**HACK:** Helm, AWS, CI/CD, Kubernetes

**Secure and Scalable Microservices Deployment with AWS, Terraform, Helm, and Vault**

**Project Description:** This project involves setting up a secure and scalable microservices architecture on AWS using Terraform for infrastructure provisioning, Helm charts for Kubernetes application deployment, HashiCorp Vault for secrets management, and Prometheus and Grafana for monitoring.

**Key Components:**

1. **Infrastructure Provisioning with Terraform:**
   * Use Terraform to provision AWS resources such as EC2 instances, EKS (Elastic Kubernetes Service) clusters, VPCs, subnets, security groups, and IAM roles.
   * Implement Infrastructure as Code (IaC) practices to ensure reproducibility and version control of your infrastructure setup.
2. **Application Deployment with Helm:**
   * Use Helm charts to package, configure, and deploy your microservices applications onto the EKS cluster.
   * Leverage Helm's templating capabilities to manage different configurations across various environments (e.g., development, staging, production).
3. **Secrets Management with HashiCorp Vault:**
   * Integrate HashiCorp Vault for secure storage and management of sensitive information like API keys, database credentials, and other secrets.
   * Configure dynamic secrets generation for AWS services, ensuring that credentials are short-lived and rotated automatically.
4. **Monitoring with Prometheus and Grafana:**
   * Set up Prometheus to collect metrics from your Kubernetes cluster and microservices.
   * Use Grafana to visualize the collected metrics and create dashboards for monitoring the health and performance of your applications.
   * Set up alerting rules to notify the team in case of any performance issues or system failures.
5. **Automation and CI/CD Integration:**
   * Integrate the infrastructure and application deployment process with CI/CD tools like Jenkins or GitLab CI/CD.
   * Automate the deployment pipeline, ensuring that every code change triggers a build, test, and deployment process.
6. **Security Enhancements:**
   * Utilize AWS Identity and Access Management (IAM) roles and policies to control access to resources.
   * Use Terraform and Vault to manage and rotate secrets dynamically, reducing the risk of credential leakage.

This project demonstrates your proficiency in using Helm charts for Kubernetes deployments, HashiCorp Vault for secure secrets management, Terraform for infrastructure automation, and AWS for scalable cloud solutions, combined with monitoring systems to ensure the reliability and stability of your microservices architecture.

**Project File Structure**

/microservices-project/

├── /infrastructure/

│ ├── /terraform/

│ │ ├── /modules/

│ │ │ ├── /networking/ # VPC, Subnets, Security Groups

│ │ │ ├── /eks-cluster/ # EKS Cluster setup

│ │ │ ├── /iam/ # IAM roles and policies

│ │ │ └── /storage/ # S3 buckets, EBS volumes

│ │ ├── main.tf # Terraform root configuration

│ │ ├── variables.tf # Input variables for Terraform

│ │ ├── outputs.tf # Outputs from Terraform resources

│ │ ├── provider.tf # AWS provider configurations

│ │ ├── terraform.tfvars # Values for the input variables

│ │ └── backend.tf # Remote backend configuration

│

├── /deployment/

│ ├── /helm-chart/

│ │ ├── Chart.yaml # Helm chart metadata for the microservice

│ │ ├── values.yaml # Default configuration values

│ │ └── /templates/ # Kubernetes YAML templates

│

├── /secrets-management/

│ ├── /vault/

│ │ ├── policies/ # Vault access policies

│ │ ├── terraform-integration/ # Vault setup with Terraform

│ │ ├── secrets-config.hcl # Configuration for dynamic secrets

│ │ └── vault-agent-config/ # Auto-auth configuration for Vault Agent

│

├── /monitoring/

│ ├── /prometheus/

│ │ ├── prometheus-config.yaml # Prometheus configuration file

│ │ ├── /kubernetes-manifests/ # Deployment files for Prometheus in EKS

│ ├── /grafana/

│ │ ├── grafana-dashboards/ # Pre-configured dashboards for visualization

│ │ ├── grafana-config.yaml # Grafana configuration file

│ │ └── /datasource-configs/ # Datasource configurations for Prometheus

│

├── /ci-cd/

│ └── /github-actions/

│ ├── main.yml # GitHub Actions workflow file for CI/CD pipeline

│

├── README.md # Project overview and setup instructions

├── LICENSE # Licensing information

└── docs/

├── architecture-diagram.png # Diagram of the infrastructure architecture

├── helm-usage-guide.md # Documentation on using the Helm chart

├── terraform-guide.md # Guide on setting up and deploying with Terraform

└── vault-integration.md # Instructions for integrating Vault with EKS

### Explanation of the Structure:

1. **/infrastructure/terraform/**: Contains the Terraform code for provisioning AWS infrastructure, organized using a modular approach to make it reusable and maintainable.
2. **/deployment/helm-charts/**: Holds the Helm charts for each microservice. Each service has its own Helm chart, allowing for independent deployment and configuration.
3. **/secrets-management/vault/**: Includes configurations for integrating HashiCorp Vault into the deployment for secure secrets management.
4. **/monitoring/**: Contains Prometheus and Grafana configurations to set up a robust monitoring system for your Kubernetes cluster and deployed applications.
5. **/ci-cd/**: Includes Jenkins and GitLab CI/CD configurations to automate the deployment pipeline, ensuring smooth and continuous integration and delivery of new changes.
6. **/docs/**: Contains detailed documentation to help understand the project architecture, usage of Helm, Terraform setup, and how to integrate Vault.

**Step-By-Step Guide For Implementations:**

**Step 1: Set Up Infrastructure with Terraform**

1. **Initialize the Terraform Project:**
   * Navigate to the /infrastructure/terraform directory.
   * Run terraform init to initialize the Terraform working directory.
2. **Configure Terraform Modules:**
   * Use modules to define reusable components like networking, EKS cluster, IAM roles, and storage.
   * Update variables.tf with input variables for AWS credentials, region, and other configurations.
3. **Provision AWS Resources:**
   * Define resources in main.tf to create:
     + **VPC, Subnets, and Security Groups** in the /networking/ module.
     + **EKS Cluster** in the /eks-cluster/ module.
     + **IAM Roles** in the /iam/ module.
   * Run terraform apply to create the infrastructure on AWS.
4. **Configure Backend State:**
   * Set up backend.tf to use a remote backend (like S3) for Terraform state files.
   * Store state securely to maintain the current state of the infrastructure.

