MLOps Experimental Learning Assignment:
End-to-End ML Model Development, CI/CD, and Production Deployment Experimental Learning

Assignment 1
Group 90

1. Introduction

Machine Learning Operations (MLOps) is a discipline that focuses on the **reliable, scalable, and reproducible deployment of machine learning systems in production environments**. While traditional machine learning projects often emphasize model accuracy, real-world applications require additional considerations such as automation, version control, continuous integration, deployment pipelines, and monitoring. This assignment is designed to simulate an end-to-end MLOps workflow that closely mirrors industry practices.

In this project, an end-to-end machine learning solution is developed to predict the **risk of heart disease** using patient health data from the UCI Heart Disease dataset. The dataset contains clinical attributes such as age, sex, blood pressure, cholesterol levels, and electrocardiographic measurements, along with a target variable indicating the presence or absence of heart disease. The objective is to build a robust binary classification model and deploy it as a production-ready, cloud-deployable API.

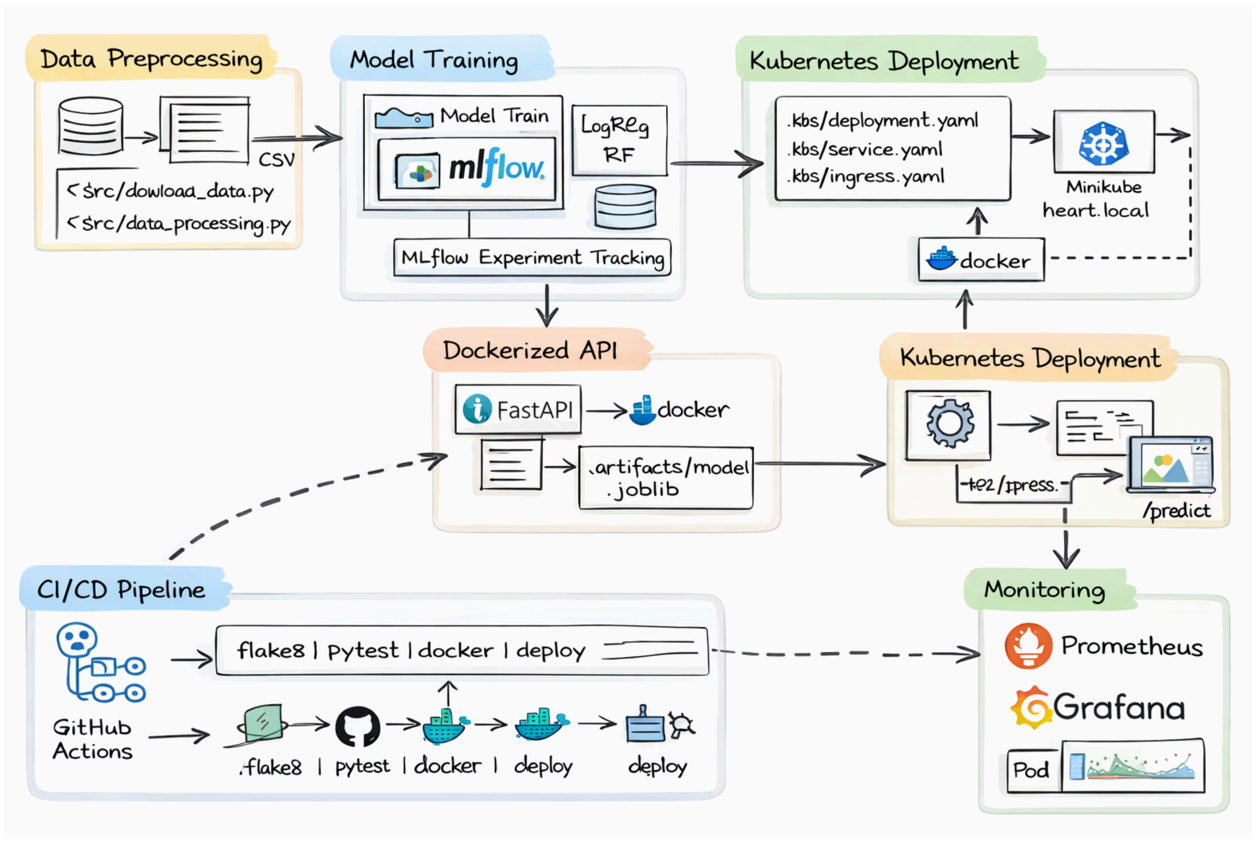
Beyond model development, this assignment emphasizes the **complete machine learning lifecycle**, including data acquisition, exploratory data analysis, feature engineering, model training, experiment tracking, automated testing, containerization, and deployment using Kubernetes. Modern MLOps tools such as **MLflow** are used for experiment tracking, **GitHub Actions** for continuous integration, **Docker** for containerization, and **Kubernetes** for orchestration and deployment.

The final system exposes a RESTful API using **FastAPI**, enabling real-time inference through a /predict endpoint that returns both the prediction and its associated confidence score. The solution is designed to be fully reproducible, automated, and production-oriented, satisfying key MLOps principles such as traceability, portability, and reliability.

This report documents each stage of the pipeline in detail, supported by code snippets, experiment results, screenshots, and deployment evidence. The complete source code, container configurations, CI/CD workflows, and deployment manifests are made available through a version-controlled GitHub repository, along with an end-to-end demonstration video and deployment access instructions.

2. Project Architecture

The architecture diagram given below illustrates the end-to-end architecture of the MLOps pipeline implemented for heart disease prediction. The system begins with data preprocessing scripts that prepare raw data into usable format. This data is then used for model training using MLflow to track experiments. The best-performing model is serialized and served via a FastAPI-based Docker container. The containerized API is deployed to Kubernetes using manifests. CI/CD is implemented with GitHub Actions to automate linting, testing, and deployment. Prometheus and Grafana are used for monitoring the deployed service.



3. Setup & Installation Instructions

This section describes the environment setup and installation steps required to reproduce the complete MLOps pipeline. The same setup was validated both on a **local development machine** and on an **AWS (Osha) instance** to ensure consistency between development and deployment environments.

