ENPM 818N - Cloud Computing

Final Group Project

Deploying OpenTelemetry Application in EKS

Submitted By – Group 18

Name	Directory ID	University ID	Section
Aryan Kulshrestha	ak17	120342035	0201
Aashay Mehta	amehta01	119577028	0201
Long Le	lle38	116223775	0201
Gideon Marcus Jayakumar	gideonmm	121182067	0201
Ali Huangz	ahuangz	122061767	0201

Video Walkthrough Link: https://drive.google.com/file/d/1_hY2j5fRadtY-bleq-tVfo8An1EsEnmW/view?usp=sharing

Repository link: https://github.com/Aashay011/opentelemetry-demo

INDEX

Introduction

This project explores the complete lifecycle of deploying, monitoring, and automating a cloud-native application using DevOps tools. It is structured across multiple independent phases to simulate real-world production scenarios using container orchestration, monitoring stacks, performance testing, and CI/CD pipelines, all within AWS cloud infrastructure and GitHub.

1.1 Project Overview

The application runs a real-world OpenTelemetry demo platform (similar to an e-commerce system), backed by containerized microservices such as frontend, checkout, cart-service, recommendation-service, etc. These services interact within a Kubernetes cluster to simulate realistic user flows.

The project is divided into four distinct phases:

- **Phase 1** involves local deployment using Docker Compose on an EC2 instance to validate basic service interaction and health. Kubernetes cluster is set up in AWS using eksctl and kubectl to complete task 2.
- **Phase 2** transitions the deployment to Amazon EKS using Helm charts, enabling scalability and native Kubernetes resource management.
- **Phase 3** focuses on service alerting and notifications, with Locust used to simulate load and assess system performance under stress.
- **Phase 4** emphasizes cloud-native observability and DevOps automation, integrating Prometheus, Grafana, and Jaeger for monitoring. A CI/CD pipeline is implemented using GitHub Actions to enable continuous delivery and rollback support.

1.2 Objective

The core objective of this project is to develop practical expertise in deploying and managing distributed microservices using industry-standard DevOps practices. We learned to containerize and orchestrate services, provision cloud infrastructure using Amazon EKS with eksctl, and implement observability through monitoring of metrics, logs, and traces. Automation was achieved through a CI/CD pipeline, enabling streamlined deployments and version control. Key focus areas included reliability and resilience, achieved through rollback mechanisms, secure secret handling, and namespace isolation. Ultimately, the project replicates a real-world production workflow centered on scalability, automation, and system observability.

Architecture Diagram of Infrastructure

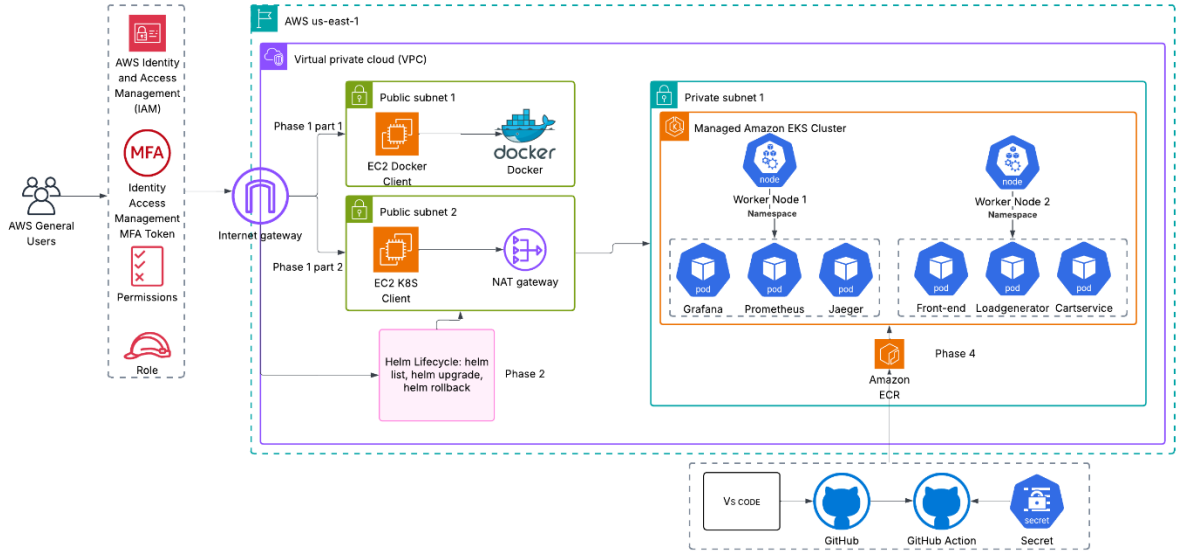


Figure 1: Main Architectural Diagram for the Project

Link: https://lucid.app/lucidchart/afb40287-ed7d-430e-82fd-29d4a1a145e0/edit?viewport_loc=-2726%2C-299%2C3867%2C1645%2C0_0&invitationId=inv_46ec701d-24dc-4376-97ec-b7e89e08f692

The above architecture illustrates the complete lifecycle of our microservices-based project across all four phases.

- Phase 1 begins with Docker-based deployment on an EC2 instance, followed by Kubernetes setup in a separate EC2 client.
- Phase 2 transitions into deploying Helm charts from the EC2 K8s client into a managed EKS cluster across private subnets.
- Phase 3 accounted for the monitoring tools like Prometheus, Grafana, and Jaeger, along with application microservices which are distributed across two worker nodes.
- Phase 4 integrates GitHub Actions for CI/CD, pushing Docker images to Amazon ECR and pulling them into the cluster for deployment. IAM roles, MFA, and secrets management help secure the environment end-to-end.

This architecture highlights how multiple AWS services and DevOps tools work simultaneously to allow scalable, securing, and automating the deployments.

Phase 1: Environment and Initial Application Setup

This phase lays down the base for the Kubernetes infrastructure setup. The Docker Deployment is independent from the other phases of the project.

3.1 Architecture Overview:

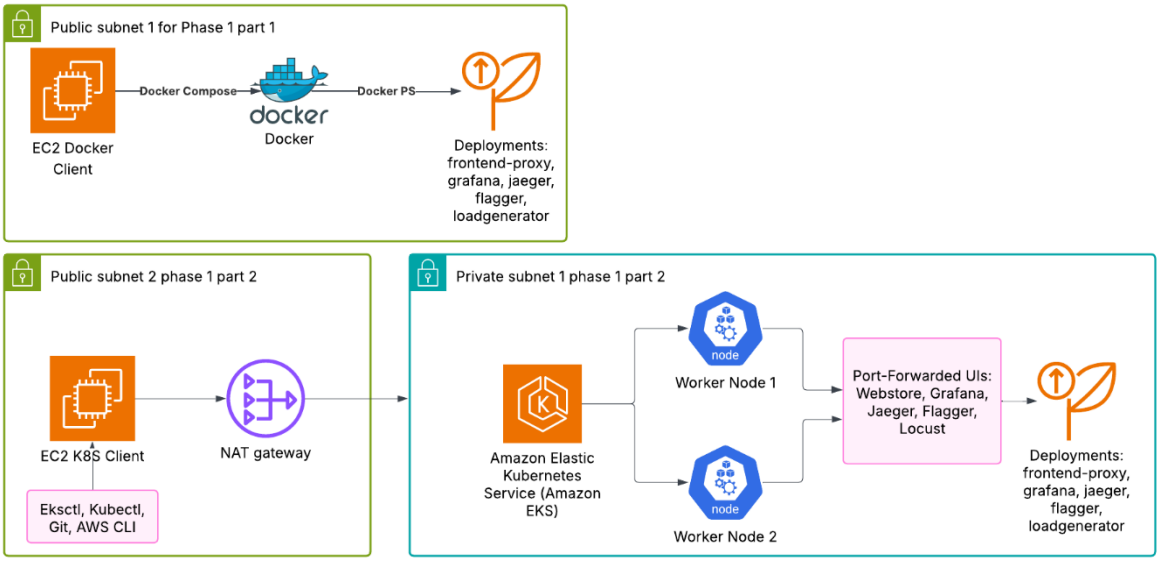


Figure 2: High-level architecture of Phase 1

Link: https://lucid.app/lucidchart/b6ecad76-6d34-43d0-8eb4-6f5f5398d04f/edit?viewport_loc=2701%2C-2956%2C3075%2C1308%2C0_0&invitationId=inv_0d133ed2-c055-4e0c-a968-18a24569f5a1

3.2 Implementation

Before beginning the deployment tasks, we prepared a secure and scalable cloud environment. IAM users were created and assigned to a user group named Developer, adhering to the principle of least privilege by attaching only the necessary policies required to interact with the AWS environment. Multi-Factor Authentication (MFA) was enforced for all users to enhance identity security. Additionally, we provisioned a custom Virtual Private Cloud (VPC) with two public and two private subnets, and configured a NAT Gateway to allow internet access from private subnets. The IPv4 CIDR range of the VPC was 10.0.0.0/16. Appropriate security groups were defined to allow traffic only on specific ports required for application access and cluster communication.

Phase 1 involved setting up the development environment and deploying the OpenTelemetry Demo application using two approaches: Docker and Kubernetes. The phase was divided into two primary tasks to demonstrate both local container orchestration and cloud-based microservices deployment.

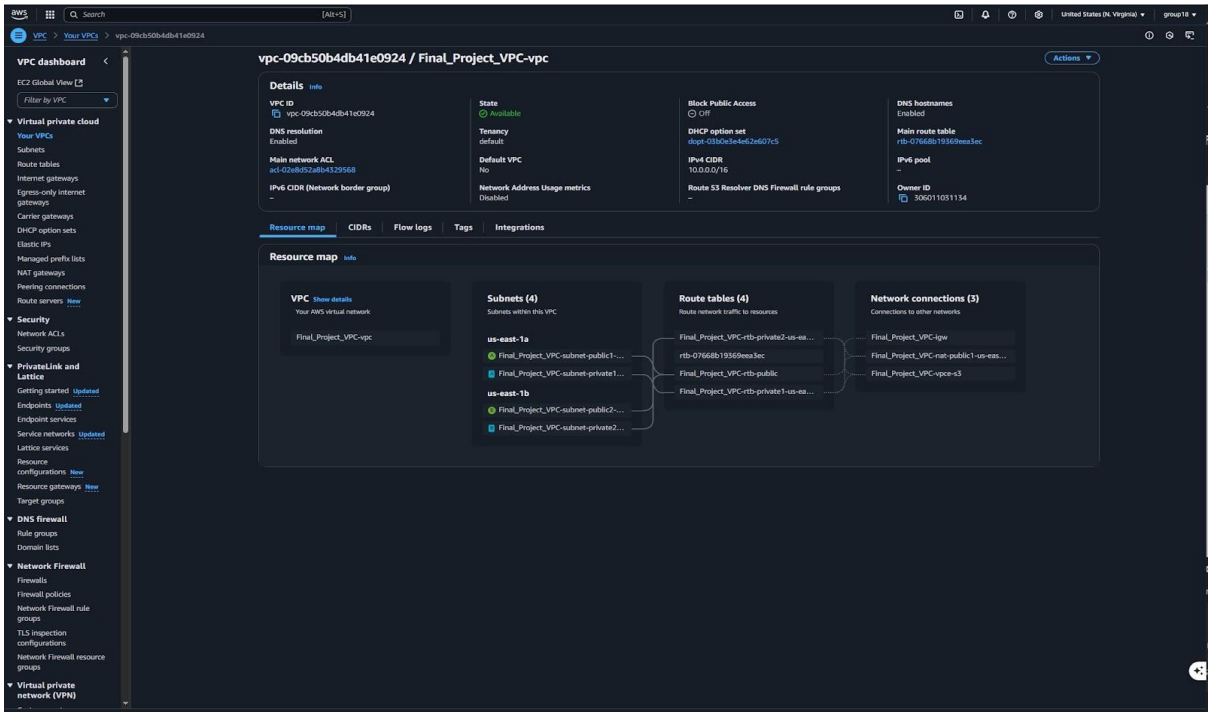


Figure 3: VPC Setup

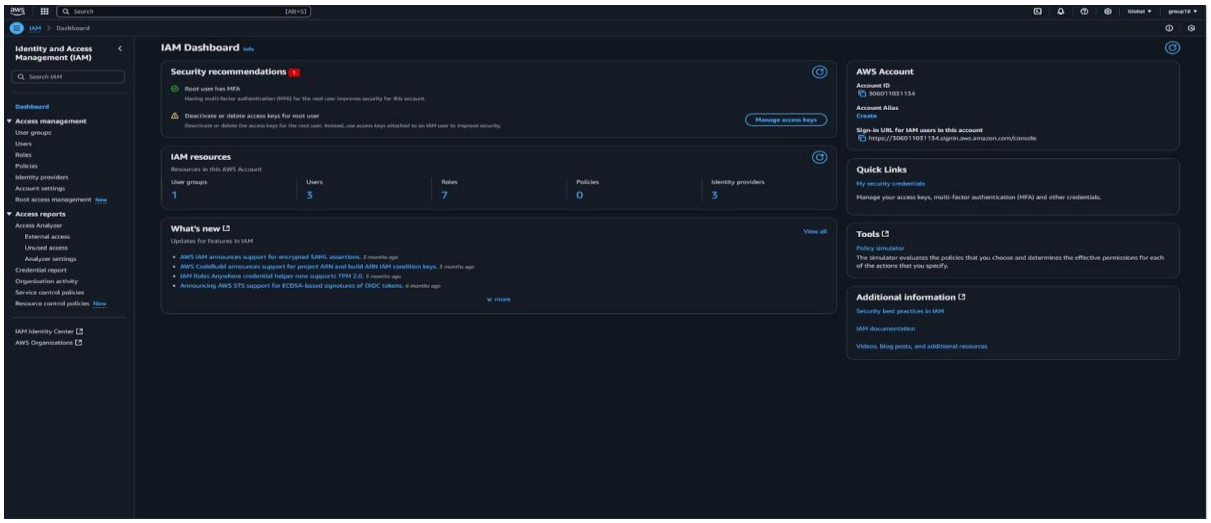


Figure 4: Root user with MFA

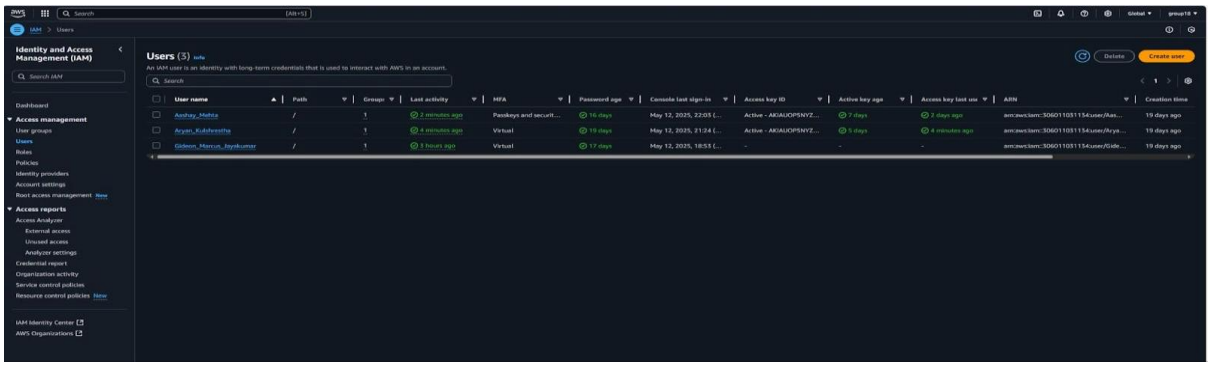


Figure 5: MFA enabled for every user

3.3 Task 1: Docker Deployment

The project began with deploying the OpenTelemetry Demo application on a single EC2 instance using Docker Compose. Git and Docker were installed on the instance, followed by cloning the official OpenTelemetry demo repository. After setting up the environment, Docker Compose was used to launch multiple services defined in the "docker-compose.yaml" file, including the frontend, backend services, Grafana, Jaeger, and load generator. Each container was validated using docker ps, and application endpoints were accessed via the EC2's public IP and mapped ports. The setup allowed the application to run in a self-contained environment for functional verification.

Step 1: Create VPC for our environment

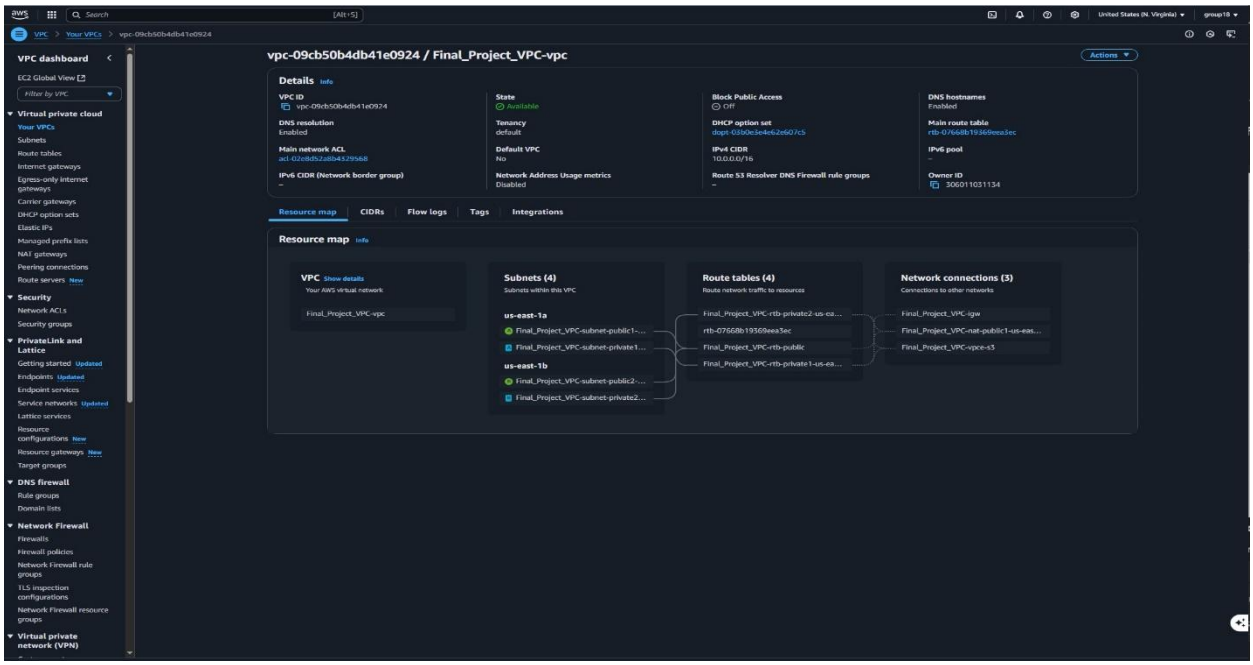


Figure 6: VPC

We created a custom VPC(Virtual Private Cloud) to provide an isolated network for the EC2 instance hosting our Docker containers.

Step 2: Create EC2 to host docker

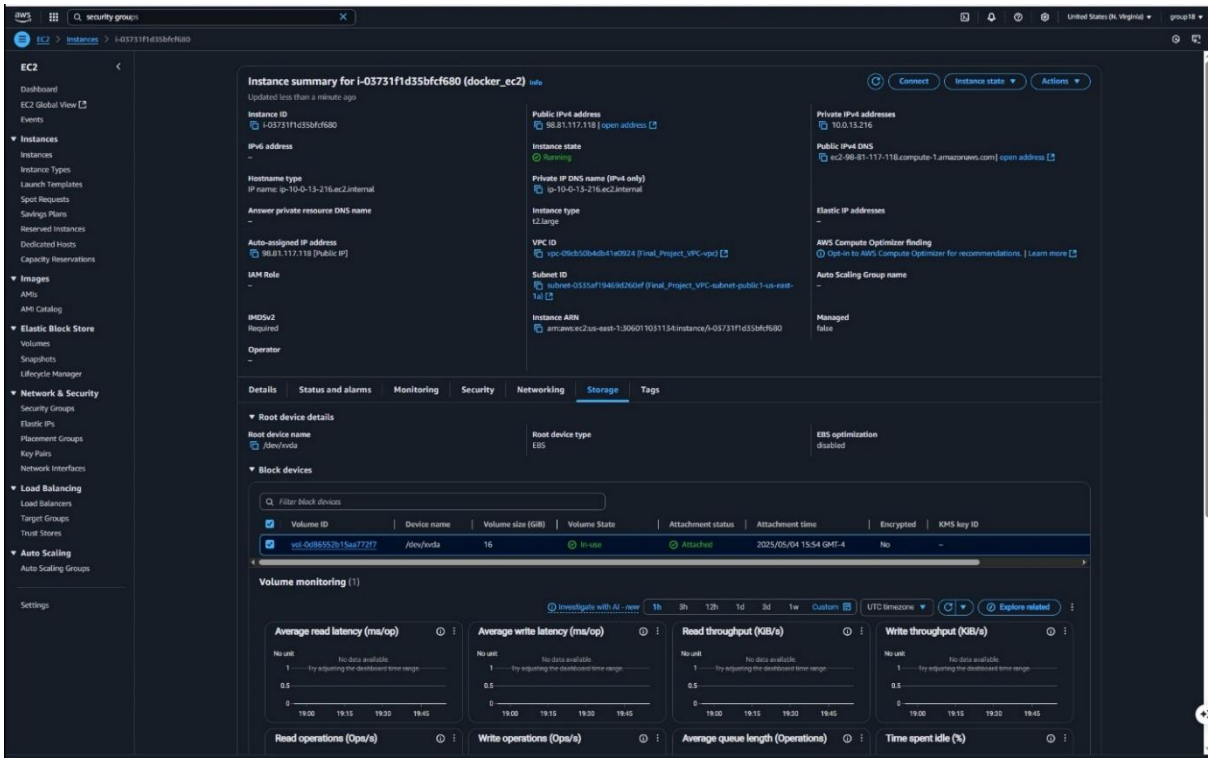


Figure 7: EC2 for Docker

We setup a dedicated EC2 instance provisioned using the Amazon Linux 2 AMI to install and configure Docker acting as a host for Docker.

Step 3: Create Security Group and allow port 22 for SSH connection.