**Step 2: Deploy Microservice Using Helm**

1. **Set Up Kubernetes Context:**
   * Connect to your newly created EKS cluster using AWS CLI:

aws eks --region <region> update-kubeconfig --name <cluster-name>

1. **Create a Helm Chart for the Microservice:**
   * Navigate to /deployment/helm-chart/.
   * Customize Chart.yaml and values.yaml with your microservice's configuration.
2. **Deploy the Microservice:**
   * Use the following Helm command to deploy:

helm install <release-name> ./helm-chart

* + Verify that the microservice is running on the Kubernetes cluster.

**Step 3: Integrate HashiCorp Vault for Secrets Management**

1. **Deploy Vault on Kubernetes:**
   * Set up Vault in the /secrets-management/vault/ directory.
   * Follow the Vault documentation to deploy it securely on your EKS cluster.
2. **Configure Vault Policies:**
   * Create Vault policies and roles that allow applications to access only the secrets they need.
3. **Integrate Vault with Terraform:**
   * Use the Vault provider in Terraform to securely fetch secrets and integrate them into your infrastructure setup.
   * Store sensitive information like database credentials, API keys, and other secrets in Vault.

**Step 4: Set Up Monitoring with Prometheus and Grafana**

1. **Deploy Prometheus to Kubernetes:**
   * Navigate to /monitoring/prometheus/.
   * Apply the Kubernetes manifests to deploy Prometheus:

kubectl apply -f prometheus-config.yaml

1. **Deploy Grafana:**
   * Navigate to /monitoring/grafana/.
   * Deploy Grafana to visualize the metrics collected by Prometheus.
   * Configure data sources and dashboards in Grafana to monitor the health of the microservice and the EKS cluster.
2. **Create Alerts:**
   * Set up alerting rules in Prometheus to get notified about performance issues or resource exhaustion.

**Step 5: CI/CD Pipeline with GitHub Actions**

1. **Create a GitHub Repository:**
   * Push your project code to a GitHub repository.
2. **Configure GitHub Actions Workflow:**
   * In the /ci-cd/github-actions/ directory, create a main.yml file.
   * Define your CI/CD pipeline steps:
     + **Build** the microservice.
     + **Test** the code to ensure functionality.
     + **Deploy** the microservice to the EKS cluster using Helm.
3. **Example GitHub Actions Workflow (main.yml):**

name: CI/CD Pipeline

on:

push:

branches:

- main

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Set up Kubernetes

uses: azure/setup-kubectl@v2

- name: Deploy with Helm

run: helm upgrade --install <release-name> ./deployment/helm-chart

1. **Test and Deploy Automatically:**
   * Every time you push code to the main branch, the GitHub Actions workflow will trigger, automating the deployment process.

**Step 6: Testing and Validation**

1. **Check Microservice Deployment:**
   * Verify that the microservice is correctly deployed on the EKS cluster using:

kubectl get pods -n <namespace>

1. **Verify Vault Integration:**
   * Ensure that the microservice can access secrets stored in HashiCorp Vault.
2. **Monitor Application Performance:**
   * Use Grafana dashboards to monitor the health and performance of the microservice.
3. **Resolve Alerts:**
   * Address any issues raised by Prometheus alerts to ensure system stability.

**Step 7: Document the Setup**

1. **Update the README.md:**
   * Provide clear instructions on how to set up, deploy, and manage the project.
2. **Create Detailed Documentation:**
   * Add information to /docs/ about Helm, Terraform, Vault integration, and monitoring setup.

**Step 8: Maintain and Optimize**

1. **Regularly Update Infrastructure:**
   * Use Terraform to manage updates and changes to the infrastructure efficiently.
2. **Enhance Monitoring and Security:**
   * Continuously monitor your application and improve security settings with Vault.
3. **Optimize CI/CD Pipeline:**
   * Refine the GitHub Actions pipeline to improve deployment times and add more automated tests.

* **Necessary Files for Infrastructure**

This setup includes the core files: main.tf, variables.tf, outputs.tf, provider.tf, and backend.tf. These files will create an AWS EKS cluster, networking components, and necessary IAM roles for your infrastructure.

**1. main.tf**

This file contains the main configuration to set up the VPC, EKS cluster, and other resources.

provider "aws" {

region = var.aws\_region

}

# VPC Setup

module "vpc" {

source = "terraform-aws-modules/vpc/aws"

version = "3.14.2"

name = "${var.project\_name}-vpc"

cidr = "10.0.0.0/16"

azs = ["${var.aws\_region}a", "${var.aws\_region}b"]

private\_subnets = ["10.0.1.0/24", "10.0.2.0/24"]

public\_subnets = ["10.0.3.0/24", "10.0.4.0/24"]

enable\_nat\_gateway = true

single\_nat\_gateway = true

public\_subnet\_tags = { "kubernetes.io/role/elb" = "1" }

private\_subnet\_tags = { "kubernetes.io/role/internal-elb" = "1" }

}

# EKS Cluster Setup

module "eks" {

source = "terraform-aws-modules/eks/aws"

cluster\_name = "${var.project\_name}-eks-cluster"

cluster\_version = "1.21"

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"

key\_name = var.ec2\_key\_pair

}

}

}

**2. variables.tf**

This file contains input variables that are used in the Terraform configuration. You can update these values based on your requirements.

variable "aws\_region" {

description = "AWS region to deploy resources"

type = string

default = "us-east-1"

}

variable "project\_name" {

description = "Project name for resource naming"

type = string

default = "microservices-project"

}

variable "ec2\_key\_pair" {

description = "Name of the existing EC2 Key Pair to use"

type = string

default = "my-key-pair"

}

**3. outputs.tf**

This file defines the output values that provide information about the deployed infrastructure.