3.1 Prerequisites

The following prerequisites are required:

- Operating System:
  - Windows 10/11 (local development)
  - Linux-based AWS EC2 instance (Ubuntu)
- Python version: **3.10**

- Conda (Miniconda or Anaconda)
- Git
- Docker (with WSL2 backend on Windows)
- Internet access for dependency installation

3.2 Local Development Environment Setup (Laptop)

3.2.1 Create Conda Environment

A dedicated Conda environment was created to isolate dependencies and ensure reproducibility.

```
conda create -n mlops-a1-group90 python=3.10 -y
conda activate mlops-a1-group90
```

3.2.2 Project Setup

The project repository was cloned and the required directory structure was created.

```
git clone
https://github.com/ranga-srinivasan/mlops-assignment1-group90.git
cd mlops-assignment1-group90
```

3.2.3 Install Python Dependencies

All required Python libraries are listed in requirements.txt [Appendix A1].

```
pip install -r requirements.txt
```

3.2.4 Run Unit Tests

Unit tests were executed to validate data processing and API schemas.

```
pytest -q
```

3.2.5 Model Training and Experiment Tracking

The model training script was executed locally, and experiments were tracked using MLflow.

```
mlflow ui --host 127.0.0.1 --port 5000
python -m src.train
```

The MLflow UI was accessed at: http://127.0.0.1:5000

3.2.6 Run FastAPI Service Locally

The trained model was served locally using FastAPI.

```
uvicorn api.app:app --host 0.0.0.0 --port 8000
```

The interactive API documentation was accessed at: http://127.0.0.1:8000/docs

3.2.7 Docker Installation and Validation (Local)

Docker Desktop with WSL2 backend was used for containerization.

Docker installation was verified using: **`docker --version`**

3.2.8 Build and Run Docker Container (Local)

The FastAPI application was containerized using Docker.

```
docker build -t heart-api:latest .
docker run -p 8000:8000 heart-api:latest
```

The containerized API was tested using the Swagger UI at: http://127.0.0.1:8000/docs

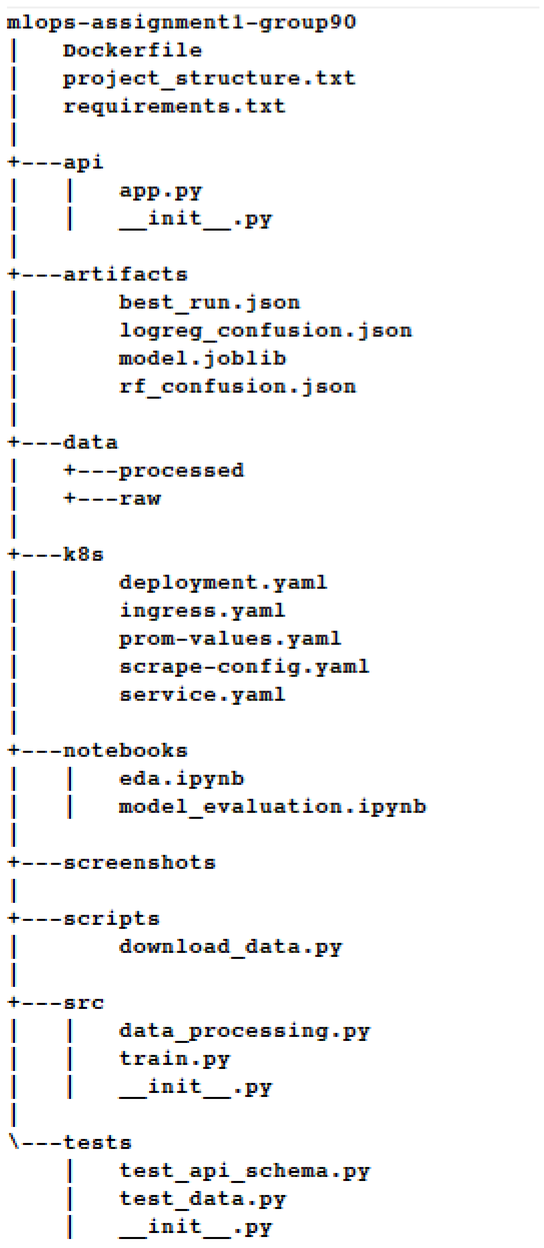
3.3 AWS (Osha) Production Environment Setup

To validate production deployment, the same setup was replicated on an AWS EC2 instance provided through Osha.

3.3.1 Project Setup

Clone the project repository: `git clone https://github.com/ranga-srinivasan/mlops-assignment1-group90.git`

Project structure is shown in the screenshot below:



```
mlops-assignment1-group90
|   Dockerfile
|   project_structure.txt
|   requirements.txt
|
+---api
|   |   app.py
|   |   __init__.py
|
+---artifacts
|       best_run.json
|       logreg_confusion.json
|       model.joblib
|       rf_confusion.json
|
+---data
|   +---processed
|   +---raw
|
+---k8s
|       deployment.yaml
|       ingress.yaml
|       prom-values.yaml
|       scrape-config.yaml
|       service.yaml
|
+---notebooks
|   |   eda.ipynb
|   |   model_evaluation.ipynb
|
+---screenshots
|
+---scripts
|       download_data.py
|
+---src
|   |   data_processing.py
|   |   train.py
|   |   __init__.py
|
\---tests
    |   test_api_schema.py
    |   test_data.py
    |   __init__.py
```

```
cd mlops-assignment1-group90
```

3.3.2 Run Unit Tests

Run the unit tests for sanity check: `pytest -q`

3.3.3 Model Training

Train the model: `python -m src.train`

3.3.4 Docker Installation and Validation

Docker is already available on OSHA AWS. Hence there is no need to install it. Docker installation was verified using: **`docker --version`**

3.3.5 Build and Run Docker Container

The FastAPI application was containerized using Docker.

```
docker build -t heart-api:latest .
docker run -p 8000:8000 heart-api:latest
```

The containerized API can be tested using the Swagger UI at: http://127.0.0.1:8000/docs

3.3.6 Install Kubernetes & minikube

Install Kubernetes:

```
sudo curl -LO
https://storage.googleapis.com/kubernetes-release/release/`curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt`
/bin/linux/amd64/kubectl

sudo chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

Install minikube:

```
sudo curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-linu
x-amd64

sudo chmod +x minikube-linux-amd64
sudo mv minikube-linux-amd64 /usr/local/bin/minikube
```

3.3.7 Kubernetes Deployment Steps

1. Start minikube: `minikube start --driver=docker`

2. Point docker to minikube (for local builds): `eval $(minikube -p minikube docker-env)`
3. Build docker image inside minikube: `docker build -t heart-api:latest .`
4. Apply Kubernetes manifests:

```
kubectl apply -f k8s/deployment.yaml
kubectl apply -f k8s/service.yaml
kubectl apply -f k8s/ingress.yaml
```

5. Enable minikube addons: minikube addons enable ingress

4. Data Acquisition & Exploratory Data Analysis (EDA)

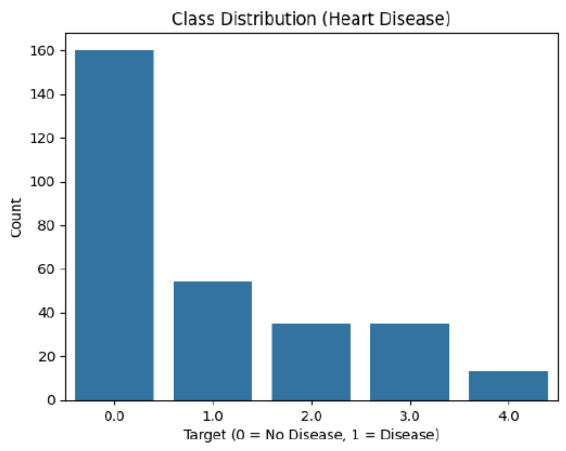
The dataset used is the UCI Heart Disease Dataset, included as a zip archive in **data/raw/heart+disease.zip**. A preprocessing script handles missing values, parses numeric fields, and transforms the target column into a binary label.

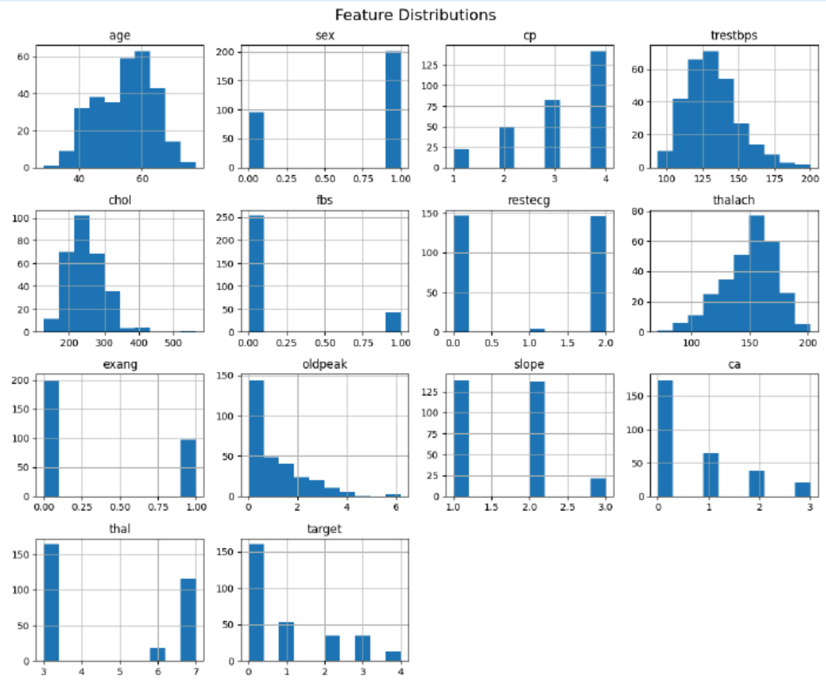
Missing values denoted by '?' are replaced with NaN, and such rows are dropped. All features are converted to float for modeling. The cleaned dataset is used for training and EDA.

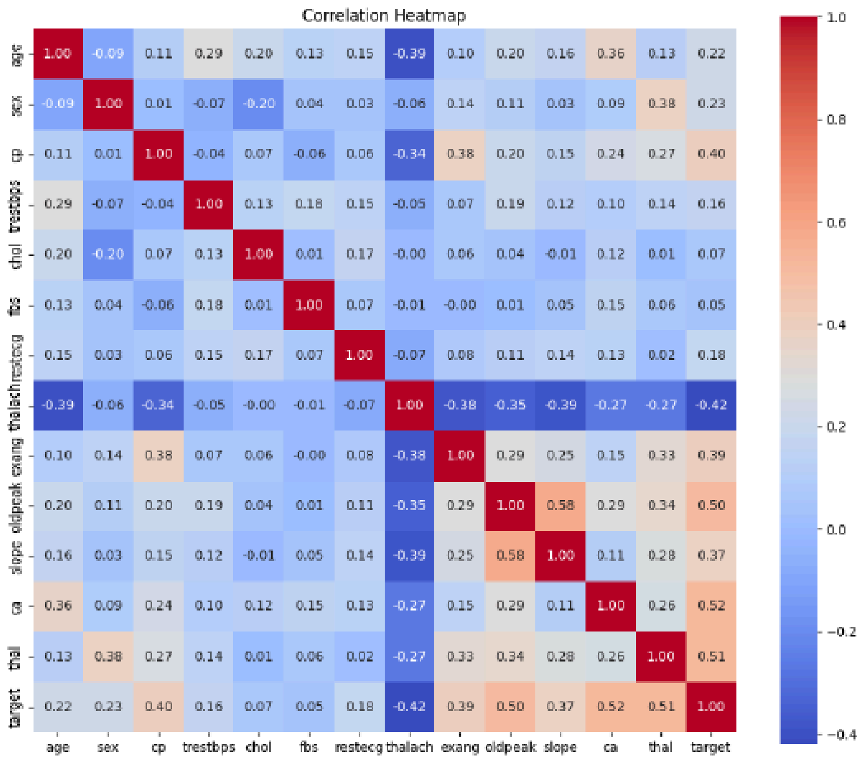
EDA was conducted in a Jupyter notebook (notebooks/eda.ipynb) and includes:

- Histograms of numeric features
- Correlation heatmap
- Class balance plot

These visualizations helped identify key feature distributions, collinearity, and confirmed moderate class imbalance.







	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
age	1.00	-0.09	0.11	0.29	0.20	0.13	0.15	-0.39	0.10	0.20	0.16	0.36	0.13	0.22
sex	-0.09	1.00	0.01	-0.07	-0.20	0.04	0.03	-0.06	0.14	0.11	0.03	0.09	0.38	0.23
cp	0.11	0.01	1.00	-0.04	0.07	-0.06	0.06	-0.34	0.38	0.20	0.15	0.24	0.27	0.40
trestbps	0.29	-0.07	-0.04	1.00	0.13	0.18	0.15	-0.05	0.07	0.19	0.12	0.10	0.14	0.16
chol	0.20	-0.20	0.07	0.13	1.00	0.01	0.17	-0.00	0.06	0.04	-0.01	0.12	0.01	0.07
fbs	0.13	0.04	-0.06	0.18	0.01	1.00	0.07	-0.01	-0.00	0.01	0.05	0.15	0.06	0.05
restecg	0.15	0.03	0.06	0.15	0.17	0.07	1.00	-0.07	0.08	0.11	0.14	0.13	0.02	0.18
thalach	-0.39	-0.06	-0.34	-0.05	-0.00	-0.01	-0.07	1.00	-0.38	-0.35	-0.39	-0.27	-0.27	-0.42
exang	0.10	0.14	0.38	0.07	0.06	-0.00	0.08	-0.38	1.00	0.29	0.25	0.15	0.33	0.39
oldpeak	0.20	0.11	0.20	0.19	0.04	0.01	0.11	-0.35	0.29	1.00	0.58	0.29	0.34	0.50
slope	0.16	0.03	0.15	0.12	-0.01	0.05	0.14	-0.39	0.25	0.58	1.00	0.11	0.28	0.37
ca	0.36	0.09	0.24	0.10	0.12	0.15	0.13	-0.27	0.15	0.29	0.11	1.00	0.26	0.52
thal	0.13	0.38	0.27	0.14	0.01	0.06	0.02	-0.27	0.33	0.34	0.28	0.26	1.00	0.51
target	0.22	0.23	0.40	0.16	0.07	0.05	0.18	-0.42	0.39	0.50	0.37	0.52	0.51	1.00

5. Feature Engineering & Model Development

Feature preparation was done using a scikit-learn pipeline with StandardScaler. Since the dataset contains only numeric or ordinal fields, no additional encoding was required.

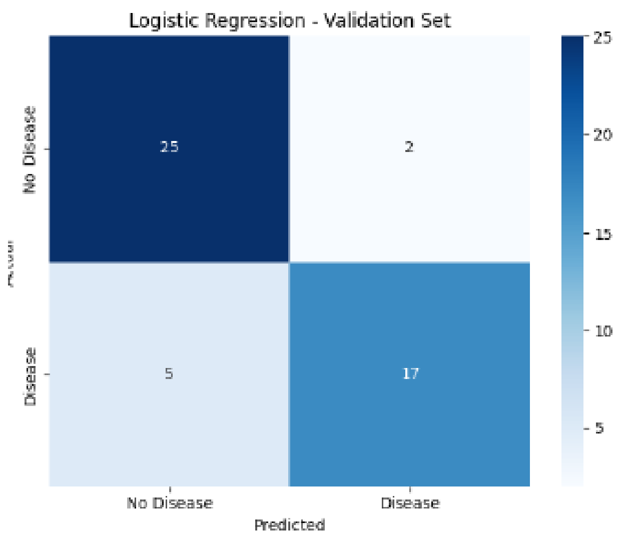
Two models were trained and compared:

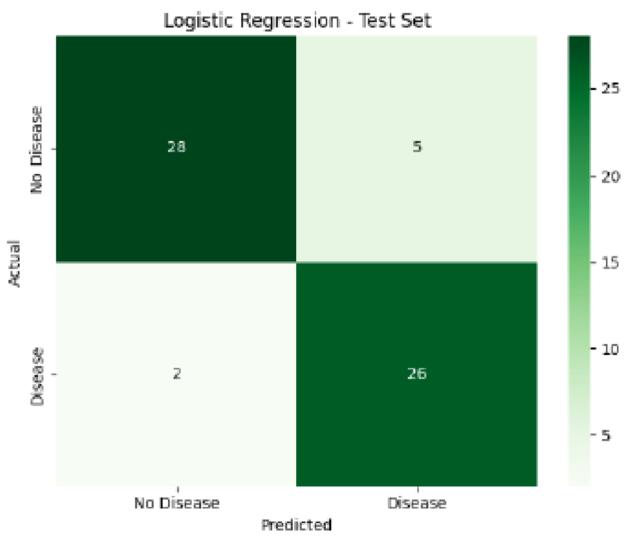
- Logistic Regression
- Random Forest

Evaluation was done using 5-fold stratified cross-validation. Metrics computed included:

- Accuracy
- Precision
