Infrastructure as Code (IaC) and CI/CD Pipeline for ***Scalable Multi-Cloud Kubernetes Deployment***

Project Objective:

We will design and implement a scalable multi-cloud Kubernetes deployment using Infrastructure as Code (IaC) tools. The project will include creating Kubernetes clusters, deploying a microservices application, and implementing a CI/CD pipeline for automated application updates.

Project Components:

1. Infrastructure as Code (IaC):

- **Tools**: Terraform, AWS CloudFormation, or Azure Resource Manager. I prefer Terraform for this project.
- **Tasks**:
    - Write Terraform configurations to provision:
        - Kubernetes clusters in AWS (EKS) and Azure (AKS).
        - Networking components such as VPCs, subnets, and security groups.
        - Load balancers for exposing services.
        - Monitoring and logging solutions (CloudWatch for AWS, Azure Monitor for AKS).
    - Implement a Terraform module for reusable infrastructure.

2. Kubernetes Setup:

- **Tools**: kubectl, Helm, Kubernetes Dashboard.
- **Tasks**:
    - Set up Kubernetes clusters in AWS and Azure.
    - Configure namespaces, resource quotas, and limits.
    - Deploy a monitoring stack (e.g., Prometheus + Grafana).
    - Set up centralized logging (e.g., Fluentd or Elasticsearch-Logstash-Kibana).

3. Microservices Deployment:

- **Tools**: Docker, Kubernetes, Helm.
- **Tasks**:
    - Containerize a sample microservices application (e.g., a to-do app or e-commerce app).
    - Create Helm charts for the microservices deployment.
    - Configure horizontal pod autoscaling and rolling updates.

4. CI/CD Pipeline:

- **Tools**: Jenkins/GitHub Actions/GitLab CI/CD.
- **Tasks**:
    - Automate build and testing of Docker images.
    - Push images to a container registry (e.g., AWS ECR, Azure Container Registry).
    - Automate Kubernetes deployment using Helm charts.
    - Implement a Canary deployment strategy to minimize downtime.

5. Observability:

- **Tools**: Prometheus, Grafana, Jaeger.
- **Tasks**:
    - Monitor Kubernetes clusters for resource usage and application performance.
    - Set up alerts for resource thresholds.
    - Implement distributed tracing to visualize microservices interactions.

6. Multi-Cloud Setup:

- **Tasks**:
    - Use multi-cloud load balancers (e.g., AWS Global Accelerator or Azure Traffic Manager) to route traffic.
    - Implement DNS failover to ensure high availability across cloud platforms.

7. Security:

- **Tools**: Vault, Kubernetes RBAC, OPA (Open Policy Agent).
- **Tasks**:
    - Manage secrets and configuration securely using HashiCorp Vault.
    - Enforce RBAC policies for cluster access.
    - Use OPA to validate Kubernetes manifests.

Infrastructure as Code (IaC) with Terraform

This section explains how to provision Kubernetes clusters in AWS (EKS) and Azure (AKS), along with the necessary networking and monitoring components, using Terraform. We'll include Terraform code and explanations for each task.

Prerequisites:

1. Install **Terraform**
2. Configure **AWS CLI** and **Azure CLI**:
    - AWS CLI Configuration.
    - Azure CLI Configuration.

Provision Kubernetes Clusters

Terraform Code for AWS (EKS):

```
provider "aws" {
  region = "us-east-1"
}

# Create VPC
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "3.18.1"

  name = "eks-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["us-east-1a", "us-east-1b"]
  public_subnets  = ["10.0.1.0/24", "10.0.2.0/24"]
  private_subnets = ["10.0.3.0/24", "10.0.4.0/24"]
  enable_nat_gateway = true
}

# Create EKS Cluster
module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  cluster_name    = "eks-cluster"
  cluster_version = "1.24"

  subnets         = module.vpc.private_subnets
  vpc_id          = module.vpc.vpc_id

  node_groups = {
    eks_nodes = {
      desired_capacity = 2
      max_capacity     = 3
      min_capacity     = 1

      instance_type = "t3.medium"
    }
  }
}

output "eks_cluster_endpoint" {
  value = module.eks.cluster_endpoint
}
```

Terraform Code for Azure (AKS):

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "aks_rg" {
  name     = "aks-resource-group"
  location = "East US"
```