output "vpc\_id" {

description = "The ID of the VPC"

value = module.vpc.vpc\_id

}

output "eks\_cluster\_id" {

description = "The ID of the EKS Cluster"

value = module.eks.cluster\_id

}

output "eks\_cluster\_endpoint" {

description = "The endpoint of the EKS Cluster"

value = module.eks.cluster\_endpoint

}

output "eks\_node\_security\_group\_id" {

description = "Security Group ID for EKS nodes"

value = module.eks.node\_security\_group\_id

}

**4. provider.tf**

This file configures the AWS provider and is used to specify the AWS region where the resources will be deployed.

terraform {

required\_providers {

aws = {

source = "hashicorp/aws"

version = "~> 4.0"

}

}

required\_version = ">= 1.0.0"

}

provider "aws" {

region = var.aws\_region

}

**5. backend.tf**

This file configures the remote backend to store the Terraform state securely in an S3 bucket.

terraform {

backend "s3" {

bucket = "my-terraform-state-bucket"

key = "microservices-project/terraform.tfstate"

region = "us-east-1"

encrypt = true

dynamodb\_table = "terraform-lock-table"

}

}

**Explanation of the Terraform Files:**

1. **main.tf**: Configures the main infrastructure components, including the VPC, subnets, and EKS cluster.
2. **variables.tf**: Holds input variables, making it easy to configure the infrastructure based on different environments.
3. **outputs.tf**: Displays important details about the infrastructure after deployment.
4. **provider.tf**: Specifies the AWS provider and version requirements.
5. **backend.tf**: Sets up a remote backend to store the Terraform state securely in an S3 bucket, which helps with state management and collaboration.

**Next Steps:**

1. **Initialize Terraform**: Run terraform init to initialize the Terraform workspace.
2. **Validate Configuration**: Run terraform validate to ensure the configuration files are syntactically correct.
3. **Plan and Apply**: Execute terraform plan to see the changes that will be made, followed by terraform apply to deploy the infrastructure.

### Deploy Sample Microservice Using Helm

#### 1. ****Create a Sample Node.js Microservice****

First, we'll create a simple Node.js application with a basic HTTP server. Here's how you can set up the structure for your sample microservice:

**File Structure for the Sample Microservice:**

/sample-microservice/

├── Dockerfile

├── app.js

├── package.json

└── README.md

**1.1. Create the app.js file:**

javascript

Copy code

// app.js

const http = require('http');

const hostname = '0.0.0.0';

const port = 3000;

const server = http.createServer((req, res) => {

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Hello, this is a sample microservice running on Kubernetes!\n');

});

server.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`);

});

**1.2. Create the package.json file:**

{

"name": "sample-microservice",

"version": "1.0.0",

"description": "A sample Node.js microservice",

"main": "app.js",

"scripts": {

"start": "node app.js"

},

"dependencies": {

"http": "latest"

}

}

**1.3. Create the Dockerfile:**

Dockerfile

Copy code

# Dockerfile

FROM node:16-alpine

WORKDIR /usr/src/app

COPY package\*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "start"]

This Dockerfile builds a lightweight Node.js image with your sample microservice, ready to be deployed on Kubernetes.

#### 2. ****Build and Push Docker Image to a Container Registry****

To deploy this microservice with Helm, you'll need to push the Docker image to a container registry like Docker Hub or Amazon ECR.

# Build the Docker image

docker build -t your-dockerhub-username/sample-microservice:latest .

# Push the Docker image to Docker Hub

docker push your-dockerhub-username/sample-microservice:latest

Replace your-dockerhub-username with your Docker Hub username.

#### 3. ****Create a Helm Chart for the Microservice****

Navigate to the /deployment/helm-chart/ directory in your project and create a Helm chart for your microservice.

**3.1. Create the Helm chart structure:**

helm create sample-microservice

This will create the following structure:

/sample-microservice/

├── Chart.yaml

├── values.yaml

└── templates/

├── deployment.yaml

├── service.yaml

├── hpa.yaml

└── ingress.yaml

**3.2. Update Chart.yaml:** Make sure your Chart.yaml file contains appropriate metadata for your microservice:

apiVersion: v2

name: sample-microservice

version: 1.0.0

description: A Helm chart for deploying a sample microservice

**3.3. Update values.yaml:** Update values.yaml to include your Docker image:

image:

repository: your-dockerhub-username/sample-microservice

tag: latest

pullPolicy: IfNotPresent

service:

type: LoadBalancer

port: 80

targetPort: 3000

Replace your-dockerhub-username with your Docker Hub username.

#### 4. ****Deploy the Microservice to Kubernetes using Helm****

Now, let's deploy the Helm chart to your EKS cluster:

helm install sample-microservice ./sample-microservice

This command will create a release of your microservice on the Kubernetes cluster.

#### 5. ****Verify the Deployment****

Check the status of your microservice deployment with:

kubectl get pods

Check the service to see if it's correctly exposing the application:

kubectl get services

You should see an external IP or DNS endpoint associated with your microservice, allowing you to access it via your browser.

### Summary of this step:

* We created a sample Node.js microservice.
* We built and pushed the Docker image to a container registry.
* We created a Helm chart to manage the deployment of the microservice.
* We deployed the microservice to the Kubernetes cluster using Helm.

### Integrate HashiCorp Vault for Secrets Management

#### 1. ****Deploy HashiCorp Vault on Kubernetes****

1.1. **Helm Installation of Vault**

* To deploy Vault in your Kubernetes cluster, we'll use the official Helm chart.
* Add the HashiCorp Helm repository and update it:

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

helm repo update

1.2. **Install Vault using Helm**

helm install vault hashicorp/vault --set "server.ha.enabled=true"

* This command installs Vault in High Availability (HA) mode in your Kubernetes cluster.
* Verify that Vault pods are running:

kubectl get pods -l app.kubernetes.io/name=vault

#### 2. ****Initialize and Unseal Vault****

2.1. **Initialize Vault**

* You need to initialize Vault to set up its encryption keys. Run:

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

* This command will generate unseal keys and a root token. **Store these securely**, as they are required to unseal Vault and manage secrets.

2.2. **Unseal Vault**

* Vault needs to be unsealed using the keys generated during initialization:

kubectl exec -it vault-0 -- vault operator unseal <Unseal\_Key\_1>

kubectl exec -it vault-0 -- vault operator unseal <Unseal\_Key\_2>

kubectl exec -it vault-0 -- vault operator unseal <Unseal\_Key\_3>

* You must enter three of the five unseal keys to unlock Vault.

