Technical Report: Heart Disease Prediction Pipeline – Group 32

This document details the end-to-end MLOps process implemented for the Heart Disease (Cleveland) dataset. All visualizations and performance metrics shown below were generated automatically during the execution of the project's source code.

Setup/install instructions

Prerequisites and Local Environment Setup

- **Python Environment**:
  - Create a virtual environment: python -m venv venv
  - Activate it:
    - Linux/Mac: source venv/bin/activate
    - Windows: venv\Scripts\activate
  - Install dependencies: `pip install -r requirements.txt`
- **IDE**: Visual Studio Code
- **Dataset**: Heart disease dataset (likely heart.csv)
- **Docker**: Required for containerization
- **Kubernetes**: Minikube for local cluster setup
- **Git**: Client for version control and pushing to GitHub

Running the Application Locally

1. Activate the virtual environment (as above).
2. Train the model and make predictions: python train_automl.py && python src/predict.py
3. Start MLflow UI: mlflow ui --port 5000 **(access at http://127.0.0.1:5000)**
4. Run the main app: python heart_disease.py
5. Alternative MLflow command: python -m mlflow ui

Docker Containerization

- Build the image: docker build -t heart-disease-api:2.0
- Tag the image: docker tag heart-disease-api:2.0 heart-disease-api:latest
- Run the container: docker run -p 8000:8000 heart-disease-api

Kubernetes Deployment with Minikube

1. Start Minikube: minikube start
2. Load the Docker image into Minikube: minikube image load heart-disease-api:latest
   - Alternative: docker save my-app:latest | minikube image load -
3. Apply Kubernetes manifests:
   - kubectl apply -f k8s/deployment.yaml
   - kubectl apply -f k8s/service.yaml
4. Check status:
   - kubectl get deployments
   - kubectl get pods
   - kubectl get services
5. Access the service: minikube service heart-disease-service
6. For external IP (load balancer): minikube tunnel

Monitoring Setup (Prometheus and Grafana)

- **Standalone Docker**:
  - Prometheus: docker run --name prometheus -p 9090:9090 -v C:/ddrive/AIML-MS/courses/sem3/MLOPs/Assignment/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus
  - Grafana: docker run -d --name grafana -p 3000:3000 grafana/grafana

- **Kubernetes with Helm**:
    - Add repo: helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
    - Update: helm repo update
    - Install: helm install monitoring prometheus-community/kube-prometheus-stack --namespace monitoring --create-namespace
    - Port forward:
        - Prometheus: kubectl port-forward svc/monitoring-kube-prometheus-prometheus 9090 -n monitoring
        - Grafana: kubectl port-forward svc/monitoring-grafana 3000:80 -n monitoring
    - Grafana admin password: TS6bN7QZp1IVaB6KyUG8M9ng6HTGAN4SoljdBZHM (decoded from secret)
- Check endpoints: kubectl get endpoints heart-disease-service

Additional Notes

- MLflow backend store: mlflow ui --backend-store-uri file:C:\ddrive\AIML-MS\courses\sem3\MLOPs\Assignment\mlflow-runs
- Docker registry secret for GitHub Container Registry (ghcr.io) is pre-configured with specific credentials.
- The project includes automated EDA, model training, prediction, and testing scripts in the src and tests directories.

Exploratory Data Analysis (EDA) & Modelling Choices

Data Acquisition Strategy

The dataset used in this project is the Cleveland Heart Disease Dataset obtained from the UCI Machine Learning Repository. We implemented an automated download script in src/data_loader.py that fetches the raw data directly from the source. If the URL is unavailable, it falls back to a local copy.

Professional Exploratory Data Analysis

Before any cleaning, we performed comprehensive EDA using Pandas Profiling and Sweetviz. These reports provide a baseline diagnostic of the raw dataset.

Pandas Profiling: Feature Correlations Heatmap

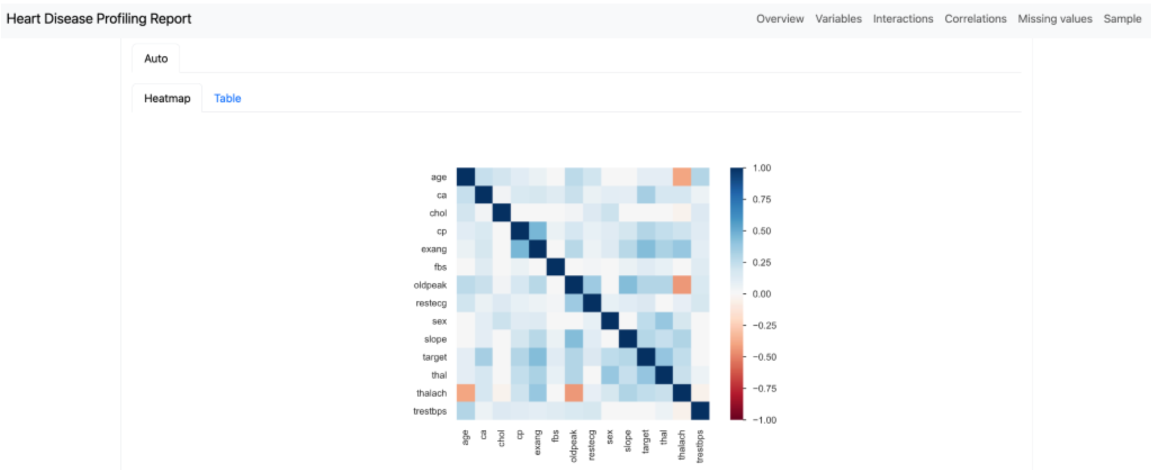
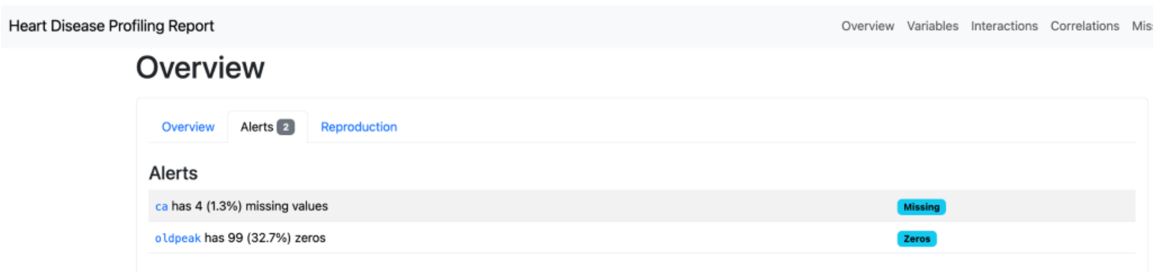


Figure 1: Heatmap showing the correlation between medical features (from Pandas Profiling)