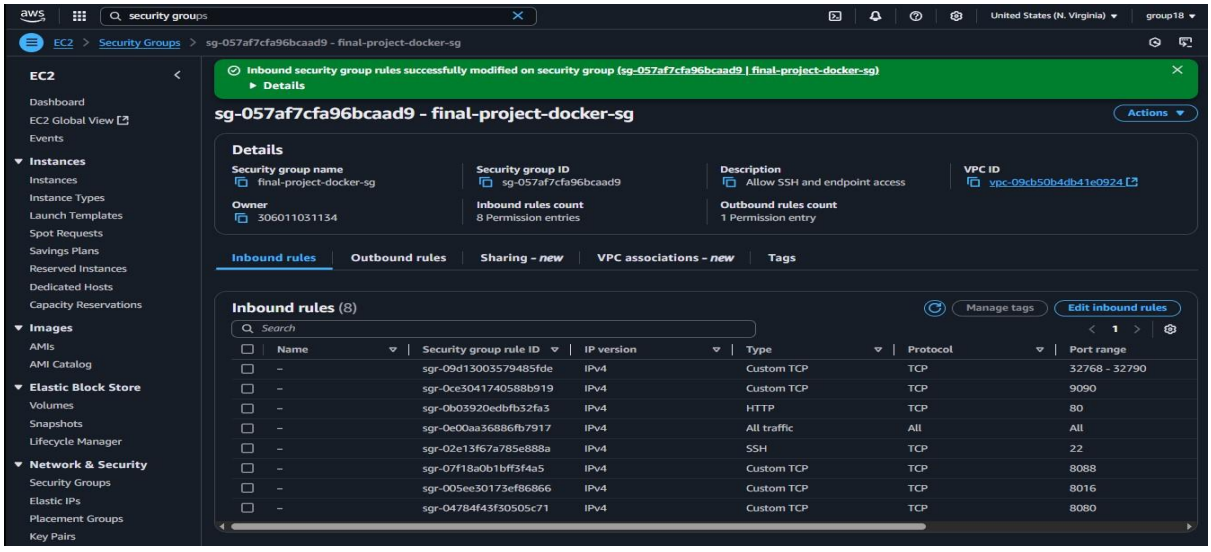


Figure 8: Security Group

We edited the inbound rules of the security group to allow for SSH access via port 22, this also enables future access to Docker containers via exposed ports such as port 8080.

Step 4: SSH into EC2 instance and install Git, docker and docker-compose

- Git: Sudo yum install git

Figure 9: Git Installation

- Docker: sudo amazon-linux-extras install docker -y

Figure 10: Docker Installation

- Docker-compose:

→ sudo curl -L https://github.com/docker/compose/release/latest/download/docker-compose-$(uname -s) -$(uname -m) \ -o /usr/local/bin/docker-compose

→sudo chmod +x /usr/local/bin/docker-compose

→ docker-compose –version

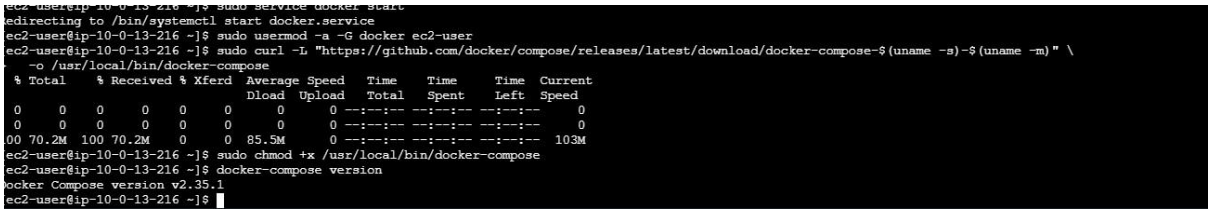


Figure 11: Docker-Compose Installation

- Git clone the repository: git clone https://github.com/open-telemetry/opentelemetry-demo.git

- Change Directory (cd) to the Opentelemtry folder: cd opentelemetry-demo

Step 5: Launch the Docker Environment:

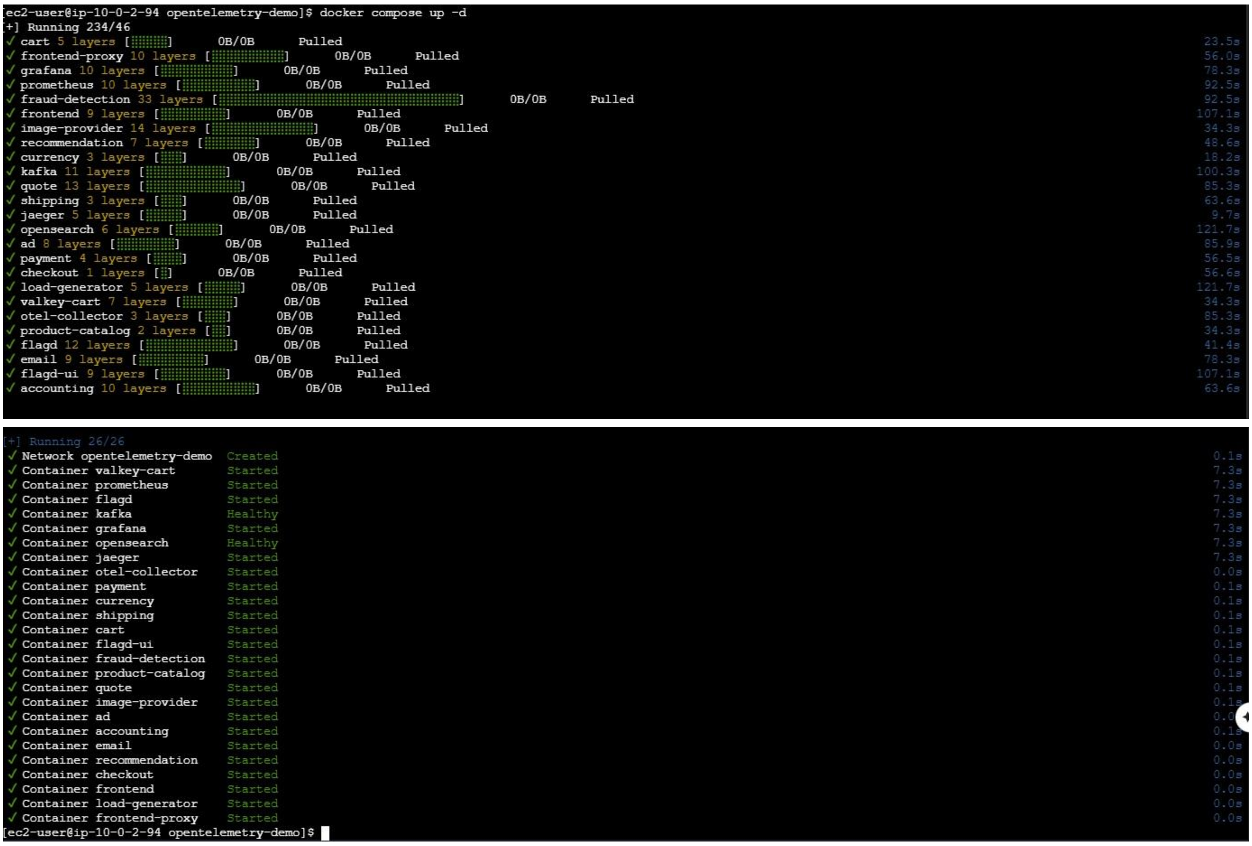


Figure 12-13: Launching Docker Environment

Step 6: Run Docker ps to get all the containers

Figure 14: Docker ps

- Validating by checking Docker-Compose Logs:

Figure 15: Docker Compose Logs

Step 7: View All the running containers on browser using:
Http://<EC2 public IP>: 8080(Port Number)

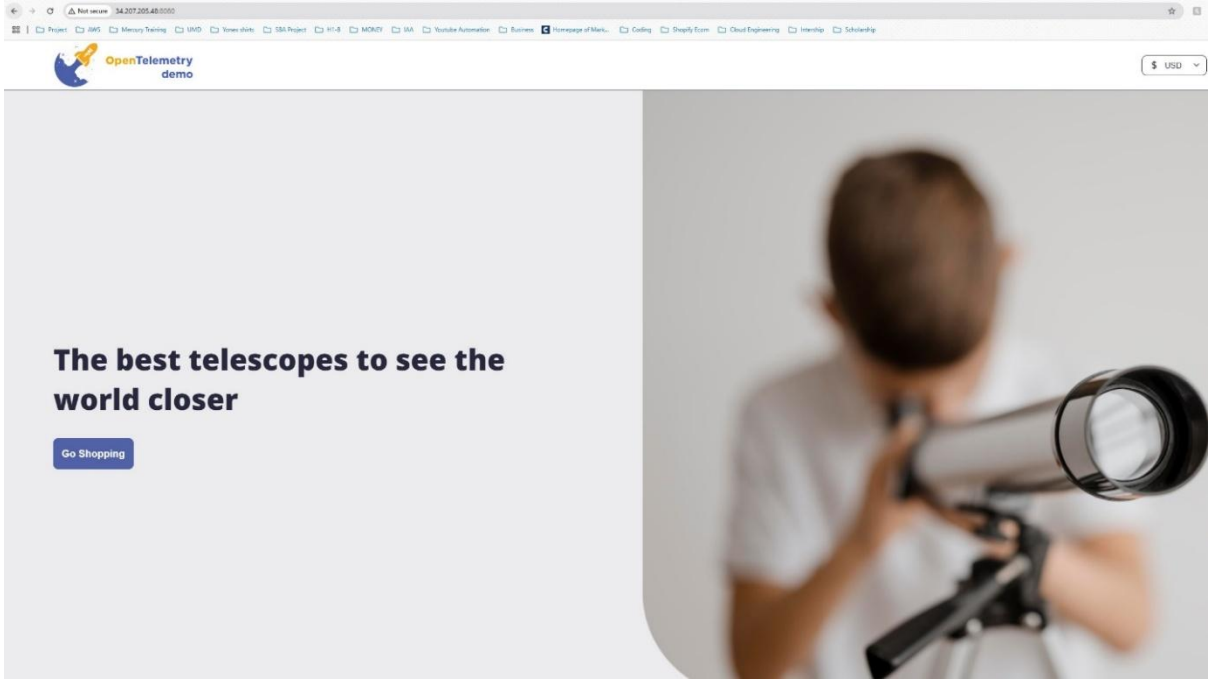


Figure 16: Frontend with port 8080: Http://<EC2 public IP>: 8080

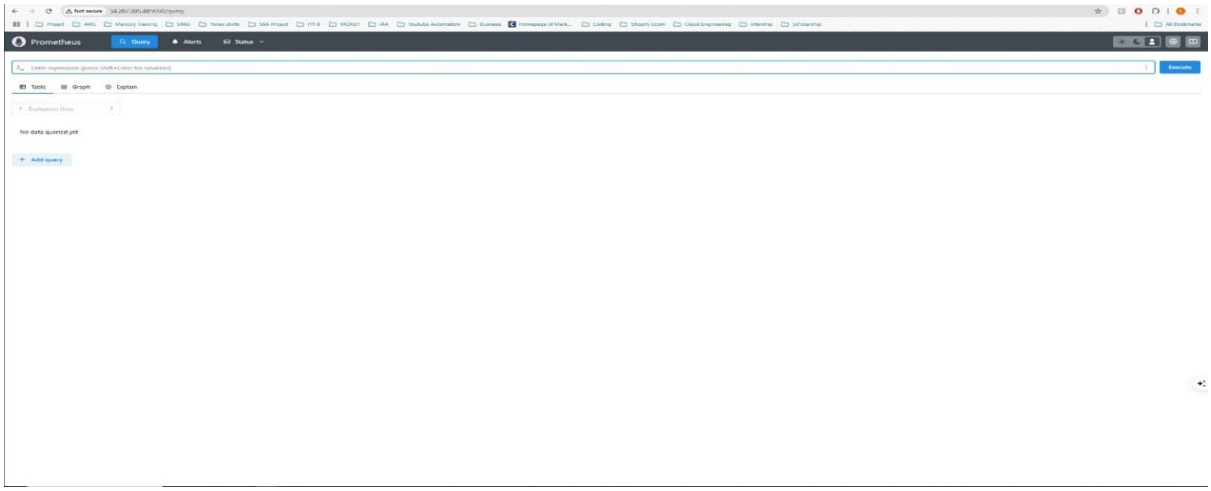


Figure 17: Prometheus with port 9090: Http://<EC2 public IP>: 9090

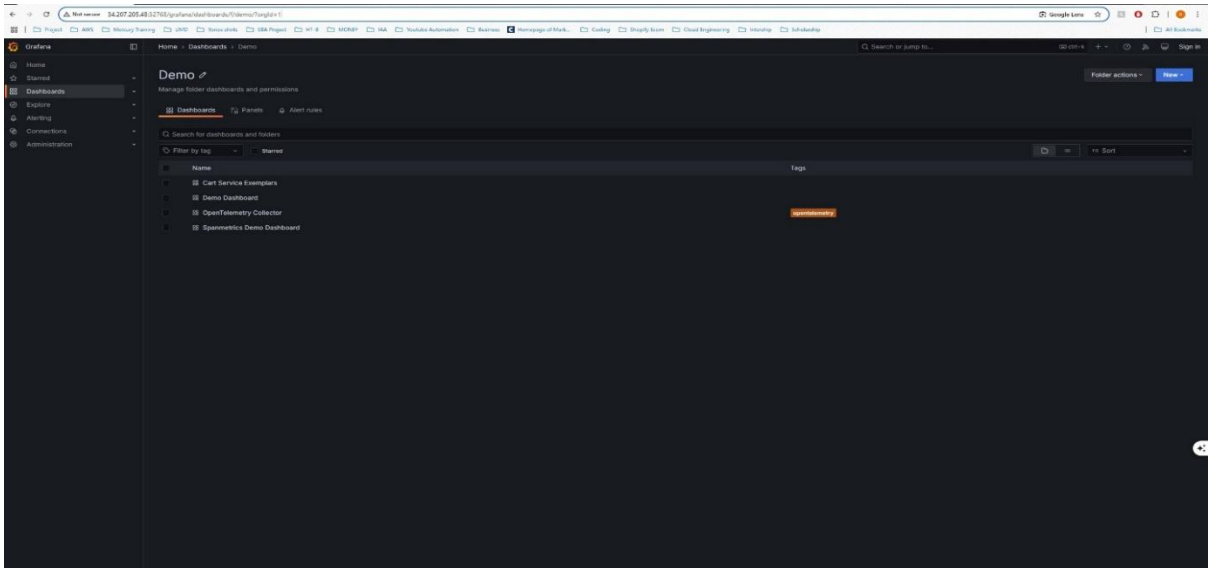


Figure 18: Grafana with port 32768: Http://<EC2 public IP>: 32768

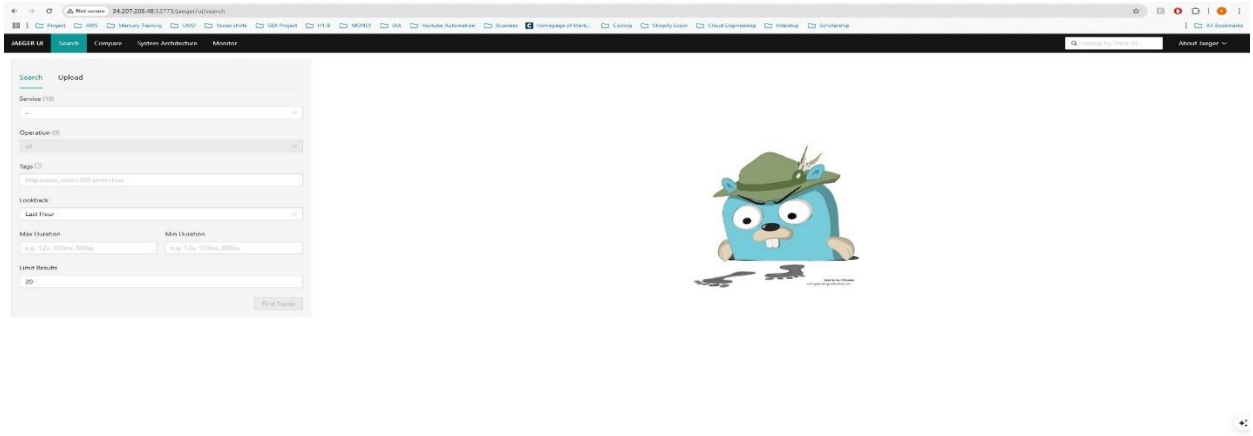


Figure 19: Jaeger with port 32778: Http://<EC2 public IP>: 3277

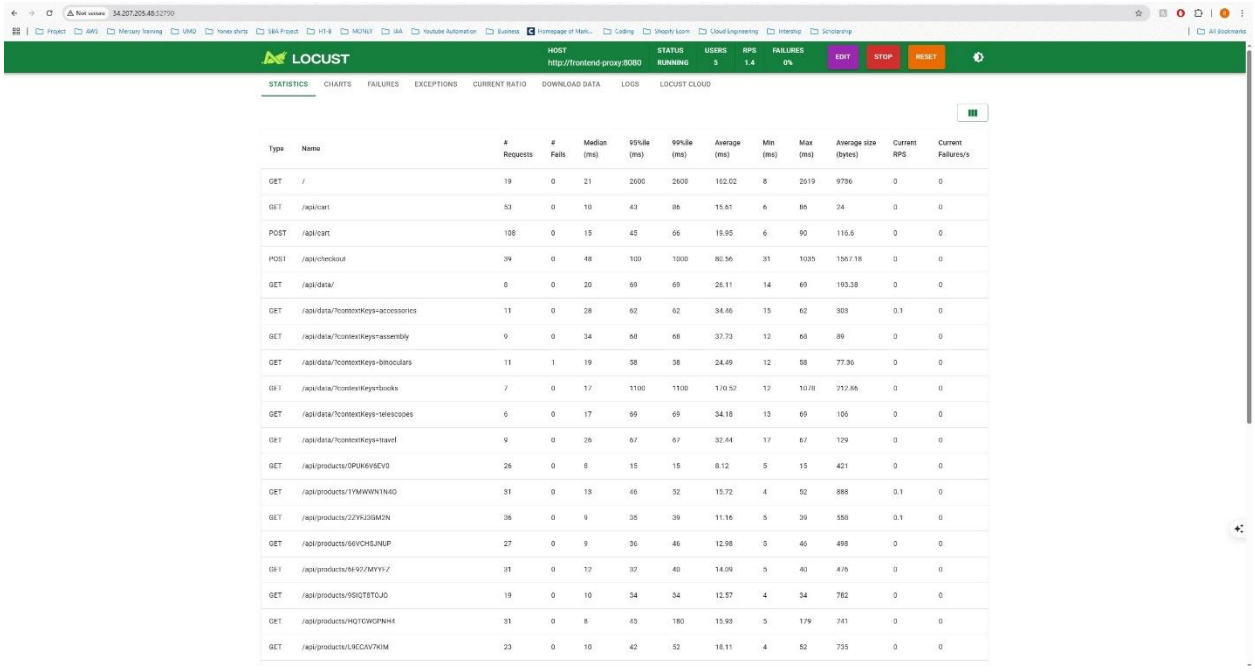


Figure 20: Load Generator with port 32790: Http://<EC2 public IP>: 32790

Step 8: Exiting and cleaning the environment
→ Command: *docker-compose down.* This is used to stop Docker and Exit from our EC2 instance. Further delete the EC2 to properly clean up phase 1 part 1.

Task 2: Kubernetes Setup Tasks

In the second part of Phase 1, the application was deployed to a cloud-based Kubernetes environment using Amazon EKS. A dedicated EC2 instance was provisioned as a client machine, and it was configured with kubectl, awscli, and eksctl to interact with the EKS cluster. Using this client, we created a new EKS cluster with two managed worker nodes. These worker nodes were launched in private subnets and configured to serve as the compute layer for deploying and running application workloads. Once the cluster was created and the client EC2 was mapped to it using aws eks update-kubeconfig, we deployed the OpenTelemetry Demo application using the provided opentelemetry-demo.yaml manifest file. All Kubernetes resources, including pods, services, and deployments, were launched in a dedicated namespace. We validated the deployment using kubectl get all -n otel-demo and confirmed that services such as the frontend, backend APIs, and observability components (e.g., Jaeger, Grafana) were running correctly on the worker nodes. Port forwarding and curl were used to access and verify the application endpoints from the client instance.

