Solution Development and Testing for Leveraging Docker and Kubernetes for a Multi-Cloud Strategy

College Name: Vijaya Vittala Institute of Technology

Group Members:

• Name: Narasimhamurthy D V CAN ID: 33837191

• Name: P P Jyothsna Sai CAN ID: 34043056

• Name: Mahalakshmi A CAN ID: 33822962

Solution Development and Testing for Leveraging Docker and Kubernetes for a Multi-Cloud Strategy

SECTION 1: SOLUTION DEVELOPMENT

Step 1: Define Multi-Cloud Strategy Objectives

1. **Portability**: Ensure applications can be deployed on multiple cloud platforms (IBM Cloud, AWS, Azure, Google Cloud).
2. **Scalability**: Enable horizontal scaling across cloud environments.
3. **Resilience**: Distribute workloads to ensure high availability.
4. **Security**: Implement strong security policies for containerized workloads.

Step 2: Setup Multi-Cloud Environment

1. Configure IBM Cloud

- Create an IBM Cloud account.
- Set up IBM Cloud Kubernetes Service (IKS) or OpenShift.
- Create IBM Cloud Container Registry for storing images.
- Authenticate IBM Cloud CLI.

Step 3: Containerize Applications Using Docker

1. Create Dockerfiles

Frontend Dockerfile:

```
FROM node:16-alpine

WORKDIR /app

COPY . .

RUN npm install

EXPOSE 3000

CMD ["npm", "start"]
```

Backend Dockerfile:

```
FROM node:16-alpine

WORKDIR /app

COPY . .

RUN npm install

EXPOSE 5000

CMD ["node", "server.js"]
```

2. Build and Tag Docker Images

```
docker build -t frontend-app:1.0 ./public

docker build -t backend-app:1.0 ./server
```

3. Push Images to Multiple Cloud Registries

```
docker tag frontend-app:1.0 <region>.icr.io/<namespace>/frontend-app:1.0

docker tag backend-app:1.0 <region>.icr.io/<namespace>/backend-app:1.0


# IBM Cloud

ibmcloud cr login

docker push <region>.icr.io/<namespace>/frontend-app:1.0
```

docker push <region>.icr.io/<namespace>/backend-app:1.0

Step 4: Deploy Applications Using Kubernetes

1. Create Kubernetes Deployment and Service YAML Files

Frontend Deployment (frontend-deployment.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: <cloud_registry>/frontend-app:1.0
        ports:
        - containerPort: 3000
---
```

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: LoadBalancer
  selector:
    app: frontend
  ports:
  - port: 3000
    targetPort: 3000
```

Backend Deployment (backend-deployment.yaml)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
```

```
    containers:
    - name: backend
      image: <cloud_registry>/backend-app:1.0
      ports:
      - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  type: LoadBalancer
  selector:
    app: backend
  ports:
  - port: 5000
    targetPort: 5000
```

2. Apply Kubernetes Configurations

```
kubectl apply -f frontend-deployment.yaml
kubectl apply -f backend-deployment.yaml
```

SECTION 2: TESTING THE SOLUTION

Step 1: Validate Kubernetes Deployments

```
kubectl get pods
kubectl get svc
```

Step 2: Test Application Accessibility

- Use cloud provider LoadBalancer IPs to access services.

- Verify application responses using curl or browser.

Step 3: Automate CI/CD Pipeline with GitHub Actions

1. Create a .github/workflows/deploy.yml file.
2. Define CI/CD steps for building, testing, and deploying containers.

Step 4: Perform Stress and Load Testing

- Use Apache JMeter to simulate concurrent users.
- Monitor performance using Prometheus and Grafana.

SECTION 3: FUTURE IMPROVEMENTS

1. **Implement Autoscaling**
    - Configure Kubernetes Horizontal Pod Autoscaler (HPA).
2. kubectl autoscale deployment frontend-deployment --cpu-percent=50 --min=1 --max=10
3. **Integrate Advanced CI/CD Pipelines**
    - Use Jenkins, GitHub Actions, or Tekton for automated deployments.
4. **Enhance Security Measures**
    - Enable Kubernetes Network Policies.
    - Implement role-based access control (RBAC).
5. **Multi-Cloud Service Mesh Implementation**
    - Deploy Istio for traffic management and security policies across clouds.

By implementing this solution, businesses can achieve a robust multi-cloud strategy using Docker and Kubernetes, ensuring resilience, scalability, and security across different cloud providers.