- Recall
- ROC-AUC

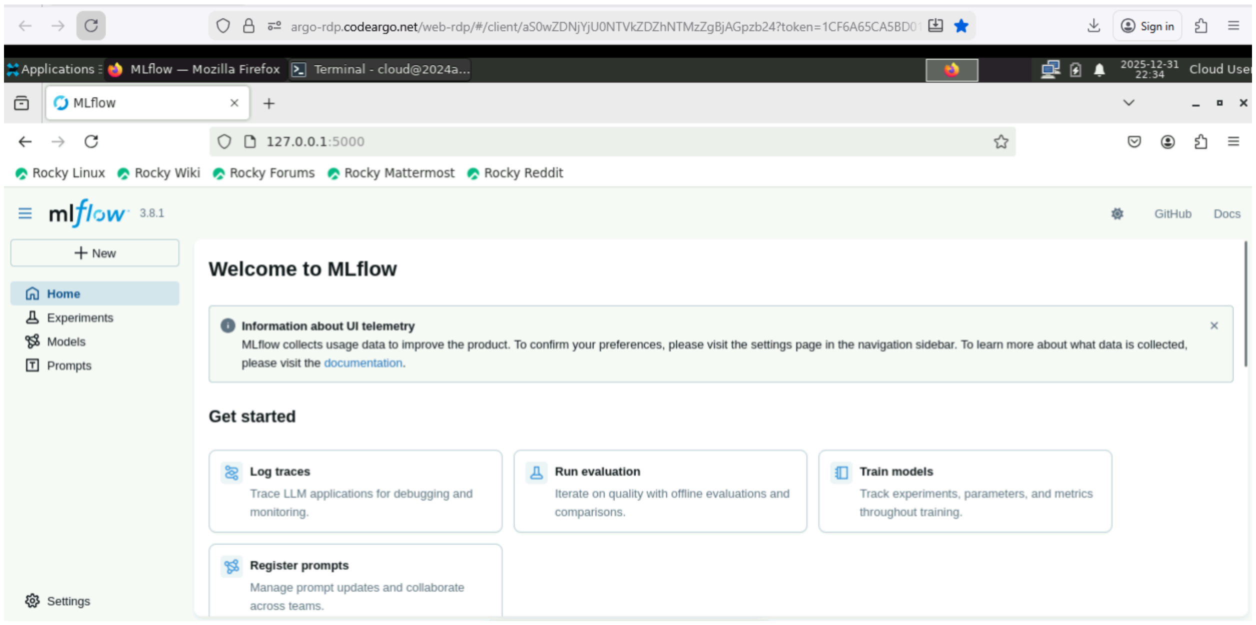
The best model was selected based on validation ROC-AUC and persisted for inference. Metrics and confusion matrices were logged using MLflow. The trained pipeline is saved in `artifacts/model.joblib`.





6. Experiment Tracking Summary

MLflow was used for end-to-end experiment tracking as shown in the screenshot below:



Welcome to MLflow

Information about UI telemetry
MLflow collects usage data to improve the product. To confirm your preferences, please visit the settings page in the navigation sidebar. To learn more about what data is collected, please visit the documentation.

Get started

Log traces
Trace LLM applications for debugging and monitoring.

Run evaluation
Iterate on quality with offline evaluations and comparisons.

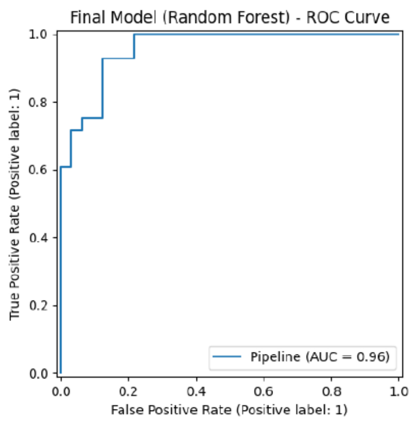
Train models
Track experiments, parameters, and metrics throughout training.

Register prompts
Manage prompt updates and collaborate across teams.

Each run logs:

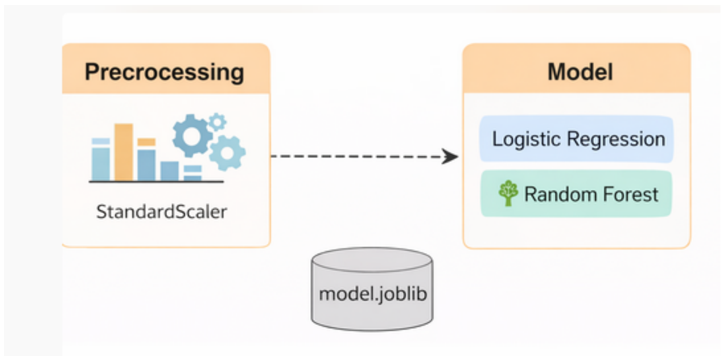
- **Parameters:** model type, scaling, and CV settings
- **Metrics:** accuracy, precision, recall, ROC-AUC
- **Artifacts:**
    - Trained model (model.joblib)
    - Confusion matrix JSON files
    - Best run metadata (best_run.json)

These are organized under artifacts/ and tracked via the MLflow UI for reproducibility and comparison. The best model was selected by ROC-AUC and promoted to production.



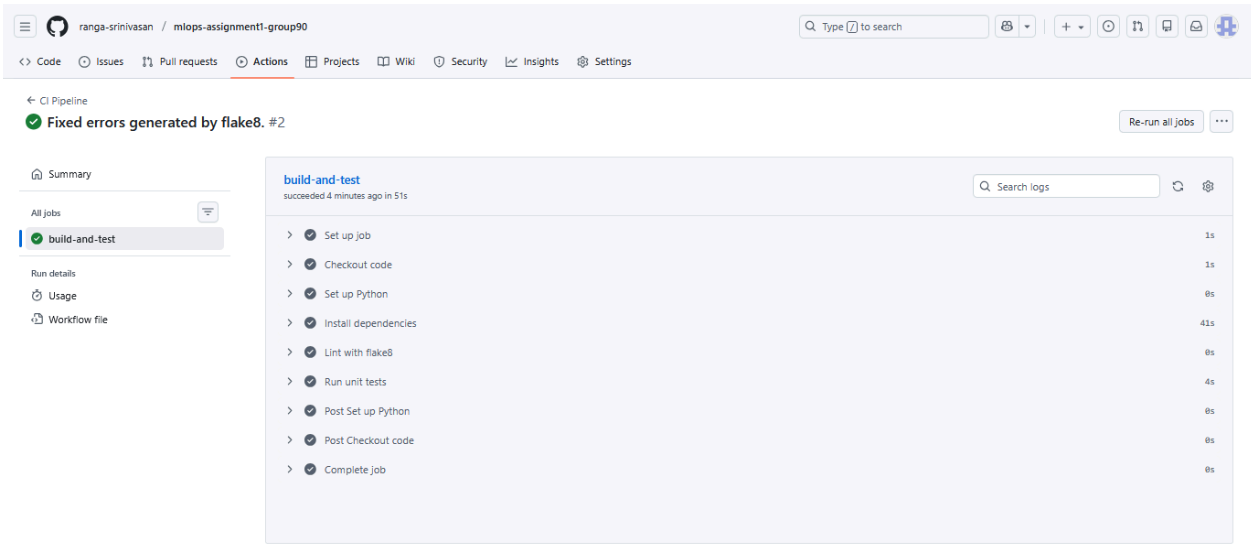
7. Model Packaging & Reproducibility

- The final model (including preprocessing pipeline) was saved as a serialized joblib file: `artifacts/model.joblib`.
- This model is loaded by the FastAPI app during container startup to ensure consistent predictions.
- All package dependencies are specified in requirements.txt, which is used for both local development and Docker-based deployment.
- The use of a pipeline from scikit-learn ensures that scaling and feature transformation are identical between training and inference, guaranteeing full reproducibility.
- The trained models and metrics are also logged in MLflow for traceability.



8. CI/CD Pipeline & Automated Testing

- Unit tests were written using pytest and are located in the tests/ folder.
- The tests validate data loading, preprocessing assumptions, and FastAPI schema validation.
- A CI pipeline was configured using GitHub Actions (.github/workflows/main.yml) to automate:
  - Linting via flake8
  - Unit testing with pytest
- Each run produces logs visible in the GitHub Actions dashboard for traceability and debugging.



Fixed errors generated by flake8. #2

build-and-test

build-and-test
succeeded 4 minutes ago in 51s