```
}

resource "azurerm_virtual_network" "aks_vnet" {
  name                = "aks-vnet"
  address_space       = ["10.1.0.0/16"]
  location            = azurerm_resource_group.aks_rg.location
  resource_group_name = azurerm_resource_group.aks_rg.name
}

resource "azurerm_subnet" "aks_subnet" {
  name                 = "aks-subnet"
  resource_group_name  = azurerm_resource_group.aks_rg.name
  virtual_network_name = azurerm_virtual_network.aks_vnet.name
  address_prefixes     = ["10.1.1.0/24"]
}

resource "azurerm_kubernetes_cluster" "aks" {
  name                = "aks-cluster"
  location            = azurerm_resource_group.aks_rg.location
  resource_group_name = azurerm_resource_group.aks_rg.name
  dns_prefix          = "akscluster"

  default_node_pool {
    name       = "default"
    node_count = 2
    vm_size    = "Standard_DS2_v2"
  }

  identity {
    type = "SystemAssigned"
  }

  network_profile {
    network_plugin    = "azure"
    load_balancer_sku = "standard"
  }
}

output "aks_cluster_endpoint" {
  value = azurerm_kubernetes_cluster.aks.kube_config.0.host
}
```

Networking Components

Networking Details:

- **AWS**: The `terraform-aws-modules/vpc` module handles VPC, subnets, NAT gateway, and route tables.
- **Azure**: Use the `azurerm_virtual_network` and `azurerm_subnet` resources to create a VNet and subnets.

Load Balancers

AWS Load Balancer:

EKS automatically provisions a load balancer for services of type `LoadBalancer`. You can customize it using an ingress controller like **AWS ALB Ingress Controller**.

Azure Load Balancer:

AKS automatically provisions a load balancer for services of type LoadBalancer. Use Azure-specific annotations to configure the load balancer.

Monitoring and Logging

AWS (CloudWatch):

Add the following Terraform configuration to enable monitoring:

```
resource "aws_cloudwatch_log_group" "eks_logs" {
  name              = "/eks/cluster"
  retention_in_days = 7
}

module "eks" {
  ...
  enable_logging = true
  log_types      = ["api", "audit", "authenticator"]
}
```

Azure (Azure Monitor):

Enable monitoring during AKS creation:

```
resource "azurerm_kubernetes_cluster" "aks" {
  ...
  addon_profile {
    oms_agent {
      enabled                    = true
      log_analytics_workspace_id = azurerm_log_analytics_workspace.law.id
    }
  }
}

resource "azurerm_log_analytics_workspace" "law" {
  name                = "aks-logs"
  location            = azurerm_resource_group.aks_rg.location
  resource_group_name = azurerm_resource_group.aks_rg.name
  sku                 = "PerGB2018"
}
```

Terraform Module for Reusability

Create a module structure like this:

```
modules/
└── eks/
    ├── main.tf
    ├── variables.tf
    └── outputs.tf
```

#main.tf:

```
resource "aws_eks_cluster" "this" {
  name     = var.cluster_name
  role_arn = var.role_arn

  vpc_config {
    subnet_ids = var.subnet_ids
  }
}
```

#variables.tf:

```
variable "cluster_name" {}

variable "role_arn" {}

variable "subnet_ids" {

  type = list(string)

}
```

#outputs.tf:

```
output "cluster_endpoint" {

  value = aws_eks_cluster.this.endpoint

}
```

Execution Steps:

Initialize Terraform:

```
terraform init
```

Validate the configuration and apply the changes:

```
terraform validate
terraform apply
```

Kubernetes Setup

We will configure Kubernetes clusters in AWS (EKS) and Azure (AKS), followed by setting up namespaces, resource quotas, and limits. We will also deploy a monitoring stack (Prometheus + Grafana) and set up centralized logging (Fluentd or Elasticsearch-Logstash-Kibana).

Prerequisites:

1. Install **kubectl**: Download and Install kubectl.
2. Install **Helm**: Install Helm.
3. Install **AWS CLI** and **Azure CLI** as configured in Step 1.
4. Ensure that Kubernetes clusters (EKS and AKS) are up and running.

Set Up Kubernetes Clusters

Connect to AWS (EKS) Cluster:

Update kubeconfig for your EKS cluster:

```
aws eks update-kubeconfig --region us-east-1 --name eks-cluster
```

Verify the connection:

```
kubectl get nodes
```

Connect to Azure (AKS) Cluster:

Update kubeconfig for your AKS cluster:

```
az aks get-credentials --resource-group aks-resource-group --name aks-cluster
```

Verify the connection:

```
kubectl get nodes
```

Configure Namespaces, Resource Quotas, and Limits

Create Namespaces:

Namespaces logically isolate resources. Create separate namespaces for development, staging, and production.

```
kubectl create namespace development

kubectl create namespace staging

kubectl create namespace production
```

Apply Resource Quotas:

Define resource quotas to limit the CPU, memory, and other resources within a namespace.

resource-quota.yaml:

```
apiVersion: v1

kind: ResourceQuota

metadata:

  name: compute-resources

  namespace: development

spec:

  hard:

    requests.cpu: "2"

    requests.memory: "4Gi"

    limits.cpu: "4"

    limits.memory: "8Gi"
```

Apply the resource quota:

```
kubectl apply -f resource-quota.yaml
```

Set Default Resource Limits:

Use a LimitRange to define default resource requests and limits.

limit-range.yaml:

```
apiVersion: v1

kind: LimitRange

metadata:

  name: resource-limits

  namespace: development
```

```
spec:
  limits:
  - default:
      cpu: "500m"
      memory: "1Gi"
    defaultRequest:
      cpu: "250m"
      memory: "512Mi"
    type: Container
```

Apply the limit range:

```
kubectl apply -f limit-range.yaml
```

Deploy Monitoring Stack (Prometheus + Grafana)

Install Prometheus:

Add the Helm repository:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
```

Install Prometheus:

```
helm install prometheus prometheus-community/prometheus --namespace monitoring
--create-namespace
```

Verify the installation:

```
kubectl get pods -n monitoring
```

Install Grafana:

Add the Helm repository:

```
helm repo add grafana https://grafana.github.io/helm-charts
```

```
helm repo update
```

Install Grafana:

```
helm install grafana grafana/grafana --namespace monitoring
```

Retrieve the admin password:

```
kubectl get secret --namespace monitoring grafana -o jsonpath="{.data.admin-password}" |
base64 --decode
```

Access Grafana:

Use a port-forward to access Grafana:

```
kubectl port-forward svc/grafana 3000:80 -n monitoring
```

-
- Open Grafana at http://localhost:3000.

Set Up Centralized Logging

Install Fluentd for Log Forwarding:

Deploy Fluentd:

```
kubectl apply -f
https://raw.githubusercontent.com/fluent/fluentd-kubernetes-daemonset/master/fluentd-daemon
set-elasticsearch-rbac.yaml
```

Install Elasticsearch:

Add the Helm repository:

```
helm repo add elastic https://helm.elastic.co
```

```
helm repo update
```

Install Elasticsearch:

```
helm install elasticsearch elastic/elasticsearch --namespace logging --create-namespace
```

Install Kibana:

Install Kibana:

```
helm install kibana elastic/kibana --namespace logging
```

Access Kibana:

Use a port-forward to access Kibana:

```
kubectl port-forward svc/kibana-kibana 5601:5601 -n logging
```

-
- Open Kibana at http://localhost:5601.

Verification:

List namespaces:

```
kubectl get namespaces
```

Check resource quotas:

```
kubectl get resourcequota -n development
```

Access Grafana dashboards and verify Prometheus data.

View logs in Kibana to ensure Fluentd is forwarding logs.

Microservices Deployment

We will deploy a sample microservices application (e.g., a to-do app) to Kubernetes clusters. This will involve containerizing the application, creating Helm charts for deployment, and configuring features like horizontal pod autoscaling and rolling updates.

Prerequisites

1. Docker is installed and configured. (Install Docker)
2. A Kubernetes cluster is up and running. (Configured in Step 1 and Step 2)
3. Helm is installed and initialized. (Install Helm)
4. Basic understanding of Dockerfiles, Kubernetes deployments, and Helm charts.

Containerize the Application

Sample Microservices Application

We will use a simple **to-do app** with two microservices:

1. **Backend**: A Python Flask app connected to a database.
2. **Frontend**: A React app.

Backend: Flask App

#backend/app.py:

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/api/todos", methods=["GET"])
def get_todos():
    return jsonify([{"id": 1, "task": "Learn Kubernetes"}, {"id": 2, "task": "Deploy with Helm"}])

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

#backend/Dockerfile:

```
# Use Python base image
FROM python:3.9-slim

# Set working directory
WORKDIR /app
```

```
# Copy application files
COPY app.py /app/

# Install dependencies
RUN pip install flask

# Expose port
EXPOSE 5000

# Run the app
CMD ["python", "app.py"]
```

Build and tag the Docker image:

```
docker build -t todo-backend:1.0 ./backend
```

Frontend: React App

#frontend/Dockerfile:

```
# Use Node.js base image
FROM node:16

# Set working directory
WORKDIR /app
```

```
# Copy application files
COPY . /app

# Install dependencies
RUN npm install

# Build the app
RUN npm run build

# Serve the app
RUN npm install -g serve
CMD ["serve", "-s", "build"]

# Expose port
EXPOSE 3000
```

Build and tag the Docker image:

```
docker build -t todo-frontend:1.0 ./frontend
```

Create Helm Charts for Deployment

Create a Helm chart structure:

```
helm create todo-app
cd todo-app
```

Edit values.yaml:

Define the Docker image details and replica counts.

#values.yaml:

yaml

Copy code

```yaml
replicaCount: 2

image:
  backend:
    repository: todo-backend
    tag: "1.0"
  frontend:
    repository: todo-frontend
    tag: "1.0"

service:
  type: LoadBalancer
  port: 80

resources:
  limits:
    cpu: "500m"
    memory: "512Mi"
  requests:
    cpu: "250m"
    memory: "256Mi"
```

Edit templates/deployment.yaml:

Define Kubernetes deployments for both services.