#### 3. ****Configure Vault Policies and Roles****

3.1. **Log into Vault using the Root Token**

* Log in to Vault using the root token provided during initialization:

kubectl exec -it vault-0 -- vault login <Root\_Token>

3.2. **Create a Vault Policy**

* Create a policy that grants access to specific secrets for the microservice. Here's an example policy for a microservice named sample-microservice:

path "secret/data/sample-microservice/\*" {

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

}

* Save the above policy in a file called sample-microservice-policy.hcl and then add it to Vault:

vault policy write sample-microservice-policy sample-microservice-policy.hcl

3.3. **Enable Kubernetes Authentication in Vault**

* To allow your microservice to authenticate with Vault, enable the Kubernetes authentication method:

vault auth enable kubernetes

* Configure Vault to communicate with your Kubernetes API server:

vault write auth/kubernetes/config \

token\_reviewer\_jwt="$(kubectl get secret $(kubectl get serviceaccount vault -o jsonpath="{.secrets[0].name}") -o jsonpath="{.data.token}" | base64 --decode)" \

kubernetes\_host="https://$KUBERNETES\_PORT\_443\_TCP\_ADDR:443" \

kubernetes\_ca\_cert="$(kubectl get secret $(kubectl get serviceaccount vault -o jsonpath="{.secrets[0].name}") -o jsonpath="{.data['ca.crt']}" | base64 --decode)"

3.4. **Create a Role for the Microservice**

* Create a role that binds the Vault policy to the microservice:

vault write auth/kubernetes/role/sample-microservice-role \

bound\_service\_account\_names=sample-microservice-sa \

bound\_service\_account\_namespaces=default \

policies=sample-microservice-policy \

ttl=24h

#### 4. ****Inject Vault Secrets into the Microservice****

4.1. **Update the Helm Chart of the Microservice**

* Update the Helm chart's values.yaml file to include Vault annotations to inject secrets:

annotations:

vault.hashicorp.com/agent-inject: "true"

vault.hashicorp.com/role: "sample-microservice-role"

vault.hashicorp.com/agent-inject-secret-aws-credentials: "secret/data/sample-microservice/aws"

* This configuration tells Kubernetes to inject AWS credentials from Vault into the microservice's environment.

#### 5. ****Verify Vault Integration****

5.1. **Deploy the Updated Microservice**

* Redeploy your microservice with the updated Helm chart:

helm upgrade sample-microservice ./sample-microservice

5.2. **Check Environment Variables**

* Once deployed, check the microservice pod's environment to verify that the secrets were injected:

kubectl exec -it <pod-name> -- env | grep VAULT

### Summary of this step:

* We deployed HashiCorp Vault using Helm.
* We initialized and unsealed Vault, then configured policies and roles.
* We enabled Kubernetes authentication and linked it to the microservice.
* We updated the microservice Helm chart to inject secrets using Vault.

### Set Up Monitoring with Prometheus and Grafana

We'll use Helm to deploy both Prometheus and Grafana in your Kubernetes cluster.

#### 1. ****Deploy Prometheus using Helm****

1.1. **Add the Prometheus Helm Repository**

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

helm repo update

1.2. **Install Prometheus**

* Use the following command to install Prometheus in your Kubernetes cluster:

bash

Copy code

helm install prometheus prometheus-community/prometheus

* This command deploys Prometheus with its default configuration. To verify that the Prometheus pods are running:

kubectl get pods -l app=prometheus

#### 2. ****Deploy Grafana using Helm****

2.1. **Add the Grafana Helm Repository**

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

helm repo update

2.2. **Install Grafana**

* Use the following command to install Grafana in your Kubernetes cluster:

helm install grafana grafana/grafana

* Check that the Grafana pods are running:

kubectl get pods -l app.kubernetes.io/name=grafana

#### 3. ****Expose Prometheus and Grafana Services****

3.1. **Prometheus Service Exposure**

* By default, Prometheus might be exposed as a ClusterIP service. To access it externally, you can change it to a LoadBalancer or use port-forwarding for testing:

kubectl port-forward svc/prometheus-server 9090:80

* Now you can access the Prometheus UI at http://localhost:9090.

3.2. **Grafana Service Exposure**

* Similarly, for Grafana, use port-forwarding or change it to a LoadBalancer service:

kubectl port-forward svc/grafana 3000:80

* Access the Grafana dashboard at http://localhost:3000.

#### 4. ****Configure Grafana to Use Prometheus as a Data Source****

4.1. **Access Grafana Dashboard**

* Open your web browser and go to http://localhost:3000.
* Login with the default credentials:
  + **Username:** admin
  + **Password:** admin (you might be prompted to change this upon first login)

4.2. **Add Prometheus as a Data Source**

* Navigate to **Configuration** > **Data Sources** > **Add data source**.
* Select **Prometheus**.
* In the URL field, enter the Prometheus server address: http://prometheus-server.
* Click on **Save & Test** to confirm the connection.

#### 5. ****Set Up Dashboards in Grafana****

5.1. **Import a Pre-Built Dashboard**

* Grafana has many pre-built dashboards available for monitoring Kubernetes and Prometheus.
* Navigate to **Dashboards** > **Manage** > **Import**.
* Enter the dashboard ID (you can find popular IDs on the Grafana website, such as 6417 for Kubernetes cluster monitoring).
* Click on **Load**, choose the Prometheus data source, and import the dashboard.

#### 6. ****Configure Prometheus to Monitor Kubernetes Components****

6.1. **Scrape Configurations**

* The default installation of Prometheus should already be set up to scrape Kubernetes components. However, you may want to customize it by editing the Prometheus values.yaml file to include your microservice.
* To add your microservice, include a configuration similar to this:

- job\_name: 'sample-microservice'

kubernetes\_sd\_configs:

- role: pod

relabel\_configs:

- source\_labels: [\_\_meta\_kubernetes\_pod\_label\_app]

action: keep

regex: sample-microservice

#### 7. ****Verify Metrics Collection****

7.1. **Check Prometheus Targets**

* Open the Prometheus UI (http://localhost:9090) and go to **Status** > **Targets**.
* Ensure that your microservice and Kubernetes components are listed and in the "UP" state.