```
Set up job              1s
Checkout code           1s
Set up Python           0s
Install dependencies    41s
Lint with flake8        0s
Run unit tests          4s
Post Set up Python      0s
Post Checkout code      0s
Complete job            0s
```

9. Model Containerization

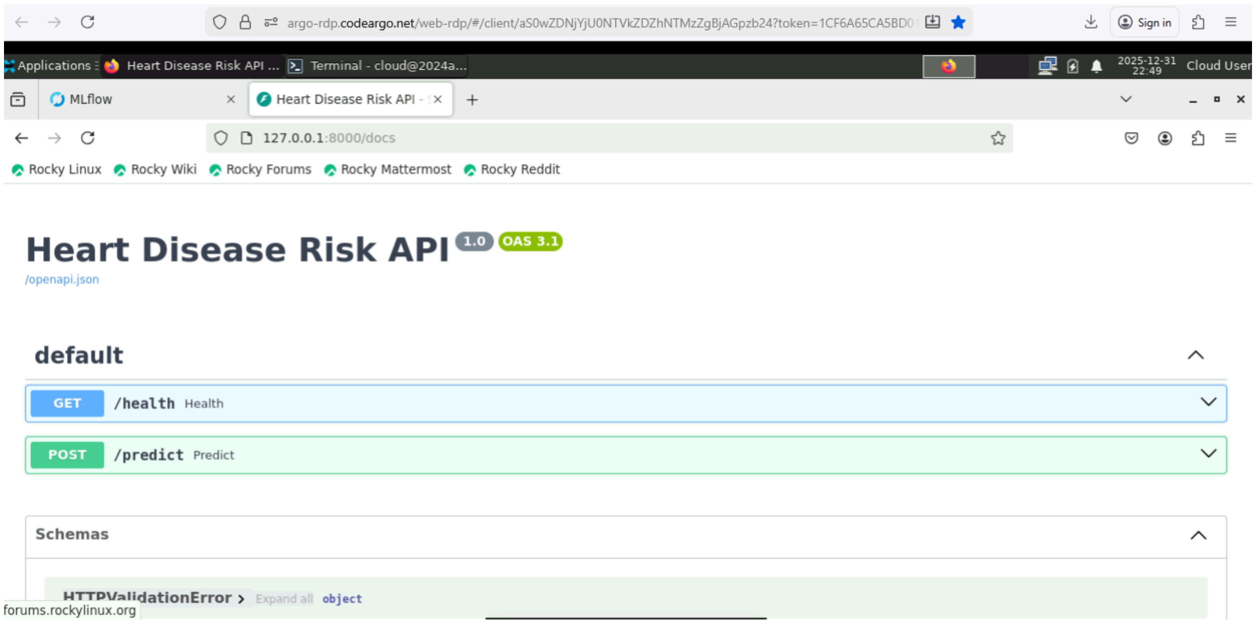
- The model-serving API was implemented using FastAPI.
- A Dockerfile was created to containerize the API, which installs dependencies and serves the app via `uvicorn`.
- The container exposes the `/predict` endpoint, accepts structured JSON input, and returns a classification result along with its probability.
- Sample requests were successfully executed using curl and via Swagger UI.
- The container was built and tested locally on Docker and later deployed to Minikube on the OSHA AWS instance.

