**Assignment – 3: DevOps**

**Build a Scalable Web Application with CI/CD Pipeline**

# Objective:

1. Create a simple web application.  
2. Automate infrastructure provisioning using Infrastructure as Code (IaC).  
3. Build a CI/CD pipeline to deploy the application on a cloud environment.  
4. Containerize the application using Docker.  
5. Monitor the application performance and logs.

# Steps:

## Create a Simple Web Application and Push to GitHub repository

Develop a basic web application (e.g., a simple Node.js app or a Python Flask app). Include a simple API or static HTML page that displays 'Hello World'.

* 1. Create the project
  2. Put it on GitHub
  3. Provide the GitHub link

## Containerize the Application Using Docker

Write a Dockerfile to containerize the application.

**Example Dockerfile for Node.js app:**

*FROM node:14  
WORKDIR /app  
COPY package\*.json ./  
RUN npm install  
COPY . .  
EXPOSE 3000  
CMD ["npm", "start"]*

**Build and run the Docker image locally to ensure it works:**

*docker build -t myapp .  
docker run -p 3000:3000 myapp*

## Write Infrastructure as Code (IaC) Using Terraform/CloudFormation

Use Terraform or AWS CloudFormation to automate the provisioning of cloud infrastructure (e.g., AWS, GCP, or Azure). Provision resources such as Virtual Machines (EC2, GCP VM), a Load Balancer, and Security Groups/Firewall Rules.

**Example for an AWS EC2 instance in Terraform:**

*provider "aws" {  
 region = "us-east-1"  
}  
  
resource "aws\_instance" "app" {  
 ami = "ami-12345678"  
 instance\_type = "t2.micro"  
  
 tags = {  
 Name = "MyAppInstance"  
 }  
}*

## Set Up a CI/CD Pipeline Using Jenkins/GitLab CI

Use Jenkins or GitLab CI to build a pipeline. Configure the pipeline to pull code from your Git repository, build and test the application, build the Docker image, and deploy the Docker container to your cloud infrastructure.

**Example Jenkinsfile:**

*pipeline {  
 agent any  
   
 stages {  
 stage('Build') {  
 steps {  
 sh 'docker build -t myapp .'  
 }  
 }  
 stage('Test') {  
 steps {  
 // Run unit tests here  
 sh 'npm test'  
 }  
 }  
 stage('Deploy') {  
 steps {  
 // Push the Docker image to a container registry  
 sh 'docker tag myapp myregistry/myapp:latest'  
 sh 'docker push myregistry/myapp:latest'  
 }  
 }  
 }  
}*

## Deploy and Scale the Application Using Kubernetes

Deploy the Docker container on a Kubernetes cluster. Define a Deployment and a Service in Kubernetes to manage the app.

**Example Kubernetes YAML file:**

*apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: myapp  
spec:  
 replicas: 2  
 selector:  
 matchLabels:  
 app: myapp  
 template:  
 metadata:  
 labels:  
 app: myapp  
 spec:  
 containers:  
 - name: myapp  
 image: myregistry/myapp:latest  
 ports:  
 - containerPort: 3000  
  
---  
apiVersion: v1  
kind: Service  
metadata:  
 name: myapp-service  
spec:  
 type: LoadBalancer  
 selector:  
 app: myapp  
 ports:  
 - port: 80  
 targetPort: 3000*

Use kubectl to deploy and manage the application.

*kubectl apply -f deployment.yaml*

## Implement Monitoring and Logging

Set up monitoring using Prometheus and Grafana or cloud-native monitoring solutions like AWS CloudWatch or GCP Stackdriver. Add application logging with tools like ELK Stack (Elasticsearch, Logstash, Kibana) or any centralized logging service (e.g., Fluentd with Kubernetes).

## Scale and Autoscale

Use Kubernetes Horizontal Pod Autoscaler (HPA) to automatically scale your application based on CPU or memory usage.

*kubectl autoscale deployment myapp --cpu-percent=50 --min=2 --max=10*

# Key Learning Areas:

1. **Automation**: Use Terraform or CloudFormation to automate the infrastructure.
2. **CI/CD**: Build a Jenkins or GitLab CI pipeline for continuous integration and deployment.
3. **Containerization**: Use Docker to containerize the application.
4. **Orchestration**: Deploy and scale your application using Kubernetes.
5. **Monitoring & Logging**: Set up real-time monitoring and log aggregation.