#templates/deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  labels:
    app: todo-app
    tier: backend
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: todo-app
      tier: backend
  template:
    metadata:
      labels:
        app: todo-app
        tier: backend
    spec:
      containers:
      - name: backend
```

```
        image: "{{ .Values.image.backend.repository }}:{{ .Values.image.backend.tag }}"
        ports:
        - containerPort: 5000
        resources:
          limits:
            memory: {{ .Values.resources.limits.memory }}
            cpu: {{ .Values.resources.limits.cpu }}
          requests:
            memory: {{ .Values.resources.requests.memory }}
            cpu: {{ .Values.resources.requests.cpu }}

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: todo-app
    tier: frontend
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: todo-app
      tier: frontend
  template:
    metadata:
      labels:
        app: todo-app
        tier: frontend
    spec:
      containers:
      - name: frontend
        image: "{{ .Values.image.frontend.repository }}:{{ .Values.image.frontend.tag }}"
        ports:
        - containerPort: 3000
```

Edit templates/service.yaml:

Define Kubernetes services to expose the application.

#templates/service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: backend
  labels:
    app: todo-app
spec:
  type: {{ .Values.service.type }}
  ports:
  - port: 5000
```

```yaml
    targetPort: 5000
  selector:
    app: todo-app
    tier: backend

---

apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: todo-app
spec:
  type: {{ .Values.service.type }}
  ports:
  - port: 80
    targetPort: 3000
  selector:
    app: todo-app
    tier: frontend
```

Configure Horizontal Pod Autoscaling and Rolling Updates

Enable Autoscaling:

Use kubectl to enable autoscaling for the backend.

```
kubectl autoscale deployment backend --cpu-percent=50 --min=2 --max=5
```

Rolling Updates:

Helm automatically performs rolling updates when you change configuration. Test it by updating the replica count:

Edit values.yaml:

```
replicaCount: 3
```

Upgrade the deployment:

```
helm upgrade todo-app ./todo-app
```

Deploy the Application

Package the Helm chart:

```
helm package ./todo-app
```

Install the chart:

```
helm install todo-app ./todo-app
```

Verification

Check the status of the deployments:

```
kubectl get deployments
```

Verify autoscaling:

```
kubectl get hpa
```

Access the frontend service:

```
kubectl get svc
```

CI/CD Pipeline

We will create a CI/CD pipeline to automate the building and testing of Docker images, push the images to a container registry, deploy to Kubernetes using Helm charts, and implement a Canary deployment strategy to minimize downtime.

Prerequisites

1. Jenkins/GitHub Actions/GitLab CI/CD setup.
2. Access to a container registry (AWS ECR, Azure Container Registry, or Docker Hub).
3. Kubernetes cluster and Helm configured (from Step 1 and Step 2).
4. Docker images for the microservices (from Step 3).

Automate Build and Testing of Docker Images

Jenkins Example Pipeline

1. Install required Jenkins plugins:
   - Docker Pipeline
   - Kubernetes CLI
   - GitHub Integration or GitLab Integration
2. Create a Jenkinsfile for the CI pipeline:

#Jenkinsfile:

```
pipeline {

  agent any

  environment {

    REGISTRY = 'your-container-registry-url'

    BACKEND_IMAGE = "${REGISTRY}/todo-backend:latest"

    FRONTEND_IMAGE = "${REGISTRY}/todo-frontend:latest"

  }
  stages {

    stage('Checkout') {

      steps {

        git branch: 'main', url: 'https://github.com/your-repo/todo-app.git'

      }

    }

    stage('Build Docker Images') {

      steps {

        sh 'docker build -t $BACKEND_IMAGE ./backend'

        sh 'docker build -t $FRONTEND_IMAGE ./frontend'

      }

    }

    stage('Test Docker Images') {

      steps {

        sh 'docker run --rm $BACKEND_IMAGE pytest'

        sh 'docker run --rm $FRONTEND_IMAGE npm test'

      }

    }

    stage('Push to Registry') {

      steps {

        withDockerRegistry([credentialsId: 'your-registry-credentials-id', url:
'https://$REGISTRY']) {

          sh 'docker push $BACKEND_IMAGE'

          sh 'docker push $FRONTEND_IMAGE'

        }

      }

    }  }  }
```

GitHub Actions Example Workflow

# .github/workflows/docker-build.yml:

```
name: CI Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout Code
      uses: actions/checkout@v3

    - name: Log in to Container Registry
      uses: docker/login-action@v2
      with:
        registry: your-container-registry-url
        username: ${{ secrets.REGISTRY_USERNAME }}
        password: ${{ secrets.REGISTRY_PASSWORD }}

    - name: Build Docker Images
      run: |
        docker build -t your-container-registry-url/todo-backend:latest ./backend
        docker build -t your-container-registry-url/todo-frontend:latest ./frontend

    - name: Push Docker Images
      run: |
        docker push your-container-registry-url/todo-backend:latest
        docker push your-container-registry-url/todo-frontend:latest
```

Push Images to Container Registry

AWS ECR

Authenticate with ECR:

```
aws ecr get-login-password --region us-east-1 | docker login --username ubuntu
--password-stdin <ecr-repository-url>
```

Push images:

```
docker tag todo-backend:1.0 <your-ecr-repository-url>/todo-backend:1.0
docker push <your-ecr-repository-url>/todo-backend:1.0

docker tag todo-frontend:1.0 <your-ecr-repository-url>/todo-frontend:1.0
docker push <your-ecr-repository-url>/todo-frontend:1.0
```