```
(mlops-a1-group90) cloud@2024aa05119:~/mlops-assignment1-group90$ docker build -t
heart-api:latest .
[+] Building 94.0s (10/10) FINISHED
docker:default
 => [internal] load build definition from Dockerfile
0.0s
 => => transferring dockerfile: 479B
0.0s
 => [internal] load metadata for docker.io/library/python:3.10-slim
2.6s
 => [internal] load .dockerignore
0.0s
 => => transferring context: 2B
0.0s
 => [1/5] FROM
docker.io/library/python:3.10-slim@sha256:fdf096ff53241fba39e55c557c703bdd8fef55535a
bc2454e5a944ed1d53fb2a        5.2s
 => => resolve
```

```
docker.io/library/python:3.10-slim@sha256:fdf096ff53241fba39e55c557c703bdd8fef55535a
bc2454e5a944ed1d53fb2a        0.0s
 => => sha256:05be60a538e21a03887c1b1ecbc37e18a204d7abd2ff9d18ec9e95a868d83364
1.75kB / 1.75kB                                  0.0s
 => => sha256:ce19342c5d49287e941ab558824eb6b0a4244066b18b5a1a9024b6c0f2f818a8
5.48kB / 5.48kB                                  0.0s
 => => sha256:02d7611c4eae219af91448a4720bdba036575d3bc0356cfe12774af85daa6aff
29.78MB / 29.78MB                                1.2s
 => => sha256:8715e552fa1374bdde269437d9a1c607c817289c2ebbceb9ed9ab1aa9ca86763
1.29MB / 1.29MB                                  1.0s
 => => sha256:9c27bc7ba63d1ac690daefc68302197d3ab9a91fc5c0e19f447cd57eda92d87c
13.82MB / 13.82MB                                0.8s
 => => sha256:fdf096ff53241fba39e55c557c703bdd8fef55535abc2454e5a944ed1d53fb2a
10.37kB / 10.37kB                                0.0s
 => => sha256:7da4424a113245eb185ea22f2512eceb36f80ca1d0547c64b117f28495d3c3e5 250B
/ 250B                                      1.4s
 => => extracting
sha256:02d7611c4eae219af91448a4720bdba036575d3bc0356cfe12774af85daa6aff
1.9s
 => => extracting
sha256:8715e552fa1374bdde269437d9a1c607c817289c2ebbceb9ed9ab1aa9ca86763
0.2s
 => => extracting
sha256:9c27bc7ba63d1ac690daefc68302197d3ab9a91fc5c0e19f447cd57eda92d87c
1.5s
 => => extracting
sha256:7da4424a113245eb185ea22f2512eceb36f80ca1d0547c64b117f28495d3c3e5
0.0s
 => [internal] load build context
0.1s
 => => transferring context: 3.38MB
0.1s
 => [2/5] WORKDIR /app
0.3s
 => [3/5] COPY requirements.txt /app/requirements.txt
0.0s
 => [4/5] RUN pip install --no-cache-dir -r /app/requirements.txt
80.1s
 => [5/5] COPY . /app
0.1s
 => exporting to image
5.4s
 => => exporting layers
5.4s
 => => writing image
sha256:f258dcb25903b50babec1d99938fe89a2e4ff7b807b39a100a4e7642d50166b8
0.0s
 => => naming to docker.io/library/heart-api:latest

(mlops-a1-group90) cloud@2024aa05119:~/mlops-assignment1-group90$ docker run -p
8000:8000 heart-api:latest
INFO:     Started server process [1]
INFO:     Waiting for application startup.
[2025-12-31 17:16:49,629] INFO - Loaded model from: /app/artifacts/model.joblib
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     172.17.0.1:48422 - "GET /docs HTTP/1.1" 200 OK
INFO:     172.17.0.1:48422 - "GET /openapi.json HTTP/1.1" 200 OK
```

The containerized API was tested using the Swagger UI at http://127.0.0.1:8000/docs as shown in the screenshot below:



10. Production Deployment

- The Dockerized API was deployed to a local Kubernetes cluster using Minikube on the OSHA AWS instance.
- Deployment was managed with Kubernetes manifests (deployment.yaml, service.yaml, and ingress.yaml) placed in the k8s/ folder.
- The service was exposed via Ingress with the host heart.local, mapped to the Minikube IP.
- /predict and /health endpoints were verified using curl and browser access. Application logs confirm inference activity.
- Screenshots of pod status, service availability, and response logs are attached below.