Step 1: Create an EC2 instance as an EKS Client

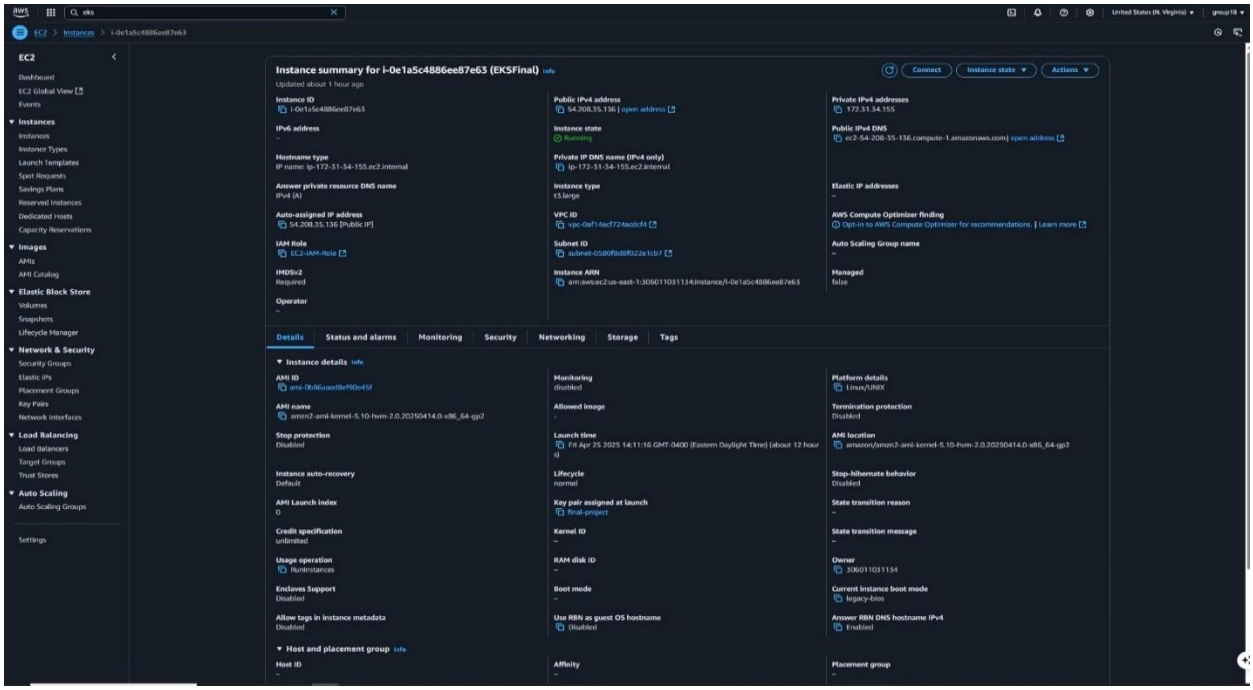


Figure 21:EC2 instance as EKS Client

Step 2: SSH into EC2 and install AWS CLI, kubetctl and eksctl

- AWS CLI Command:
  → curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
  → unzip awscliv2.zip
  → sudo ./aws/install
  → aws configure (for validation)
- Kubectl:
  →curl -LO "https://dl.k8s.io/release/v1.28.0/bin/linux/amd64/kubectl"
  →chmod +x kubectl
  →sudo mv kubectl /usr/local/bin
  →kubectl version –client (for validation)
- Eksctl:
  →curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
  →sudo mv /tmp/eksctl /usr/local/bin
  →eksctl version (for validation)

Step 3: Create EKS cluster

Missing picture for command and when the cluster is created

- EKS Cluster command:

  eksctl create cluster \ (EKS name)
  --name Final-Project \
  --version 1.27 \ (version of EKS)
  --region us-east-1 \ (region of EKS)
  --nodegroup-name final-workers \ (Node group and worker nodes for EKS)
  --node-type t3. large \ (Worker node size)
  --nodes 2\ (2 worker nodes)
  --nodes-min 2 \ (Minimum worker node set to 2)
  --nodes-max 2 \ (Maximum worker node set 2)
  --managed

- EKS Cluster Configuration:

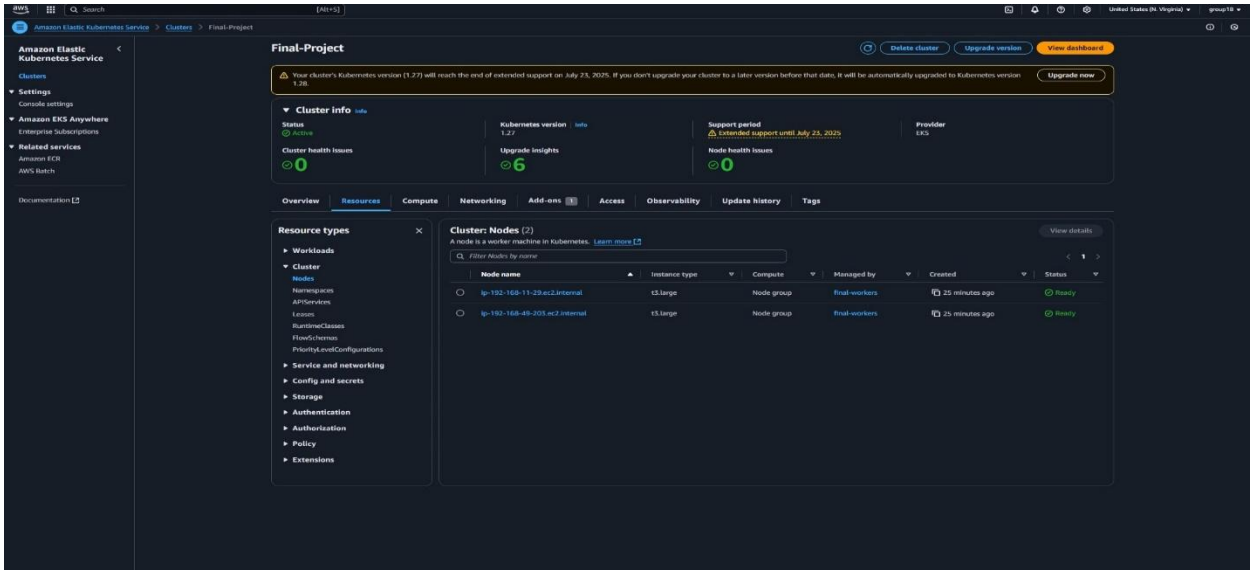


Figure 22: EKS Cluster Configuration

- EKS cluster information:

```
[ec2-user@ip-172-31-34-155 ~]$ kubectl cluster-info
Kubernetes control plane is running at https://FCFF40AD82FCB49387A760727E6760EF.gr7.us-east-1.eks.amazonaws.com
CoreDNS is running at https://FCFF40AD82FCB49387A760727E6760EF.gr7.us-east-1.eks.amazonaws.com/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Figure 23: EKS Cluster Information

- EKS Worker Nodes

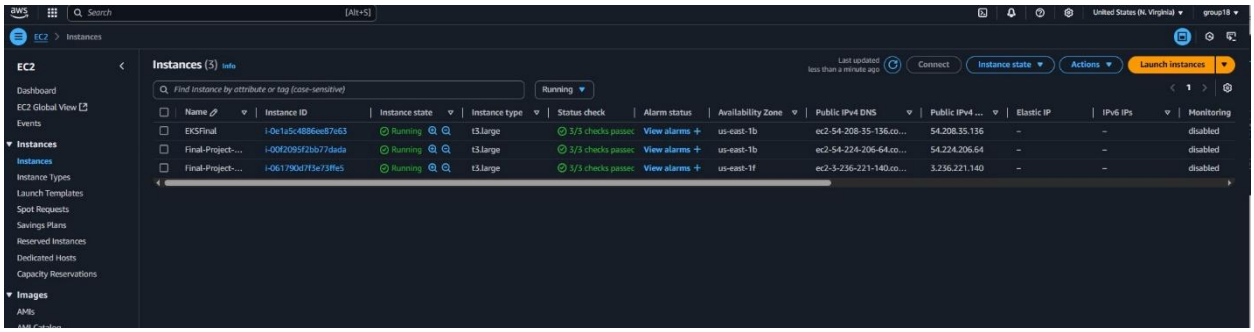


Figure 24: EKS Worker Nodes

- Worker Node configuration:

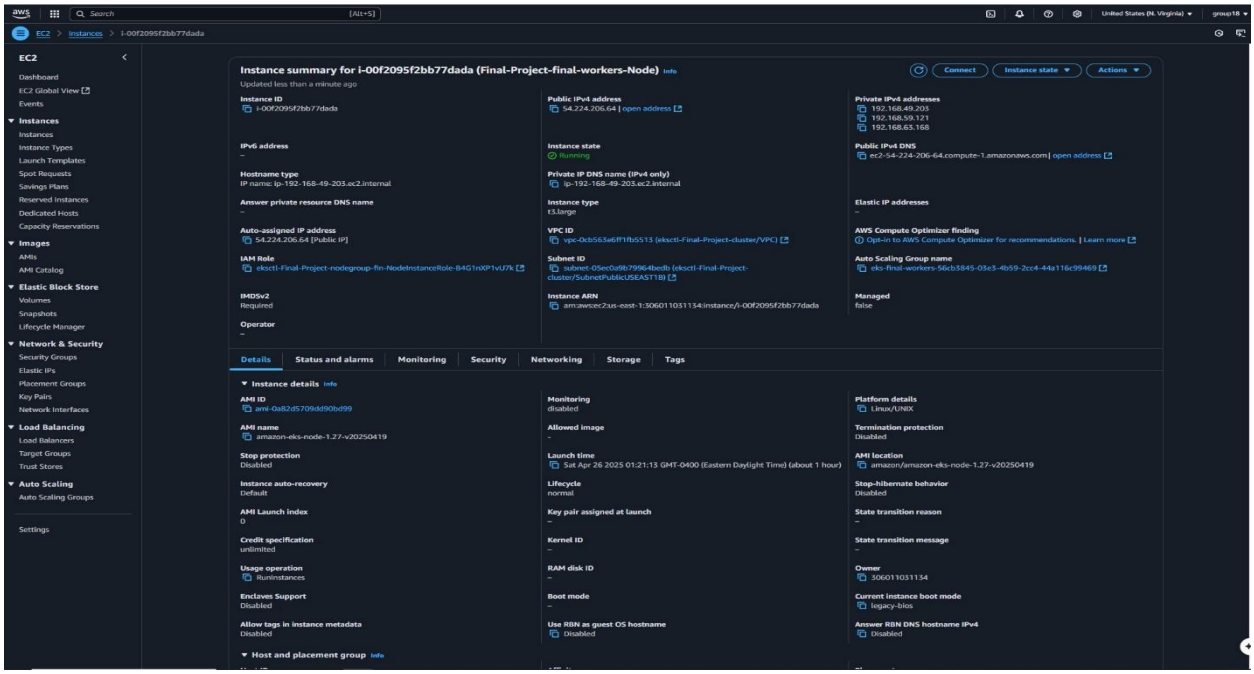


Figure 25: Worker Nodes configuration

Step 4: Git clone the Application, Create namespace, Deploy the application and validate

- git clone https://github.com/open-telemetry/opentelemetry-demo.git
- Change Directory: cd opentelemetry-demo/kubernetes-manifests
- Create namespace: kubectl create namespace otel-demo
- Deploy the application: kubectl apply -f opentelemetry-demo.yaml -n otel-demo
- Validating the deployment by getting all the pods: kubectl get all -n otel-demo (must use the correct namespace) and kubectl get all -n Final-Project
- Confirm Cluster is working: Kubectl get nodes

```
[ec2-user@ip-172-31-34-155 ~]$ kubectl get all -n otel-demo
NAME                                       READY   STATUS              RESTARTS   AGE
pod/accounting-589fc85bd6-d9vgn            1/1     Running             0          45m
pod/ad-6f9f7c55b5-9ppfb                    1/1     Running             0          45m
pod/cart-c45d9c65b-7bz84                   1/1     Running             0          45m
pod/checkout-cc7d8b89b-kvmzz               1/1     Running             0          45m
pod/currency-76f56875ff-blglf              1/1     Running             0          45m
pod/email-687f7c5796-pdcg8                 1/1     Running             0          45m
pod/flagd-cb96db58d-gj44x                  2/2     Running             0          45m
pod/fraud-detection-586d6d65f7-dxh4h       1/1     Running             0          45m
pod/frontend-7bb85994d9-45rl9              1/1     Running             0          45m
pod/frontend-proxy-bdc67d8cd-2pwg8         1/1     Running             0          45m
pod/grafana-dd7df5895-lv8hd                0/1     ContainerCreating   0          45m
pod/image-provider-56bdc5ffdc-jd29c        1/1     Running             0          45m
pod/jaeger-95c8ddd97-cqjt4                 1/1     Running             0          45m
pod/kafka-689c8dcd79-cvdhx                 1/1     Running             0          45m
pod/load-generator-867546c577-5pvfq        1/1     Running             0          45m
pod/opensearch-0                           1/1     Running             0          45m
pod/otel-collector-795db585d5-h2lrz        1/1     Running             0          45m
pod/payment-7b7578c9b4-dbj9c               1/1     Running             0          45m
pod/product-catalog-745c446cf7-jwzhg       1/1     Running             0          45m
pod/prometheus-8844b5f97-thqzm             1/1     Running             0          45m
pod/quote-7c664fcf59-5n6lc                 1/1     Running             0          45m
pod/recommendation-7c9db7dc9c-dl2qq        1/1     Running             0          45m
pod/shipping-5d9cfd9677-tkpmz              1/1     Running             0          45m
pod/valkey-cart-54ccd9fd94-kjmfm           1/1     Running             0          45m
```

Figure 26: All pods running in EKS

```
[ec2-user@ip-172-31-34-155 kubernetes]$ kubectl get all -n final-project
NAME                                       READY   STATUS              RESTARTS      AGE
pod/accounting-589fc85bd6-cdrlj            1/1     Running             1 (14m ago)   16m
pod/ad-6f9f7c55b5-mqdm7                    1/1     Running             0             16m
pod/cart-c45d9c65b-kbdrm                   1/1     Running             0             16m
pod/checkout-cc7d8b89b-bjwk2               1/1     Running             0             16m
pod/currency-76f56875ff-xzbj4              1/1     Running             0             16m
pod/email-687f7c5796-z9sks                 1/1     Running             0             16m
pod/flagd-cb96db58d-x7lnn                  0/2     Init:0/1            0             16m
pod/fraud-detection-586d6d65f7-qjmrz       1/1     Running             0             16m
pod/frontend-7bb85994d9-xr5th              1/1     Running             0             16m
pod/frontend-proxy-bdc67d8cd-fsnt9         1/1     Running             0             16m
pod/image-provider-56bdc5ffdc-tzt58        1/1     Running             0             16m
pod/jaeger-95c8ddd97-hw77c                 1/1     Running             0             16m
pod/kafka-689c8dcd79-nr8qt                 1/1     Running             0             16m
pod/load-generator-867546c577-ds2wd        1/1     Running             0             16m
pod/opensearch-0                           1/1     Running             0             16m
pod/payment-7b7578c9b4-mr9dm               1/1     Running             0             16m
pod/product-catalog-745c446cf7-s5rhf       0/1     ContainerCreating   0             16m
pod/quote-7c664fcf59-czrpq                 1/1     Running             0             16m
pod/recommendation-7c9db7dc9c-t6kps        1/1     Running             0             16m
pod/shipping-5d9cfd9677-p4vcr              1/1     Running             0             16m
pod/valkey-cart-54ccd9fd94-ppzk2           1/1     Running             0             16m

NAME                            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                                                    AGE
service/ad                      ClusterIP   10.100.233.79    <none>        8080/TCP                                                   16m
service/cart                    ClusterIP   10.100.24.220    <none>        8080/TCP                                                   16m
service/checkout                ClusterIP   10.100.184.92    <none>        8080/TCP                                                   16m
service/currency                ClusterIP   10.100.182.221   <none>        8080/TCP                                                   16m
service/email                   ClusterIP   10.100.120.115   <none>        8080/TCP                                                   16m
service/flagd                   ClusterIP   10.100.27.153    <none>        8013/TCP,8016/TCP,4000/TCP                                 16m
service/frontend                ClusterIP   10.100.228.174   <none>        8080/TCP                                                   16m
service/frontend-proxy          ClusterIP   10.100.194.122   <none>        8080/TCP                                                   16m
service/image-provider          ClusterIP   10.100.10.178    <none>        8081/TCP                                                   16m
service/jaeger-agent            ClusterIP   None             <none>        5775/UDP,5778/TCP,6831/UDP,6832/UDP                        16m
service/jaeger-collector        ClusterIP   None             <none>        9411/TCP,14250/TCP,14267/TCP,14268/TCP,4317/TCP,4318/TCP   16m
service/jaeger-query            ClusterIP   None             <none>        16686/TCP,16685/TCP                                        16m
service/kafka                   ClusterIP   10.100.45.38     <none>        9092/TCP,9093/TCP                                          16m
service/load-generator          ClusterIP   10.100.97.27     <none>        8089/TCP                                                   16m
service/opensearch              ClusterIP   10.100.106.233   <none>        9200/TCP,9300/TCP,9600/TCP                                 16m
service/opensearch-headless     ClusterIP   None             <none>        9200/TCP,9300/TCP,9600/TCP                                 16m
service/payment                 ClusterIP   10.100.200.92    <none>        8080/TCP                                                   16m
service/product-catalog         ClusterIP   10.100.122.158   <none>        8080/TCP                                                   16m
service/quote                   ClusterIP   10.100.40.42     <none>        8080/TCP                                                   16m
service/recommendation          ClusterIP   10.100.228.59    <none>        8080/TCP                                                   16m
service/shipping                ClusterIP   10.100.15.173    <none>        8080/TCP                                                   16m
service/valkey-cart             ClusterIP   10.100.201.123   <none>        6379/TCP                                                   16m

NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/accounting            1/1     1            1           16m
deployment.apps/ad                    1/1     1            1           16m
deployment.apps/cart                  1/1     1            1           16m
deployment.apps/checkout              1/1     1            1           16m
deployment.apps/currency              1/1     1            1           16m
deployment.apps/email                 1/1     1            1           16m
deployment.apps/flagd                 0/1     1            0           16m
deployment.apps/fraud-detection       1/1     1            1           16m
deployment.apps/frontend              1/1     1            1           16m
deployment.apps/frontend-proxy        1/1     1            1           16m
deployment.apps/image-provider        1/1     1            1           16m
deployment.apps/jaeger                1/1     1            1           16m
deployment.apps/kafka                 1/1     1            1           16m
deployment.apps/load-generator        1/1     1            1           16m
deployment.apps/payment               1/1     1            1           16m
deployment.apps/product-catalog       0/1     1            0           16m
deployment.apps/quote                 1/1     1            1           16m
deployment.apps/recommendation        1/1     1            1           16m
```

Figure 27: Status of Pods, Services and Deployments

Step 5: Check Logs: kubectl logs -n otel-demo deployment/frontend-proxy (for frontend)

Figure 28: Logs of Frontend-proxy

Step 6: Access the Application via port forwarding on Web Browser

- kubectl port-forward svc/frontendproxy 8080:8080 -n otel-demo

Figure 29: Logs of Frontend-proxy

After running SSH into the same EC2 and run curl http://localhost:8080 so this way, the port-forward stays open in the first session, and curl works in the second.

- Front-end Proxy: localhost:8080

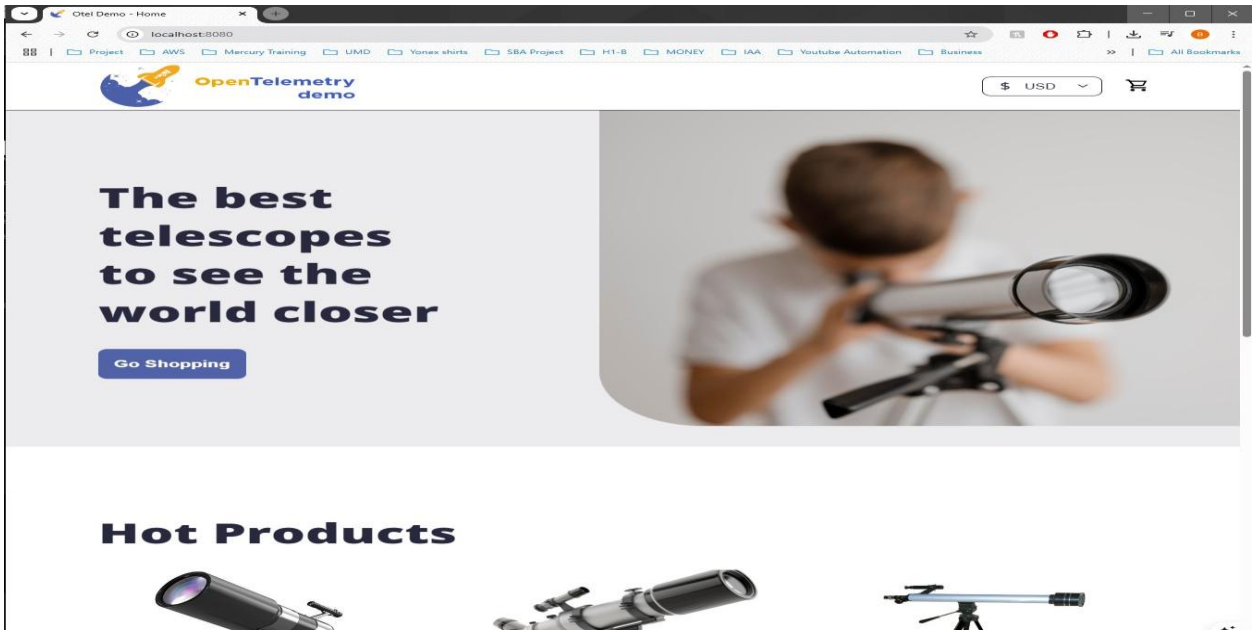


Figure 30: Frontend-proxy

- Jaeger: Localhost:8080/jaeger

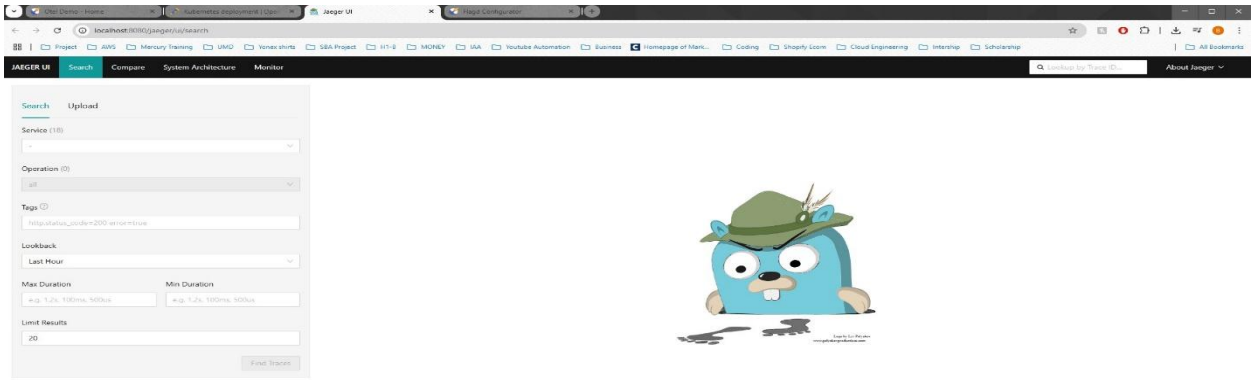


Figure 31: Jaeger

- Load Generator: localhost:8080/loadgen

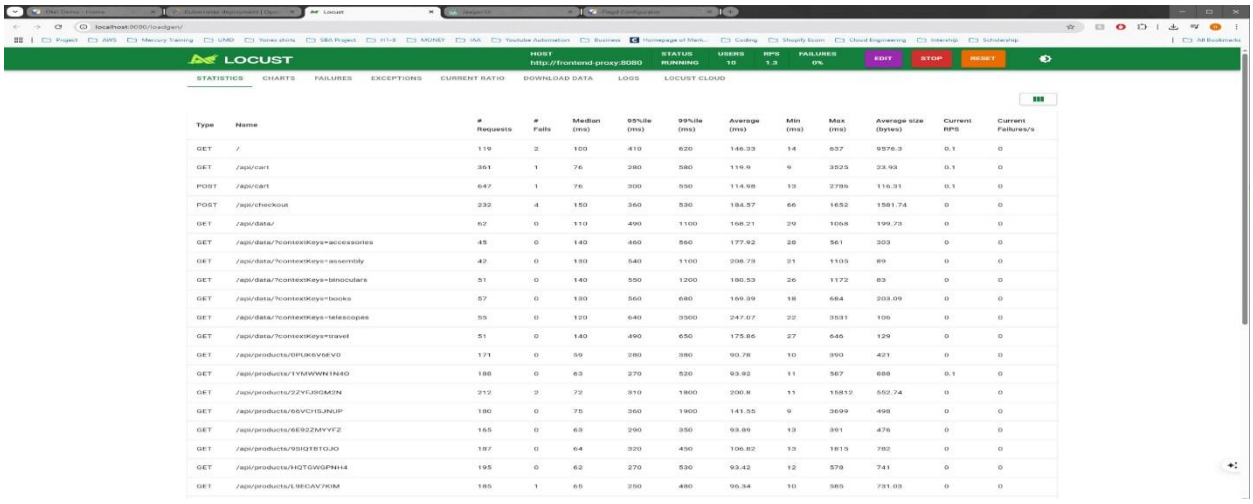


Figure 32: Load Generator

- Flagd: localhost:8080/feature

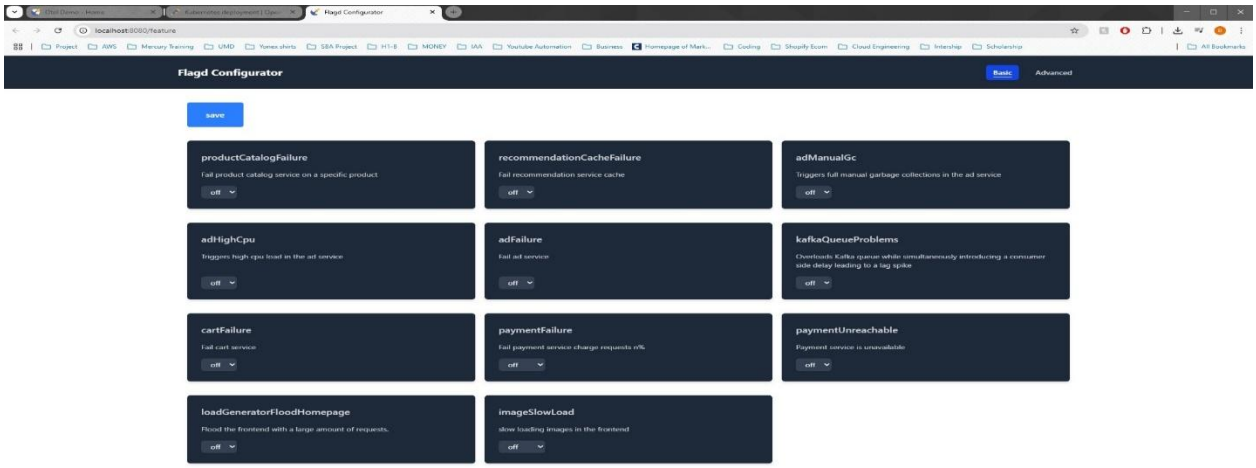


Figure 33: FlagD

- Grafana: localhost:8080/grafana/

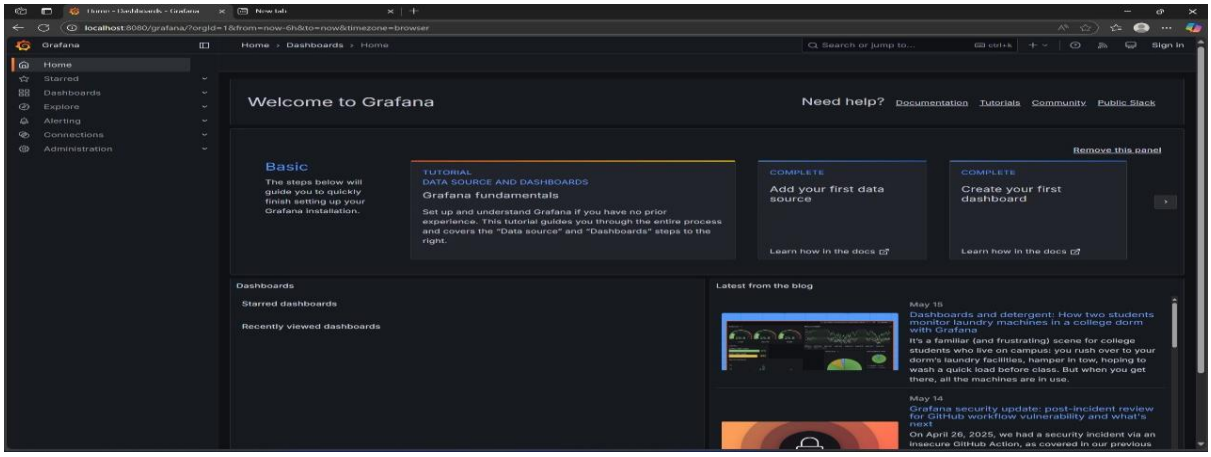


Figure 34: Grafana

3.4 Challenges Encountered

During Task 2, one of the main challenges encountered was related to the Grafana service. While most components of the OpenTelemetry Demo application deployed successfully to the EKS cluster, Grafana failed to start properly due to a configuration size limit in Kubernetes. The prebuilt opentelemetry-demo.yaml manifest attempted to load large dashboard definitions into a single ConfigMap, exceeding the default size limit of 1MB. This caused the Grafana pod to crash and it remained in a "creating" state.

To address this issue, we troubleshooted and figured out that commenting out other services apart from Grafana, will allow the Grafana pod to create when we run the below command. *"kubectl create --namespace otel-demo -f https://raw.githubusercontent.com/open-telemetry/opentelemetry-demo/main/kubernetes/opentelemetry-demo.yaml"*. Once the pod is loaded, we perform port-forwarding on localhost port 8080, making the Grafana dashboard load properly this time. The original command states to use "kubectl apply" instead of "kubectl create" but using this the dashboard loaded, pod status changed to 1, and with the help of documentation provided to us, we came across the major fix that is covered in Phase 2 of the project i.e. Helm. Helm provides a way to manage Kubernetes resources with its templates helping in automatically splitting large ConfigMaps; which in our case helped in resolving the issue to ensure Grafana dashboard loads correctly (with large byte size) in a production environment.

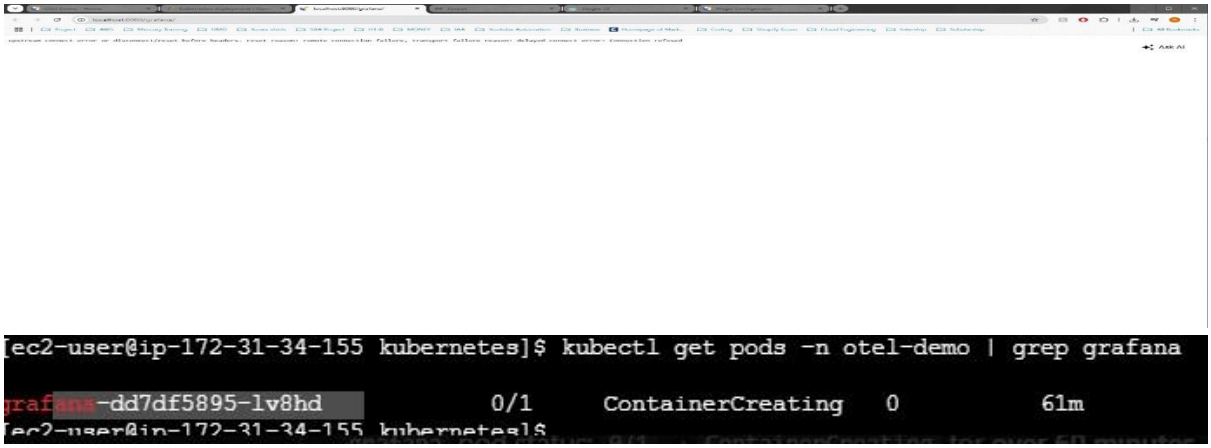


Figure 35: Grafana is not working

The above two screenshots show that grafana didn't work at first but we managed to get the endpoint, which is presented in task 2 of the phase 1.

Phase 2: Integrating Helm for Deployment

This phase shifts from manual YAML deployments to using Helm, a tool that simplifies managing Kubernetes applications. We redeployed the OpenTelemetry Demo with Helm to make deployments more scalable, modular, and easier to manage.

4.1 Architecture Overview

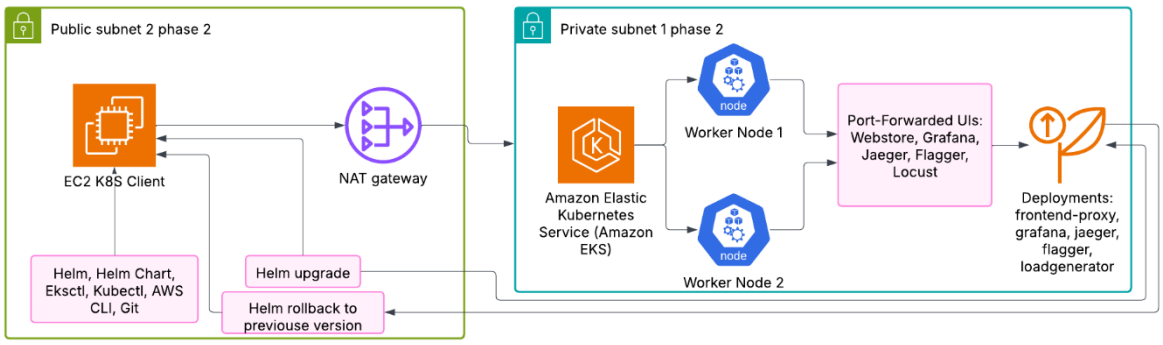


Figure 36: High-level architecture of Phase 2

Link: https://lucid.app/lucidchart/31484001-bcc1-43c4-b6bb-f4e8f24c80f3/edit?viewport_loc=-1317%2C172%2C3694%2C1571%2C0_0&invitationId=inv_d1c02ae7-c631-495c-97c5-1641121493cd

4.2 Implementation

To implement Helm-based deployment, we began by preparing our EKS environment on an EC2 instance that served as the control node. We installed and validated the required CLI tools, helm (v3.17.3), kubectl (v1.27), and connected them to our Kubernetes cluster running (v1.27). A dedicated namespace, *"otel-helm-demo"*, was created for the deployment. After adding the official OpenTelemetry Helm chart repository, we deployed all required microservices and observability tools, including Grafana, Jaeger, and the load generator. Helm's use of a values.yaml file allowed us to define configuration parameters such as replica counts, ports, and environment variables, which were dynamically inserted into reusable Kubernetes templates. This approach simplified application management and made version-controlled updates and rollbacks more reliable. It also helped resolve earlier deployment challenges, such as oversized ConfigMaps and manual YAML edits.

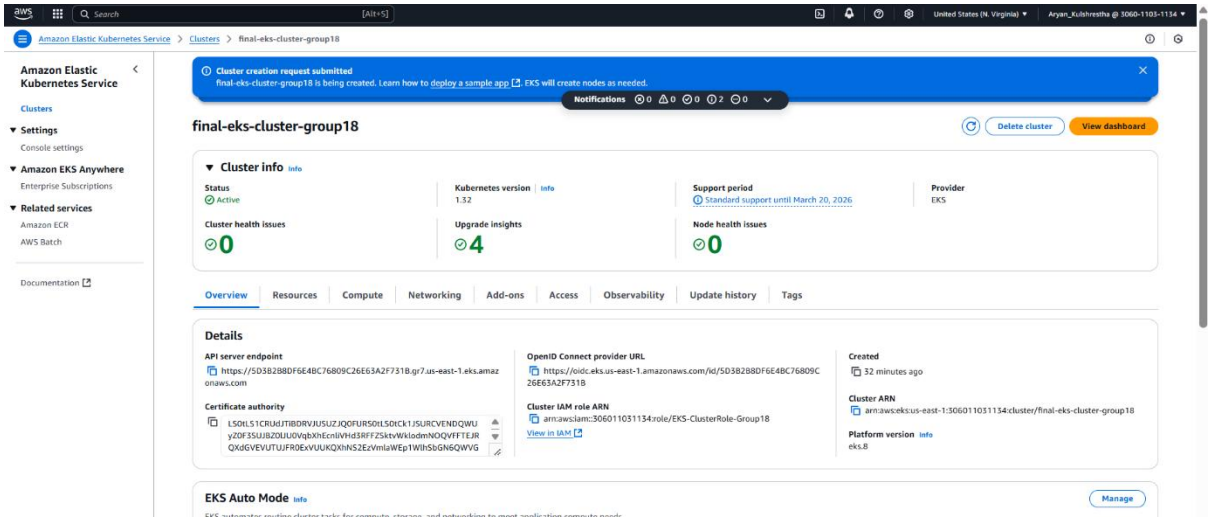


Figure 37: Cluster Created Manually in AWS

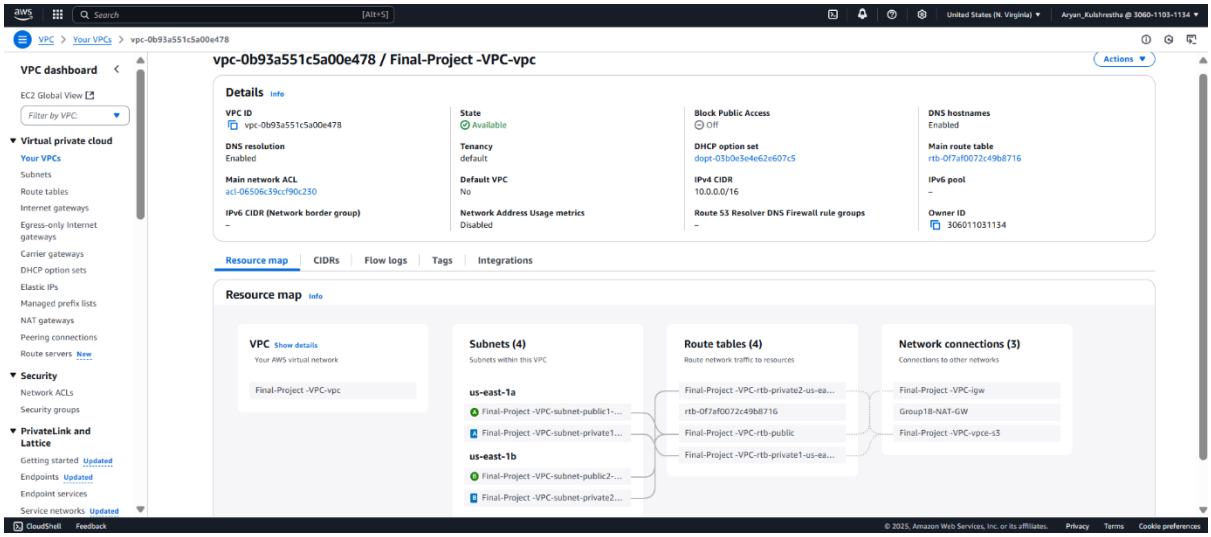


Figure 38: VPC Setup

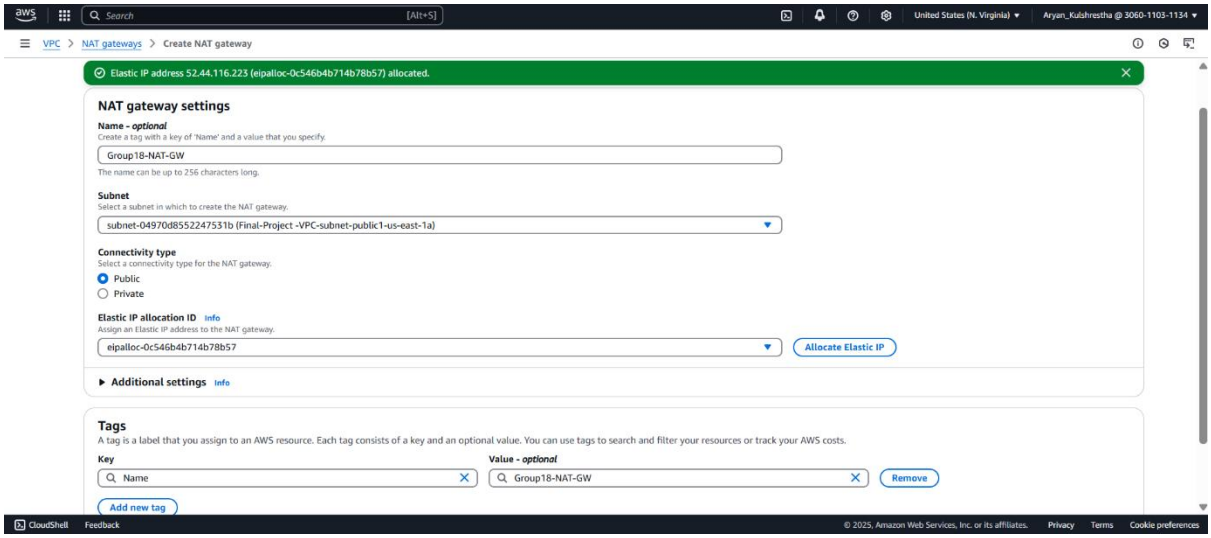


Figure 39: NAT Gateway Setup for VPC

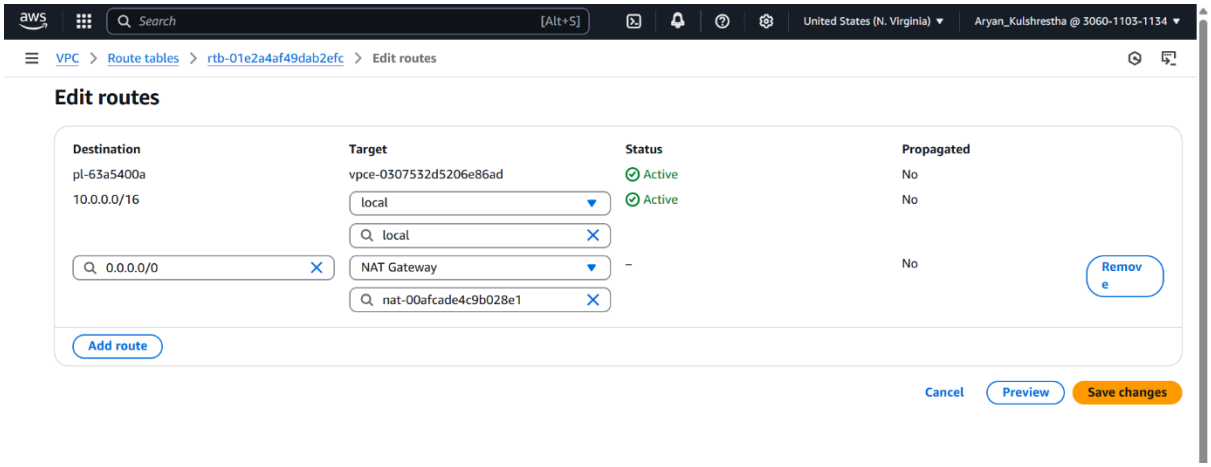


Figure 40: Route Table Rule for VPC

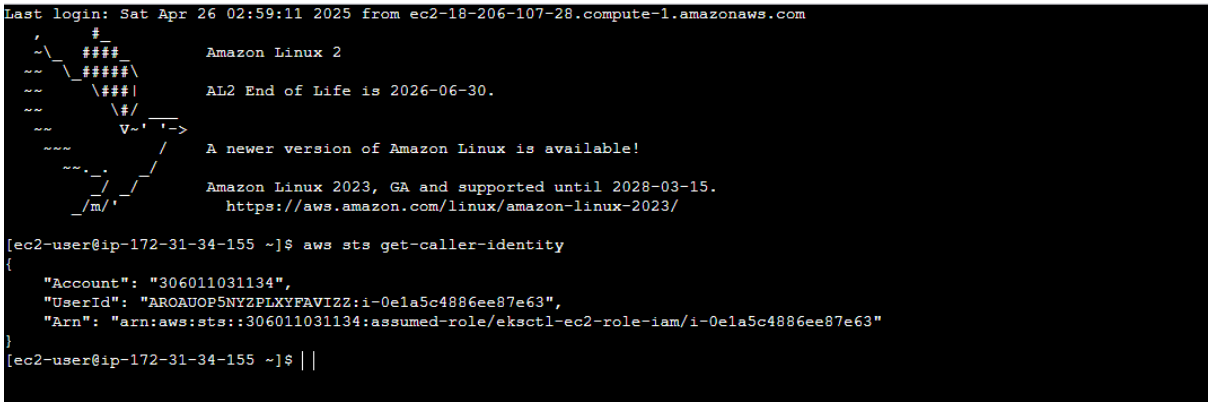


Figure 41: Once the IAM policy for cluster was attached to EC2 – SSH works

4.3 Task 1: Use of Helm Chart

Step 1: Installation of kubectl and eksctl.

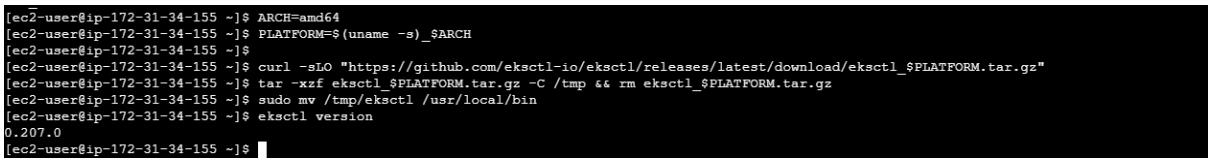


Figure 42: eksctl Installed

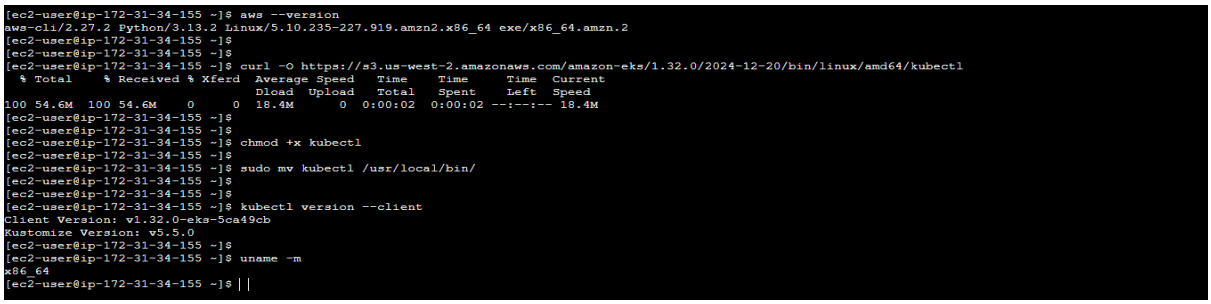


Figure 43: kubectl Installed

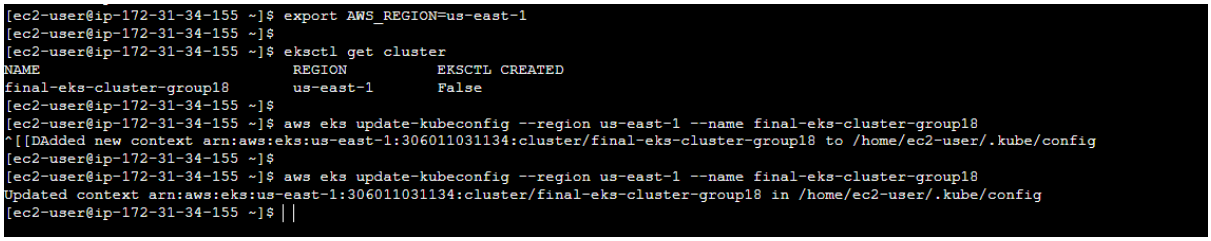


Figure 44: kubectl mapped with EKS Cluster

Step 2: Helm Installation with Add and Update OpenTelemetry Helm Repository

Helm was installed and then the official OpenTelemetry Helm chart repository was also added and updated using helm repo add and helm repo update. This is done to ensure we have access to the latest Helm charts before deployment.

Command:

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

→helm repo add open-telemetry https://open-telemetry.github.io/opentelemetry-helm-charts

→helm repo update

Figure 45: Helm Installed with Repo being Added and Updated

4.4 Task 2: Deploy the Application Using Helm

Step 1: Create Namespace for Helm Deployment

Created a dedicated namespace otel-helm-demo to isolate all Helm-related resources.

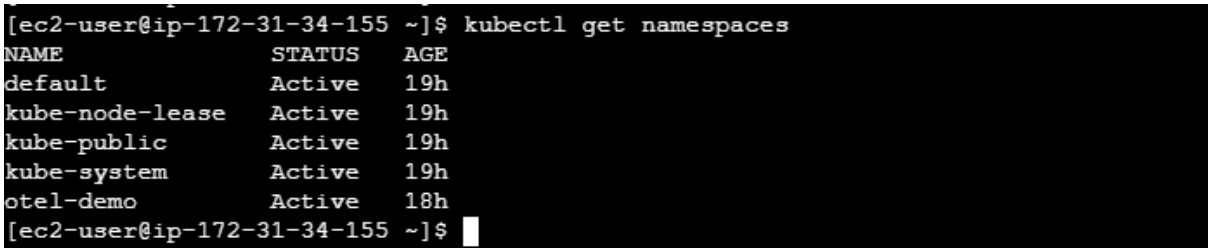


Figure 46: Previous namespaces in the cluster before otel-helm-demo was created

Step 2: Install OpenTelemetry Demo with Helm

The OpenTelemetry demo was deployed using Helm into the EKS cluster. The command creates a new namespace *"otel-helm-demo"* where it installed the application

Command: *helm install my-otel-demo open-telemetry/opentelemetry-demo --namespace otel-helm-demo --create-namespace*

Figure 47: OTEL-DEMO Application Deployment using Helm

Step 3: Verifying all Services and Pods which were created in the deployment of Otel-Demo Application in previous step.