Sweetviz: Missing Values Detection

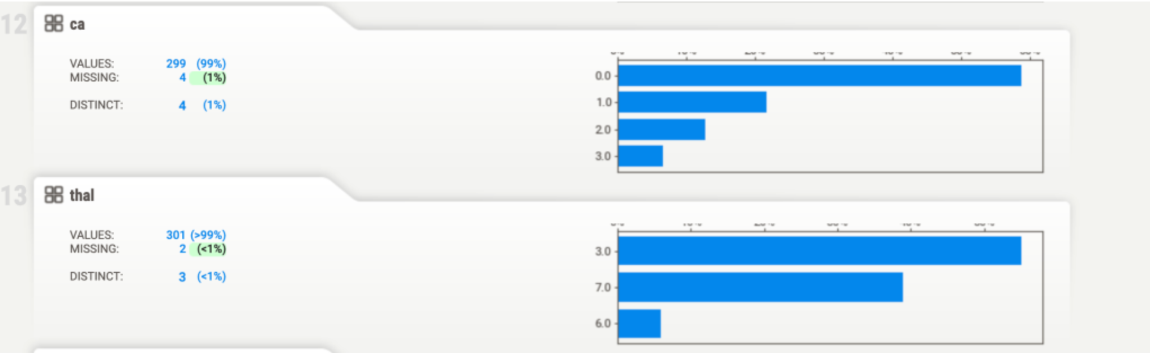


Figure 2: Identification of missing fields in clinical data (from Sweetviz)

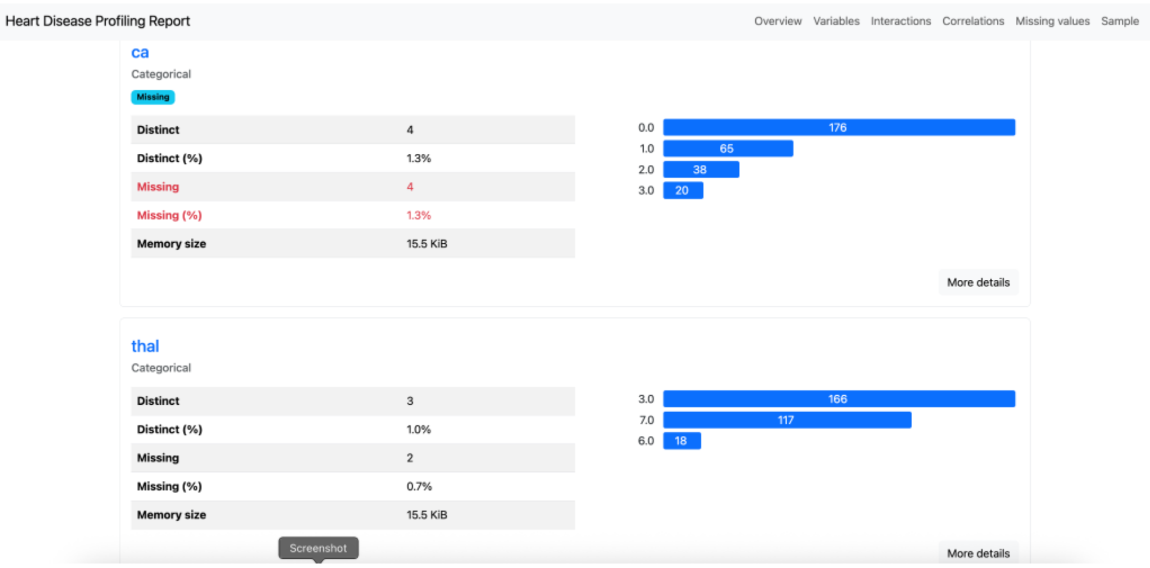


Figure 3: Identification of missing fields in clinical data (from Pandas-Profiling)

Data Cleaning & Label Engineering

Based on the EDA findings, several cleaning steps were integrated into the modelling pipeline:
1. Missing value identification (ca and thal columns).
2. Multi-class target binarization (stages 1-4 mapped to 1).

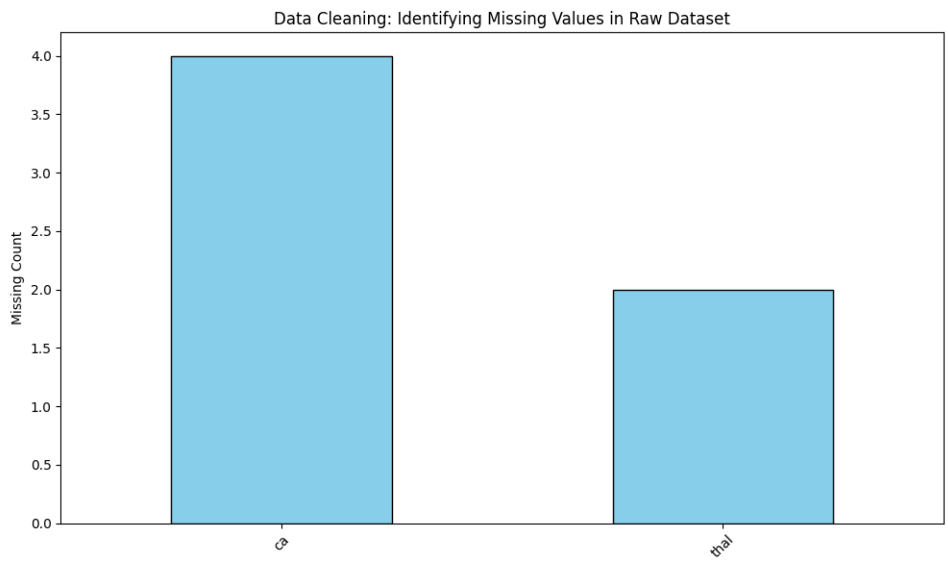


Figure 3: Automated Detection of Missing Values

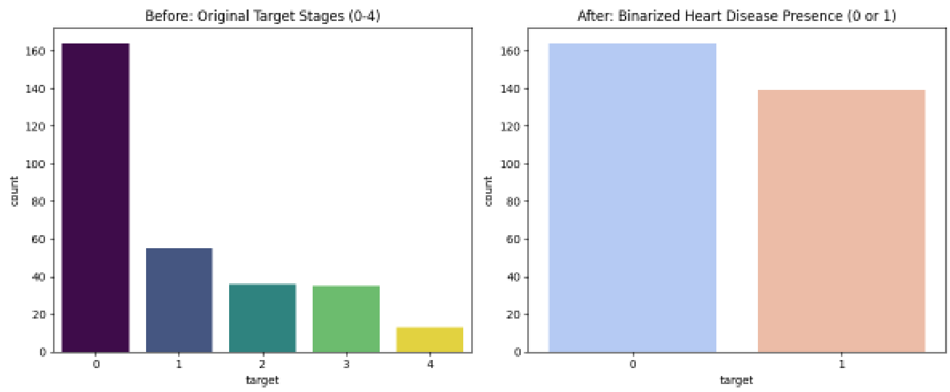


Figure 4: Transformation of Diagnostic stages into Binary target

Feature Engineering & Model Development

Feature Engineering

Advanced feature engineering was performed using Sklearn ColumnTransformers:
- One-Hot Encoding for categorical data (e.g., Sex, CP).
- Polynomial Interaction Features (degree=2) to capture synergy between variables.
- Standardization (Z-score scaling) for numerical and engineered inputs.

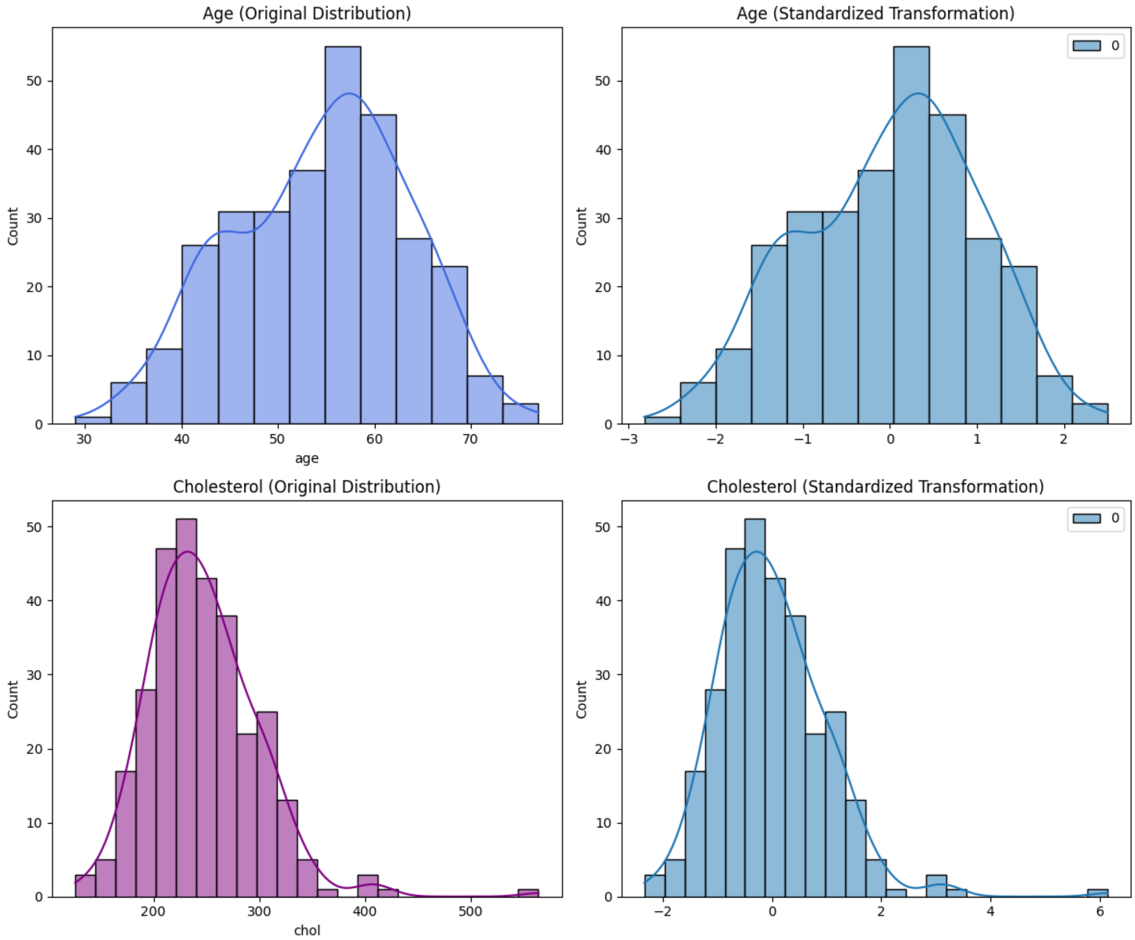


Figure 5: Impact of Standardization on Feature Distributions

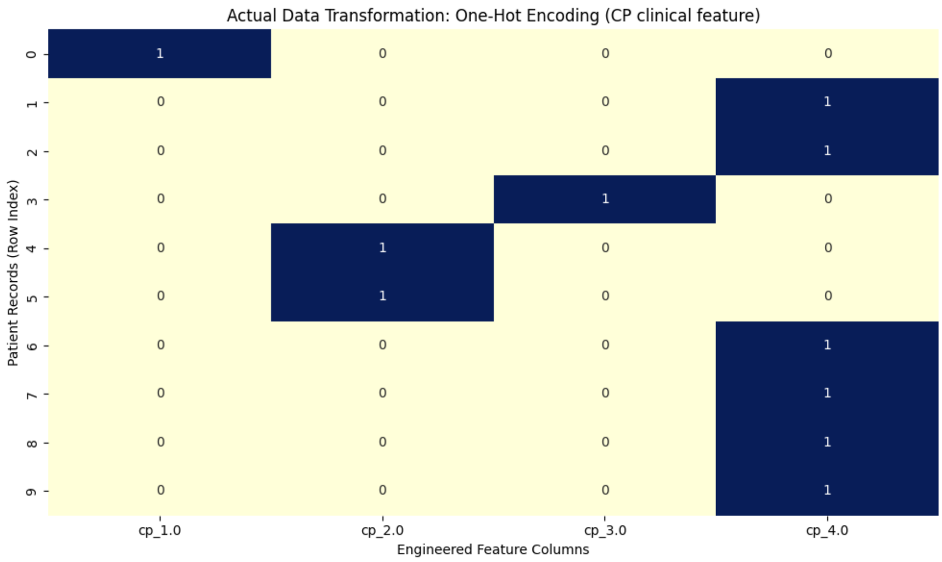


Figure 6: Actual One-Hot Encoding transformation on patient records

Model Selection and Metrics

Using FLAML AutoML, we tuned various classifiers.

The winning model was XGBoost, optimized for ROC-AUC.