```
mlops-a1-group90) cloud@2024aa05119:~/mlops-assignment1-group90$ kubectl get
pods
NAME                        READY   STATUS             RESTARTS   AGE
heart-api-d6f965b6b-5bfr5   0/1     ImagePullBackOff   0          2m1s

(mlops-a1-group90) cloud@2024aa05119:~/mlops-assignment1-group90$ kubectl get
svc
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
heart-api-svc   ClusterIP   10.99.21.103   <none>        80/TCP    115s
kubernetes      ClusterIP   10.96.0.1      <none>        443/TCP   5h58m

(mlops-a1-group90) cloud@2024aa05119:~/mlops-assignment1-group90$ kubectl get
ingress
```

```
NAME                CLASS   HOSTS         ADDRESS        PORTS   AGE
heart-api-ingress   nginx   heart.local   192.168.49.2   80      115s
```

```
(mlops-a1-group90) cloud@2024aa05119:~/mlops-assignment1-group90$ minikube ip
192.168.49.2

(mlops-a1-group90) cloud@2024aa05119:~/mlops-assignment1-group90$ curl -i -H
"Host: heart.local" http://$(minikube ip)/health
HTTP/1.1 200 OK
Date: Wed, 31 Dec 2025 18:37:05 GMT
Content-Type: application/json
Content-Length: 15
Connection: keep-alive

{"status":"ok"}

(mlops-a1-group90) cloud@2024aa05119:/mlops-assignment1-group90$ curl -i -X
POST http://$(minikube ip)/predict \e ip)/predict \
  -H "Host: heart.local" \
  -H "Content-Type: application/json" \
  --data-binary
'{"age":50,"sex":1,"cp":3,"trestbps":130,"chol":250,"fbs":0,"restecg":0,"thala
ch":150,"exang":0,"oldpeak":1.0,"slope":2,"ca":0,"thal":3}'
HTTP/1.1 200 OK
Date: Wed, 31 Dec 2025 18:37:41 GMT
Content-Type: application/json
Content-Length: 50
Connection: keep-alive

{"prediction":0,"probability":0.22834714773737422}
```

11. Monitoring & Logging

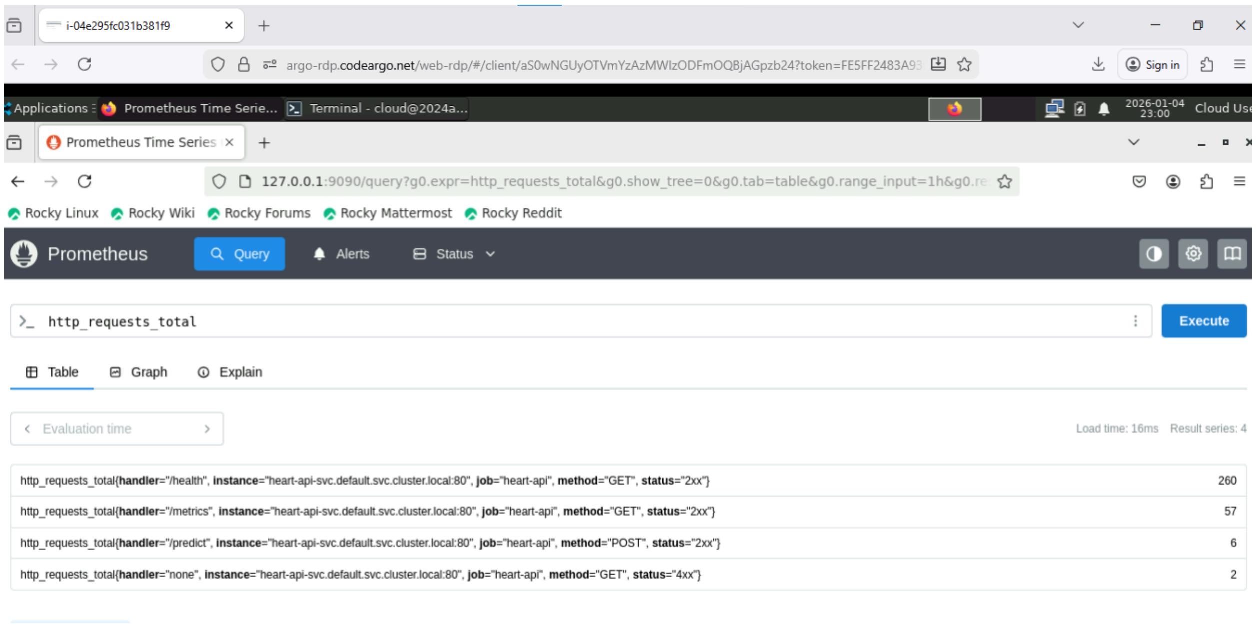
- Logging was implemented using Python's logging module and FastAPI middleware. Each API request is logged with its method, path, status code, and duration.
- The FastAPI application exposes a /metrics endpoint using prometheus_fastapi_instrumentator, which outputs Prometheus-compatible metrics.
- Prometheus and Grafana were deployed on the Minikube cluster using Helm. Prometheus was configured to scrape the FastAPI service.
- While Grafana panels for specific routes (e.g., /predict) were partially functional due to Prometheus relabeling issues, the metrics endpoint and logs confirm monitoring is operational.
- Screenshots of /metrics, API logs, and Grafana dashboards are included.