7.2. **Check Grafana Dashboards**

* In the Grafana UI, explore the imported dashboards to visualize metrics from your Kubernetes cluster and microservice.
* You should see metrics like CPU usage, memory usage, request rates, and other key performance indicators.

### Summary of this Step

* We deployed Prometheus and Grafana using Helm in the Kubernetes cluster.
* We exposed both services to access their dashboards.
* We configured Grafana to use Prometheus as the data source and imported a dashboard for monitoring.
* We set up Prometheus to scrape metrics from your microservice and Kubernetes components.

### Set Up CI/CD Pipeline with GitHub Actions

GitHub Actions provides a flexible way to automate your software workflows directly in your repository. We will create a GitHub Actions workflow that will handle the following steps:

1. **Build the Docker image** for your microservice.
2. **Push the Docker image** to a container registry (e.g., Docker Hub).
3. **Deploy the microservice** to the EKS cluster using Helm.

#### 1. ****Prepare the GitHub Repository****

Make sure that your project is in a GitHub repository. If you haven't pushed your project to GitHub yet, you can do so with these commands:

git init

git remote add origin https://github.com/your-username/your-repo-name.git

git add .

git commit -m "Initial commit"

git push -u origin main

Replace your-username and your-repo-name with your actual GitHub username and repository name.

#### 2. ****Create a GitHub Actions Workflow File****

2.1. **Create the Workflow Directory**

* In your repository, create the following directory structure:

/.github/workflows/

2.2. **Create the Workflow File**

* Create a file named ci-cd-pipeline.yml inside the .github/workflows/ directory with the following content:

name: CI/CD Pipeline

on:

push:

branches:

- main

pull\_request:

branches:

- main

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Set up Node.js

uses: actions/setup-node@v2

with:

node-version: '16'

- name: Install dependencies

run: npm install

- name: Run tests

run: npm test

- name: Build Docker image

run: docker build -t your-dockerhub-username/sample-microservice:${{ github.sha }} .

- name: Log in to Docker Hub

uses: docker/login-action@v2

with:

username: ${{ secrets.DOCKER\_USERNAME }}

password: ${{ secrets.DOCKER\_PASSWORD }}

- name: Push Docker image

run: docker push your-dockerhub-username/sample-microservice:${{ github.sha }}

deploy:

runs-on: ubuntu-latest

needs: build

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Set up kubectl

uses: azure/setup-kubectl@v1

with:

version: 'latest'

- name: Configure AWS credentials

uses: aws-actions/configure-aws-credentials@v1

with:

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-region: us-east-1

- name: Update kubeconfig

run: aws eks update-kubeconfig --name your-cluster-name --region us-east-1

- name: Deploy to EKS using Helm

run: |

helm upgrade --install sample-microservice ./deployment/helm-chart/sample-microservice \

--set image.repository=your-dockerhub-username/sample-microservice \

--set image.tag=${{ github.sha }}

Replace your-dockerhub-username with your Docker Hub username, and your-cluster-name with your EKS cluster name.

#### 3. ****Configure GitHub Secrets****

You need to store sensitive information as GitHub secrets to ensure security. Navigate to your GitHub repository settings, go to **Secrets and variables > Actions**, and add the following secrets:

* DOCKER\_USERNAME: Your Docker Hub username.
* DOCKER\_PASSWORD: Your Docker Hub password.
* AWS\_ACCESS\_KEY\_ID: Your AWS access key ID.
* AWS\_SECRET\_ACCESS\_KEY: Your AWS secret access key.

#### 4. ****How the CI/CD Pipeline Works****

* **Build Stage**:
  + The workflow triggers on every push or pull request to the main branch.
  + It checks out the code, installs Node.js dependencies, runs tests, and builds the Docker image.
  + It pushes the Docker image to Docker Hub with a tag based on the commit SHA.
* **Deploy Stage**:
  + This stage runs after the build stage completes successfully.
  + It configures the AWS CLI and kubectl to interact with your EKS cluster.
  + Finally, it deploys the updated Docker image to the EKS cluster using Helm.

#### 5. ****Testing the Pipeline****

* Push changes to the main branch of your repository:

git add .

git commit -m "Test CI/CD pipeline"

git push origin main

* Check the **Actions** tab in your GitHub repository to monitor the pipeline's progress.

### Summary of this Step

* We created a GitHub Actions workflow for the CI/CD pipeline.
* The pipeline automates building, testing, and deploying the microservice to the Kubernetes cluster.
* We configured GitHub secrets for securely storing sensitive information.
* We verified that the pipeline is set to trigger on pushes to the main branch.

### Testing and Validation

We'll break down this stage into different parts to cover all aspects of the project:

1. **Infrastructure Validation**
2. **Microservice Deployment Validation**
3. **HashiCorp Vault Secrets Validation**
4. **Monitoring System Validation**
5. **End-to-End Testing**

#### 1. ****Infrastructure Validation****

1.1. **Check AWS Resources**

* Confirm that all AWS resources (like EKS cluster, VPC, subnets, security groups) are correctly created and running.
* Use the AWS CLI to verify EKS cluster status:

aws eks describe-cluster --name your-cluster-name --region your-region

1.2. **Validate Kubernetes Cluster**

* Verify that the Kubernetes nodes are running:

kubectl get nodes

* Ensure that all pods in the cluster are in the Running state:

kubectl get pods --all-namespaces

#### 2. ****Microservice Deployment Validation****

2.1. **Verify Helm Deployment**

* Check that the Helm deployment of your microservice was successful:

helm list

* Ensure that the microservice pod is running:

kubectl get pods -l app=sample-microservice

2.2. **Access the Microservice**

* If you’ve exposed the microservice through a LoadBalancer service, get its external IP:

kubectl get svc sample-microservice

* Open the microservice in your browser using the external IP address or use a command-line tool like curl to test its endpoints.

