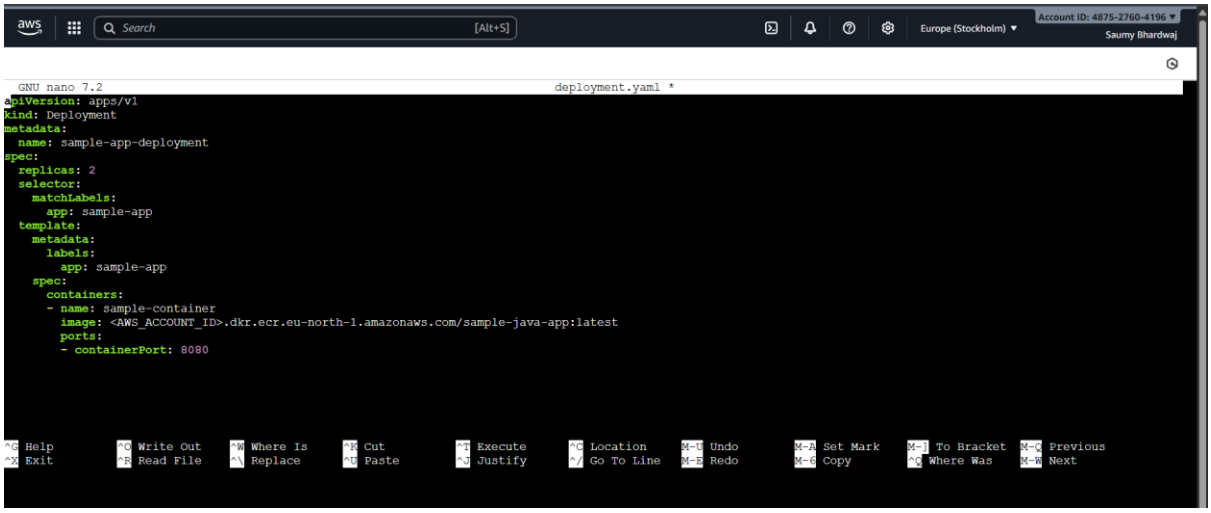
```
pipeline {
  agent any

  tools {
    maven 'Maven 3.8.1'
  }

  environment {
    DOCKER_IMAGE = "yourdockerhubusername/sample-java-app:${BUILD_NUMBER}"
  }

  stages {
    stage('Checkout Code') {
      steps {
        git 'https://github.com/your-username/sample-java-app.git'
      }
```

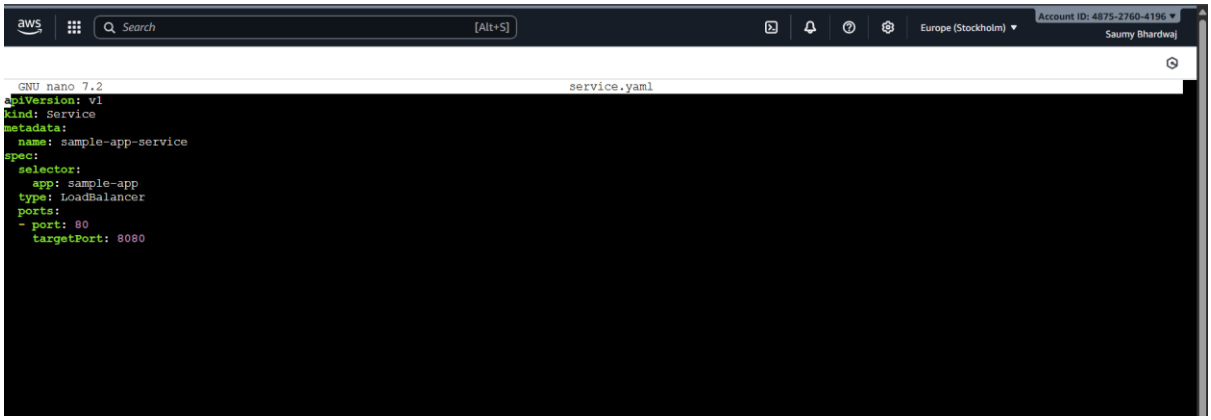
```
        }
        stage('Build with Maven') {
            steps {
                sh 'mvn clean package'
            }
        }
        stage('Build Docker Image') {
            steps {
                sh 'docker build -t $DOCKER_IMAGE .'
            }
        }
        stage('Push to Docker Hub') {
            steps {
                withDockerRegistry([credentialsId: 'docker-hub-creds']) {
                    sh 'docker push $DOCKER_IMAGE'
                }
            }
        }
        stage('Deploy to Kubernetes') {
            steps {
                sh 'kubectl set image deployment/sample-app-deployment sample-
container=$DOCKER_IMAGE'
            }
        }
    }
}
```



- **STEP 6: Create Kubernetes Deployment Files**

1. Write YAML files for:
    - Deployment
    - Service (NodePort/LoadBalancer)





2. Deploy manually once:

   kubectl apply -f deployment.yaml

   kubectl apply -f service.yaml

```
ubuntu@ip-10-0-1-88:~/sample-java-app$ kubectl apply -f k8s/deployment.yaml
deployment.apps/sample-app-deployment created
ubuntu@ip-10-0-1-88:~/sample-java-app$ kubectl apply -f k8s/service.yaml
service/sample-app-service created
ubuntu@ip-10-0-1-88:~/sample-java-app$ kubectl get svc sample-app-service -w
NAME                 TYPE           CLUSTER-IP     EXTERNAL-IP                                                                PORT(S)        AGE
sample-app-service   LoadBalancer   172.20.85.47   a7f74207c45e64ea7b2526bd7a8d3668-479627857.eu-north-1.elb.amazonaws.com   80:31734/TCP   8s
```

- **STEP 7: Verify CI/CD Pipeline Execution**

1. Make a code change in GitHub (e.g., update a string in the Java app).
2. Jenkins will:
   - Pull code
   - Build with Maven
   - Build and push Docker image
   - Deploy new version to Kubernetes
3. Access the app via:

http://<LoadBalancer-DNS or NodePort-IP>

✅ **Expected Outcome:**

- Fully functional CI/CD pipeline
- Automatic code build, Docker packaging, and deployment to Kubernetes on AWS

Rubric (Total: 6 Marks)

Criteria	Excellent (1 Marks)	Good (0.5 Mark)	Needs Improvement (0 Mark)
1. Infrastructure Provisioning (Terraform)	Fully functional AWS infra with EC2 & EKS	Partially functional infra, minor issues	Infrastructure not provisioned or fails entirely
2. Configuration Management (Ansible)	Jenkins and dependencies correctly installed	Partial setup; minor misconfigurations	Jenkins not installed or major errors
3. CI/CD Pipeline in Jenkins	Pipeline works end-to-end with all stages	Pipeline executes partially; minor stage issues	Pipeline fails or is not configured
4. Dockerization & Image Management	Docker image built and pushed to Docker Hub	Image built but push or tag missing	Docker image not built or pushed
5. Kubernetes Deployment	Application deployed and accessible via service	App deployed but not accessible properly	Deployment failed or not attempted
6. Version Control Integration (GitHub)	GitHub repo connected and automated builds work	GitHub integrated but manual triggers used	No GitHub integration or broken webhook