Azure Container Registry (ACR)

Authenticate with ACR:

```
az acr login --name your-acr-name
```

Push images:

```
docker tag todo-backend:1.0 your-acr-name.azurecr.io/todo-backend:1.0
docker push your-acr-name.azurecr.io/todo-backend:1.0
docker tag todo-frontend:1.0 your-acr-name.azurecr.io/todo-frontend:1.0

docker push your-acr-name.azurecr.io/todo-frontend:1.0
```

Automate Kubernetes Deployment Using Helm Charts

Jenkins Deployment Pipeline

Add to Jenkinsfile:

```
stage('Deploy to Kubernetes') {

    steps {

        sh 'helm upgrade --install todo-app ./helm/todo-app --set
image.backend.repository=$BACKEND_IMAGE --set
image.frontend.repository=$FRONTEND_IMAGE'

    }

}
```

GitHub Actions Deployment Workflow

# .github/workflows/deploy.yml:

```
name: CD Pipeline


on:

  push:

    branches:

      - main


jobs:

  deploy:

    runs-on: ubuntu-latest
    steps:
    - name: Checkout Code
      uses: actions/checkout@v3

    - name: Set up kubectl
      uses: azure/setup-kubectl@v3
      with:
        version: 'v1.27.0'

    - name: Deploy to Kubernetes
      run: |
        helm upgrade --install todo-app ./helm/todo-app \
          --set image.backend.repository=your-container-registry-url/todo-backend:latest \
          --set image.frontend.repository=your-container-registry-url/todo-frontend:latest
```

Implement a Canary Deployment Strategy

Update Helm Chart for Canary Deployment

Add a `canary` section in `values.yaml`:

```
canary:
  enabled: true
  weight: 20
```

Update deployment.yaml to include the canary weight:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend-canary
  labels:
    app: todo-app
    tier: backend
    version: canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: todo-app
      tier: backend
      version: canary
  template:
    metadata:
      labels:
        app: todo-app
        tier: backend
        version: canary
    spec:
      containers:
      - name: backend
        image: "{{ .Values.image.backend.repository }}:{{ .Values.image.backend.tag }}"
        ports:

        - containerPort: 5000
```

Deploy with Helm:

```
helm upgrade --install todo-app ./helm/todo-app --set canary.enabled=true --set canary.weight=20
```

Verification

CI Pipeline: Check Jenkins/GitHub Actions build logs to ensure Docker images are built and pushed.

Kubernetes Deployment:

```
kubectl get pods

kubectl get svc
```

Canary Deployment: Monitor traffic split between stable and canary pods:

```
kubectl get pods -l version=canary
```

Observability

We will set up monitoring, logging, and tracing tools to ensure the Kubernetes cluster and microservices are observable. This involves installing Prometheus and Grafana for monitoring, setting up alerts for resource thresholds, and using Jaeger for distributed tracing.

Prerequisites

1. Kubernetes clusters (AWS EKS and Azure AKS) set up from Step 1 and Step 2.
2. Helm installed on your local machine or CI/CD pipeline.

Monitor Kubernetes Clusters for Resource Usage

Install Prometheus using Helm

Prometheus will collect metrics from Kubernetes and microservices.

Add the Prometheus Helm repository:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm repo update
```

Deploy Prometheus:

```
helm upgrade --install prometheus prometheus-community/kube-prometheus-stack \
  --namespace monitoring --create-namespace
```

Verify the deployment:

```
kubectl get pods -n monitoring
```

Access Prometheus (port-forward for local access):

```
kubectl port-forward svc/prometheus-kube-prometheus-prometheus -n monitoring 9090:9090
```

Access Prometheus at http://localhost:9090.

Install Grafana using Helm

Grafana will visualize metrics collected by Prometheus.

Grafana is included in the Prometheus Stack above, but if needed as a standalone deployment:

```
helm repo add grafana https://grafana.github.io/helm-charts

helm repo update

helm upgrade --install grafana grafana/grafana \
  --namespace monitoring --create-namespace
```

Verify the deployment:

```
kubectl get pods -n monitoring
```

Access Grafana:

```
kubectl port-forward svc/grafana -n monitoring 3000:3000
```

Access Grafana at `http://localhost:3000`. The default username is `admin` and the password is retrieved with:

```
kubectl get secret -n monitoring grafana -o jsonpath="{.data.admin-password}" | base64
--decode
```

Add Prometheus as a data source in Grafana:

- Navigate to **Configuration > Data Sources > Add Data Source**.
- Select Prometheus and provide the Prometheus URL (`http://prometheus-kube-prometheus-prometheus:9090`).

Set Up Alerts for Resource Thresholds

Configure Prometheus Alerts

Edit Prometheus rules:

```
kubectl edit configmap prometheus-kube-prometheus-prometheus-rulefiles -n monitoring
```

Add alerting rules, for example:

```
groups:

- name: resource-alerts

  rules:

  - alert: HighCPUUsage

    expr: sum(rate(container_cpu_usage_seconds_total[1m])) by (instance) > 0.8

    for: 2m

    labels:

      severity: warning

    annotations:

      summary: "High CPU usage detected on {{ $labels.instance }}"

      description: "CPU usage has been above 80% for the past 2 minutes."
```

Reload Prometheus to apply the changes:

```
kubectl rollout restart deployment prometheus-kube-prometheus-prometheus -n monitoring
```

Alerts can be integrated with tools like Slack or email using Alertmanager, which is included in the Prometheus stack.

Implement Distributed Tracing

Install Jaeger using Helm

Add the Jaeger Helm repository:

```
helm repo add jaegertracing https://jaegertracing.github.io/helm-charts

helm repo update
```

Deploy Jaeger:

```
helm upgrade --install jaeger jaegertracing/jaeger \

  --namespace observability --create-namespace \

  --set agent.enabled=true \

  --set allInOne.enabled=true
```

Verify the deployment:

```
kubectl get pods -n observability
```

Access Jaeger (port-forward for local access):

```
kubectl port-forward svc/jaeger-all-in-one-query -n observability 16686:16686
```

Access Jaeger at http://localhost:16686.

Instrument Microservices for Tracing

Add OpenTelemetry or Jaeger client libraries to your microservices codebase.

Example (Python Flask app with opentelemetry):

```
pip install opentelemetry-api opentelemetry-sdk opentelemetry-instrumentation-flask
opentelemetry-exporter-jaeger
```

Instrument the application:

```
from flask import Flask

from opentelemetry.instrumentation.flask import FlaskInstrumentor

from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.sdk.trace import TracerProvider

from opentelemetry.sdk.trace.export import BatchSpanProcessor


app = Flask(__name__)

FlaskInstrumentor().instrument_app(app)


jaeger_exporter = JaegerExporter(

    agent_host_name="jaeger-agent",

    agent_port=6831,

)


tracer_provider = TracerProvider()

tracer_provider.add_span_processor(BatchSpanProcessor(jaeger_exporter))
```

Deploy the updated microservices to Kubernetes.

Visualize Microservices Interactions

1. Access the Jaeger UI at http://localhost:16686.
2. Search for traces of your application by service name.
3. Analyze spans to debug latency or errors in microservice interactions.

Verification

Prometheus Metrics:

```
kubectl port-forward svc/prometheus-kube-prometheus-prometheus -n monitoring 9090:9090
```

1. Query for metrics like `container_cpu_usage_seconds_total`.
2. **Grafana Dashboards**:
    - Import Kubernetes dashboards from Grafana's library (e.g., Dashboard ID 6417).
3. **Jaeger Traces**:
    - Check for end-to-end traces of microservices in the Jaeger UI.

Multi-Cloud Setup

We will configure a multi-cloud environment to ensure high availability and redundancy by routing traffic between AWS and Azure Kubernetes clusters using multi-cloud load balancers and DNS failover. This setup provides resilience in case one cloud provider's services go down.

Tools Required

- **AWS Global Accelerator**
- **Azure Traffic Manager**
- **DNS Provider**: AWS Route 53, Azure DNS, or a third-party service like Cloudflare.

Tasks

Set Up Multi-Cloud Load Balancers

Configure AWS Global Accelerator

AWS Global Accelerator distributes traffic globally across multiple AWS Regions, but it can also route traffic to Azure or other clouds using custom origins.

1. **Create a Global Accelerator**
    - Navigate to the AWS Global Accelerator Console.
    - Click **Create Accelerator** and provide:
        - **Name**: MultiCloudAccelerator.
        - **Listener**: Define the port for traffic (e.g., port 80 for HTTP or 443 for HTTPS).
2. **Add Endpoints**
    - Choose **Custom Routing** or **Standard**.
    - Add two endpoints:
        - **AWS**: Use the AWS Application Load Balancer (ALB) or Network Load Balancer (NLB) DNS name for the EKS cluster.
        - **Azure**: Use the public IP or DNS name of the Azure AKS ingress.
3. Example setup:
    - AWS Endpoint: my-eks-alb-1234567890.us-west-2.elb.amazonaws.com
    - Azure Endpoint: my-aks-ingress.eastus.cloudapp.azure.com
4. **Configure Health Checks**
    - Set up health checks to monitor the status of each endpoint.
5. **Save and Deploy** the Global Accelerator.

Configure Azure Traffic Manager

Azure Traffic Manager routes traffic based on DNS queries and can failover between Azure and external services.

1. **Create a Traffic Manager Profile**
    - Navigate to the Azure Portal > **Traffic Manager Profiles** > **Create**.
    - Profile Settings:
        - **Name**: MultiCloudTrafficManager.
        - **Routing Method**: Use Priority for failover or Performance for optimal latency.
2. **Add Endpoints**
    - Add an Azure AKS ingress endpoint:
        - Type: Azure Endpoint
        - Target Resource: Select the AKS ingress public IP.
    - Add an AWS EKS ingress endpoint:
        - Type: External Endpoint
        - Target Resource: Use the AWS ALB/NLB DNS.
3. **Configure Monitoring**
    - Add a health probe for each endpoint to detect failures.
4. **Save and Deploy** the Traffic Manager.

Implement DNS Failover

DNS failover ensures traffic is rerouted if one cloud's services fail.

1: Use a DNS Provider

- Use AWS Route 53, Azure DNS, or a third-party provider like Cloudflare.

2: Configure Route 53 (AWS Example)

1. **Create a Hosted Zone**
    - Go to Route 53 and create a hosted zone (e.g., example.com).
2. **Add Record Sets**
    - Add an A or CNAME record for your domain:
        - Name: app.example.com
        - Type: A or CNAME.
        - Routing Policy: **Failover**.
        - Primary:
            - Value: AWS Global Accelerator DNS name.
            - Health Check: Link to the Global Accelerator health check.
        - Secondary:
            - Value: Azure Traffic Manager DNS name.
            - Health Check: Link to Traffic Manager health check.
3. **Test Failover**
    - Simulate a failure by disabling one of the endpoints and ensuring traffic routes to the other.

Verification

1. **Test Traffic Routing**
   - Access your application through the domain (e.g., app.example.com) and verify it routes correctly to both AWS and Azure based on the health checks.
2. **Failover Testing**
   - Simulate a failure by shutting down an endpoint and verify that traffic automatically switches to the healthy endpoint.

Security

Here we focus on implementing robust security measures for your multi-cloud setup. We will manage secrets securely using HashiCorp Vault, enforce role-based access control (RBAC) in Kubernetes, and validate Kubernetes manifests with Open Policy Agent (OPA).

Prerequisites

1. Kubernetes clusters set up on AWS (EKS) and Azure (AKS).
2. Helm installed for deploying Vault and OPA.

Tasks

Manage Secrets Securely with HashiCorp Vault

Vault provides a secure method for storing and accessing secrets such as API keys, credentials, and configuration data.

1: Deploy Vault on Kubernetes

Add the Helm Repository:

```
helm repo add hashicorp https://helm.releases.hashicorp.com

helm repo update
```

Deploy Vault:

```
helm install vault hashicorp/vault \

  --namespace vault --create-namespace \

  --set "server.ha.enabled=true"
```

Verify the Deployment:

```
kubectl get pods -n vault
```

Initialize Vault:

```
kubectl exec -it vault-0 -n vault -- vault operator init
```

Save the unseal keys and root token securely.

Unseal Vault:

```
kubectl exec -it vault-0 -n vault -- vault operator unseal <unseal-key-1>

kubectl exec -it vault-0 -n vault -- vault operator unseal <unseal-key-2>

kubectl exec -it vault-0 -n vault -- vault operator unseal <unseal-key-3>
```

2: Configure Vault for Kubernetes Authentication

Enable the Kubernetes auth method:

```
kubectl exec -it vault-0 -n vault -- vault auth enable kubernetes
```

Configure Vault to talk to Kubernetes:

```
kubectl exec -it vault-0 -n vault -- vault write auth/kubernetes/config \

  token_reviewer_jwt="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" \

  kubernetes_host="https://$KUBERNETES_PORT_443_TCP_ADDR:443" \

  kubernetes_ca_cert="@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt"
```

Define policies for accessing secrets:

```
kubectl exec -it vault-0 -n vault -- vault policy write my-policy - <<EOF

path "secret/*" {

  capabilities = ["create", "read", "update", "delete", "list"]

}

EOF
```

Map Kubernetes service accounts to Vault policies:

```
kubectl exec -it vault-0 -n vault -- vault write auth/kubernetes/role/my-role \
  bound_service_account_names=my-service-account \

  bound_service_account_namespaces=default \

  policies=my-policy \

  ttl=24h
```

2: Enforce RBAC Policies for Kubernetes Access

RBAC restricts access to Kubernetes resources based on user roles.

1: Create a Namespace and Role

Create a Namespace:

```
kubectl create namespace dev
```

Define a Role:

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

kind: Role

metadata:

  namespace: dev

  name: dev-role

rules:

- apiGroups: [""]

  resources: ["pods", "services"]

  verbs: ["get", "list", "watch"]
```

Apply the Role:

```
kubectl apply -f dev-role.yaml
```

2: Bind the Role to a User or Service Account

Create a RoleBinding:

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

```
kind: RoleBinding
metadata:
  name: dev-rolebinding
  namespace: dev
subjects:
- kind: User
  name: dev-user
roleRef:
  kind: Role
  name: dev-role
  apiGroup: rbac.authorization.k8s.io
```

Apply the RoleBinding:

```
kubectl apply -f dev-rolebinding.yaml
```

Test Access:

- Switch context to the user dev-user and verify that only pods and services in the dev namespace are accessible.

3: Use OPA to Validate Kubernetes Manifests

OPA is a policy engine to enforce compliance by validating Kubernetes manifests.

1: Deploy OPA Gatekeeper

Add the Gatekeeper Helm Repository:

```
helm repo add gatekeeper https://open-policy-agent.github.io/gatekeeper/charts
helm repo update
```

Install Gatekeeper:

```
helm install gatekeeper gatekeeper/gatekeeper \
  --namespace gatekeeper-system --create-namespace
```

Verify the Deployment:

```
kubectl get pods -n gatekeeper-system
```

2: Create a Constraint Template

Define a Template:

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
  targets:
  - target: admission.k8s.gatekeeper.sh
    rego: |
      package k8srequiredlabels

      violation[{"msg": msg}] {
        missing := {"app", "env"} - {label | input.review.object.metadata.labels[label]}
        count(missing) > 0
        msg := sprintf("Missing required labels: %v", [missing])
      }
```

Apply the Template:

```
kubectl apply -f constraint-template.yaml
```

3: Create a Constraint

Define the Constraint:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: must-have-labels
spec:
  match:
    kinds:
    - apiGroups: [""]
      kinds: ["Pod"]
```

Apply the Constraint:

```
kubectl apply -f constraint.yaml
```

Test Manifest Validation:

Apply a manifest missing the required labels:

```
kubectl apply -f pod-without-labels.yaml
```

- Ensure the validation fails with an error about missing labels.

Verification

1. **HashiCorp Vault**:
   - Verify that secrets are accessible only via authorized service accounts.
2. **RBAC**:
   - Test role-based access control by attempting unauthorized actions.
3. **OPA**:
   - Validate manifests against defined policies and check for compliance.

1. Terraform Configuration Files for Multi-Cloud Kubernetes Setup

Structure: Create a folder structure for Terraform files:

```
multi-cloud-k8s/
├── aws/
│   ├── eks.tf
│   ├── vpc.tf
│   ├── outputs.tf
│   └── variables.tf
├── azure/
│   ├── aks.tf
│   ├── networking.tf
│   ├── outputs.tf
│   └── variables.tf
├── modules/
│   └── kubernetes/
│       ├── main.tf
│       ├── outputs.tf
│       └── variables.tf
├── providers.tf
├── main.tf
├── outputs.tf
└── variables.tf
```

Contents

1. **AWS EKS Setup**: eks.tf provisions an EKS cluster, VPC, and subnets.
2. **Azure AKS Setup**: aks.tf provisions an AKS cluster, VNet, and subnets.
3. **Modules**: Use reusable modules for Kubernetes cluster provisioning.
4. **Documentation**: Include comments for each resource.

2. Helm Charts for the Microservices Application

Structure

Folder structure for Helm charts:

```
helm-charts/
└── my-microservices-app/
    ├── charts/
    ├── templates/
    │   ├── deployment.yaml
    │   ├── service.yaml
    │   └── hpa.yaml
    ├── values.yaml
    ├── Chart.yaml
    └── README.md
```

Contents

1. **Templates**:
   - deployment.yaml: Defines Kubernetes deployments for microservices.
   - service.yaml: Exposes services via NodePort or LoadBalancer.
   - hpa.yaml: Configures horizontal pod autoscaling.
2. **Values**:
   - Parameterize image names, replicas, and resource limits in values.yaml.
3. **Documentation**:
   - Add usage instructions in README.md.

---

3. CI/CD Pipeline Scripts for Building, Testing, and Deploying

Structure

Folder for CI/CD configuration:

```
ci-cd/
├── jenkins/
│   ├── Jenkinsfile
│   ├── scripts/
│   │   ├── build.sh
│   │   ├── test.sh
│   │   └── deploy.sh
├── github-actions/
│   └── main.yml
└── gitlab-ci/
    └── .gitlab-ci.yml
```

Contents

1. **Jenkins Pipeline**: Automates the build-test-deploy pipeline using a declarative Jenkinsfile.
2. **Scripts**:
   - build.sh: Builds Docker images and pushes to a registry.
   - test.sh: Runs unit tests and security scans.
   - deploy.sh: Deploys Helm charts to Kubernetes clusters.
3. **GitHub Actions/GitLab CI**:
   - YAML configuration for building, testing, and deploying the application.

---

4. Monitoring Dashboards in Grafana and Tracing Visuals in Jaeger

Grafana Dashboards

- Pre-built dashboards for:
  - Kubernetes Cluster Metrics (CPU, memory, node health).
  - Application Metrics (request latency, error rates).
- Export JSON configurations from Grafana for easy sharing.

Jaeger Tracing

- Visualize service calls, response times, and errors in Jaeger.
- Include exported tracing diagrams and instructions for setting up the Jaeger UI.

Architecture Diagrams

- Use diagramming tools like Draw.io or Lucidchart to create:
  - High-level architecture (multi-cloud setup, DNS routing, CI/CD).
  - Detailed Kubernetes resource flow (services, pods, ingress).

Post-Deployment Verification Steps

- Verify the following:
    1. Kubernetes clusters are provisioned in AWS and Azure.
    2. Microservices are deployed and accessible via ingress.
    3. Traffic is routed and balanced across clouds.
    4. Monitoring dashboards show metrics.
    5. Traces in Jaeger correctly show service interactions.

Thanks

Aman Kr Choudhary