DevOps Engineer
Assignment Documentation

Submitted by - Sameera Sandeepa
Submission Date -02/04/2025

Content

1. Python Model Description

KServe Model Service

The **KServe Model Service** is a Python-based service that leverages **KServe** to deploy and serve an **OCR (Optical Character Recognition) model** using **Tesseract**. This service is responsible for processing incoming image data, extracting text using Tesseract OCR, and returning the recognized text as output. The model is implemented in the model.py script, which defines how the service loads and processes images for text recognition.

FastAPI Gateway Service

The **FastAPI Gateway Service** acts as an entry point for users to interact with the OCR model. This service is built using **FastAPI**, a high-performance web framework for Python. It provides an API endpoint where users can upload images, which are then encoded in **Base64** format and forwarded to the KServe Model Service for inference. The gateway handles communication between clients and the model service, ensuring efficient request processing and response handling. The implementation is in api-gateway.py.

Terminology

- **KServe**: An open-source model serving platform for Kubernetes that simplifies the deployment and scaling of machine learning models. It provides built-in features for inference request handling, autoscaling, and monitoring.
- **FastAPI**: A modern web framework for building APIs in Python. It is designed for high performance and ease of use, making it ideal for creating API gateways and backend services.
- **OCR (Optical Character Recognition)**: A technology that extracts text from images, enabling machines to recognize and process printed or handwritten characters.
- **Tesseract**: An open-source OCR engine developed by Google, widely used for text recognition in scanned documents and images.
- **Base64 Encoding**: A method of converting binary data (such as images) into an ASCII string format, making it easier to transmit over text-based communication protocols.
- **Inference**: The process of using a trained machine learning model to make predictions or extract information from input data.

a. Architecture Diagram

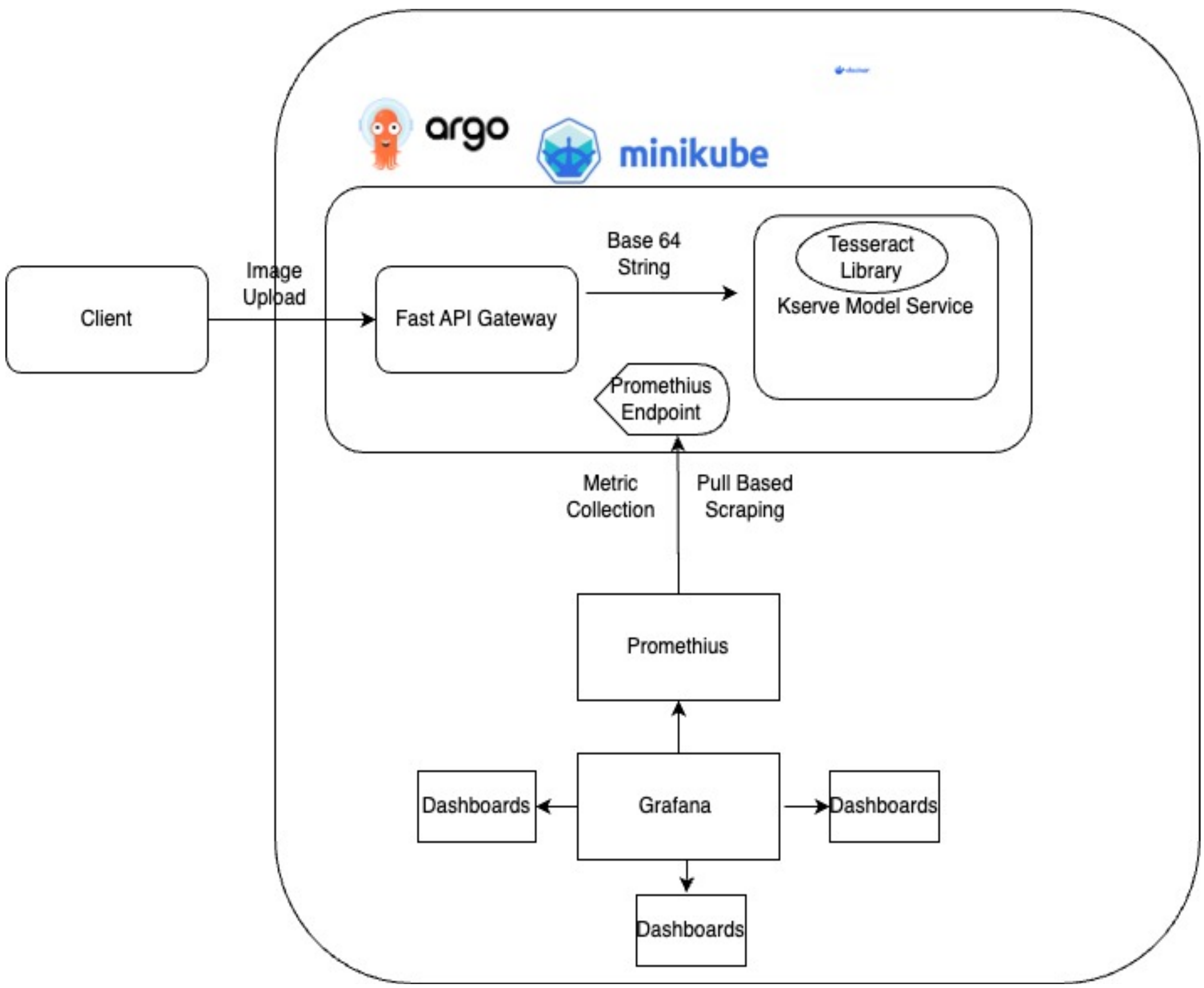

FIGURE 1- ARCHTECTURE DIAGRAM

b. Code Adjustments from Local Setup to Container

1. Kserve URL

```
KSERVE_URL = "http://model:8080/v2/models/ocr-model/infer"  # Your KServe model server URL
```

FIGURE 2- KSERVE URL

In Docker (Single Container or Docker Compose)

- Each container has its own network namespace.
- If the FastAPI container tries to call http://localhost:8080, it will look for the service **inside its own container**, which won't work unless both services are inside the same container (not recommended).
- Instead, we reference the **service name** (model) as defined in **Docker Compose** or Kubernetes.

In Kubernetes (Minikube, KServe)

- Kubernetes provides a **DNS system** where each pod or service can be reached by its **service name** inside the cluster.
- Instead of localhost, we use the **Kubernetes Service name (model)**, assuming KServe is deployed with a service named model

2. Output Data Type

```python
output = InferOutput(
    name="output-0",
    shape=[1],
    datatype="BYTES",
    data=[extracted_text] # Base64 encode the output List
)
```

FIGURE 3- EXTRACTED TEXT

Local Execution (Non-Containerized)

When you run the model **locally**, Python is more forgiving. If data=extracted_text (a string), KServe might internally handle it without enforcing strict type validation.