```
(mlops-a1-group90) cloud@2024aa05119:~/mlops-assignment1-group90$ kubectl logs
```

```
deploy/heart-api
INFO:     Started server process [1]
INFO:     Waiting for application startup.
[2026-01-01 17:18:44,668] INFO - Loaded model from: /app/artifacts/model.joblib
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
INFO:     10.244.0.1:53752 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:33384 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:33394 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:54930 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:35006 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:35022 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:48844 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:53624 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:53632 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:40702 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:42400 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:42414 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:43768 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:32996 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:33012 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:57980 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:58824 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:58834 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:39212 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:40000 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:40010 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:44758 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:41346 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:41354 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:52420 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:51262 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:51276 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:48504 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:33724 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:33732 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:51012 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:48116 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:48118 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:55500 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:42774 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:42782 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:52840 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:36262 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:36278 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:50330 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:51074 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:51090 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:53684 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:42396 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:42398 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:55378 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:44390 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:44398 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:39410 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:53562 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:53572 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:39390 - "GET /health HTTP/1.1" 200 OK
```

```
INFO:     10.244.0.1:42922 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:42926 - "GET /health HTTP/1.1" 200 OK
[2026-01-01 17:24:50,287] INFO - Received request: {'age': 50.0, 'sex': 1.0, 'cp': 
3.0, 'trestbps': 130.0, 'chol': 250.0, 'fbs': 0.0, 'restecg': 0.0, 'thalach': 150.0, 
'exang': 0.0, 'oldpeak': 1.0, 'slope': 2.0, 'ca': 0.0, 'thal': 3.0}
[2026-01-01 17:24:50,299] INFO - Prediction=0, Probability=0.2283
INFO:     10.244.0.11:40774 - "POST /predict HTTP/1.1" 200 OK
INFO:     10.244.0.1:60038 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:60750 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:60758 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:56620 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:48134 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:48142 - "GET /health HTTP/1.1" 200 OK
INFO:     10.244.0.1:50868 - "GET /health HTTP/1.1" 200 OK
```

11.1 Prometheus Dashboard

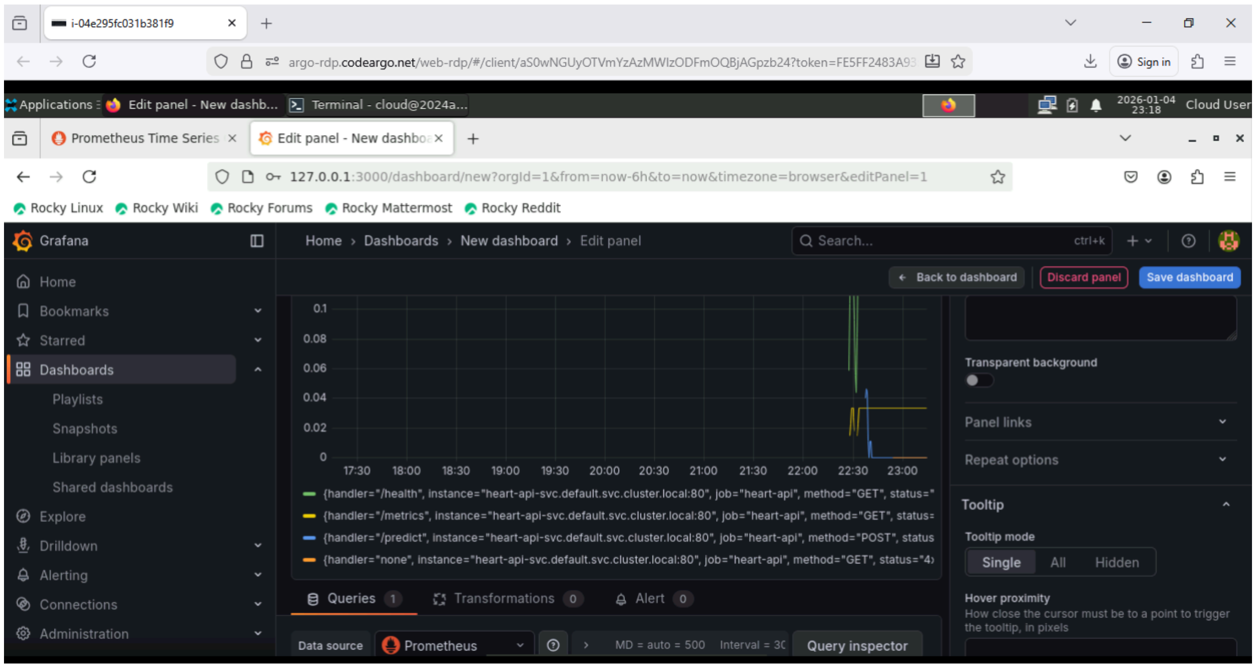


```
http_requests_total
```

Load time: 16ms   Result series: 4

```
http_requests_total{handler="/health", instance="heart-api-svc.default.svc.cluster.local:80", job="heart-api", method="GET", status="2xx"}    260
http_requests_total{handler="/metrics", instance="heart-api-svc.default.svc.cluster.local:80", job="heart-api", method="GET", status="2xx"}    57
http_requests_total{handler="/predict", instance="heart-api-svc.default.svc.cluster.local:80", job="heart-api", method="POST", status="2xx"}    6
http_requests_total{handler="none", instance="heart-api-svc.default.svc.cluster.local:80", job="heart-api", method="GET", status="4xx"}    2
```

11.2 Grafana Dashboard



```
{handler="/health", instance="heart-api-svc.default.svc.cluster.local:80", job="heart-api", method="GET", status="
{handler="/metrics", instance="heart-api-svc.default.svc.cluster.local:80", job="heart-api", method="GET", status=
{handler="/predict", instance="heart-api-svc.default.svc.cluster.local:80", job="heart-api", method="POST", status
{handler="none", instance="heart-api-svc.default.svc.cluster.local:80", job="heart-api", method="GET", status="4›
```

Data source: Prometheus   MD = auto = 500   Interval = 3C

12. Link to Code Repository

This is covered in section 3.2.2 and 3.3.1. The code repository link is given below explicitly for clarity:

`https://github.com/ranga-srinivasan/mlops-assignment1-group90.git`

13. Conclusion

This MLOps project successfully demonstrated the end-to-end lifecycle of a machine learning solution for heart disease risk prediction using the UCI Cleveland dataset. The work followed a modular and reproducible pipeline covering all critical aspects of modern ML engineering.

From data acquisition and EDA to model training, evaluation, and packaging, the project emphasized clarity, traceability, and reproducibility. Experiment tracking with MLflow enabled transparent model comparison and selection, while Docker containerization ensured consistent deployment across environments. The trained pipeline was deployed to a local Kubernetes cluster using Minikube, with an exposed **`/predict`** endpoint to serve real-time inferences. Basic request logging and Prometheus-based metrics were integrated to support observability.

Key steps include:

- A fully automated training and evaluation flow using GitHub Actions.
- Robust experiment tracking with versioned models and artifacts.
- A clean CI/CD pipeline and reproducible deployment via Docker and Kubernetes.
- A REST API built with FastAPI, suitable for production inference.
- Prometheus + Grafana for observability.

The project not only meets the academic objectives of MLOps coursework but also aligns with industry best practices in terms of model reproducibility, automation, and infrastructure readiness. With all code, artifacts, and deployment manifests version-controlled and documented, this serves as a strong baseline for future production ML workflows.