#### 3. ****HashiCorp Vault Secrets Validation****

3.1. **Verify Vault Pod Status**

* Check that all Vault pods are in the Running state:

kubectl get pods -l app.kubernetes.io/name=vault

3.2. **Check Secrets Injection**

* Verify that Vault is correctly injecting secrets into the microservice by examining the environment variables of the microservice pod:

kubectl exec -it <sample-microservice-pod-name> -- env | grep VAULT

* Ensure that the secrets from Vault are present as expected.

#### 4. ****Monitoring System Validation****

4.1. **Prometheus Validation**

* Open the Prometheus UI (http://localhost:9090) and navigate to **Status > Targets**.
* Check that all targets (including your microservice and Kubernetes components) are listed and in the "UP" state.
* Run some queries to ensure metrics are being collected:

up

node\_cpu\_seconds\_total

4.2. **Grafana Validation**

* Open the Grafana dashboard (http://localhost:3000) and log in.
* Check that the Prometheus data source is connected and dashboards are displaying metrics correctly.
* Verify that the dashboard shows the health and performance of your microservice and Kubernetes components.

#### 5. ****End-to-End Testing****

5.1. **Functional Testing**

* Test the main functionalities of the microservice to ensure that it behaves as expected.
* Use tools like curl, Postman, or automated test scripts to perform HTTP requests to the microservice endpoints and verify responses.

5.2. **Load Testing**

* Perform load testing to check how the microservice behaves under heavy traffic.
* Use tools like Apache JMeter or Locust to simulate high loads and analyze the microservice's performance.

5.3. **Failure Scenarios**

* Test how the system responds to simulated failures (e.g., restarting pods, killing nodes).
* Ensure that the system is resilient and automatically recovers from failures.

#### 6. ****GitHub Actions Pipeline Validation****

6.1. **Pipeline Execution**

* Trigger a new pipeline run by pushing a code change to the main branch of the GitHub repository.

git commit -am "Trigger pipeline for validation" && git push origin main

* Monitor the **Actions** tab on GitHub to ensure that each stage of the CI/CD pipeline completes successfully (build, test, deploy).

6.2. **Check Deployment**

* Verify that the microservice is redeployed to the Kubernetes cluster using the new Docker image.
* Ensure that the latest version of the microservice is up and running.

### Summary of this Step

* We validated the infrastructure setup by checking AWS and Kubernetes resources.
* We confirmed that the microservice deployment was successful and is accessible.
* We verified HashiCorp Vault integration to ensure secrets are properly injected.
* We checked Prometheus and Grafana to confirm the monitoring system is functional.
* We performed end-to-end, load, and failure scenario testing to ensure the robustness of the system.
* We validated the CI/CD pipeline to confirm that it automates the deployment process as intended.

### Document the Setup

The documentation will be organized into several sections for clarity and ease of use. You can save this documentation in a file called README.md at the root of your project repository.

#### README.md Structure

Here is a suggested structure for your documentation:

1. **Project Overview**
2. **Infrastructure Setup with Terraform**
3. **Microservice Deployment Using Helm**
4. **Secrets Management with HashiCorp Vault**
5. **Monitoring with Prometheus and Grafana**
6. **CI/CD Pipeline with GitHub Actions**
7. **Testing and Validation**
8. **Common Issues and Troubleshooting**
9. **Future Improvements**
10. **Contributors and Acknowledgments**

#### Detailed Documentation Content

Here's a detailed breakdown of what to include in each section:

### 1. ****Project Overview****

* **Project Name:** Microservice CI/CD Pipeline with Kubernetes, Terraform, Helm, and HashiCorp Vault
* **Description:** This project demonstrates the setup of a robust CI/CD pipeline for a microservice deployed on an AWS EKS cluster using Terraform, Helm, HashiCorp Vault, Prometheus, and Grafana for monitoring.
* **Technologies Used:**
  + AWS (EKS, IAM)
  + Terraform
  + Kubernetes
  + Helm
  + HashiCorp Vault
  + Prometheus and Grafana
  + GitHub Actions
* **Objective:** Automate the infrastructure setup, deployment, monitoring, and secrets management for microservices using modern DevOps tools and practices.

### 2. ****Infrastructure Setup with Terraform****

* **Location:** /terraform/
* **Description:** Terraform is used to provision AWS resources, including the EKS cluster, networking, and IAM roles.
* **Commands to Run:**

terraform init

terraform plan

terraform apply

* **Files:**
  + main.tf: Contains the main infrastructure code.
  + variables.tf: Defines the variables used in the infrastructure.
  + outputs.tf: Specifies the outputs of the Terraform setup.

### 3. ****Microservice Deployment Using Helm****

* **Location:** /deployment/helm-chart/sample-microservice/
* **Description:** Helm is used to manage Kubernetes resources and deploy the sample microservice.
* **Deployment Command:**

helm install sample-microservice ./deployment/helm-chart/sample-microservice

* **Configuration Files:**
  + values.yaml: Configures the Helm chart values.
  + Chart.yaml: Defines the structure of the Helm chart.

### 4. ****Secrets Management with HashiCorp Vault****

* **Location:** /vault/
* **Description:** HashiCorp Vault is used to securely manage and inject secrets into the microservice.
* **Vault Setup Commands:**

vault server -config=config.hcl

vault login <token>

* **Process:**
  + Secret Injection via Kubernetes annotations.
  + Environment variables setup for microservices.

### 5. ****Monitoring with Prometheus and Grafana****

* **Prometheus Deployment:**
  + Installed using the Helm chart.
  + Accessible at http://localhost:9090.
* **Grafana Deployment:**
  + Installed using the Helm chart.
  + Accessible at http://localhost:3000 (default credentials: admin/admin).
* **Dashboards:** Pre-built Grafana dashboards are used to monitor the health of the Kubernetes cluster and microservice.

### 6. ****CI/CD Pipeline with GitHub Actions****

* **Workflow File Location:** .github/workflows/ci-cd-pipeline.yml
* **Stages:**
  + **Build**: Builds the Docker image for the microservice.
  + **Test**: Runs automated tests on the code.
  + **Deploy**: Deploys the Docker image to EKS using Helm.
* **Trigger:** Automatically triggers on a push to the main branch.
* **Environment Variables:** Managed via GitHub Secrets for secure handling of sensitive information.