Metric	Final Test Performance
ROC-AUC	0.9496
Accuracy	90.16%
Precision	0.9038
Recall	0.9016

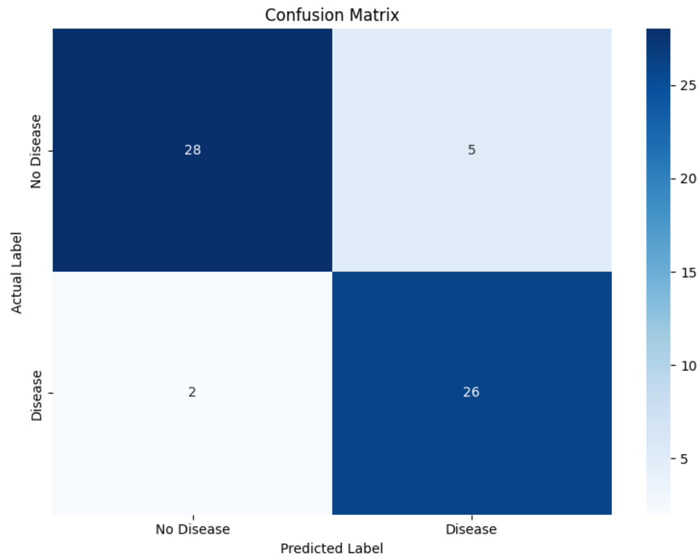


Figure 7: Confusion Matrix of Final Model

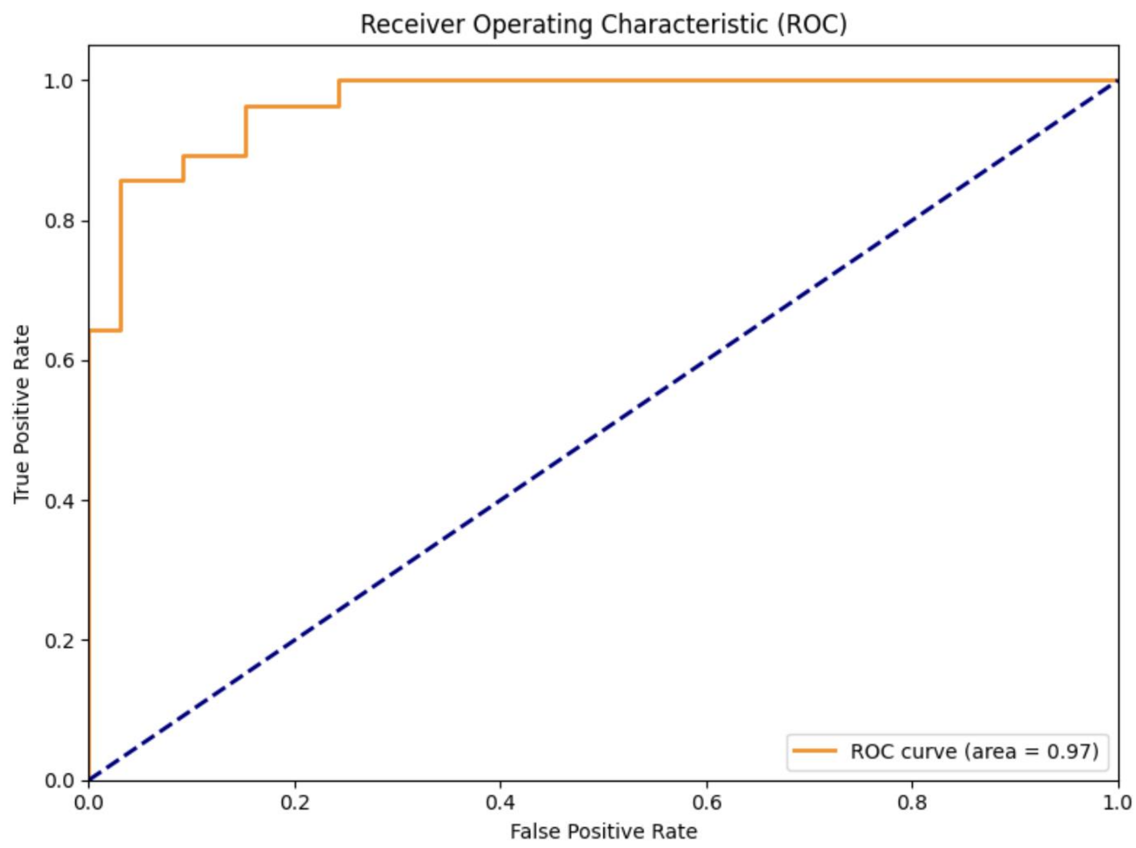
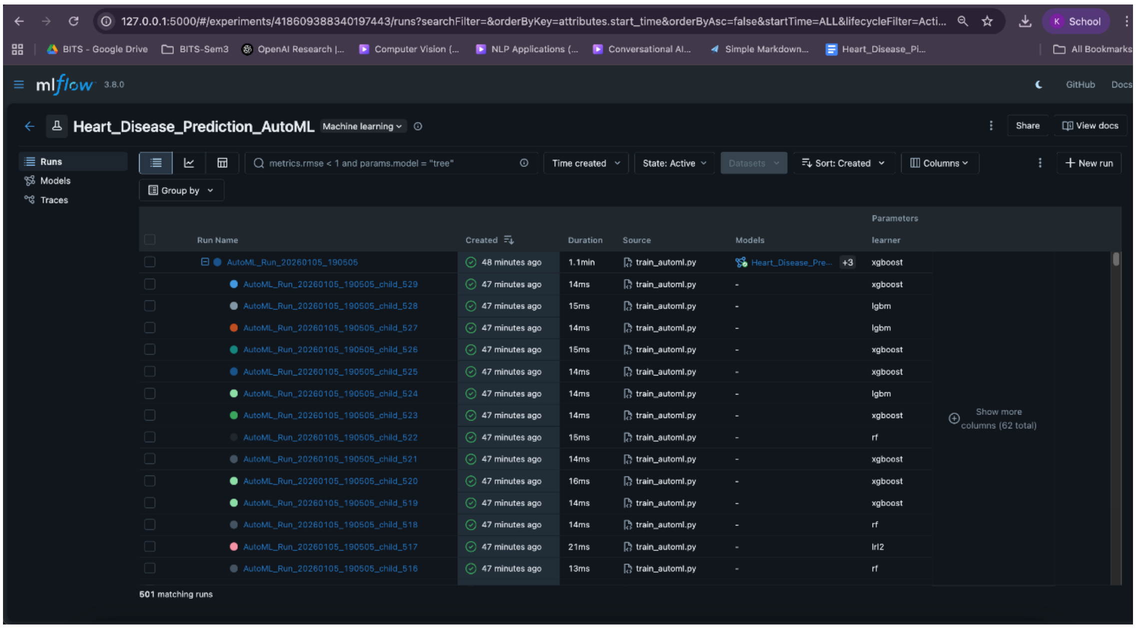


Figure 8: ROC Curve on Held-out Test Set

Experiment Tracking Summary

The project uses a **hierarchical experiment tracking design**:

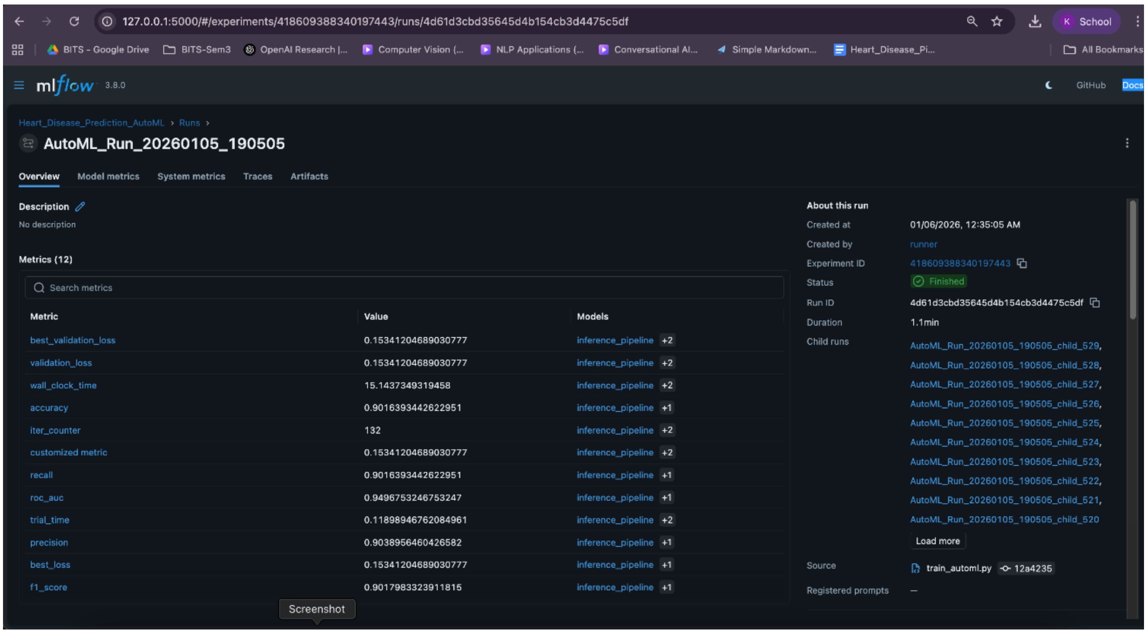
- **Experiment Name:**
  Heart_Disease_Prediction_AutoML
- **Parent Run:**
  Represents a single AutoML execution, including:
  - Dataset split
  - Preprocessing strategy
  - AutoML configuration
  - Final model evaluation
- **Nested Child Runs:**
  Each FLAML trial (model + hyperparameters) is logged as a nested MLflow run using a custom callback.



Logged Parameters:

Parent Run Parameters

The following parameters are logged once per AutoML run:

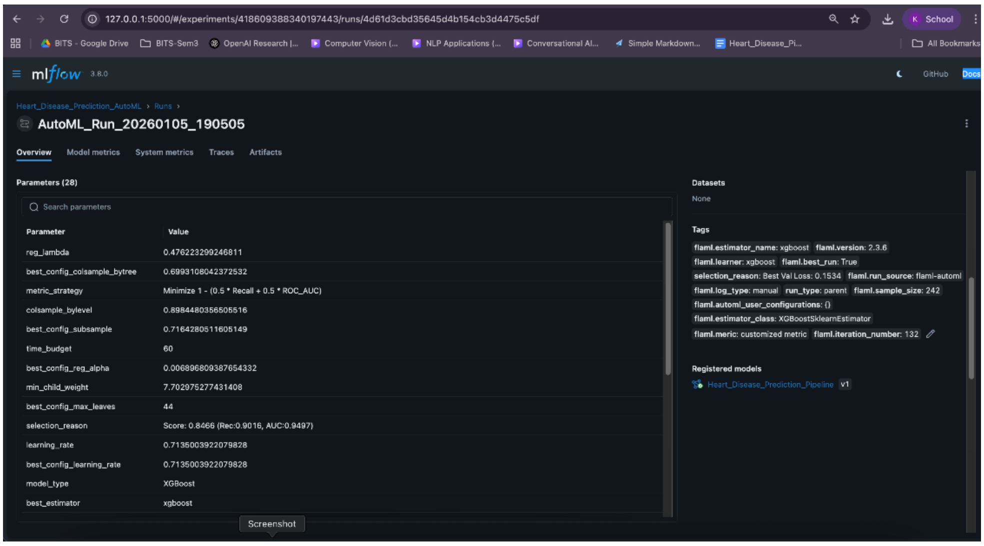


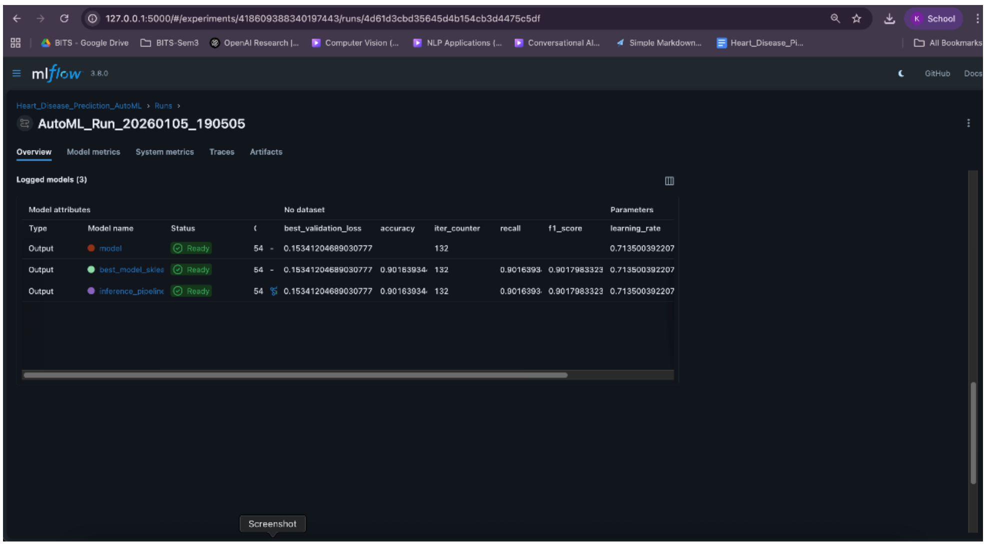
Final Test Metrics

After model selection, the best model is evaluated on the held-out test set. The following metrics are logged:

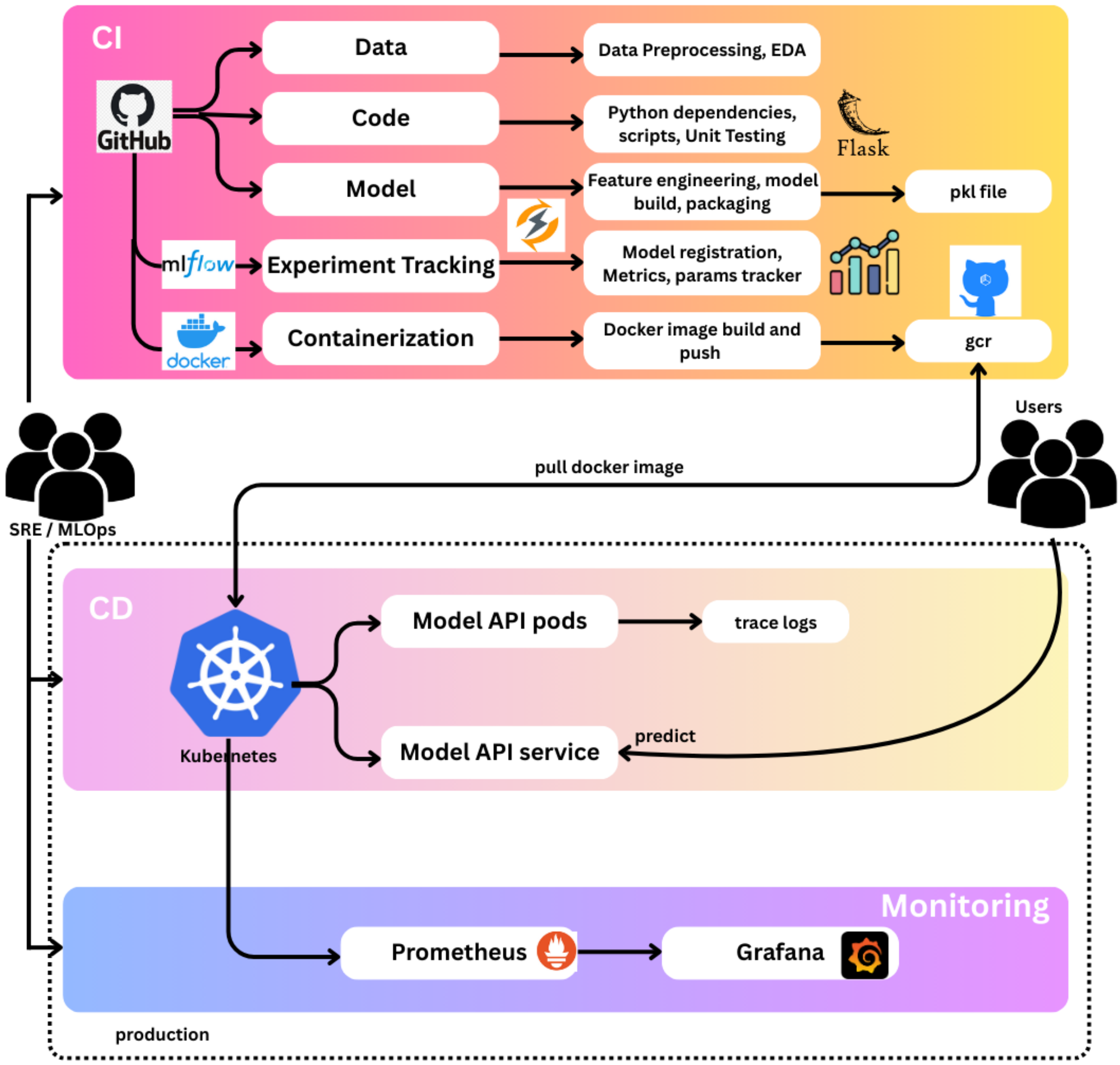
- Accuracy

- Precision (weighted)
- Recall (weighted)
- F1-score (weighted)
- ROC AUC



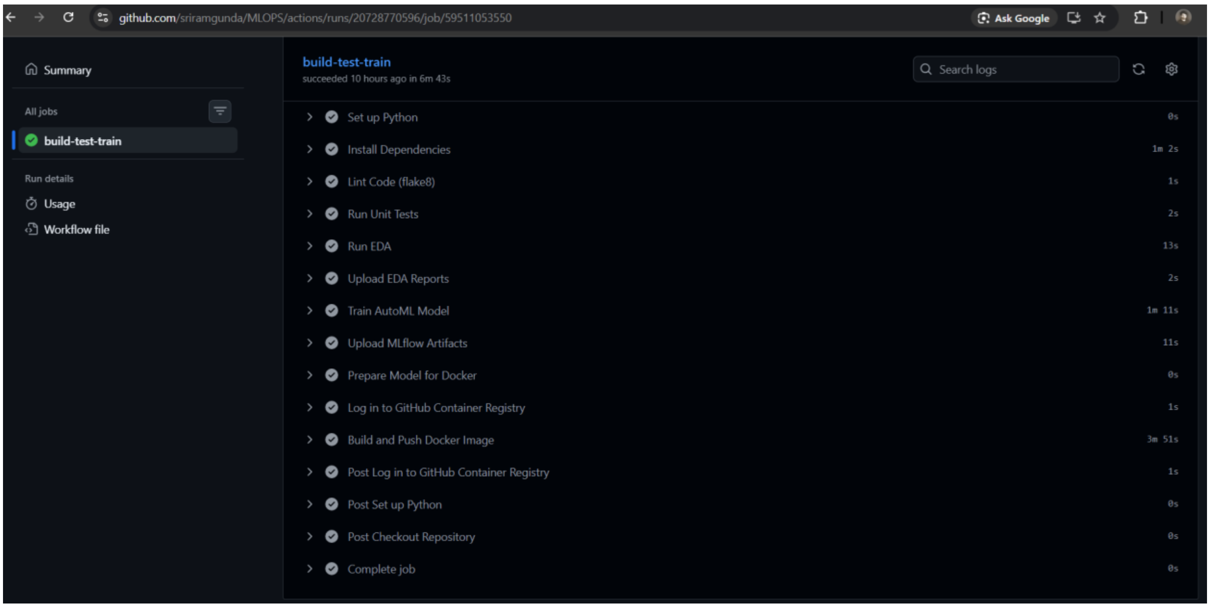


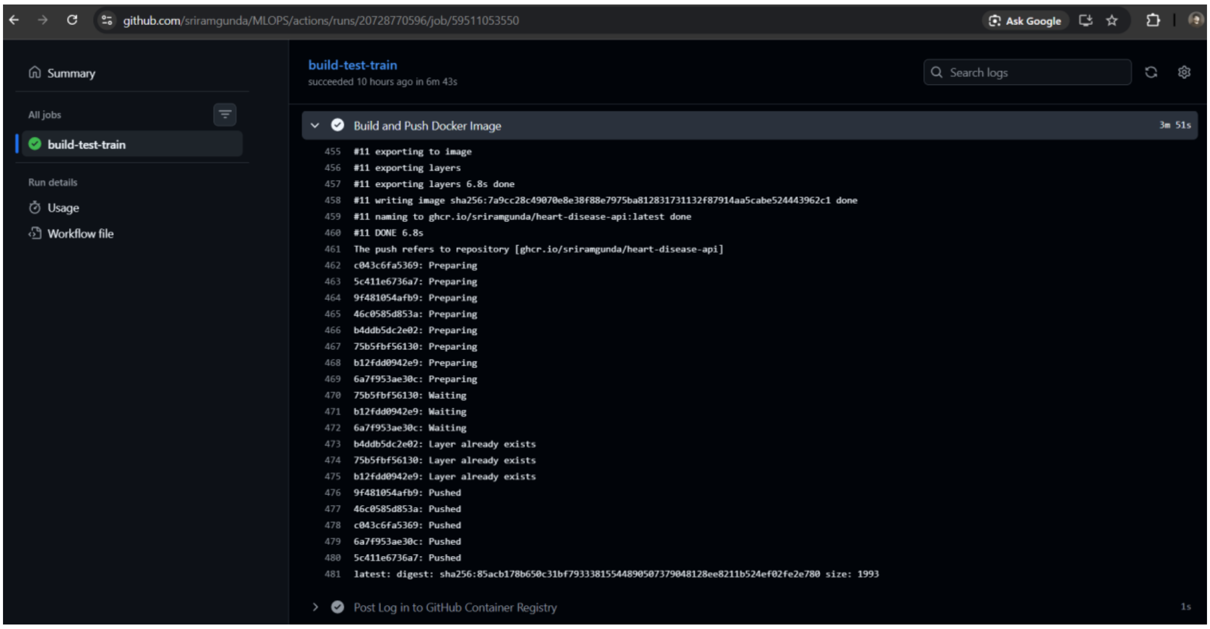
Architecture diagram:



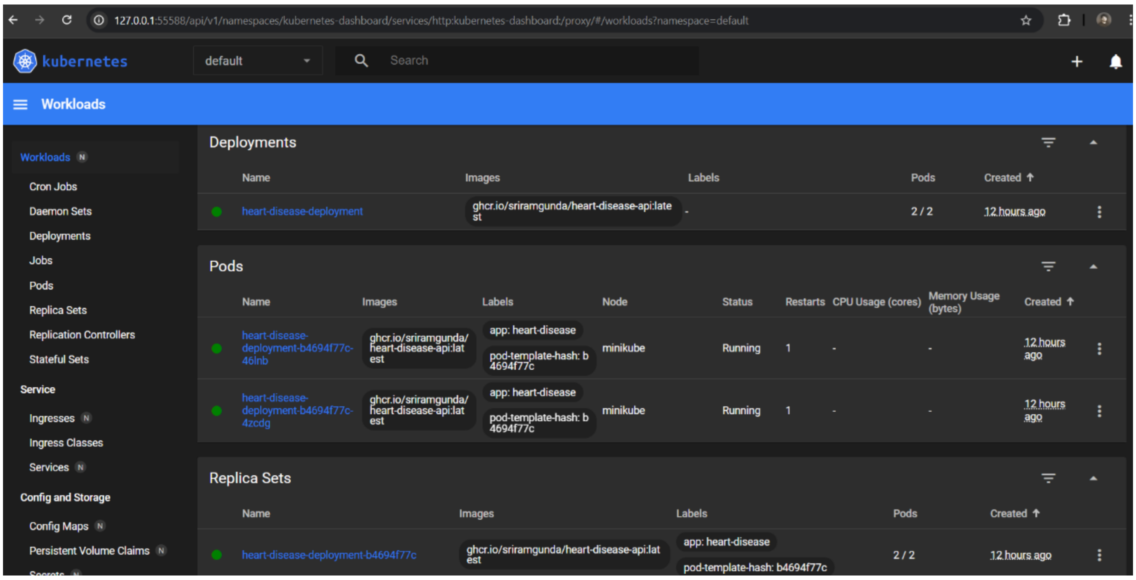
CI/CD and deployment workflow screenshots:

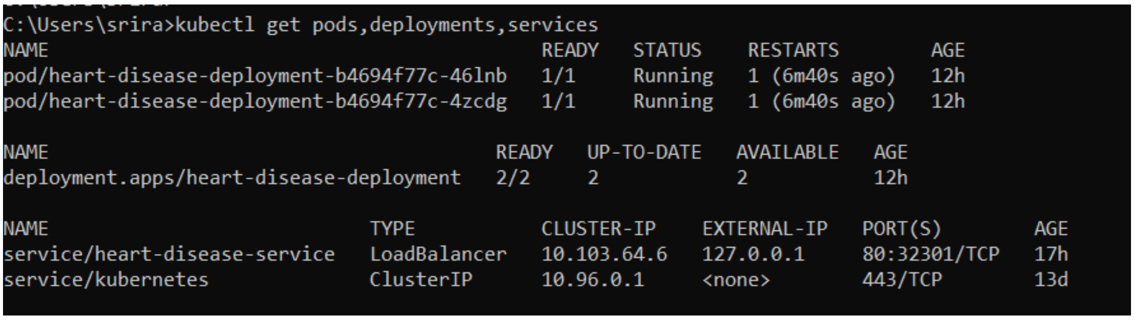
CI Screenshots:





Deployment Screenshots:

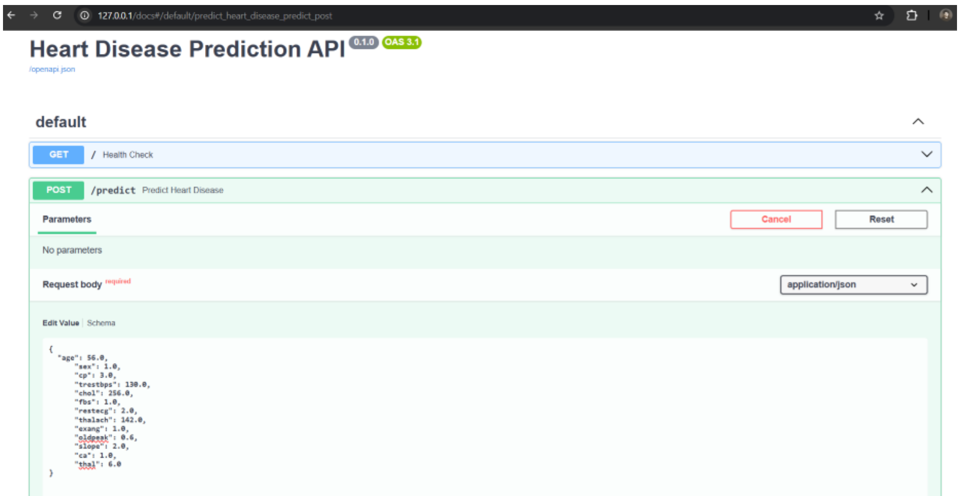


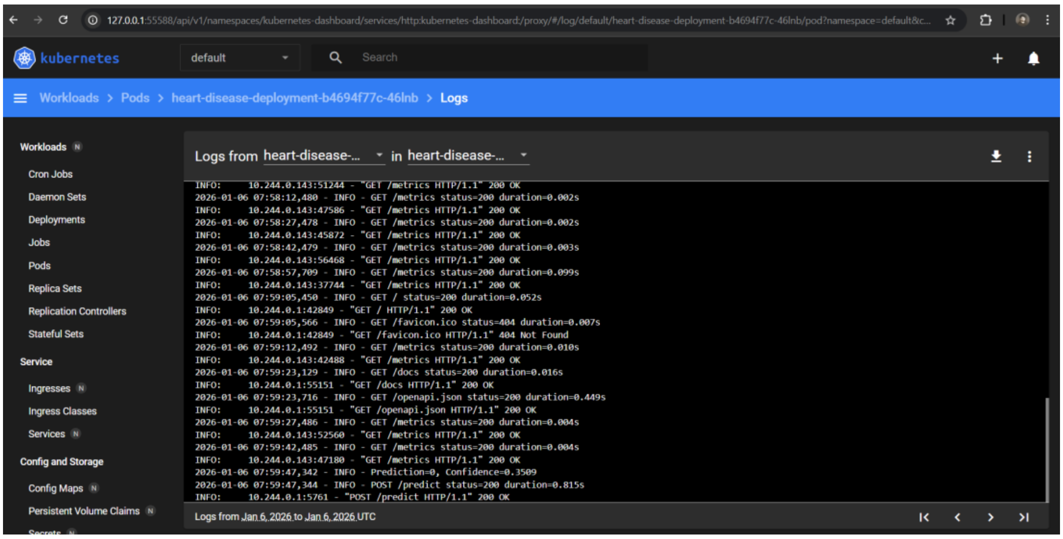


```
C:\Users\srira>kubectl get pods,deployments,services
NAME                                          READY   STATUS    RESTARTS       AGE
pod/heart-disease-deployment-b4694f77c-46lnb   1/1     Running   1 (6m40s ago)   12h
pod/heart-disease-deployment-b4694f77c-4zcdg   1/1     Running   1 (6m40s ago)   12h

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/heart-disease-deployment   2/2     2            2           12h

NAME                            TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
service/heart-disease-service   LoadBalancer   10.103.64.6    127.0.0.1     80:32301/TCP   17h
service/kubernetes              ClusterIP      10.96.0.1      <none>        443/TCP        13d
```



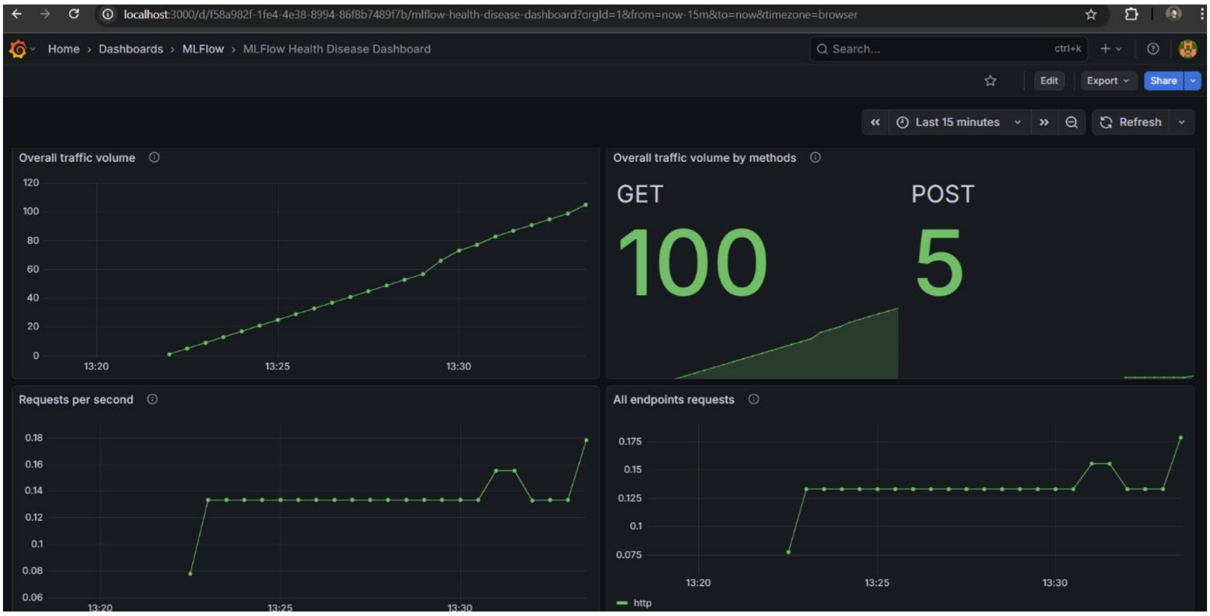


Prometheus metrics:

127.0.0.1/metrics

```
process_resident_memory_bytes 2.08457728e+08
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.76768590473e+09
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 8.129999999999999
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 9.0
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1.048576e+06
# HELP api_requests_total Total API requests
# TYPE api_requests_total counter
api_requests_total{endpoint="/metrics",method="GET"} 36.0
api_requests_total{endpoint="/predict",method="GET"} 1.0
api_requests_total{endpoint="/predict",method="POST"} 2.0
api_requests_total{endpoint="/",method="GET"} 1.0
# HELP api_requests_created Total API requests
# TYPE api_requests_created gauge
api_requests_created{endpoint="/metrics",method="GET"} 1.767685927748295e+09
api_requests_created{endpoint="/predict",method="GET"} 1.7676863576104758e+09
api_requests_created{endpoint="/predict",method="POST"} 1.7676863720557714e+09
api_requests_created{endpoint="/",method="GET"} 1.7676864481392646e+09
# HELP api_request_latency_seconds Request latency
# TYPE api_request_latency_seconds histogram
api_request_latency_seconds_bucket{le="0.005"} 17.0
api_request_latency_seconds_bucket{le="0.01"} 29.0
api_request_latency_seconds_bucket{le="0.025"} 32.0
api_request_latency_seconds_bucket{le="0.05"} 35.0
api_request_latency_seconds_bucket{le="0.075"} 35.0
api_request_latency_seconds_bucket{le="0.1"} 36.0
api_request_latency_seconds_bucket{le="0.25"} 37.0
api_request_latency_seconds_bucket{le="0.5"} 38.0
api_request_latency_seconds_bucket{le="0.75"} 38.0
api_request_latency_seconds_bucket{le="1.0"} 38.0
api_request_latency_seconds_bucket{le="2.5"} 39.0
api_request_latency_seconds_bucket{le="5.0"} 39.0
api_request_latency_seconds_bucket{le="7.5"} 39.0
api_request_latency_seconds_bucket{le="10.0"} 39.0
api_request_latency_seconds_bucket{le="+Inf"} 39.0
api_request_latency_seconds_count 39.0
api_request_latency_seconds_sum 2.03809096900045
# HELP api_request_latency_seconds_created Request latency
# TYPE api_request_latency_seconds_created gauge
api_request_latency_seconds_created 1.767685910923466 7e+09
```

Grafana Dashboard:



Link to code repository:

GitHub Link: https://github.com/sriramgunda/MLOPS

Additional Screenshots:

Please refer documentation folder for additional screenshots.
https://github.com/sriramgunda/MLOPS/tree/main/documentation

Demonstration Link:

Please refer the video uploaded to YouTube: https://www.youtube.com/watch?v=SqjqfYa2Kyc