Containerized Execution (Inside KServe)

When the model runs inside a **KServe container**, the response **must strictly follow the KServe V2 API**.

- KServe expects InferOutput.data to be a **list of values**, even for a single string.
- If you provide a **raw string**, it may fail validation or cause unexpected behavior.

2. Dependency Installation

1. Python 3.11

Python 3.11 is the primary programming language used in this project. It provides performance improvements and better error messages compared to previous versions.

2.Tesseract-OCR

Tesseract is an **open-source Optical Character Recognition (OCR) engine** used for extracting text from images. The **OCR model** in this project relies on Tesseract for text recognition.

macOS (Homebrew):

```
brew install tesseract
```

3.Poetry (Dependency & Virtual Environment Manager)

Poetry is a modern Python dependency manager used in this project to handle **package installations, dependency resolution, and virtual environments**. It simplifies project setup and ensures reproducibility.

macOS (Homebrew):

```
curl -SSL https://install.python-poetry.org | python3 -
```

4.Running the Services with Poetry

```
poetry run python model.py

poetry run python api-gateway.py
```

3.Local Test Results

Below tests has been done through postman

Request:

```
curl --location 'http://localhost:8001/gateway/ocr' \

--form
'image_file=@"/Users/sameerasandeepa/Documents/Smile.jp
g"'
```

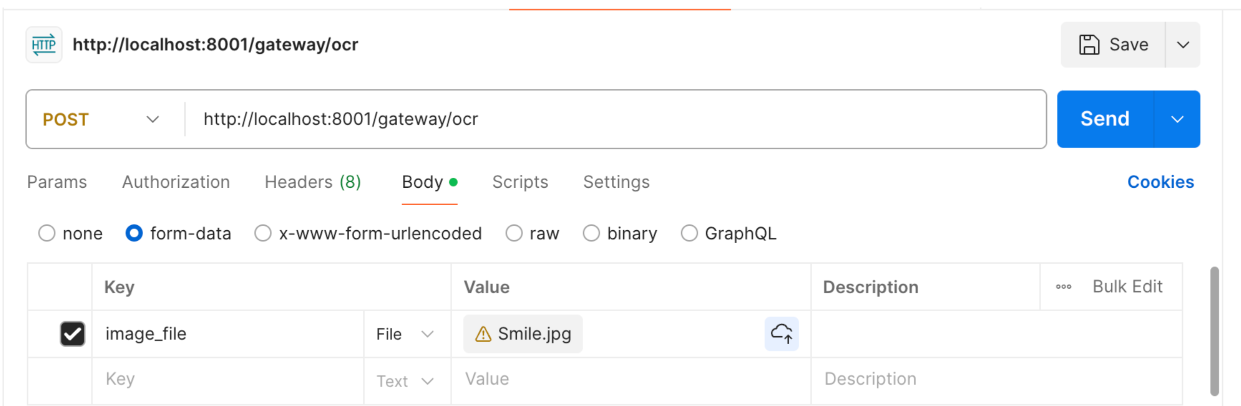

FIGURE 4- API GATEWAY TESTING

Response

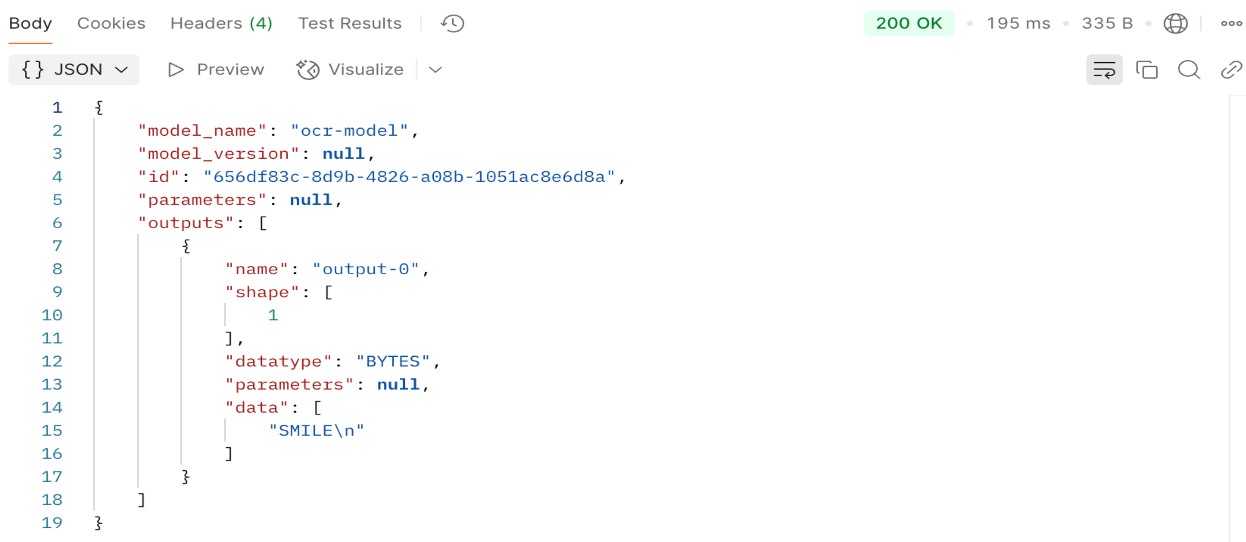

FIGURE 5- RESPONSE

Testing model alone

Get the encoded base 64 version of a known image thorough base 64 converter and input for below request

Request:

`http://localhost:8080/v2/models/ocr-model/infer`

Code snippet

cURL

```
curl --location 'http://localhost:8080/v2/models/
    ocr-model/infer' \
--header 'Content-Type: application/json' \
--data '{
    "inputs": [
        {
            "name": "input-0",
            "shape": [1],
            "datatype": "BYTES",
            "data": [
                "/9j/4AAQSkZJRgABAQAAAQABAAD//
                    gA7Q1JFQVRPUjogZ2QtanBlZyB2MS4w
                    ICh1c2luZyBJSkcgSlBFRyB2NjIpLCB
                    xdWFsaXR5ID0gODIK/
                    9sAQwAGBAQFBAQGBQUFBgYGBwkOCQkI
                    CAkSDQ0KDhUSFhYVEhQUFxohHBcYHxk
                    UFB0nHR8iIyUlJRYcKSwoJCshJCUk/
                    9sAQwEGBgYJCAkRCQkRJBgUGCQkJCQk
                    JCQkJCQkJCQkJCQkJCQkJCQkJCQkJCQ
                    kJCQkJCQkJCQkJCQkJCQkJCQkJCQk/
                    8AAEQgCMwPoAwEiAAIRAQMRAf/
                    EAB8AAAEFAQEBAQEBAAAAAAAAAAABAg
                    MEBQYHCAkKC//
                    EALUQAAIBAwMCBAMFBQQEAAABfQECAw
                    AEEQUSITFBBhNRYQcicRQygZGhCCNCs
                    cEVUtHwJDNicoIJChYXGBkaJSYnKCkq
                    NDU2Nzg5OkNERUZHSElKU1RVVldYWVp
                    jZGVmZ2hpanN0dXZ3eHl6g4SFhoeIiY
                    qSk5SVlpeYmZqio6Slpqeoqaqys7S1t
                    re4ubrCw8TFxsfIycrS09TV1tfY2drh
                    4uPk5ebn6Onq8fLz9PX29/j5+v/
```

FIGURE 6- REQUEST TO MODEL

Response

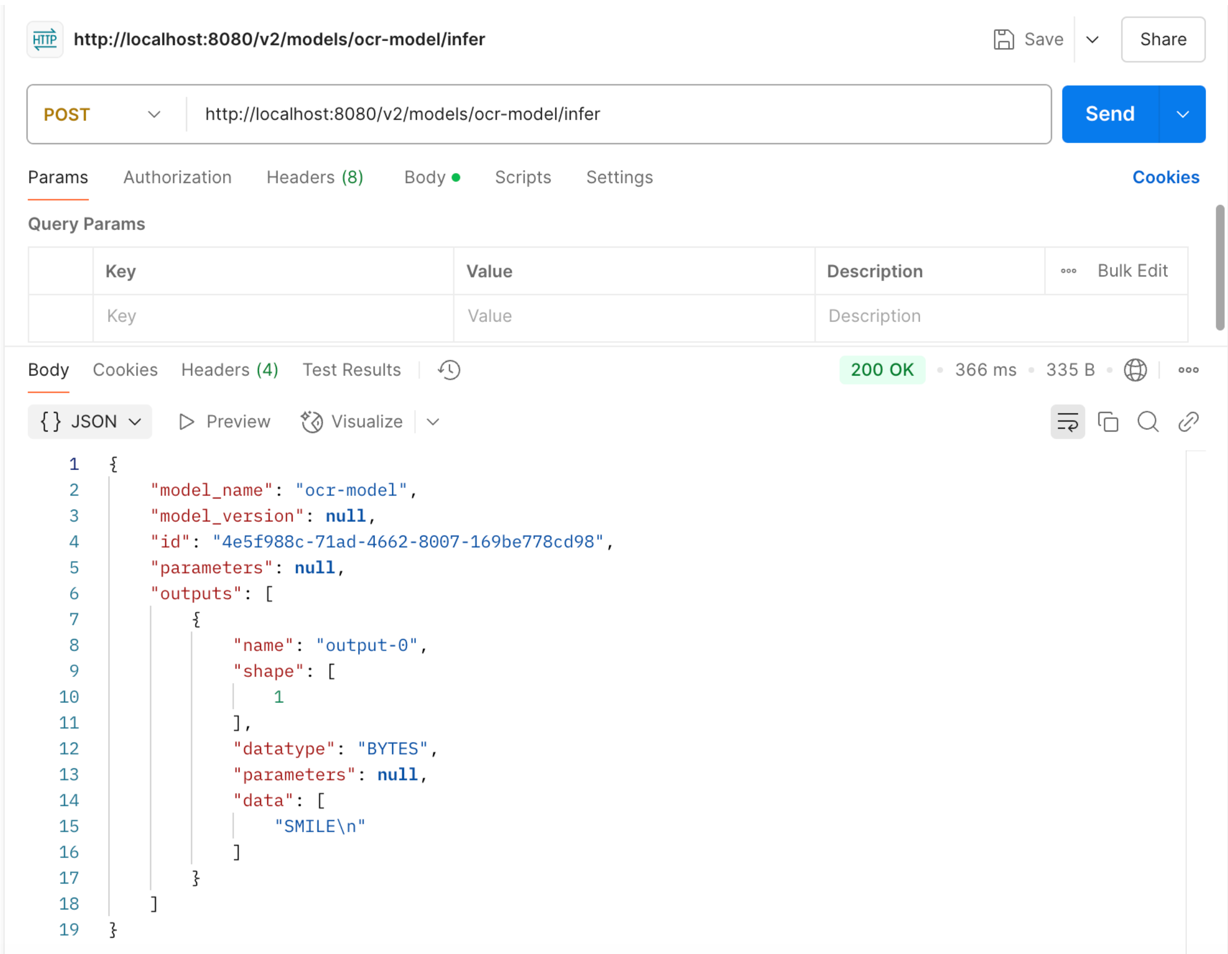

FIGURE 7-RESPONSE

4.Containerization

Below considerations need to be done when using docker for containerization

- ## Gateway Api Service

1.Base Image Selection

```
FROM python:3.11-slim
```

Uses **Python 3.11 Slim**, a minimal version of Python, reducing unnecessary dependencies and improving container efficiency.

2.Install System Dependencies

```
RUN apt-get update && apt-get install -y curl && rm -
rf /var/lib/apt/lists/*
```

Updates the package list and installs **curl** (useful for health checks and API testing).

Cleans up cache (rm -rf /var/lib/apt/lists/*) to keep the image small.

3. Copy and Install Python Dependencies

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

Below were included in requirements.txt

```
fastapi==0.110.0
uvicorn==0.27.1
requests==2.31.0
python-multipart
```

4.Set the Entry point (Application Startup Command)

```
ENTRYPOINT ["uvicorn", "api-gateway:app", "--host",
           "0.0.0.0", "--port", "8001"]
```

- **Model service**

1.Install System Dependencies (Including Tesseract-OCR)

```
RUN apt-get update && apt-get install -y \
    g++ \
    build-essential \
    tesseract-ocr \
    && apt-get clean && rm -rf /var/lib/apt/lists/*
```

2. Installs essential build tools (g++, build-essential) needed for compiling dependencies.

Installs **Tesseract-OCR**, the core OCR engine required by the model to process images.

Cleans up package lists (apt-get clean && rm -rf /var/lib/apt/lists/*) to reduce image size.