### 7. ****Testing and Validation****

* **Infrastructure Validation:** Verify AWS and Kubernetes resources.
* **Microservice Validation:** Ensure the microservice is deployed and accessible.
* **Vault Secrets Validation:** Confirm that secrets are injected into the microservice.
* **Monitoring Validation:** Ensure that Prometheus and Grafana are collecting and displaying metrics correctly.
* **End-to-End Testing:** Perform functional and load testing on the microservice.

### 8. ****Common Issues and Troubleshooting****

* **Helm Deployment Failures:** Check Helm release logs using:

helm status sample-microservice

* **Pod Failures:** View pod logs using:

kubectl logs <pod-name>

* **GitHub Actions Failures:** Check the GitHub Actions tab for detailed error messages.

### 9. ****Future Improvements****

* **Implement Blue-Green Deployments**: To minimize downtime during updates.
* **Add More Microservices**: Extend the setup to deploy multiple microservices with separate Helm charts.
* **Enhance Security**: Integrate more secure mechanisms for managing secrets and access control.

### 10. ****Contributors and Acknowledgments****

* **Lead Developer:** Kashinath Meshram
* **Contributors:** Mention any other team members or contributors here.
* **Acknowledgments:** Thank any tools, tutorials, or people who helped guide the project.

### Final Steps

* Save this content as a README.md file in the root of your repository.
* Commit the changes to your repository:

git add README.md

git commit -m "Add project documentation"

git push origin main

* **Maintain and Optimize**

This step will involve strategies for routine maintenance, monitoring, performance tuning, and optimizations for the infrastructure and applications. The focus will be on continuous improvement and reducing technical debt over time.

**1. Infrastructure Maintenance**

1.1. **Regular Updates**

* **Kubernetes Cluster Upgrades:** Keep your Kubernetes cluster up-to-date with the latest versions. Regularly check for new versions of EKS and apply upgrades in a controlled manner.
* **Helm Charts:** Periodically update Helm charts for your microservices to incorporate new features, security patches, and performance improvements.
* **Terraform Modules:** Ensure that Terraform modules and providers are regularly updated to benefit from the latest features and bug fixes.

1.2. **Backup Strategy**

* Implement a robust backup strategy for critical resources such as Vault secrets, Helm releases, and application data.
* Use AWS services like Amazon S3 and EBS Snapshots for data backup and disaster recovery.

**2. Microservice Optimization**

2.1. **Performance Tuning**

* **Resource Requests and Limits:** Fine-tune CPU and memory requests/limits for your microservice pods to optimize resource utilization and avoid over-provisioning.
* **Horizontal Pod Autoscaling:** Configure Horizontal Pod Autoscalers (HPA) to automatically scale the number of pods based on CPU and memory usage.

2.2. **Logging and Tracing**

* Integrate centralized logging using tools like **ELK Stack (Elasticsearch, Logstash, and Kibana)** or **Fluentd** to gather, store, and analyze logs from your microservices.
* Implement tracing tools such as **Jaeger** or **OpenTelemetry** to track requests across microservices and identify bottlenecks.

**3. Secrets Management Optimization**

3.1. **Vault Policies**

* Regularly review and update Vault policies to ensure that permissions are as restrictive as possible.
* Rotate secrets periodically to reduce the risk of unauthorized access if credentials are compromised.

3.2. **Vault High Availability (HA)**

* Set up Vault in High Availability mode to ensure that your secrets management system is resilient and can handle failures without service disruption.

**4. Monitoring Enhancements**

4.1. **Prometheus Alerting**

* Set up custom alerts in Prometheus to notify your team of issues such as high CPU/memory usage, pod failures, or service downtime.
* Use alert management tools like **Alertmanager** to route alerts to appropriate channels like Slack, email, or incident response tools.

4.2. **Grafana Dashboards**

* Create and maintain Grafana dashboards with more detailed and customized metrics to monitor microservice performance and Kubernetes cluster health.
* Set up threshold-based alerts in Grafana for proactive monitoring of infrastructure and application anomalies.

**5. Security Enhancements**

5.1. **IAM Role Management**

* Regularly review and update IAM roles and policies to enforce the principle of least privilege.
* Implement multi-factor authentication (MFA) for critical AWS accounts to enhance security.

5.2. **Kubernetes Security Best Practices**

* Implement **Network Policies** to control traffic between pods and ensure that only necessary communication is allowed.
* Use **Pod Security Policies** to enforce security standards for deploying containers in your cluster.

**6. CI/CD Pipeline Improvements**

6.1. **Pipeline Optimization**

* Speed up the CI/CD process by using Docker image layer caching and parallelizing steps where possible.
* Use **static code analysis** tools like **SonarQube** or **CodeQL** to automate code quality checks as part of the CI pipeline.

6.2. **Automated Rollbacks**

* Implement automated rollbacks in the pipeline to revert to the last stable version if a deployment fails or if health checks detect issues post-deployment.
* Use **Blue-Green** or **Canary** deployment strategies to reduce the risk of downtime during updates.

**7. Cost Optimization**

7.1. **AWS Cost Management**

* Regularly analyze your AWS bill and use AWS **Cost Explorer** to identify areas where costs can be reduced.
* Use spot instances for non-critical workloads and right-size your instances to optimize costs.

7.2. **Resource Optimization**

* Identify and eliminate unused or underutilized resources in your EKS cluster.
* Implement Kubernetes **cluster autoscaling** to automatically scale nodes up or down based on workload demand.

**8. Future Scalability Planning**

8.1. **Microservice Architecture Enhancements**

* Plan for adding more microservices in the future with a focus on using **service mesh** technologies like **Istio** or **Linkerd** to manage service-to-service communication.
* Break down monolithic applications into microservices to enhance scalability, fault tolerance, and development agility.

8.2. **Cloud-Native Practices**

* Continuously adopt cloud-native practices and tools that align with your project’s architecture.
* Use tools like **ArgoCD** or **Flux** to manage GitOps workflows and ensure that the state of the cluster is always in sync with your Git repository.

**Summary of this Step**

* We've outlined a detailed plan for maintaining and optimizing the infrastructure, microservice deployment, secrets management, monitoring system, and CI/CD pipeline.
* Emphasis was placed on regular updates, security best practices, cost optimization, and future scalability.
* Proactive measures were suggested for improving performance, monitoring, and incident response.