```
NAME                                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                                                        AGE
service/ad                          ClusterIP   10.100.47.114    <none>        8080/TCP                                                       8m12s
service/cart                        ClusterIP   10.100.6.208     <none>        8080/TCP                                                       8m12s
service/checkout                    ClusterIP   10.100.166.171   <none>        8080/TCP                                                       8m12s
service/currency                    ClusterIP   10.100.244.203   <none>        8080/TCP                                                       8m12s
service/email                       ClusterIP   10.100.251.128   <none>        8080/TCP                                                       8m12s
service/flagd                       ClusterIP   10.100.34.97     <none>        8013/TCP,8016/TCP,4000/TCP                                     8m12s
service/frontend                    ClusterIP   10.100.110.10    <none>        8080/TCP                                                       8m12s
service/frontend-proxy              ClusterIP   10.100.136.217   <none>        8080/TCP                                                       8m12s
service/grafana                     ClusterIP   10.100.65.119    <none>        80/TCP                                                         8m12s
service/image-provider              ClusterIP   10.100.139.31    <none>        8081/TCP                                                       8m12s
service/jaeger-agent                ClusterIP   None             <none>        5775/UDP,5778/TCP,6831/UDP,6832/UDP                            8m12s
service/jaeger-collector            ClusterIP   None             <none>        9411/TCP,14250/TCP,14267/TCP,14268/TCP,4317/TCP,4318/TCP       8m12s
service/jaeger-query                ClusterIP   None             <none>        16686/TCP,16685/TCP                                            8m12s
service/kafka                       ClusterIP   10.100.4.95      <none>        9092/TCP,9093/TCP                                              8m12s
service/load-generator              ClusterIP   10.100.87.140    <none>        8089/TCP                                                       8m12s
service/opensearch                  ClusterIP   10.100.74.210    <none>        9200/TCP,9300/TCP,9600/TCP                                     8m12s
service/opensearch-headless         ClusterIP   None             <none>        9200/TCP,9300/TCP,9600/TCP                                     8m12s
service/otel-collector              ClusterIP   10.100.177.29    <none>        6831/UDP,14250/TCP,14268/TCP,8888/TCP,4317/TCP,4318/TCP,9411/TCP   8m12s
service/payment                     ClusterIP   10.100.7.83      <none>        8080/TCP                                                       8m12s
service/product-catalog             ClusterIP   10.100.101.199   <none>        8080/TCP                                                       8m12s
service/prometheus                  ClusterIP   10.100.197.80    <none>        9090/TCP                                                       8m12s
service/quote                       ClusterIP   10.100.181.172   <none>        8080/TCP                                                       8m12s
service/recommendation              ClusterIP   10.100.230.111   <none>        8080/TCP                                                       8m12s
service/shipping                    ClusterIP   10.100.249.32    <none>        8080/TCP                                                       8m12s
service/valkey-cart                 ClusterIP   10.100.7.70      <none>        6379/TCP                                                       8m12s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/accounting          1/1     1            1           8m12s
deployment.apps/ad                  1/1     1            1           8m12s
deployment.apps/cart                1/1     1            1           8m12s
deployment.apps/checkout            1/1     1            1           8m12s
deployment.apps/currency            1/1     1            1           8m12s
deployment.apps/email               1/1     1            1           8m12s
deployment.apps/flagd               1/1     1            1           8m12s
deployment.apps/fraud-detection     1/1     1            1           8m12s
```

Figure 48: Confirming Cluster and Other services are installed after Helm

```
[ec2-user@ip-172-31-34-155 ~]$ kubectl get all -n otel-helm-demo
NAME                                    READY   STATUS    RESTARTS   AGE
pod/accounting-64cddc9b9d-nx8n6         1/1     Running   0          8m12s
pod/ad-7d5d9fc5f7-g568g                 1/1     Running   0          8m11s
pod/cart-64ff8d88d8-x51h4               1/1     Running   0          8m12s
pod/checkout-7b678db584-fc9t5           1/1     Running   0          8m10s
pod/currency-cdfd88979-rm6kt            1/1     Running   0          8m9s
pod/email-59657ccf76-kj9kt              1/1     Running   0          8m12s
pod/flagd-7c5995c9b4-t8kdp              2/2     Running   0          8m12s
pod/fraud-detection-5d5448f5c7-2tgpc    1/1     Running   0          8m12s
pod/frontend-59874fd7d9-r52j5           1/1     Running   0          8m10s
pod/frontend-proxy-69b697458-rf7c6      1/1     Running   0          8m11s
pod/grafana-7696b658d-wgkng             1/1     Running   0          8m11s
pod/image-provider-688675b9b7-jlwxx     1/1     Running   0          8m12s
pod/jaeger-68f4c88f6f-66zx4             1/1     Running   0          8m10s
pod/kafka-7fdc6c75f5-dz9wp              1/1     Running   0          8m12s
pod/load-generator-7f4cc4dd5-jxxkf      1/1     Running   0          8m12s
pod/opensearch-0                        1/1     Running   0          8m12s
pod/otel-collector-845f8f47d6-v88cc     1/1     Running   0          8m12s
pod/payment-67cfdcd6d8-kf25r            1/1     Running   0          8m10s
pod/product-catalog-f54db96df-6r7nq     1/1     Running   0          8m9s
pod/prometheus-7bfddcb4c7-fngsz         1/1     Running   0          8m12s
pod/quote-5684d479f5-fhxg6              1/1     Running   0          8m10s
pod/recommendation-7cd8dfb4cc-mrb9z     1/1     Running   0          8m11s
pod/shipping-c66788945-smb6m            1/1     Running   0          8m9s
pod/valkey-cart-776c9fbc8b-g8j2k        1/1     Running   0          8m12s

NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                     AGE
service/ad                  ClusterIP   10.100.47.114    <none>        8080/TCP                    8m12s
service/cart                ClusterIP   10.100.6.208     <none>        8080/TCP                    8m12s
service/checkout            ClusterIP   10.100.166.171   <none>        8080/TCP                    8m12s
service/currency            ClusterIP   10.100.244.203   <none>        8080/TCP                    8m12s
service/email               ClusterIP   10.100.251.128   <none>        8080/TCP                    8m12s
service/flagd               ClusterIP   10.100.34.97     <none>        8013/TCP,8016/TCP,4000/TCP  8m12s
service/frontend            ClusterIP   10.100.110.10    <none>        8080/TCP                    8m12s
service/frontend-proxy      ClusterIP   10.100.136.217   <none>        8080/TCP                    8m12s
```

Figure 49: Confirming pods are working after helm installation

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/accounting            1/1     1            1           8m12s
deployment.apps/ad                    1/1     1            1           8m12s
deployment.apps/cart                  1/1     1            1           8m12s
deployment.apps/checkout              1/1     1            1           8m12s
deployment.apps/currency              1/1     1            1           8m12s
deployment.apps/email                 1/1     1            1           8m12s
deployment.apps/flagd                 1/1     1            1           8m12s
deployment.apps/fraud-detection       1/1     1            1           8m12s
deployment.apps/frontend              1/1     1            1           8m12s
deployment.apps/frontend-proxy        1/1     1            1           8m12s
deployment.apps/grafana               1/1     1            1           8m12s
deployment.apps/image-provider        1/1     1            1           8m12s
deployment.apps/jaeger                1/1     1            1           8m12s
deployment.apps/kafka                 1/1     1            1           8m12s
deployment.apps/load-generator        1/1     1            1           8m12s
deployment.apps/otel-collector        1/1     1            1           8m12s
deployment.apps/payment               1/1     1            1           8m12s
deployment.apps/product-catalog       1/1     1            1           8m12s
deployment.apps/prometheus            1/1     1            1           8m12s
deployment.apps/quote                 1/1     1            1           8m12s
deployment.apps/recommendation        1/1     1            1           8m12s
deployment.apps/shipping              1/1     1            1           8m12s
deployment.apps/valkey-cart           1/1     1            1           8m12s

NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/accounting-64cddc9b9d       1         1         1       8m12s
replicaset.apps/ad-7d5d9fc5f7               1         1         1       8m12s
replicaset.apps/cart-64ff8d88d8             1         1         1       8m12s
replicaset.apps/checkout-7b678db584         1         1         1       8m11s
replicaset.apps/currency-cdfd88979          1         1         1       8m10s
replicaset.apps/email-59657ccf76            1         1         1       8m12s
replicaset.apps/flagd-7c5995c9b4            1         1         1       8m12s
replicaset.apps/fraud-detection-5d5448f5c7  1         1         1       8m12s
replicaset.apps/frontend-59874fd7d9         1         1         1       8m11s
replicaset.apps/frontend-proxy-69b697458    1         1         1       8m11s
```

Figure 50: Confirming services Installed after Helm

```
NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/accounting-64cddc9b9d       1         1         1       8m12s
replicaset.apps/ad-7d5d9fc5f7               1         1         1       8m12s
replicaset.apps/cart-64ff8d88d8             1         1         1       8m12s
replicaset.apps/checkout-7b678db584         1         1         1       8m11s
replicaset.apps/currency-cdfd88979          1         1         1       8m10s
replicaset.apps/email-59657ccf76            1         1         1       8m12s
replicaset.apps/flagd-7c5995c9b4            1         1         1       8m12s
replicaset.apps/fraud-detection-5d5448f5c7  1         1         1       8m12s
replicaset.apps/frontend-59874fd7d9         1         1         1       8m11s
replicaset.apps/frontend-proxy-69b697458    1         1         1       8m11s
replicaset.apps/grafana-7696b658d           1         1         1       8m11s
replicaset.apps/image-provider-688675b9b7   1         1         1       8m12s
replicaset.apps/jaeger-68f4c88f6f           1         1         1       8m10s
replicaset.apps/kafka-7fdc6c75f5            1         1         1       8m12s
replicaset.apps/load-generator-7f4cc4dd5    1         1         1       8m12s
replicaset.apps/otel-collector-845f8f47d6   1         1         1       8m12s
replicaset.apps/payment-67cfdcd6d8          1         1         1       8m11s
replicaset.apps/product-catalog-f54db96df   1         1         1       8m10s
replicaset.apps/prometheus-7bfddcb4c7       1         1         1       8m12s
replicaset.apps/quote-5684d479f5            1         1         1       8m11s
replicaset.apps/recommendation-7cd8dfb4cc   1         1         1       8m11s
replicaset.apps/shipping-c66788945          1         1         1       8m10s
replicaset.apps/valkey-cart-776c9fbc8b      1         1         1       8m12s

NAME                             READY   AGE
statefulset.apps/opensearch      1/1     8m12s
[ec2-user@ip-172-31-34-155 ~]$
```

Figure 51: Helm chart installation confirmation

4.5 Task 3: Upgrade and Rollback

After deploying the application using Helm, we performed an upgrade and rollback followed by accessing the endpoints.

Step 1: Helm Upgrade

When we perform an upgrade, the existing Helm release is overwritten with updated values or the latest available chart version, but it also retains the previous deployment.

Command: *helm upgrade my-otel-demo open-telemetry/opentelemetry-demo -n otel-helm-demo*

Figure 52: This figure shows the version of the application before upgrade

Note: This is the same version the application falls back to after rollback is complete

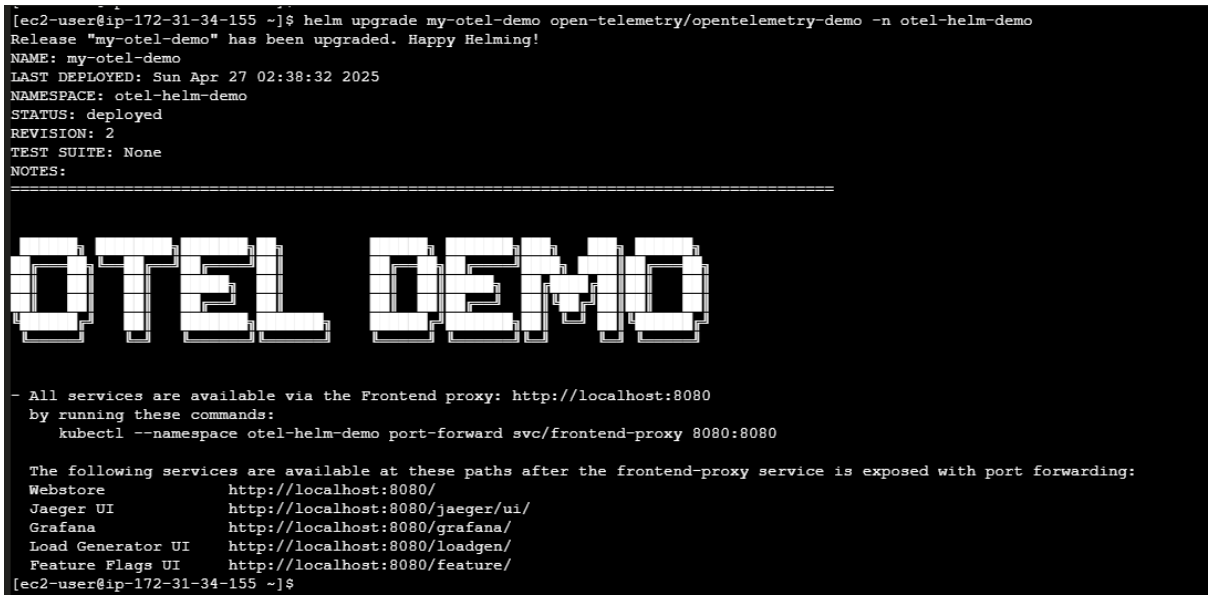


Figure 53: Upgrade was successfully completed

Step 2: Helm Rollback

To validate the update and rollback support provided in helm, we reverted the deployment to its earlier state using the helm rollback command

Command: *helm rollback my-otel-demo 1 -n otel-helm-demo*

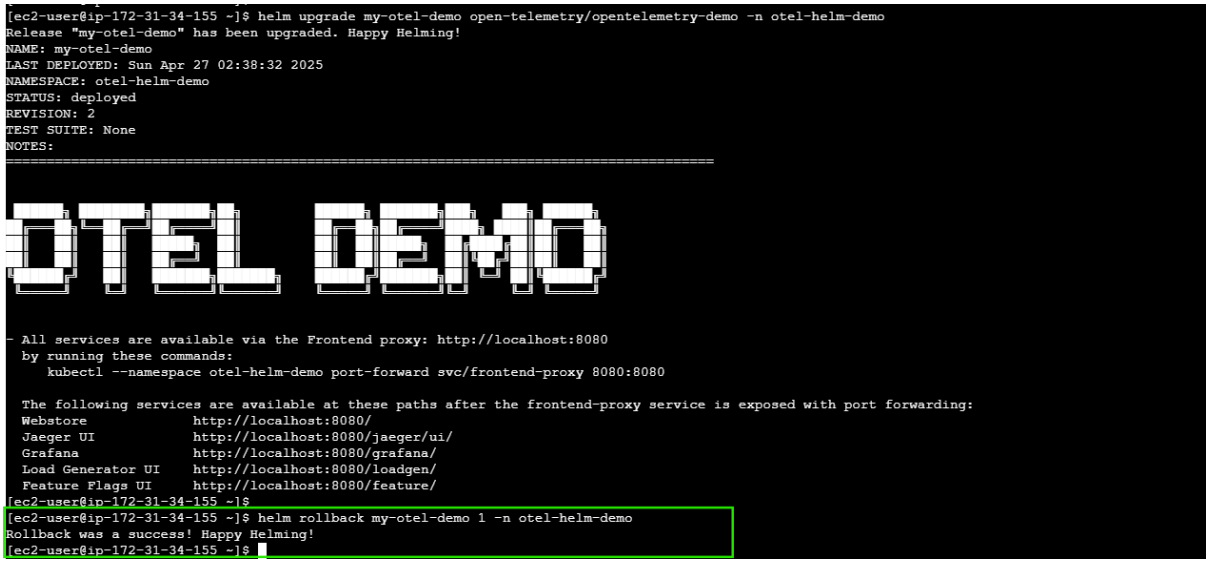


Figure 54: Rollback Implemented Successfully

Figure 55: The EKS Cluster and otel-helm-demo namespace reverts back to previous application version

Step 4: Access the Application via port forwarding on Web Browser

- kubectl port-forward svc/frontend-proxy 8080:8080 -n otel-helm-demo

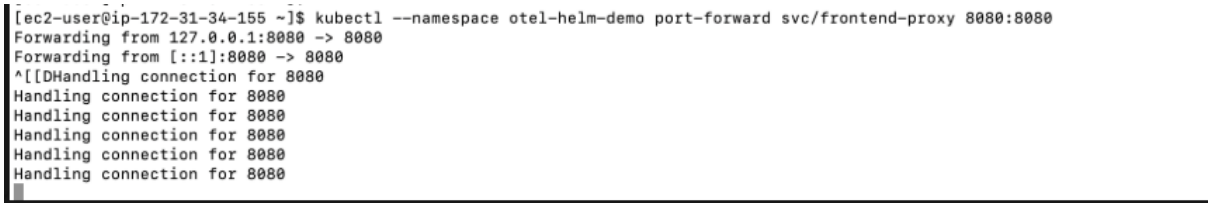


Figure 56: Logs of Frontend-proxy

- Front-end Proxy: localhost:8080

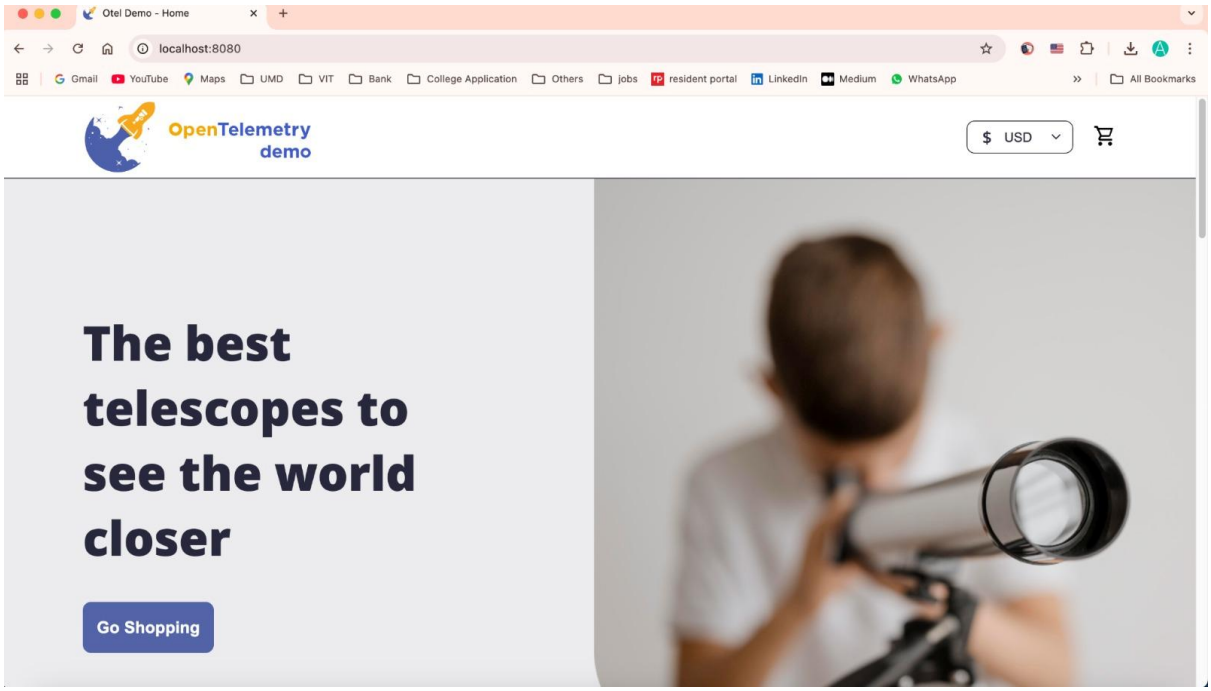


Figure 57: Frontend-proxy

- Grafana: localhost:8080/grafana/

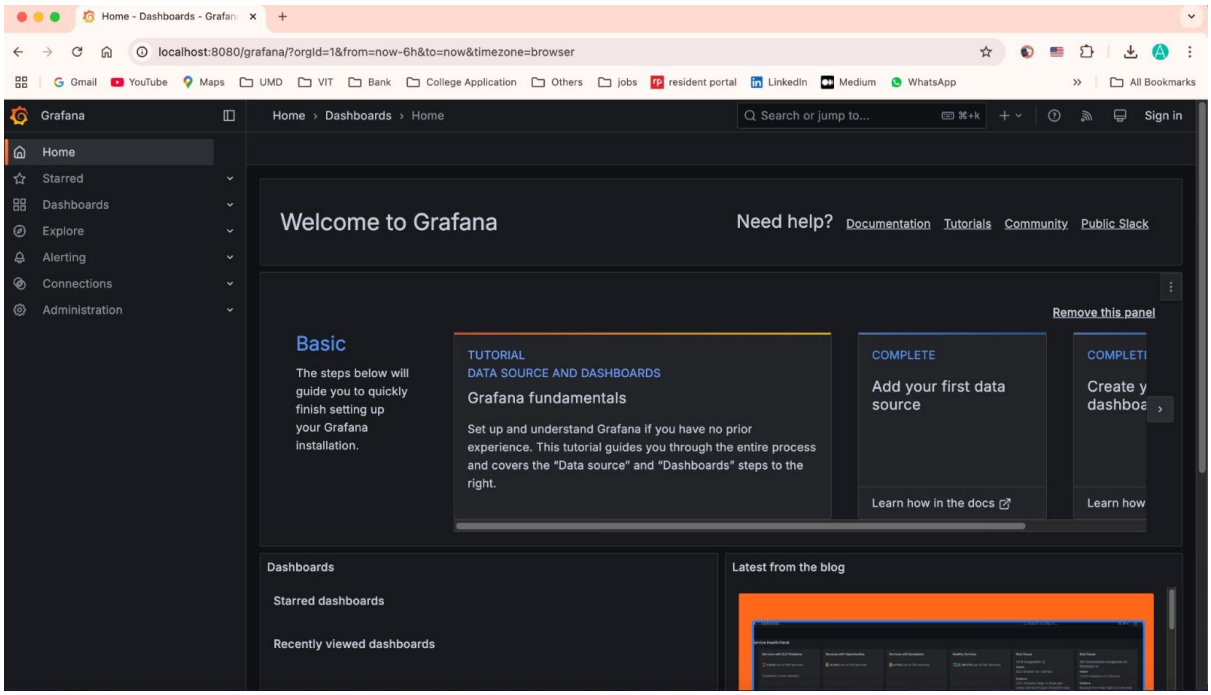


Figure 58: Grafana

- Jaeger: Localhost:8080/jaeger

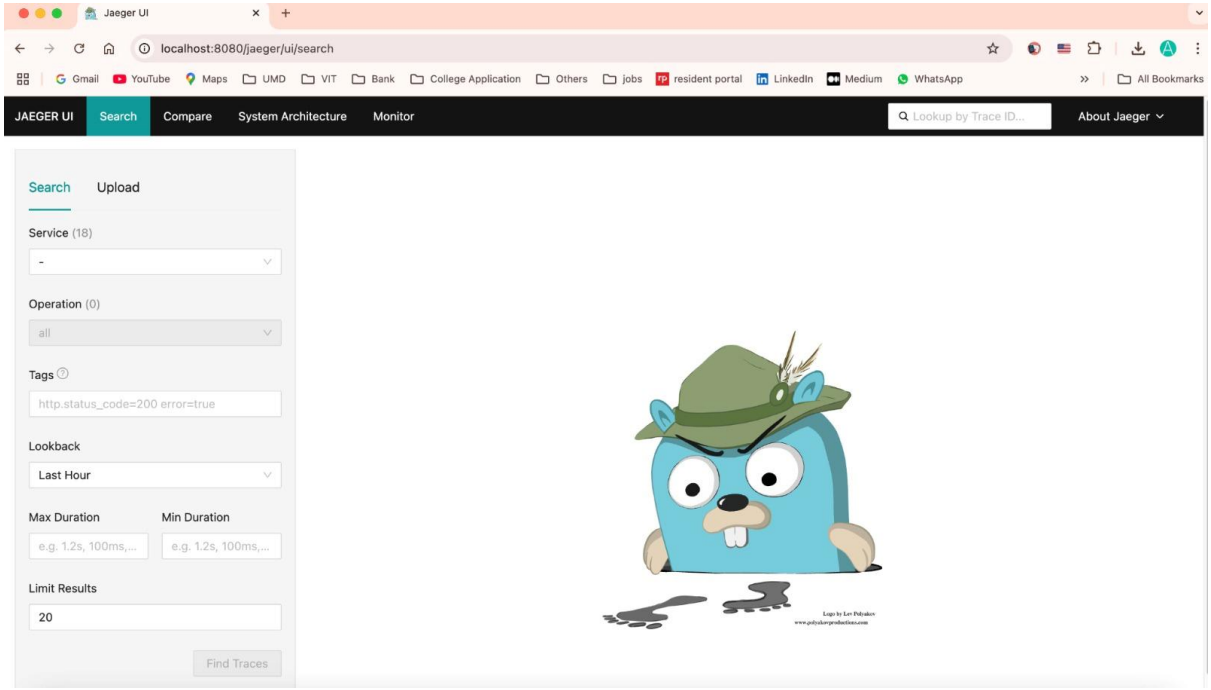
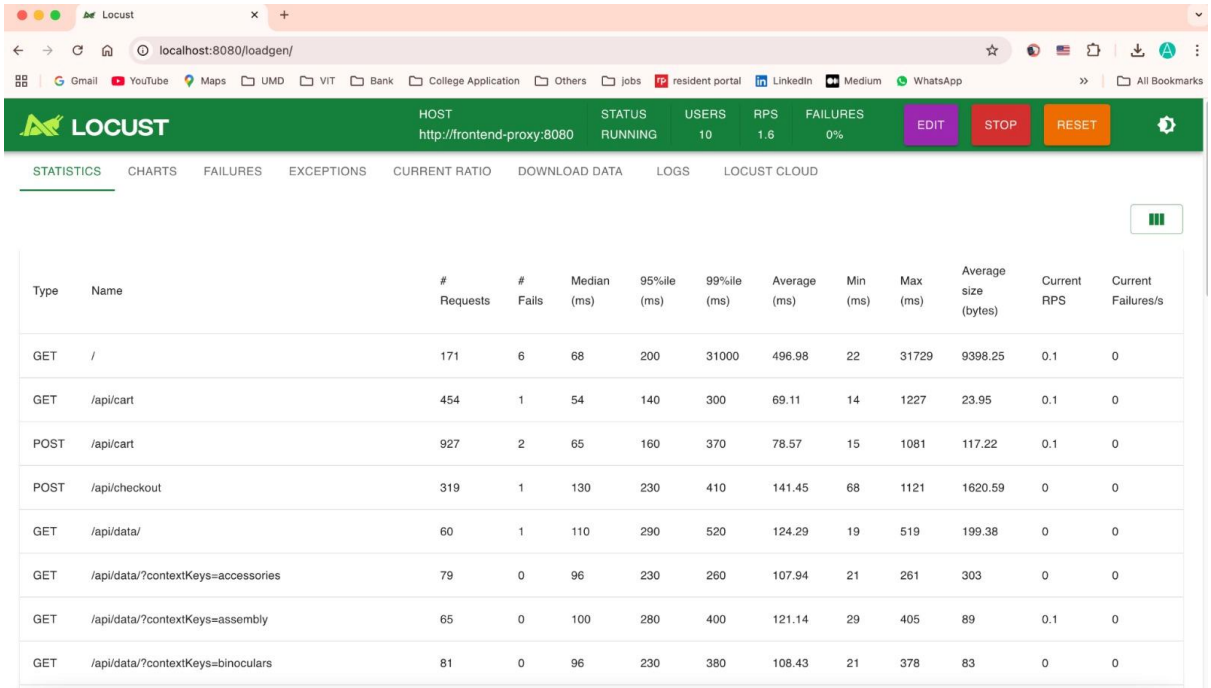


Figure 59: Jaeger

- Load Generator: localhost:8080/loadgen



Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	171	6	68	200	31000	496.98	22	31729	9398.25	0.1	0
GET	/api/cart	454	1	54	140	300	69.11	14	1227	23.95	0.1	0
POST	/api/cart	927	2	65	160	370	78.57	15	1081	117.22	0.1	0
POST	/api/checkout	319	1	130	230	410	141.45	68	1121	1620.59	0	0
GET	/api/data/	60	1	110	290	520	124.29	19	519	199.38	0	0
GET	/api/data/?contextKeys=accessories	79	0	96	230	260	107.94	21	261	303	0	0
GET	/api/data/?contextKeys=assembly	65	0	100	280	400	121.14	29	405	89	0.1	0
GET	/api/data/?contextKeys=binoculars	81	0	96	230	380	108.43	21	378	83	0	0

Figure 60: Load Generator

- Flagd: localhost:8080/feature

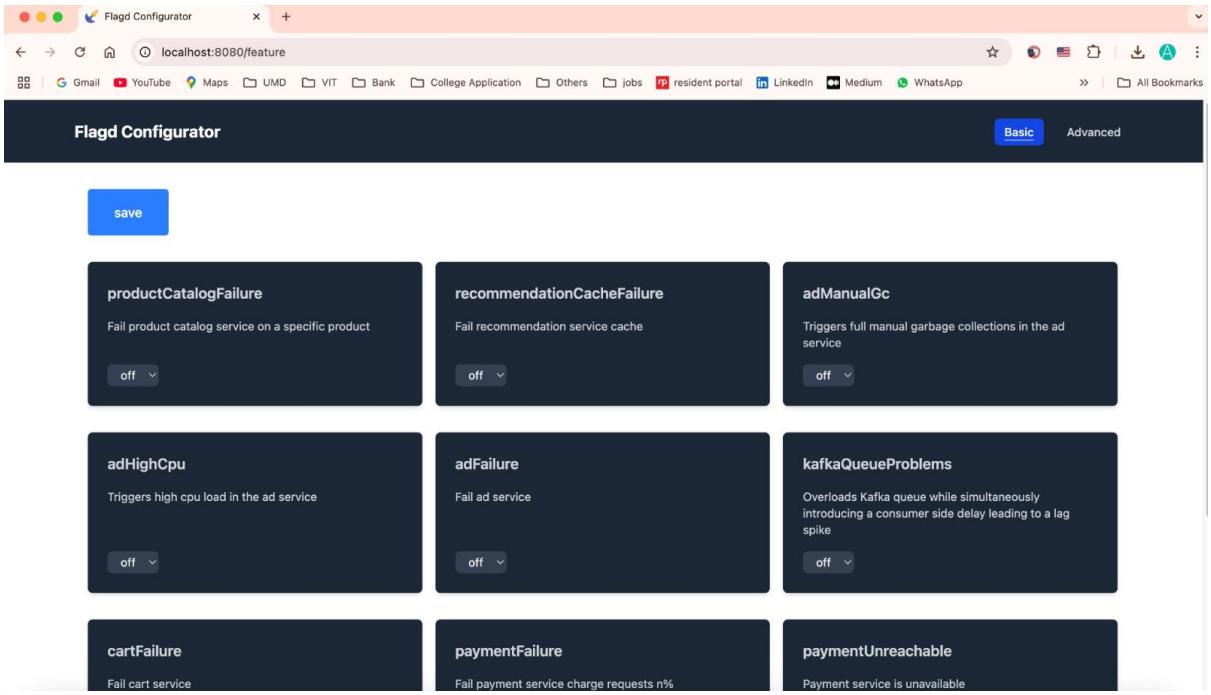


Figure 61: FlagD

4.6 Challenges Encountered

One of the initial challenges was resolving the repetition of resources from a previous namespace. To properly deploy the application under the new "otel-helm-demo" namespace, we had to manually delete several existing pods and services that were clashing with the Helm release. In Phase 1, initially Grafana dashboard didn't load properly due to ConfigMap limits, which we fixed by commenting the other services in phase 1 to only run grafana but in Phase 2, Helm handled the configuration smoothly such that Grafana dashboard was successfully loaded. This highlighted Helm’s strength in managing large, dynamic Kubernetes resources.

Phase 3: Alerting Service and Notifications

This phase focuses on enabling an Alerting Service for the Kubernetes Cluster, where the Open-Telemetry Application is deployed. Notifications should be triggered when an anomality is observed based on the configured rule-set.

5.1 Architecture Overview

Figure 62: High-level architecture of Phase 3

Link: https://lucid.app/lucidchart/b4230840-95e0-4576-9085-9e3fb15404ee/edit?viewport_loc=1853%2C-55382%2C4100%2C1744%2C0_0&invitationId=inv_115c4f0f-be50-498d-aa1a-f47c983b31b2

5.2 Implementation

The initial setup of Kubernetes cluster in Phases 1 & 2 was used to further perform the tasks in this phase. In Phase 3 we implemented an alerting system to monitor that the pods restart in our Kubernetes cluster, using Prometheus and Alertmanager. We installed Prometheus using Helm and set it up to collect metrics from the Kubernetes cluster. A custom Prometheus Rule was defined to trigger an alert (HighPodRestart). If a pod restarts at least twice or more within five minutes, then an alert will be triggered. Alertmanager was then configured to send email notifications through Gmail SMTP by securely encoding and patching the alertmanager.yaml with the sender's credentials and recipient address. We tested the alert by deploying a crash pod and confirmed the email notifications, which were successfully delivered to our inbox, thus, verifying end-to-end alert functionality.

5.3 Task 1: Set Up Monitoring

Step 1: Install and Deploy Prometheus and Alertmanager using Helm

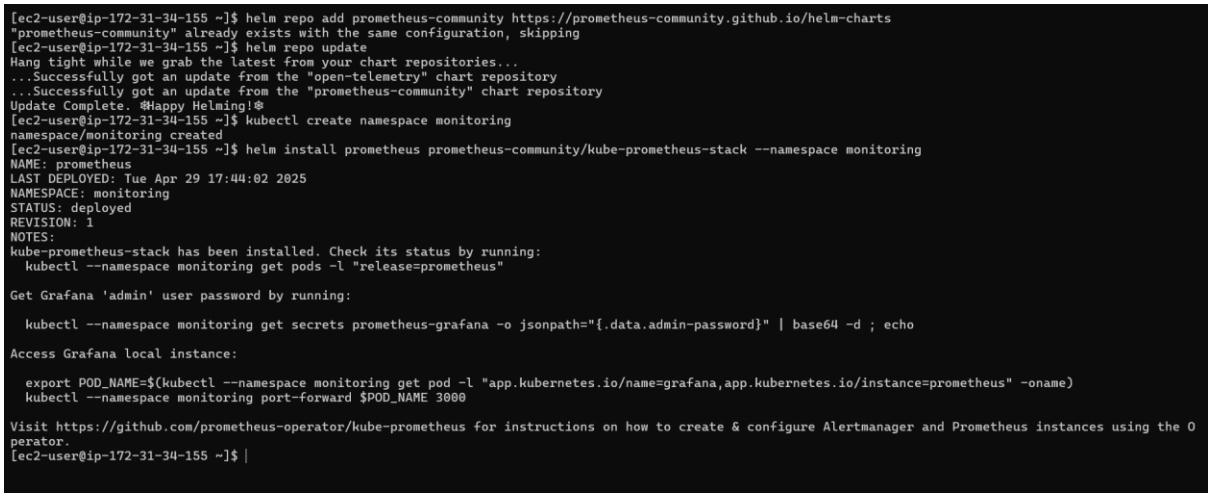


Figure 63: Prometheus Stack Deployment

We deployed the Prometheus-Grafana monitoring stack using Helm into our Kubernetes cluster. This involved adding the Prometheus Helm chart repository, creating the 'monitoring' namespace, and installing the Prometheus-Grafana stack into this namespace. *Figure 1: Prometheus Stack Deployment*, shows the terminal output confirming successful deployment. We are now ready to Check running pods and configure alerting rules.

5.4 Task 2: Define Alerting Rule

Step 2: Define the Pod Restart Alert

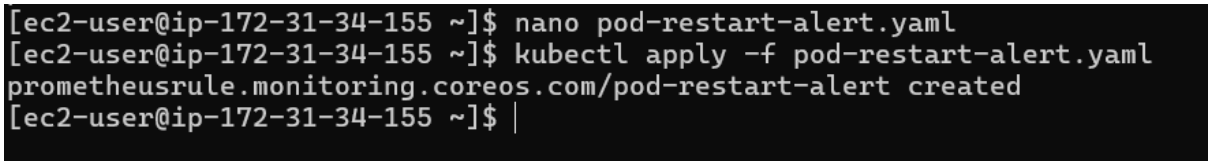


Figure 64: Pod Restart Alert Rule

We created a custom alert rule file named pod-restart-alert.yaml to monitor pod restarts. This rule was applied using the kubectl apply command and registered with Prometheus within the monitoring namespace. Prometheus successfully recognized the configuration and created the alert under the PrometheusRule resource type.

```
[ec2-user@ip-172-31-34-155 ~]$ cat pod-restart-alert.yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: pod-restart-alert
  namespace: monitoring
spec:
  groups:
  - name: pod-restart-alerts
    rules:
    - alert: HighPodRestart
      expr: increase(kube_pod_container_status_restarts_total[5m]) > 2
      for: 2m
      labels:
        severity: critical
      annotations:
        summary: "Pod has restarted more than 2 times within 5 minutes."
        description: "Pod {{ $labels.pod }} in namespace {{ $labels.namespace }} has restarted frequently."
```

Figure 65: High Pod Restart Rule present in Yaml File

In this step, we defined a Prometheus alert rule using YAML. The rule, named HighPodRestart, is part of a PrometheusRule resource called pod-restart-alert in the monitoring namespace. It monitors the number of container restarts over a 5-minute interval. If a pod restarts more than twice within that window and sustains that rate for 2 minutes, the alert is triggered. The alert is labeled as "critical" and includes a message identifying the affected pod. The configuration for this rule is shown in Figure 3: High Pod Restart Rule.

5.5 Task 3: Configure Email Notifications

Step 3: Configure Alertmanager to Send Emails

```
[ec2-user@ip-172-31-34-155 ~]$ kubectl edit secret alertmanager-prometheus-kube-prometheus-alertmanager -n monitoring
Edit cancelled, no changes made.
[ec2-user@ip-172-31-34-155 ~]$ kubectl get secret alertmanager-prometheus-kube-prometheus-alertmanager -n monitoring -o jsonpath="{.data.alertmanager\.yaml}
" | base64 --decode > alertmanager.yaml
[ec2-user@ip-172-31-34-155 ~]$ nano alertmanager.yaml
[ec2-user@ip-172-31-34-155 ~]$ cat alertmanager.yaml
global:
  smtp_smarthost: 'smtp.gmail.com:587'
  smtp_from: 'gideonmm1848@gmail.com'
  smtp_auth_username: 'gideonmm1848@gmail.com'
  smtp_auth_password: 'zetekpbayurhztta'
  smtp_require_tls: true

route:
  receiver: 'email-alert'

receivers:
- name: 'email-alert'
  email_configs:
  - to: 'gideonmm1848@gmail.com'
    send_resolved: true

[ec2-user@ip-172-31-34-155 ~]$ base64 -w0 alertmanager.yaml > alertmanager.b64
[ec2-user@ip-172-31-34-155 ~]$ kubectl patch secret alertmanager-prometheus-kube-prometheus-alertmanager -n monitoring --type='merge' -p "{\"data\":{\"alert
manager.yaml\":\"$(cat alertmanager.b64)\"}}"
secret/alertmanager-prometheus-kube-prometheus-alertmanager patched
[ec2-user@ip-172-31-34-155 ~]$ kubectl delete pod -n monitoring -l app.kubernetes.io/name=alertmanager
pod "alertmanager-prometheus-kube-prometheus-alertmanager-0" deleted
[ec2-user@ip-172-31-34-155 ~]$
```

Figure 66: Configuring Alert Manager to send Emails

We configured Alertmanager to send email alerts by first decoding its configuration file from a Kubernetes secret. After editing the alertmanager.yaml to include Gmail SMTP details, we re-encoded it using the base64 command. The updated file was then patched back into the Kubernetes secret. To apply the changes, we deleted the Alertmanager pod, which restarted automatically with

the new settings. This ensured that Alertmanager could now send email notifications when alerts are triggered.

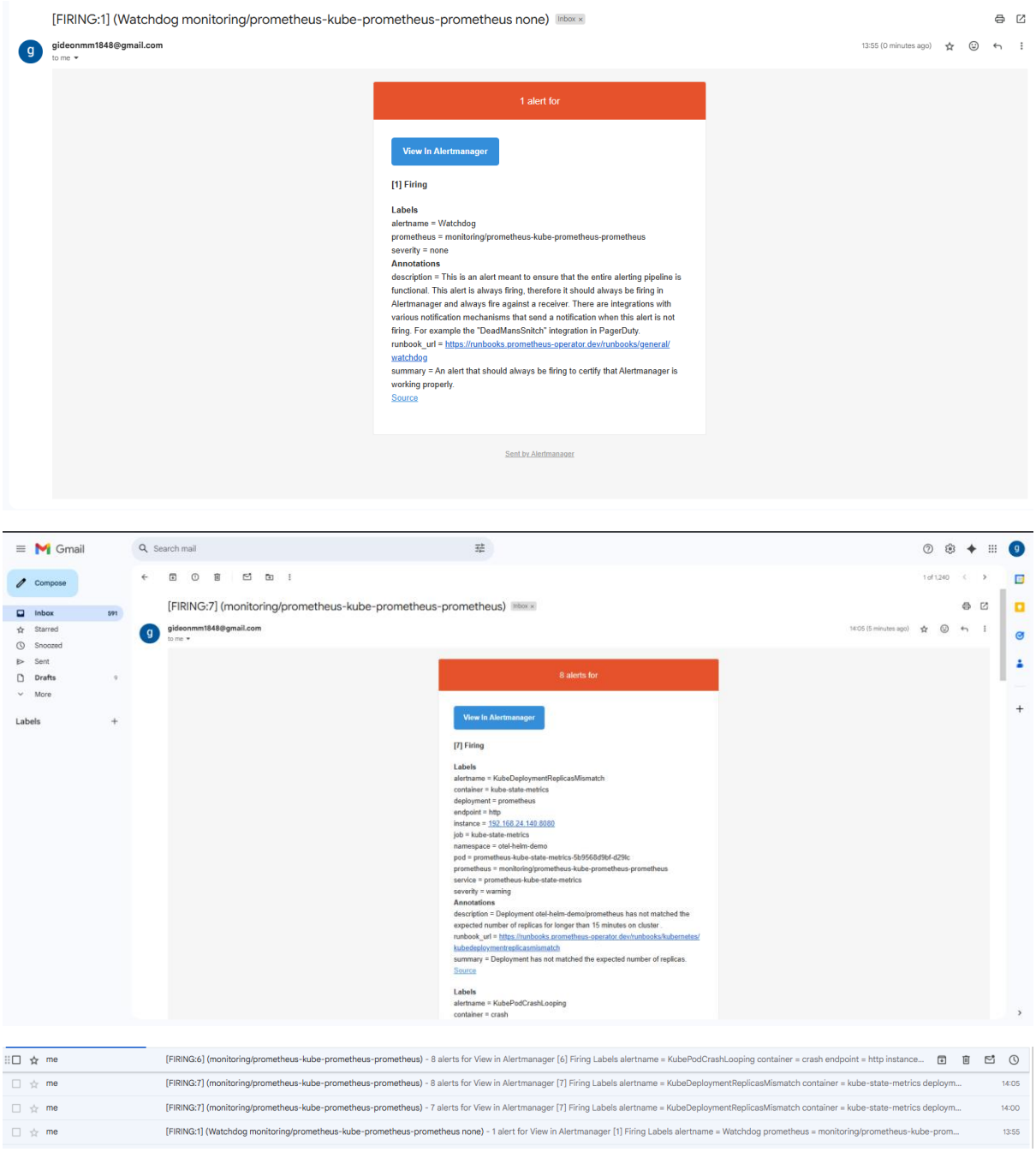


Figure 67-69: Alerting Emails received

We successfully configured Prometheus and Alertmanager to monitor our Kubernetes cluster. A custom pod restart alert was defined and applied using a PrometheusRule. Alertmanager was integrated with Gmail SMTP, and we verified email delivery by receiving a default Watchdog alert.

We then triggered actual alerts, such as KubePodCrashLooping and KubeDeploymentReplicasMismatch, which were successfully delivered via email. The presence of multiple alert emails in the inbox confirms that our alerting and notification system is fully operational.

5.6 Challenges Encountered

One of the main challenges was setting up the Alertmanager to send emails using Gmail. Since, the alertmanager.yaml is stored encoded inside a Kubernetes secret, any direct edit without proper decode – edit – encode breaks the structure. Initially, we were editing the Kubernetes Secrets (base 64 encoded) directly which didn't work. Therefore we had to carefully decode, change and re-encode the alertmanager.yaml file to make sure Alertmanager works properly. Another challenge was to make sure that the alert gets triggered. Hence, we had to crash the pod multiple times and wait for Prometheus to identify the error and trigger the alert.

Phase 4: CI/CD Integration

This phase covers the integration of all the tasks performed in previous phases into a CI/CD pipeline. The pipeline automates the process, from building and testing code to deploying it properly and connecting our AWS account such that it integrates AWS ECR and AWS EKS efficiently.

6.1 Architecture Overview

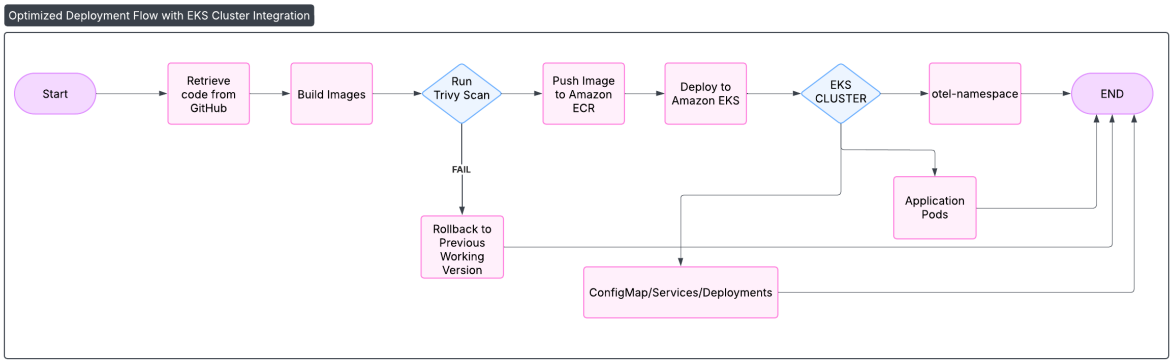


Figure 70: High-level architecture of Phase 4

Link: https://lucid.app/lucidchart/20bc0929-10f2-4003-b596-492b8efefef1/edit?viewport_loc=7085%2C2127%2C2880%2C1225%2C0_0&invitationId=inv_26b09ccc-3015-43c1-9bb1-b7af5a8973d4

6.2 Implementation

To implement Phase 4, we set up a CI/CD pipeline using GitHub Actions to automate the application deployment. The key objectives included containerizing the application, securely storing secrets using GitHub repository secrets, pushing images to AWS ECR, deploying to an EKS cluster, integrating testing and image scanning, and enabling a rollback mechanism in case of deployment failure. Sensitive data, including AWS credentials, was managed using GitHub Secrets to prevent public exposure. A YAML workflow file (cicd-pipeline.yml) was created to define and automate all the tasks required for Phase 4. Further, we also forked the provided opentelemetry-demo repository from GitHub into our own GitHub account.

Since, one of the members (Aryan) was responsible for creating the EKS cluster, he had root access by default for the EKS cluster. However, because the GitHub repository used for the CI/CD pipeline was forked into another member's (Aashay) GitHub account, therefore, we need to provide him with admin-level access to the cluster. This was essential to ensure that the pipeline could authenticate with AWS, deploy successfully to the cluster, and manage resources

without permission issues. To achieve this, we granted the access using a yaml file. Access entry YAML file is linked with the appropriate IAM policy through a configuration file.

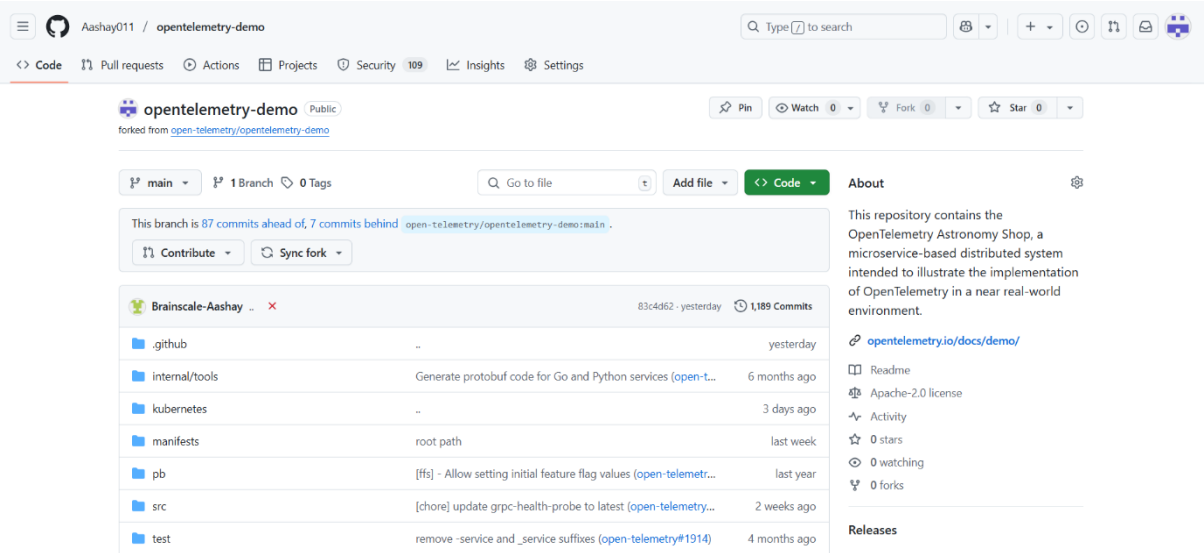


Figure 71: Forked the Opentelemetry-demo repository in our account

Command used for creating the cluster from terminal:

→ *eksctl create cluster --name otel-demo --region us-east-1 --version 1.32 --nodegroup-name otel-node-group --node-type t3.large --nodes 2 --nodes-min 1 --nodes-max 3 –managed*

Access Entry Configuration Script for our EKS Cluster:

eksctl_updated.yaml

File Edit View

```yaml
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: otel-demo
  region: us-east-1
  version: '1.32'

accessConfig:
  authenticationMode: API
  accessEntries:
    - principalARN: arn:aws:iam::306011031134:user/Aashay_Mehta
      type: STANDARD
      accessPolicies:
        - policyARN: arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy
          accessScope:
            type: cluster
```

Figure 72: YAML Script for Access Entry Configuration

6.3 Task 1: Set Up CI/CD Pipeline

We setup the CI/CD pipeline to automate the containerization and deployment process. The steps below show how we configured the pipeline using GitHub Actions and integrated it with AWS services.

Step 1: The first task involves containerizing the images.

Step 2: Decoding the cicd-pipeline.yml file: The first task in the file involves logging in to AWS with the Access Key and Secret. These credentials are stored in GitHub Actions Secrets.

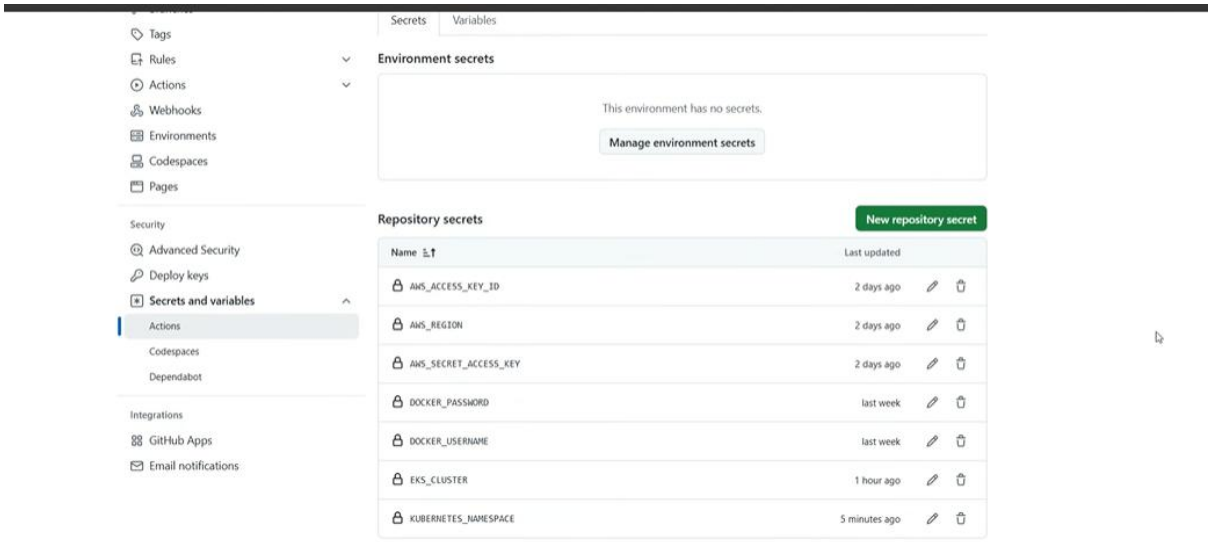


Figure 73: Environment Variables stored in GitHub Secrets

Step 3: We created the ECR on the AWS Portal in order to push and store the images.

Step 4: In order to build the images, we used the docker-compose.yml file and .env file and used the following command:

“docker compose -f ./docker-compose.yml build”

Step 5: Once the images are built, we pushed them into the ECR. We authenticated into the ECR by creating a task in cicd-pipeline.yml.

Step 6: After authenticating into the ECR, we used the following command to push the images: “docker compose push”

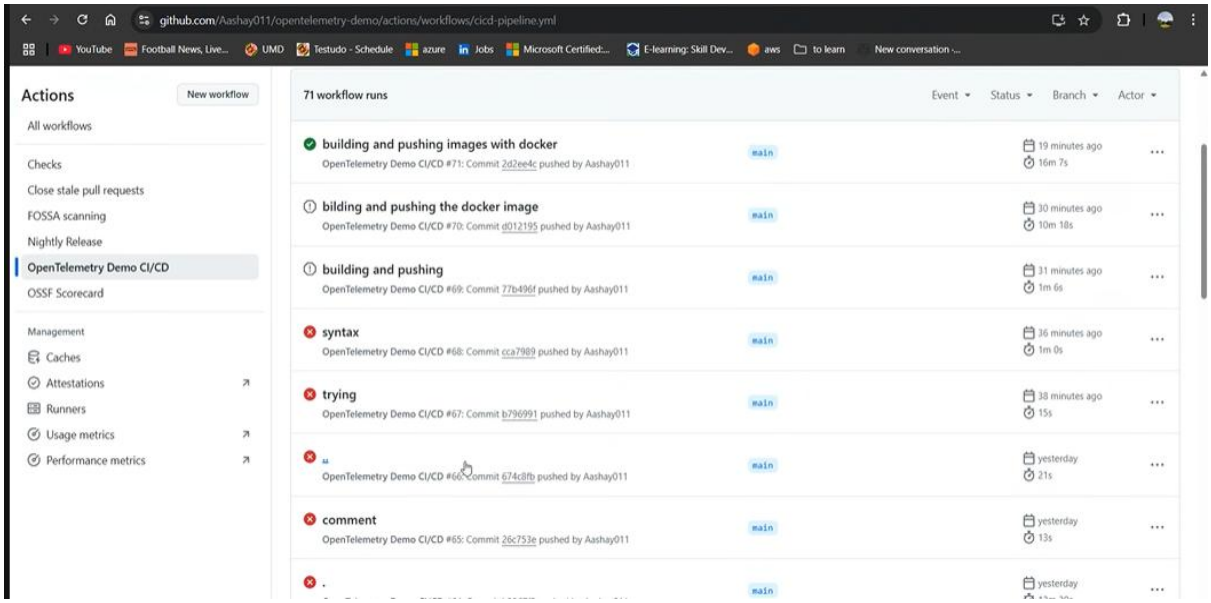


Figure 74: Successful Build and Push of the Pipeline

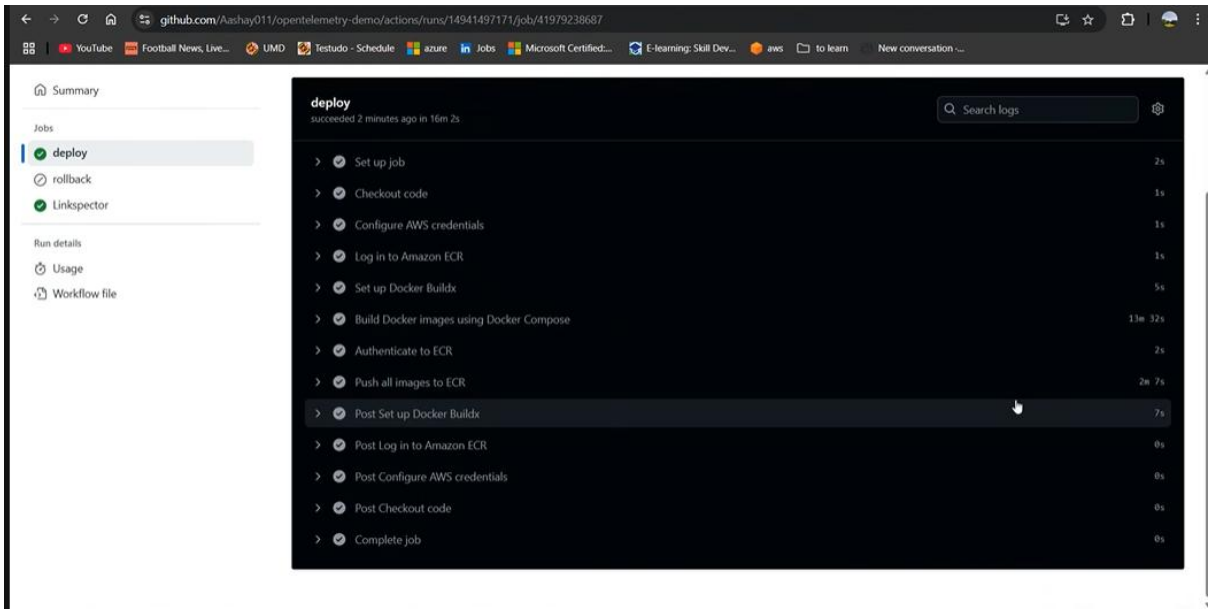


Figure 75: Successful Deployment of the Pipeline

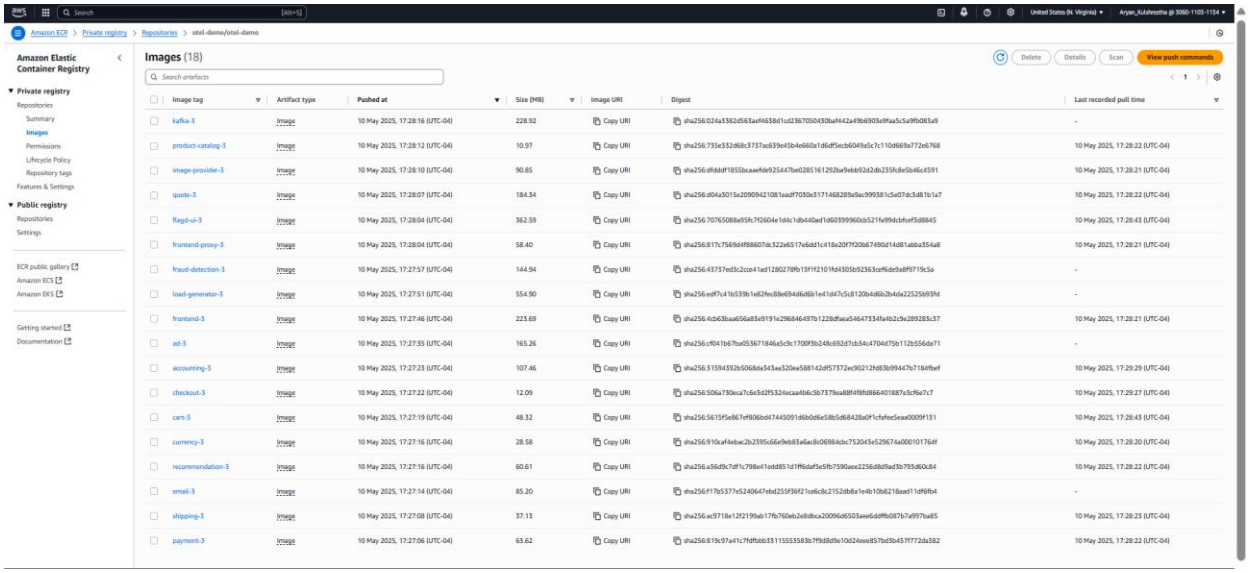


Figure 76: Images are pushed into the ECR successfully

Step 9: We also added steps for testing and checking if the images are safe by implementing Trivy. In order to run Trivy, we needed to install it. Hence, we added a task to install Trivy and another task to use Trivy for image scanning before pushing. If the images have issues, they will not get pushed. Once all the images are safe, they get pushed into the ECR.

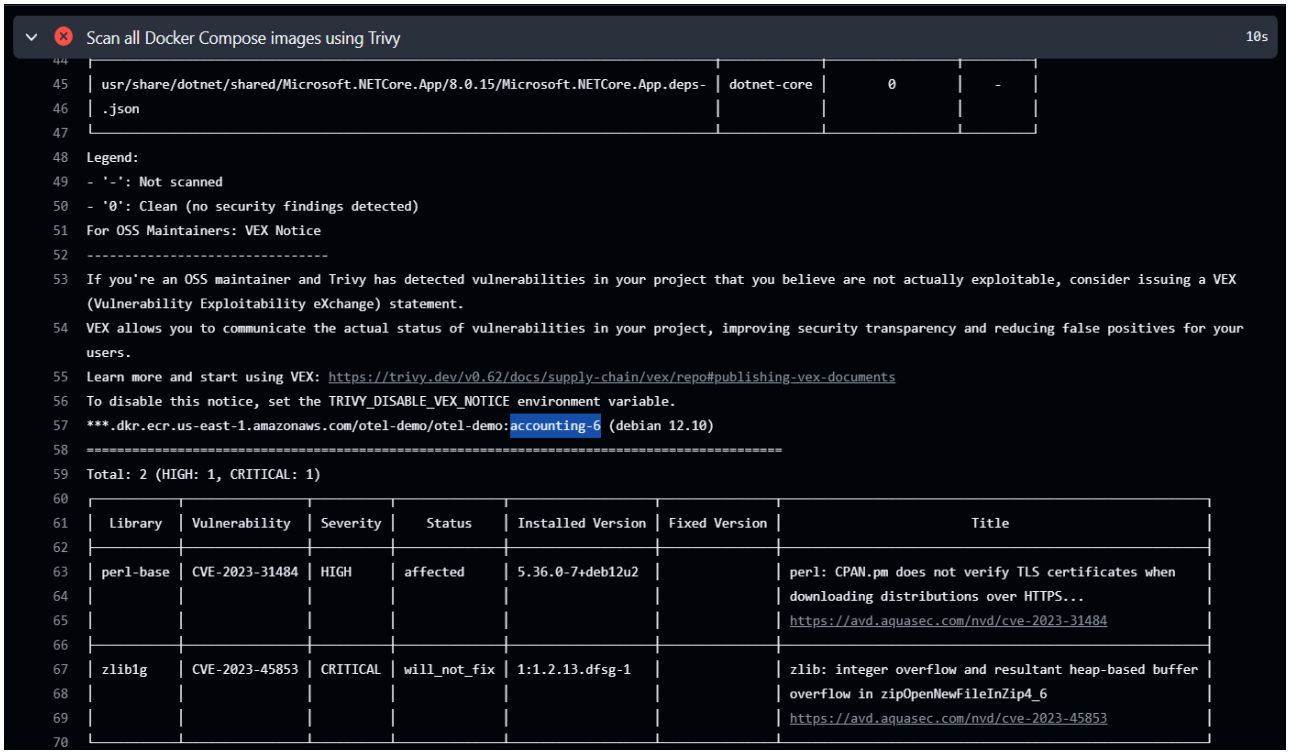


Figure 77: Trivy Test

This screenshot shows that the Trivy scan came out with results indicating that there were two errors in an image and it led to the pipeline failing and not pushing the images to ECR. Once the scan results came out to be safe, the images were pushed into the ECR.

Step 10: We used the opentelemetry-demo.yml file for deployment. The image names in the file were updated to the image URIs of the images pushed into ECR.

Step 11: We added tasks in our cicd-pipeline.yml file to set up the kubeconfig and deploy to our EKS cluster. The following commands were used in the pipeline to complete the deployment.

→aws eks update-kubeconfig --region $REGION --name $CLUSTER_NAME
→kubectl apply -f kubernetes/opentelemetry-demo.yaml -n $NAMESPACE --validate=false

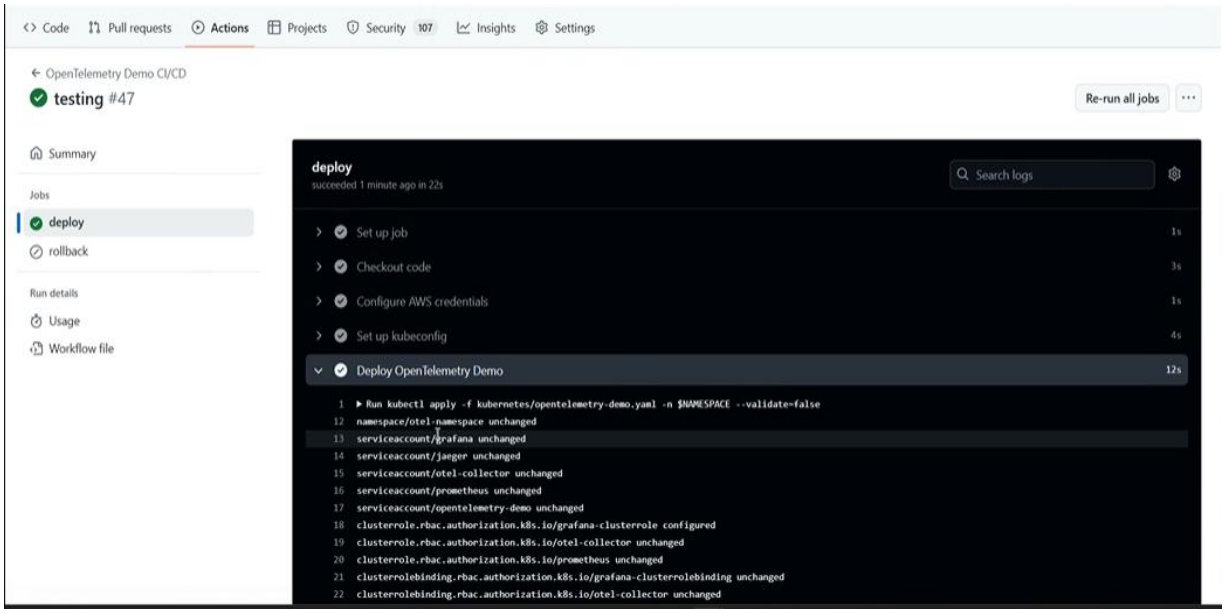


Figure 78: Successful Execution of the CI/CD Pipeline in 47th Commit

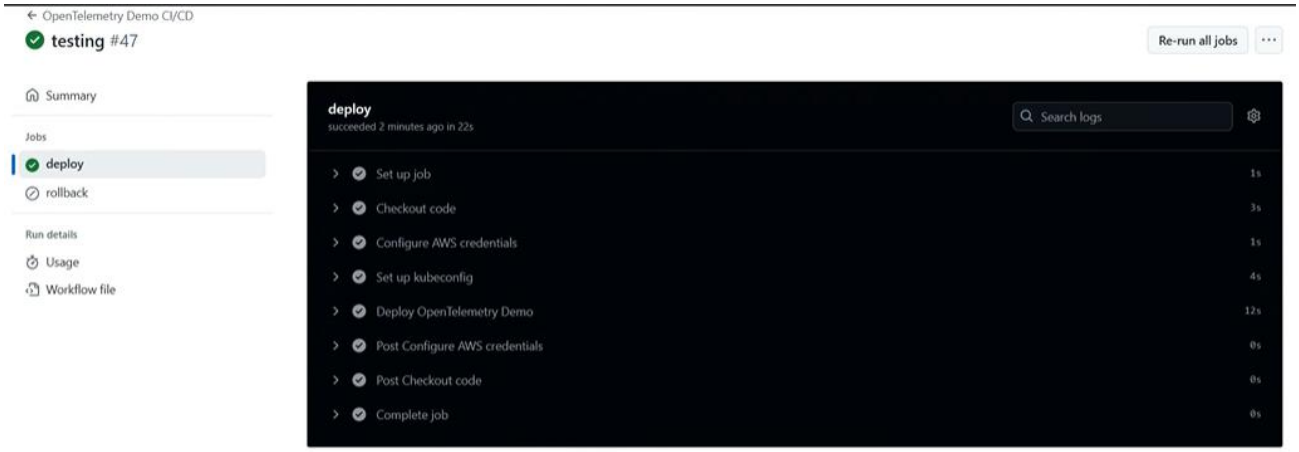


Figure 79: Deployment Successful

Step 5: We created a namespace inside our cluster called otel-namespace, and the screenshot shows that the deployment was successful with all pods running.

```
PS C:\Users\DELL\Desktop> kubectl get namespaces
NAME                STATUS   AGE
default             Active   47m
kube-node-lease     Active   47m
kube-public         Active   47m
kube-system         Active   47m
otel-namespace      Active   26m
PS C:\Users\DELL\Desktop>
```

Figure 80: All the namespaces present in our Kubernetes Cluster

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                   powershell

PS C:\Users\DELL\Desktop> kubectl get pods -n otel-namespace
NAME                                 READY   STATUS    RESTARTS   AGE
accounting-59d6774748-vdl42          1/1     Running   0          3m8s
ad-95cf4fdb9-4gkmk                   1/1     Running   0          3m8s
cart-7858b899bb-ft682                1/1     Running   0          3m8s
checkout-7dc5d5cf74-kxh82            1/1     Running   0          3m8s
currency-5ffd4456cf-n5bmm            1/1     Running   0          3m8s
email-5b5b6c76dd-bdp2m               1/1     Running   0          3m8s
flagd-6c87f4c84d-xf7pl               2/2     Running   0          3m8s
fraud-detection-5fcbfcd6d7-7l8t8     1/1     Running   0          3m8s
frontend-6b6b8b4fc6-bhvsp            1/1     Running   0          3m7s
frontend-proxy-7f85bf78f7-9sc6v      1/1     Running   0          3m7s
grafana-64f9548f8c-jt6n9             1/1     Running   0          3m5s
image-provider-66584b847f-67jk8      1/1     Running   0          3m7s
jaeger-76b4cc75dd-vgwxm              1/1     Running   0          3m8s
kafka-5b9c6db458-f59pj               1/1     Running   0          3m7s
load-generator-6bf6549cdf-kvzpv      1/1     Running   0          3m7s
opensearch-0                         1/1     Running   0          3m7s
otel-collector-7fb566966b-j7rrm      1/1     Running   0          3m5s
payment-668b9655bc-mt8d5             1/1     Running   0          3m6s
product-catalog-6f8869b65c-zfw2r     1/1     Running   0          3m6s
prometheus-59fc4944c7-qkjjr          1/1     Running   0          3m5s
quote-7dbf5d4694-tr5gf               1/1     Running   0          3m6s
recommendation-f8fb9cd4f-p5rzf       1/1     Running   0          3m6s
shipping-685b87765c-6mbsv            1/1     Running   0          3m6s
valkey-cart-b9fcd6689-46mqj          1/1     Running   0          3m6s
PS C:\Users\DELL\Desktop>
```

Figure 81: All the pods are running in our Kubernetes Cluster

6.4 Task 2: Enable Rollback Mechanism

This task focused on implementing a rollback mechanism to ensure stability during deployment. If any issue occurred during image build or deployment, the pipeline would automatically revert to the last known working version.

Step 1: We added a rollback task in the cicd-pipeline.yml file. This task was triggered if there was a failure in building or pushing a new version of the image.

Step 2: To test the rollback mechanism, we manually introduced an error in the Docker image to simulate a failure.

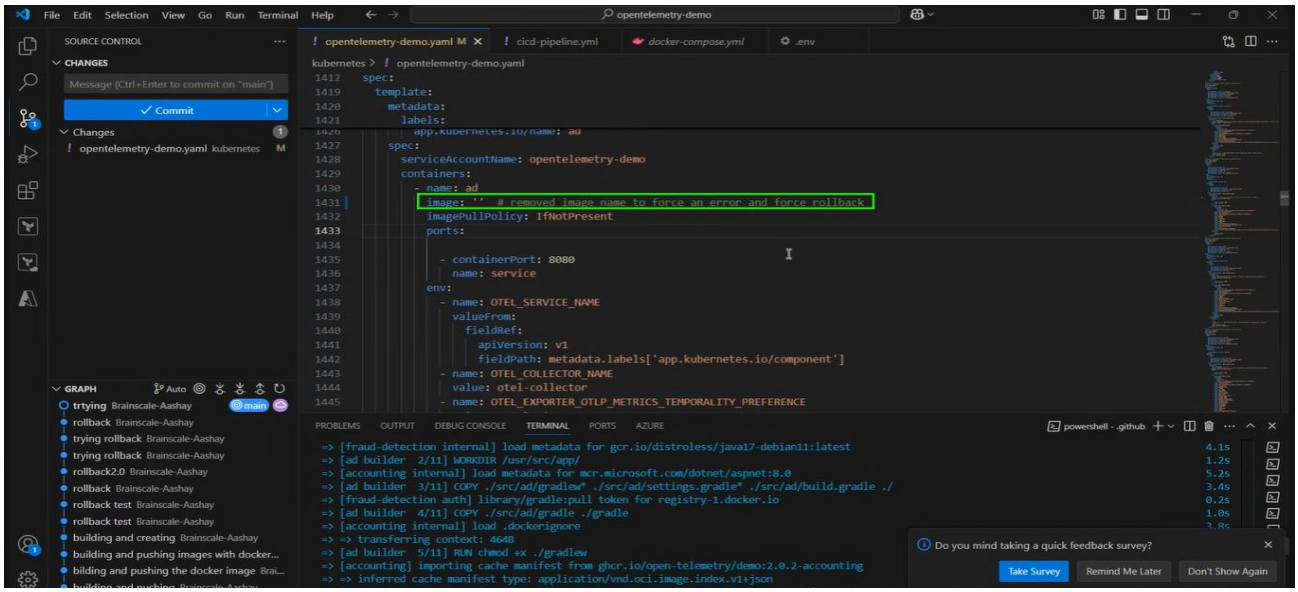


Figure 82: Error Introduced in the Code

→This error caused the pipeline to fail and triggered the rollback, which reverted the deployment to the previously working image version. The issue was introduced in the image for the "AD" service; as a result, instead of deploying the faulty version 6, the pipeline rolled back to version 3.

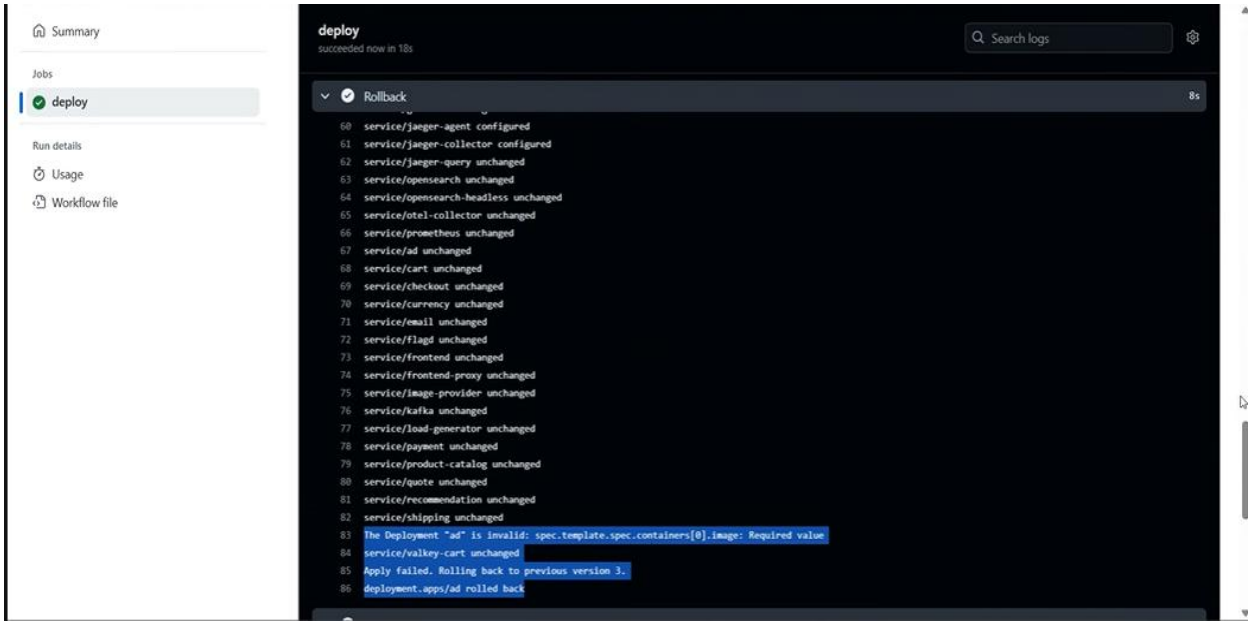


Figure 83: Rollbacked to Previous Version

→The screenshot confirms that the rollback task was executed successfully and the application reverts back to a stable state version.

6.5 Challenges Encountered

We encountered several challenges during this phase that required extensive troubleshooting. The pipeline was executed 87 times due to multiple failed attempts before achieving a successful deployment. One of the key issues was building Docker images and pushing them to AWS ECR, which involved several trials to correctly reference the docker-compose file. Establishing a reliable connection between the pipeline and the EKS cluster also proved challenging and was resolved by updating IAM role permissions. Additionally, implementing the rollback mechanism required careful configuration of conditional flags within the workflow, which involved significant research to ensure it triggered appropriately during failures.

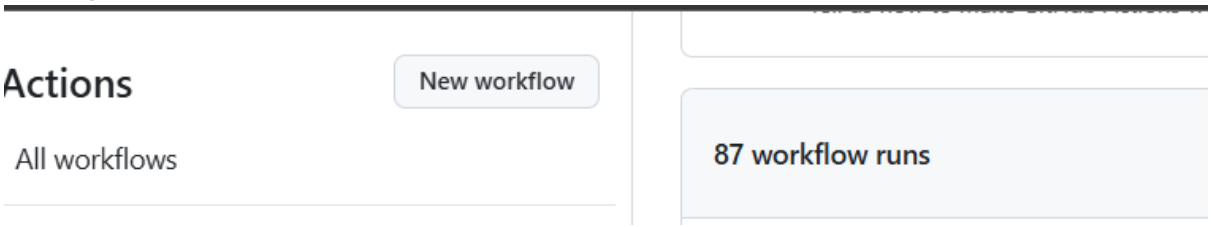


Figure 84: No. of attempts to run the pipeline

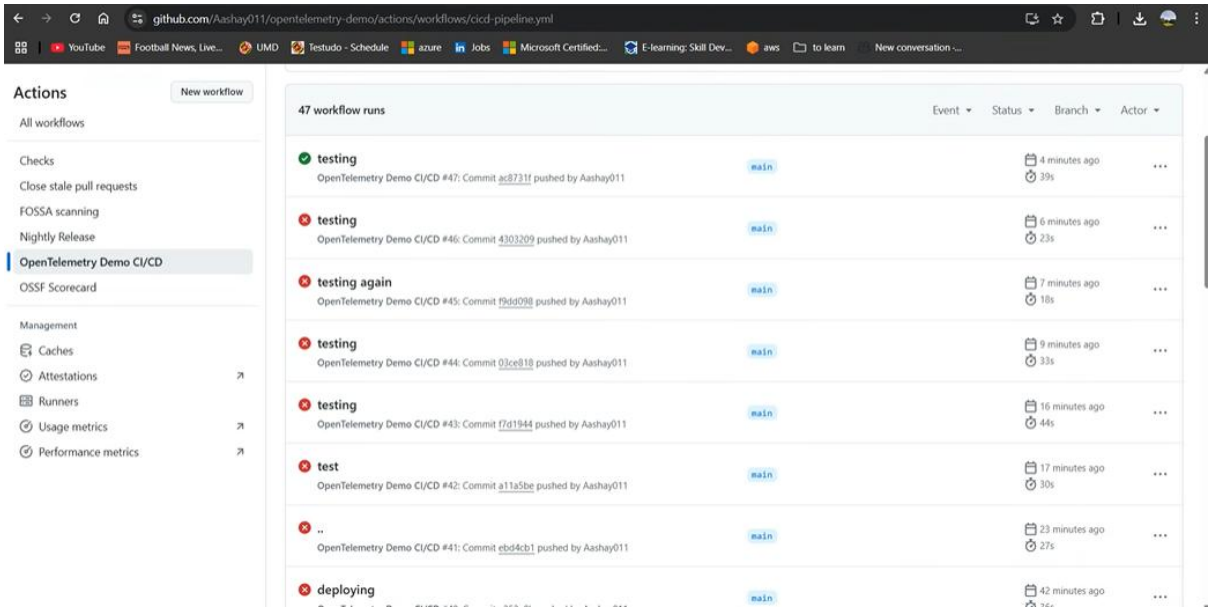


Figure 85: Workflows present in the GitHub while testing

Conclusion

7.1 Lesson Learned

1. **Deploying with Helm Simplifies Management**
   We learned that Helm streamlines complex Kubernetes deployments using templated manifests and versioned releases, improving reusability and rollback support compared to raw YAML.
2. **Namespace Isolation Matters**
   Creating a separate namespace (like otel-helm-demo) helped avoid conflicts with previous resources and allowed control of deployed services.
3. **Observability Is Essential from Day One**
   Integrating Prometheus, Grafana, and Jaeger early in the pipeline gave us visibility into system health, performance, and distributed traces which are useful for debugging and monitoring.
4. **CI/CD Pipelines Enable Consistent Delivery**
   Automating deployments through GitHub Actions not only reduced manual intervention but also allowed consistency and reduced deployment errors across environments.
5. **Resource Cleanup Is Critical for Stability**
   We encountered issues where leftover pods and services from previous namespaces caused Helm deployments to fail. This highlighted the importance of cleaning unused resources during migrations or re-deployments.
6. **Security Should Be Proactive Not Reactive**
   We recognized the importance of securing secrets, IAM roles, and networking boundaries. Tools like AWS Secrets Manager, WAF, and IAM best practices are essential for protecting cloud-based applications.
7. **Cluster Security & Access Control**
   The EKS cluster was deployed within private subnets and secured using IAM roles and Kubernetes RBAC. We learned how access to the control plane is restricted by default and recognized the value of enabling features like KMS encryption for secrets and IAM Roles for Service Accounts to follow the principle of least privilege.

7.2 Recommendations

1. **Implement AWS Secrets Manager**
   Store sensitive data like environment variables and API tokens securely using AWS Secrets Manager instead of embedding them directly in manifests or GitHub workflows.
2. **Add Multi-Factor Authentication (MFA)**
   Enforce MFA for IAM users involved in infrastructure development to enhance account-level security and access control.
3. **Include IAM Role Policies in Project Documentation**
   Clearly document IAM policies attached to infrastructure and user roles to support traceability and reproducibility.
4. **Integrate AWS Shield and WAF**
   Protect the Kubernetes API and application endpoints from DDoS attacks using AWS Shield and filter malicious traffic with AWS Web Application Firewall (WAF).
5. **Enable Cost Monitoring with AWS Budgets**
   Set up AWS Budgets and Alerts to keep track of usage and avoid unexpected expenses especially when EKS cluster is running for longer duration.
6. **Externalize Helm Values for Flexibility**
   We can move reusable configuration data (like service names, replica counts, and image tags) to external values.yaml files to support cleaner upgrades.
7. **Tag Resources for Clarity and Billing**
   Apply meaningful tags (e.g., Environment = Dev, Owner = Group 18) to AWS resources to simplify cost tracking, audits, and automated cleanup scripts.
8. **Practice Rollback Validation Regularly**
   Test Helm rollbacks and simulate failed releases in staging environments to ensure the rollback mechanism performs reliably under real-world scenarios.

References

https://opentelemetry.io/docs/demo/
https://docs.docker.com/reference/cli/docker/
https://docs.aws.amazon.com/eks/
https://docs.aws.amazon.com/ecs/
https://aws.amazon.com/blogs/security/use-iam-roles-to-connect-github-actions-to-actions-in-aws/
https://eksctl.io/

Video Walkthrough Link: https://drive.google.com/file/d/1_hY2j5fRadtY-bleq-tVfo8An1EsEnmW/view?usp=sharing

Repository link: https://github.com/Aashay011/opentelemetry-demo