3.Copy and Install Python Dependencies

Requirement.txt include below

```
kserve==0.11.0
pillow==10.0.1
pytesseract==0.3.10
```

docker desktop has been used and below are docker build steps for the two services

```
docker build -t sameerasandeepa/model-ocr:latest -f
ocr_model/dockerfile.model ocr_model/
docker run -d --name model-ocr-container -p 8080:8080
sameerasandeepa/model-ocr:latest

docker build -t sameerasandeepa/fastapi-gateway -f
dockerfile.gateway .
docker run -d --name api-gateway-container -p 8001:8001
sameerasandeepa/fastapi-gateway
```

Docker Container Setup

The Docker containers were created locally using Docker Desktop. To facilitate seamless connectivity between the two containers and streamline the setup process, **Docker Compose** was used. By leveraging a **Docker Compose file**, multiple containerized services were efficiently managed and deployed together, ensuring smooth communication between them.

This approach simplifies multi-container orchestration, enabling better resource management and easier deployment in the local development environment.

docker-compose.yaml

```yaml
version: '3.8'

services:
  model:
    build:
      context: ./ocr_model  # Change this to point to the correct directory
      dockerfile: dockerfile.model
    container_name: model-ocr-container
    ports:
      - "8080:8080"
    networks:
      - app-network

  api-gateway:
    build:
      context: .  # This is fine since Dockerfile.gateway is in the current directory
      dockerfile: dockerfile.gateway
    container_name: api-gateway-container
    ports:
      - "8001:8001"
    networks:
      - app-network

networks:
  app-network:
    driver: bridge
```

FIGURE 8- DOCKER COMPOSE FILE

Building and Pushing Docker Images

A **Docker Compose** file was created and used to build two Docker images—one for the **Model Service** and another for the **API Gateway**. These images were then tagged and pushed to the container registry for deployment.

Building Docker Images

The images were built using the following command:

docker compose build

Tagging Docker Images

Once built, the images were tagged to prepare them for repository storage:

```
docker tag devopsengineer-assignment-model:v2
sameerasandeepa/devopsengineer-assignment-model:v2

docker tag devopsengineer-assignment-api-gateway:v2
sameerasandeepa/devopsengineer-assignment-api-
gateway:v2
```

Pushing Images to Repository

The tagged images were then pushed to the repository for accessibility:

```
docker push sameerasandeepa/devopsengineer-assignment-
model:v2
docker push sameerasandeepa/devopsengineer-assignment-
api-gateway:v2
```

These Docker images were later pulled and deployed via **Helm charts**, referencing them with the **v2** tag.

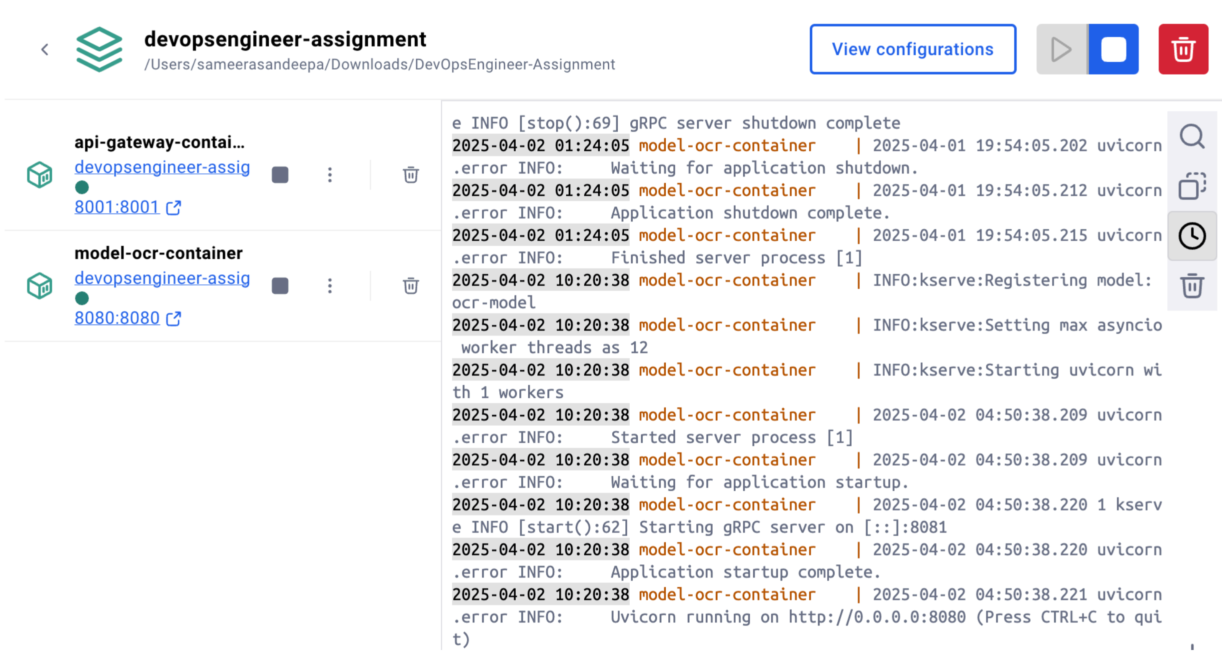

FIGURE 9-DOCKER HUB CONTAINER REGISTRY

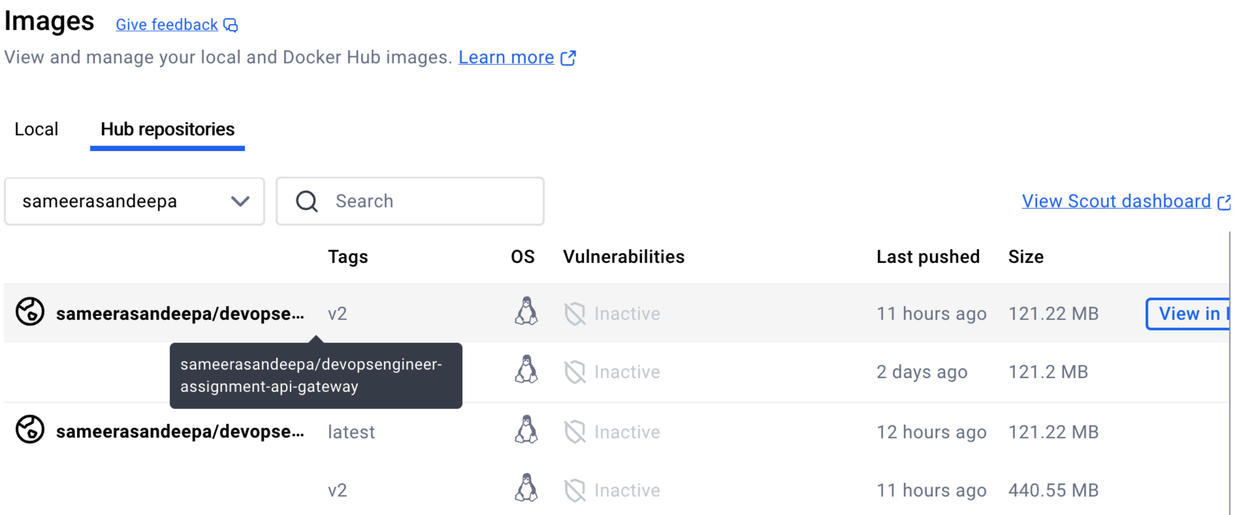

FIGURE 10 DOCKER HUB REPOSITORIES

Pushing Helm Chart to Docker Hub

The Helm charts, namely **model-chart** and **gateway-api-chart**, were pushed to Docker Hub to make them accessible for deployment. The following commands were used to tag and push the charts:

```
docker tag model-chart:v2 sameerasandeepa/model-chart:v2
docker tag gateway-api-chart:v2 sameerasandeepa/gateway-api-chart:v2

docker push sameerasandeepa/model-chart:v2
docker push sameerasandeepa/gateway-api-chart:v2
```

5.Infrastructure Setup

Minikube Cluster Setup on macOS

As part of our test infrastructure, we have set up a Minikube cluster on macOS. The installation was performed using the Homebrew package manager with the following command:

```
brew install minikube
```

After installation, the cluster was initiated using:

```
minikube start
```

This document includes images and configurations related to the Minikube cluster setup, providing insights into the cluster's architecture, resource allocation, and deployed services

```
sameerasandeepa@SYSCO-C76V65YHTK ~ % minikube start
😄  minikube v1.35.0 on Darwin 15.3.2 (arm64)
✨  Using the docker driver based on existing profile
👍  Starting "minikube" primary control-plane node in "minikube" cluster
🚜  Pulling base image v0.0.46 ...
🏃  Updating the running docker "minikube" container ...
🐳  Preparing Kubernetes v1.32.0 on Docker 27.4.1 ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: default-storageclass, storage-provisioner
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
sameerasandeepa@SYSCO-C76V65YHTK ~ % kubectl get ns
NAME              STATUS   AGE
argocd            Active   2d8h
cert-manager      Active   40h
default           Active   2d8h
kserve            Active   36h
kube-node-lease   Active   2d8h
kube-public       Active   2d8h
kube-system       Active   2d8h
monitoring        Active   2d8h
```

FIGURE 110MINIKUBE INITIALIZATION

```
sameerasandeepa@SYSCO-C76V65YHTK ~ % minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

sameerasandeepa@SYSCO-C76V65YHTK ~ % kubectl describe node minikube
Name:               minikube
Roles:              control-plane
Labels:             beta.kubernetes.io/arch=arm64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=arm64
                    kubernetes.io/hostname=minikube
                    kubernetes.io/os=linux
                    minikube.k8s.io/commit=dd5d320e41b5451cdf3c01891bc4e13d189586ed
                    minikube.k8s.io/name=minikube
                    minikube.k8s.io/primary=true
                    minikube.k8s.io/updated_at=2025_03_31T01_28_16_0700
                    minikube.k8s.io/version=v1.35.0
                    node-role.kubernetes.io/control-plane=
                    node.kubernetes.io/exclude-from-external-load-balancers=
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: unix:///var/run/cri-dockerd.sock
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Mon, 31 Mar 2025 01:28:13 +0530
Taints:             <none>
Unschedulable:      false
Lease:
  HolderIdentity:  minikube
  AcquireTime:     <unset>
  RenewTime:       Wed, 02 Apr 2025 09:51:25 +0530
```

FIGURE 12- MINIKUBE STATUS DETAILS

5.Git Ops with ArgoCD

Argo CD Deployment and Helm Chart Management

Argo CD was deployed in the Minikube cluster using **Helm charts** to facilitate GitOps-based deployment. The following command was used for the installation:

```
helm install argocd argo/argo-cd --namespace argocd --create-namespace
```

This command installs Argo CD in the **argocd** namespace, creating a centralized management platform for continuous deployment within the Minikube cluster.

```
sameerasandeepa@SYSCO-C76V65YHTK ~ % cd argo
sameerasandeepa@SYSCO-C76V65YHTK argo % ls
application-model-notproject.yaml      applicationset.yaml
application-model.yaml                 appproject.yaml
application.yaml
sameerasandeepa@SYSCO-C76V65YHTK argo %
```

FIGURE 13- ARGO CD DIRECTORY

Creating and Applying Argo CD Configurations

To manage the deployment of these services using Argo CD, the following YAML files were created within a directory and applied to the **argocd** namespace:

- **ApplicationSet.yaml**
- **Application.yaml**
- **AppProject.yaml**

These resources define the GitOps configuration for the services, specifying how Argo CD should sync and manage the applications within the cluster.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: model-api
  namespace: argocd
spec:
  destination:
    server: https://kubernetes.default.svc
    namespace: default
  source:
    # Directly using the Docker image in the container
    repository: docker.io/sameerasandeepa/devopsengineer-assignment-api-gateway
    targetRevision: v1.0
    chart: ""
  project: default
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

FIGURE 14- ARGO CD APPLICATION YAML

The following applications have been successfully deployed and managed within Argo CD for continuous delivery and deployment in the Minikube cluster:

- **Model Service**
- **API Gateway Service**

These applications are now fully integrated with Argo CD, enabling automated synchronization and management through GitOps workflows. Argo CD ensures that any changes to the application configurations are automatically reflected in the cluster, ensuring consistency and version control.

```
sameerasandeepa@SYSCO-C76V65YHTK argo % kubectl get svc -n argocd
NAME                              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
argocd-applicationset-controller  ClusterIP   10.104.99.127   <none>        7000/TCP                     2d9h
argocd-dex-server                 ClusterIP   10.97.169.172   <none>        5556/TCP,5557/TCP            2d9h
argocd-redis                      ClusterIP   10.96.126.136   <none>        6379/TCP                     2d9h
argocd-repo-server                ClusterIP   10.102.98.175   <none>        8081/TCP                     2d9h
argocd-server                     NodePort    10.101.94.37    <none>        80:30080/TCP,443:30443/TCP   2d9h
sameerasandeepa@SYSCO-C76V65YHTK argo % kubectl get po -n argocd
NAME                                               READY   STATUS             RESTARTS       AGE
argocd-application-controller-0                    1/1     Running            5 (7h57m ago)  2d9h
argocd-applicationset-controller-c456dccf7-6nw97   1/1     Running            5 (7h57m ago)  2d9h
argocd-dex-server-8479fc6486-sqqhc                 0/1     ImagePullBackOff   0              2d9h
argocd-notifications-controller-669ff4c49b-2c5zg   1/1     Running            5 (7h57m ago)  2d9h
argocd-redis-9fbdbf99-st7zn                        1/1     Running            5 (7h57m ago)  2d9h
argocd-repo-server-7949674b4-jrnzg                 1/1     Running            7 (7h57m ago)  2d9h
argocd-server-56df4bffb5-vj8n9                     1/1     Running            14 (7h57m ago) 2d9h
```

FIGURE 15- INSTALLED RESOUCES ARGO CD

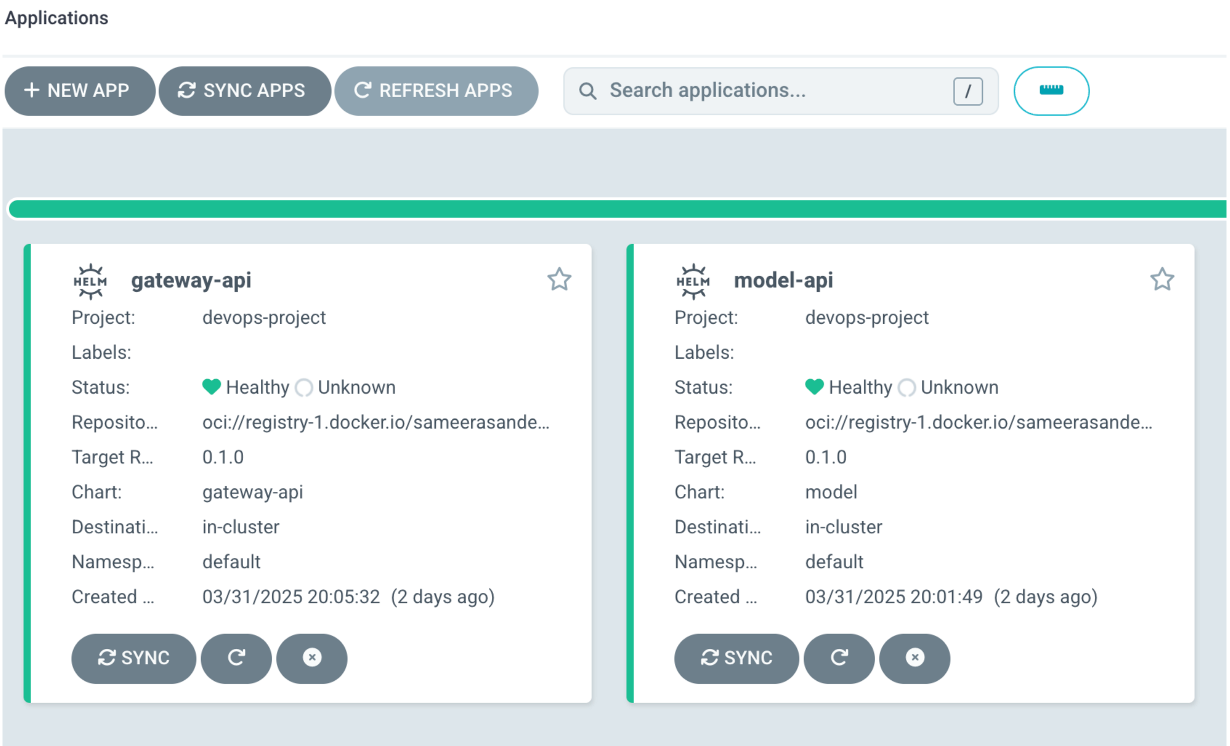

FIGURE 16- ARGO CD APPLICATIONS

7.Kubernetes Deployment

Two services have been deployed in the default namespace of the Minikube cluster using Helm charts. For these deployments, two separate Helm charts were created:

- **Model Service:** Deployed using the model-chart Helm chart.
- **API Gateway:** Deployed using the gateway-api-chart Helm chart.

The Helm charts were created using the following commands:

```
helm create model-chart

helm create gateway-api-chart
```

These charts define the necessary Kubernetes resources and configurations required for the services to function within the Minikube cluster.

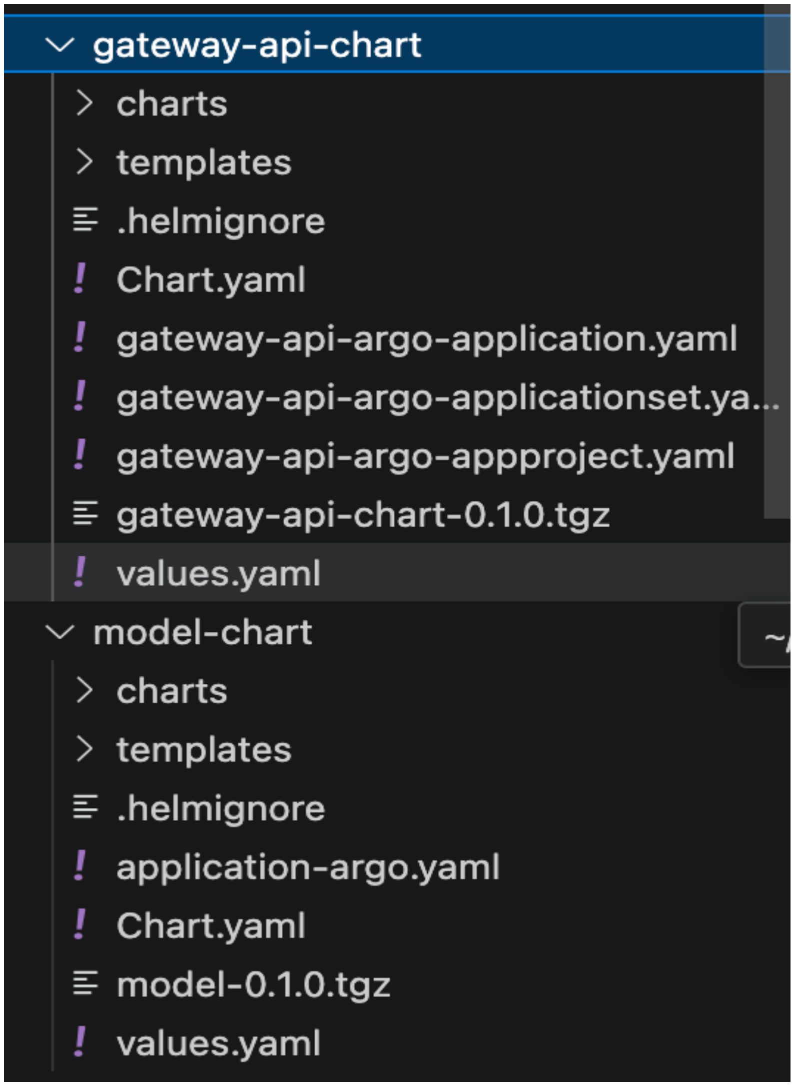

FIGURE 17- HELM CHARTS

Helm Chart Adjustments

Several modifications were made to the Helm charts to ensure proper configuration and deployment of the services. The key adjustments include:

- **values.yaml** – Configured repository details, image pull settings, IP and port configurations, and resource utilization parameters.
- **hpa.yaml** – Defined Horizontal Pod Autoscaler (HPA) settings to ensure dynamic scaling of the services based on resource utilization.
- **Chart.yaml** – Specified the chart name, version, and metadata for proper versioning and dependency management.
- **service.yaml** – Configured service-related details, including service type, ports, and selectors.
- **serviceaccount.yaml** – Defined service account settings to manage access control and permissions within the cluster.

These adjustments were made to ensure seamless deployment, scalability, and efficient resource utilization of the services.

HPA configuration

```yaml
autoscaling:
  enabled: true  # Set this to true or false based on your requirement
  minReplicas: 1
  maxReplicas: 3
  targetCPUUtilizationPercentage: 80
```

FIGURE 18- AUTO SCALING CONFIGURATIONS

```yaml
{{- if .Values.autoscaling.enabled }}
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "gateway-api-chart.fullname" . }}
  labels:
    {{- include "gateway-api-chart.labels" . | nindent 4 }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ include "gateway-api-chart.fullname" . }}
  minReplicas: {{ .Values.autoscaling.minReplicas }}
  maxReplicas: {{ .Values.autoscaling.maxReplicas }}
  metrics:
    {{- if .Values.autoscaling.targetCPUUtilizationPercentage }}
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: {{ .Values.autoscaling.targetCPUUtilizationPercentage }}
    {{- end }}
    {{- if .Values.autoscaling.targetMemoryUtilizationPercentage }}
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
```

FIGURE 19-HPA CONFIGURATIONS

Helm Installation and Upgrade Commands

The following commands were used to install and upgrade the Helm charts for the deployed services in the default namespace of the Minikube cluster:

```
helm install gateway-api gateway-api-chart --namespace default
```

- **Upgrade API Gateway Service:**

```
helm upgrade gateway-api gateway-api-chart --
namespace default
```

- Install Model Service:

```
helm install model-api model-chart --namespace
default
```

- Upgrade Model Service:

```
helm upgrade model-api model-chart --namespace
default
```

These commands ensure that the Helm charts are properly installed and updated, allowing for seamless deployment and management of the services.

8.Promethius integration

Prometheus was successfully deployed within the Kubernetes cluster using Helm. The installation process involved using the Helm package manager, which simplifies the deployment and management of Prometheus in Kubernetes environments.

Helm Install Command:

To install Prometheus in the monitoring namespace, the following Helm command was executed

```
helm install prometheus stable/prometheus --namespace monitoring
```

This command deploys the Prometheus server along with its necessary components like alert manager, node exporter, and push gateway under the monitoring namespace.

Local testing With Port Forwarding

To facilitate local testing and verification of the Prometheus service, port forwarding was enabled. This allows access to Prometheus from a local machine using a web browser or tools like curl or Postman.

Port Forwarding

To port forward the Prometheus server and make it accessible on localhost:9090, the following command was executed

```
kubectl port-forward -n monitoring svc/prometheus-server 9090:80
```

Once port forwarding was established, Prometheus could be accessed locally via the URL

http://localhost:9090

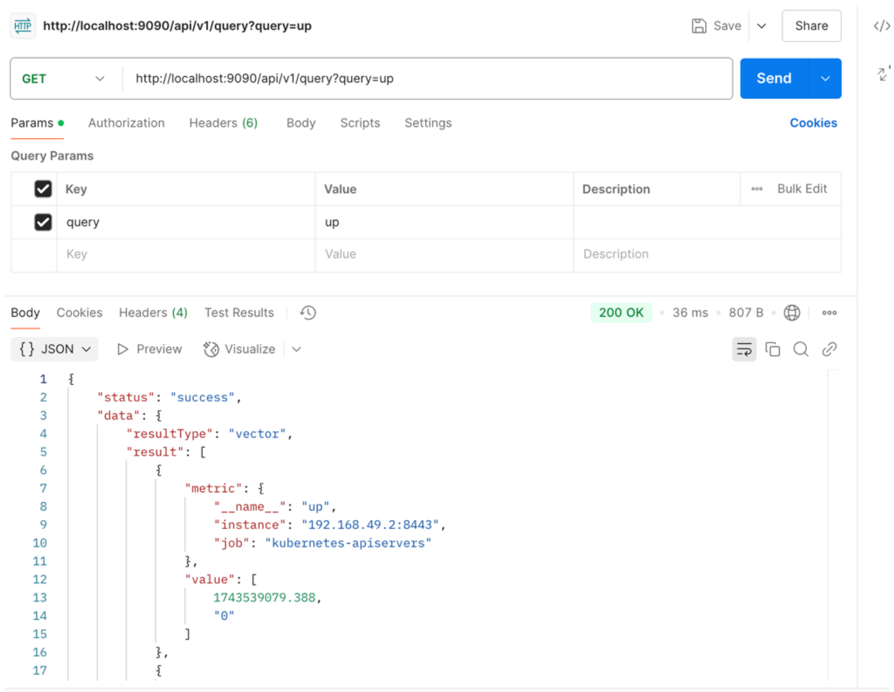

FIGURE 20 TESTING PROMETHIUS LOCALLY

Configuring Promethius to Scrape Metrics From Model- API

To enable Prometheus to collect custom metrics from the model-api service, changes were made to the Prometheus ConfigMap. This configuration modification is essential for Prometheus to begin scraping the relevant metrics exposed by the service.

1.Edit the Prometheus ConfigMap:

```
kubectl edit configmap prometheus-server -n monitoring
```

2.Add Scrape Configuration for model-api Service:

```
scrape_configs:
  - job_name: 'model-api'
    static_configs:
```

```
      - targets: ['model-
api.default.svc.cluster.local:8080']
```

In this configuration:

- The job_name is set to model-api to identify the metrics collection for this service.
- The static_configs block defines the target for the scraping, which points to the model-api service in the default namespace at port 8080. The service is accessible via the DNS name model-api.default.svc.cluster.local.

Testing and Integration

Once the configuration changes were applied and Prometheus started scraping metrics from model-api, the integration was tested:

1. **Verify Metrics Collection**:
   - Open the Prometheus web interface at http://localhost:9090.
   - Use the query up to verify that Prometheus is successfully scraping metrics from the model-api service.
2. **Local Test Verification**:
   - The metrics endpoint of model-api was tested by accessing the /metrics path locally. If metrics were correctly exposed, Prometheus should have collected and displayed them in the UI.

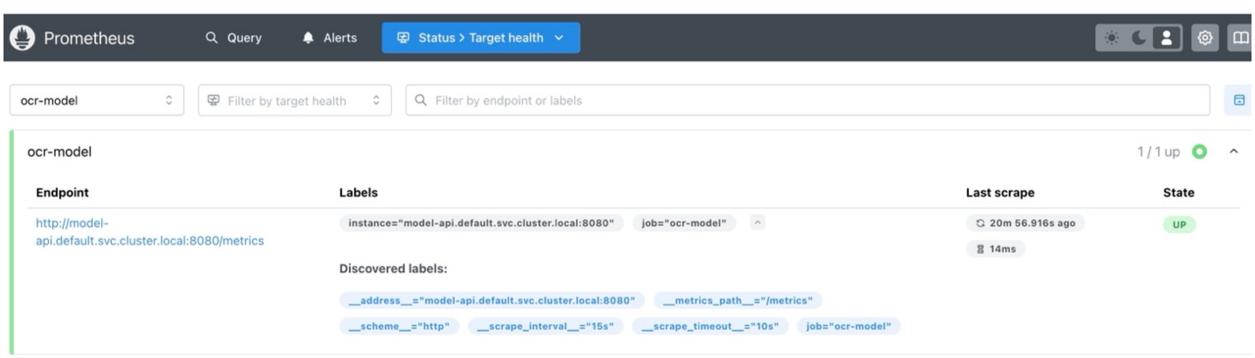

FIGURE 21 PROMETHIUS SCRAPE MATRIX

9.Grafana Integration

For visualization, Grafana was used to create dashboards that query Prometheus as a data source. The following steps were taken to integrate Grafana with Prometheus:

1.Grafana Installation

a.Add the official Grafana Helm chart repository

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

```
helm repo update
```

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

b.Installs Grafana from the Grafana Helm repository.

Uses the monitoring namespace (creates it if it doesn't exist).

c.Check if Grafana is running.

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

d.To access Grafana inside Minikube, you need to forward the service port.

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

e.Now, you can access Grafana at:

http://localhost:3000

f.Grafana's default admin username is admin. To get the default password

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

2. **Add Prometheus as a Data Source in Grafana**:

   - Open Grafana and navigate to the "Configuration" menu.
   - Under **Data Sources**, select **Prometheus** and enter the Prometheus URL: http://prometheus-server.monitoring.svc.cluster.local:80

3. **Verify the Integration**:
   - After saving the data source configuration, test the connection to ensure Grafana can communicate with Prometheus.
   - Create a dashboard and verify that metrics from the model-api service are being queried successfully.

Grafana Dashboards

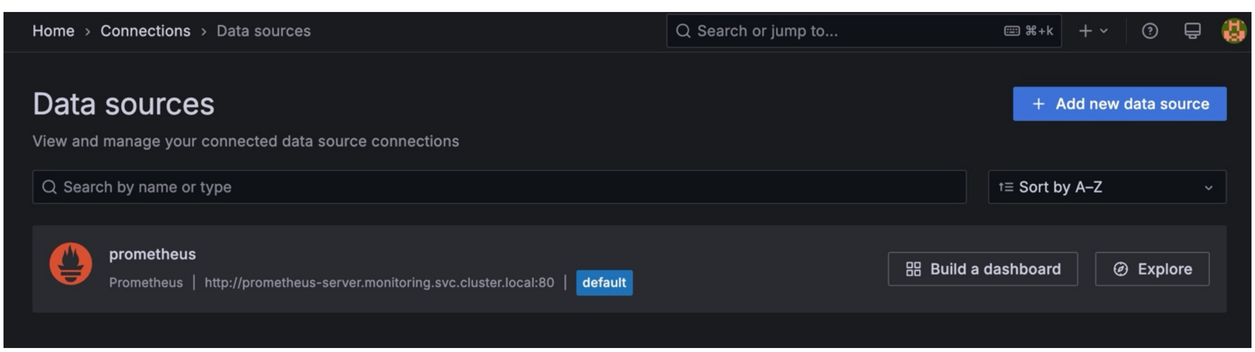

FIGURE 22- PROMETHIUS DATASOURCE

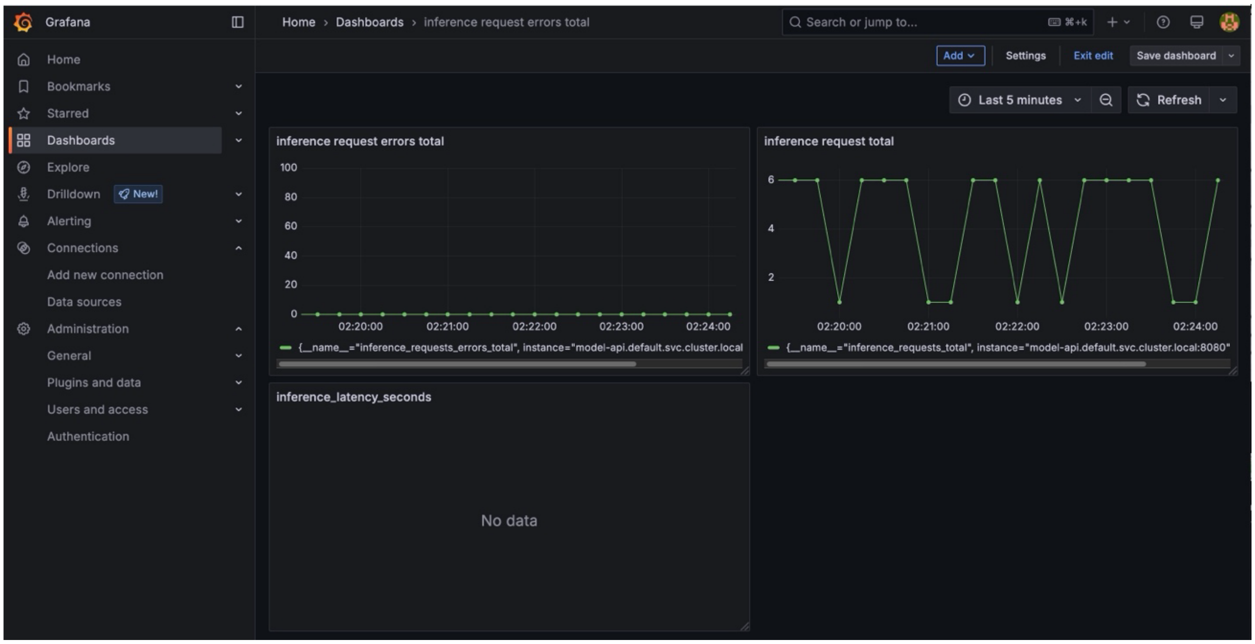

FIGURE 23- DASHBOARDS