**Final Thoughts**

This completes the project with a focus on long-term maintenance and optimization strategies. The system is now set up to be scalable, secure, and efficient with best practices for continuous improvement in place.

**Reference**

Here’s a list of valuable references that cover various aspects of this project, including Terraform, Helm, HashiCorp Vault, Kubernetes, Prometheus, Grafana, and GitHub Actions. These resources include YouTube videos, official documentation, and tutorials:

### Terraform

1. **Official Terraform Documentation**  
   Terraform by HashiCorp  
   Comprehensive guide covering everything from installation to advanced features.
2. **YouTube Tutorial: Terraform AWS Infrastructure**  
   [Terraform AWS Tutorial for Beginners](https://www.youtube.com/watch?v=2wP4pTVsC9c)  
   An introductory video on setting up AWS infrastructure using Terraform.

### Helm

1. **Official Helm Documentation**  
   Helm.sh  
   Detailed documentation on how to install and use Helm for managing Kubernetes applications.
2. **YouTube Tutorial: Kubernetes Helm Charts**  
   [Helm Tutorial - Deploying Applications on Kubernetes](https://www.youtube.com/watch?v=2f_Z5Q02_QA)  
   A video that explains Helm charts and how to deploy applications on Kubernetes using Helm.

### HashiCorp Vault

1. **Official Vault Documentation**  
   Vault by HashiCorp  
   Comprehensive documentation for installing and using Vault for secrets management.
2. **YouTube Tutorial: HashiCorp Vault Basics**  
   [HashiCorp Vault - Secrets Management](https://www.youtube.com/watch?v=9B9AWaf6LyI)  
   A beginner-friendly tutorial on how to set up and use HashiCorp Vault.

### Kubernetes

1. **Official Kubernetes Documentation**  
   Kubernetes.io  
   The official documentation for Kubernetes, covering everything from basic concepts to advanced topics.
2. **YouTube Tutorial: Kubernetes for Beginners**  
   [Kubernetes Full Course - Learn Kubernetes in 8 Hours](https://www.youtube.com/watch?v=X48VuZ4z9T4)  
   A comprehensive tutorial covering the fundamentals of Kubernetes.

### Prometheus & Grafana

1. **Official Prometheus Documentation**  
   Prometheus.io  
   The official guide on Prometheus, explaining how to set it up and monitor applications.
2. **Official Grafana Documentation**  
   Grafana.com  
   Detailed documentation on setting up and configuring Grafana.
3. **YouTube Tutorial: Monitoring with Prometheus and Grafana**  
   [Prometheus and Grafana - Monitoring System](https://www.youtube.com/watch?v=9o8FBBfT4xM)  
   A tutorial on how to set up monitoring with Prometheus and Grafana.

### GitHub Actions

1. **Official GitHub Actions Documentation**  
   [GitHub Actions Docs](https://docs.github.com/en/actions)  
   Comprehensive guide on using GitHub Actions for CI/CD pipelines.
2. **YouTube Tutorial: GitHub Actions for Beginners**  
   [GitHub Actions Tutorial - CI/CD Pipeline](https://www.youtube.com/watch?v=R8_veQiYMjI)  
   A detailed video tutorial on setting up CI/CD with GitHub Actions.

### Additional Resources

1. **Kubernetes Patterns**  
   Kubernetes Patterns Book  
   A great resource to understand design patterns for Kubernetes.
2. **DevOps Handbook**  
   The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win  
   While not a technical manual, this book provides insights into DevOps principles.

**ROUGH WORK**

**Complete File Structure with sample-microservice**

/devops-project

│

├── /terraform # Terraform configuration files

│ ├── main.tf

│ ├── variables.tf

│ ├── outputs.tf

│ └── provider.tf

│

├── /microservice-deployment # Kubernetes deployment and Helm chart files

│ ├── /charts

│ │ ├── /my-sample-service

│ │ ├── Chart.yaml

│ │ ├── values.yaml

│ │ └── /templates

│ │ ├── deployment.yaml

│ │ ├── service.yaml

│ │ └── configmap.yaml

│ ├── /scripts

│ │ ├── deploy.sh

│ │ ├── debug.sh

│ │ └── cleanup.sh

│ ├── /logs

│ │ ├── deployment-logs.log

│ │ └── error-trace.log

│ ├── /docs

│ │ ├── troubleshooting.md

│ │ └── error-codes.md

│ ├── helm-debug-notes.md

│ ├── kubectl-debug-guide.md

│ ├── terraform-debug-guide.md

│ └── README.md

│

├── /sample-microservice # Sample microservice application code

│ ├── /src

│ │ ├── app.py

│ │ └── requirements.txt

│ ├── Dockerfile

│ ├── README.md

│ └── .dockerignore

│

├── .gitignore # Ignore files for version control

├── README.md # Main documentation file for the project

└── LICENSE # License file for the project (if applicable)

# EKS Cluster Setup

module "eks" {

source = "terraform-aws-modules/eks/aws"

cluster\_name = "${var.project\_name}-eks-cluster"

cluster\_version = "1.21"

vpc\_id = module.vpc.vpc\_id

node\_groups = {

eks\_node = {

desired\_capacity = 2

max\_capacity = 3

min\_capacity = 1

instance\_type = "t2.medium"

key\_name = var.ec2\_key\_pair

}

}

}

**Checklist of Commands to Run:**

# Install Git

sudo apt-get install git

# Install Docker

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io

# Install kubectl

curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"

chmod +x kubectl

sudo mv kubectl /usr/local/bin/

# Install Terraform

sudo apt-get update && sudo apt-get install -y gnupg software-properties-common

curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg

sudo apt-add-repository "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb\_release -cs) main"

sudo apt-get update && sudo apt-get install terraform

# Install AWS CLI

curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"

unzip awscliv2.zip

sudo ./aws/install

# Install Helm

curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash

# Install Vault

sudo apt-get install vault

# Install Python 3.x

sudo apt-get install python3 python3-pip

# Install Grafana

sudo apt-get install grafana

PROBLEMS FACED:



