

Birla Institute of Technology and Science, Pilani

Work Integrated Learning Programmes Division

M. Tech. in AI & ML

AIML ZG523 - MLOps

Assignment 1: End-to-End ML Model Development, CI/CD, and Production Deployment Experimental Learning

Group 110

Sl no.	BITS ID	Student Name
1.	2024AA05210	PRIYANKA R
2.	2024AA05159	VINOTHRAJA P
3.	2023AC05062	ABHAY SURESH GUNJARKAR
4.	2024AA05864	GOWTHAM G
5.	2024AA05741	DIYA MARIA DEEPAK

GitHub Repository link - https://github.com/2024aa05210-Priyanka/Group_110_Mlops_Assignment_1_heart-disease_Prediction

The repository is publicly accessible and contains code, documentation, CI/CD workflows, deployment manifests, and screenshots.

Refer to the setup instructions at the end of the same document or check readme.md file in the repo

Video Link - https://drive.google.com/file/d/100i-clY3bdbuEJcdK_nomDc7g5Z39tvh/view?usp=drive_link

If the link doesn't work, please download the video file and then you will be able to watch.

1. Introduction

Machine Learning Operations (MLOps) bridges the gap between data science experimentation and production deployment. While traditional machine learning projects often stop at model training, real-world systems require automation, monitoring, versioning, and deployment strategies.

This project aims to implement a complete MLOps workflow that transforms a trained machine learning model into a production-ready service with automated testing, deployment, and monitoring capabilities.

2. Dataset Description and Preprocessing

2.1 Dataset Source

The dataset used in this project is the UCI Heart Disease Dataset, a publicly available dataset commonly used for benchmarking binary classification algorithms in healthcare analytics. The dataset contains clinical and diagnostic attributes collected from patients undergoing heart disease evaluation.

The objective of the task is to predict whether a patient has heart disease based on clinical measurements.

2.2 Target Variable Definition

The original dataset contains a target variable (num) with multiple integer values indicating the severity of heart disease. For this project, the problem was reformulated as a binary classification task:

- 0 → No heart disease
- 1 → Presence of heart disease

This transformation was applied as follows:

```
df["target"] = (df["target"] > 0).astype(int)
```

This approach is widely adopted in clinical ML tasks to simplify interpretation and deployment.

2.3 Missing Value Analysis

Before model training, the dataset was analyzed for missing values. The following observations were made:

Feature	Missing Values
ca	Present
thal	Present
All other features	None

The missing values were limited to two categorical/ordinal features, reducing the risk of information loss during imputation.

2.4 Missing Value Handling and Imputation Strategy

Missing values were handled during preprocessing using statistical imputation, guided by feature semantics:

- ca (number of major vessels):
 - Treated as a numerical/ordinal feature
 - Imputed using the median
 - Median imputation was chosen due to robustness against outliers

- thal (thalassemia):
 - Treated as a categorical/ordinal feature
 - Imputed using the mode
 - Mode imputation preserves the most frequent clinical category

This approach ensures:

- Minimal distortion of feature distributions
- Compatibility with tree-based and linear models
- Simplicity and reproducibility

2.5 Feature Selection

The following features were selected based on clinical relevance and dataset completeness:

- Age
- Sex
- Chest pain type (cp)
- Resting blood pressure (trestbps)
- Cholesterol (chol)
- Fasting blood sugar (fbs)
- Resting ECG (restecg)
- Maximum heart rate (thalach)
- Exercise-induced angina (exang)
- ST depression (oldpeak)
- Slope of ST segment (slope)
- Number of vessels (ca)
- Thalassemia (thal)

No automated feature elimination was applied to preserve interpretability.

3. Train–Test Split Strategy

3.1 Split Ratio

The dataset was divided into training and testing subsets using an **80:20 split**:

- **80%** training data
- **20%** testing data

Code:

```
X_train, X_test, y_train, y_test = train_test_split(
```

```
X,  
y,  
test_size=0.2,  
random_state=42,  
stratify=y  
)
```

3.2 Stratification

Stratified sampling was applied to preserve the class distribution of the target variable across both training and test sets.

This is critical for healthcare datasets where class imbalance can bias evaluation metrics.

3.3 Reproducibility

A fixed random_state=42 was used to ensure:

- Deterministic splits
- Reproducible experiments
- Consistent CI/CD execution

4. Feature Scaling and Transformation

4.1 Why Scaling Was Required

Feature scaling was applied to numerical variables because:

- Logistic Regression is sensitive to feature magnitudes
- Gradient-based optimization converges faster with normalized features
- Consistent preprocessing ensures fair model comparison

4.2 Scaling Technique Used

The **StandardScaler** was applied to numerical features:

```
StandardScaler()
```

This transforms features to have:

- Mean = 0
- Standard deviation = 1

4.3 Numerical Features Scaled

The following numerical features were scaled:

- age
- trestbps

- chol
- thalach
- oldpeak
- ca

Categorical features were left unscaled.

4.4 Persisting the Scaler

The fitted scaler was saved as a serialized artifact:

scaler.pkl

This ensures:

- Identical preprocessing during inference
- Prevention of training-serving skew
- Reproducible predictions in production

5.1 Models Trained

Two supervised learning models were trained:

1. Logistic Regression
 - Baseline linear model
 - Provides interpretability
 - Sensitive to scaling
2. Random Forest Classifier
 - Ensemble of decision trees
 - Captures non-linear feature interactions
 - Robust to noise and outliers

5.2 Evaluation Metrics

The following metrics were used:

- ROC-AUC (primary metric)
- Accuracy (secondary metric)

ROC-AUC was prioritized because:

- It is insensitive to class imbalance
- It measures ranking quality rather than threshold dependence

5.3 Model Comparison Results

Model	ROC-AUC
Logistic Regression	~0.95
Random Forest	~0.95+

Although both models performed well, Random Forest achieved the highest ROC-AUC and was therefore selected as the final production model.

5.4 Model Selection Policy

Only the best-performing model was:

- Exported as a pickle file
- Logged in MLflow as the production model
- Deployed via the API

This follows standard MLOps practice of separating experimentation from deployment.

5.5 Why Only One Model Was Deployed

Deploying a single model:

- Reduces operational complexity
- Prevents ambiguity during inference
- Aligns with CI/CD and monitoring pipelines

Logistic Regression was retained for comparison, not deployment.

6. Experiment Tracking

MLflow was used to track:

- Model hyperparameters
- Evaluation metrics
- Generated artifacts

This enabled comparison of different experiments and ensured reproducibility of results. Experiment tracking outputs are stored locally and validated through the MLflow UI.

7. CI/CD Pipeline

A Continuous Integration pipeline was implemented using **GitHub Actions** to automate the following steps on every push to the main branch:

- Code checkout
- Dependency installation
- Code linting using flake8

- Unit testing using pytest
- Model training execution

This ensures that code quality is maintained and that the system remains stable as changes are introduced.

8. Model Serving API

8.1 Framework

- FastAPI

8.2 Endpoints

Endpoint	Method	Description
/	GET	Health check
/predict	POST	Predict heart disease
/metrics	GET	Expose Prometheus metrics

8.3 Sample Request

```
{  
  "age": 55,  
  "sex": 1,  
  "cp": 2,  
  "trestbps": 140,  
  "chol": 250,  
  "fbs": 0,  
  "restecg": 1,  
  "thalach": 150,  
  "exang": 0,  
  "oldpeak": 1.5,  
  "slope": 1,  
  "ca": 0,  
  "thal": 2  
}
```

8.4 Sample Response

```
{  
  "prediction": 0,  
  "probability": 0.49  
}
```

9. Containerization

The API was containerized using Docker to ensure environment consistency.

Docker Features

- Lightweight Python base image
- Dependency installation via requirements.txt
- FastAPI application served using Uvicorn
- Port exposure for API access

The Docker image can be built and run locally, enabling easy replication of the environment.

9. Deployment Using Kubernetes

The containerized API was deployed to **local Kubernetes (Docker Desktop)**.

Kubernetes Resources

- Deployment: Manages application pods
- Service (NodePort): Exposes the API externally

Access URL

<http://localhost:30007/docs>

This allows testing of the API via Swagger UI.

10. Monitoring and Logging

10.1 Logging

Application logs capture:

- Incoming requests
- Endpoints accessed
- Prediction outputs
- Errors (if any)

Logs can be viewed using:

kubectll logs <pod-name>

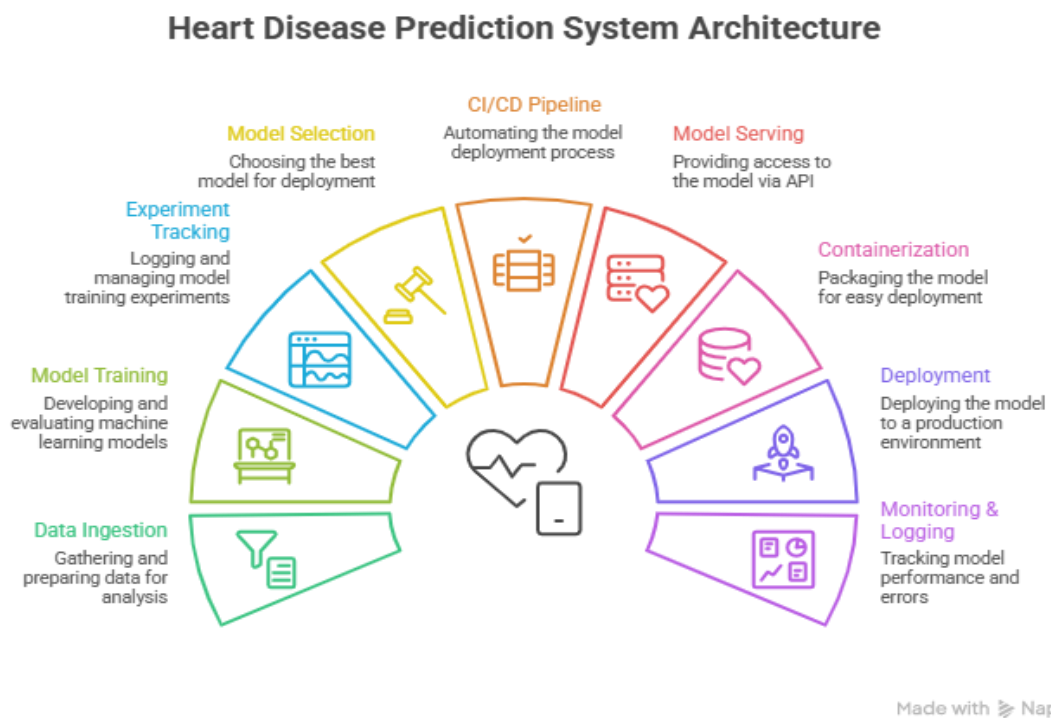
10.2 Monitoring with Prometheus

- Metrics exposed via /metrics
- Request count and latency tracked
- Prometheus configured to scrape API metrics

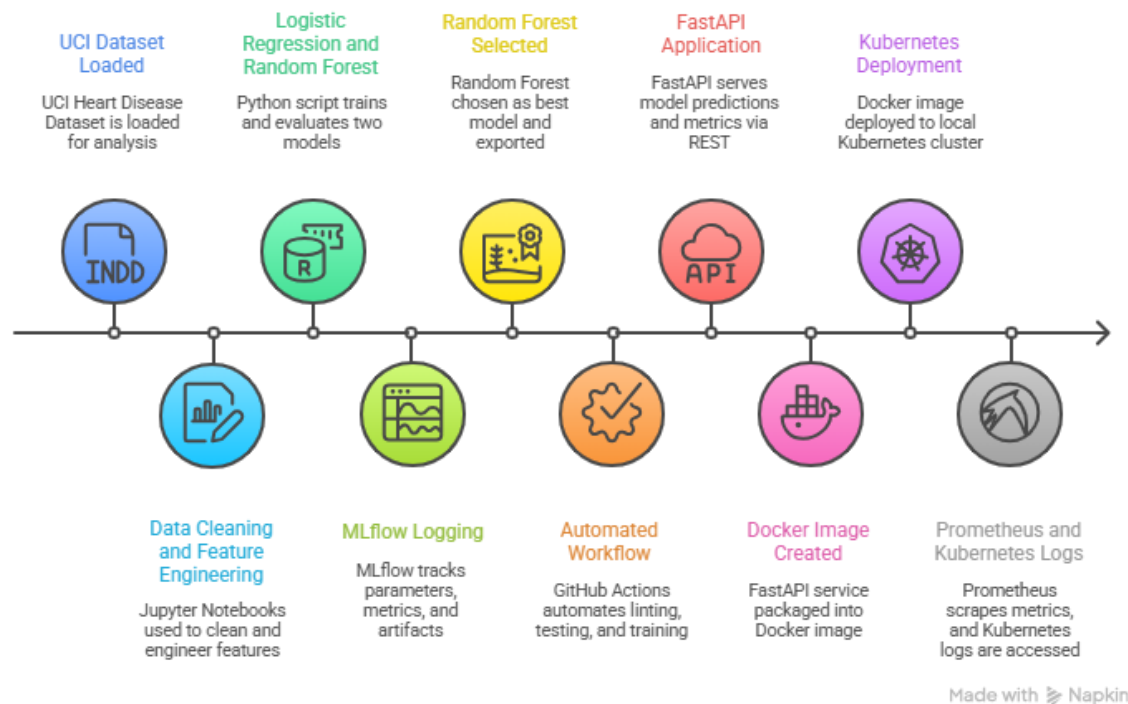
The Prometheus dashboard was used to validate:

- Successful scraping
- Endpoint availability
- Request statistics

11. Architecture Overview



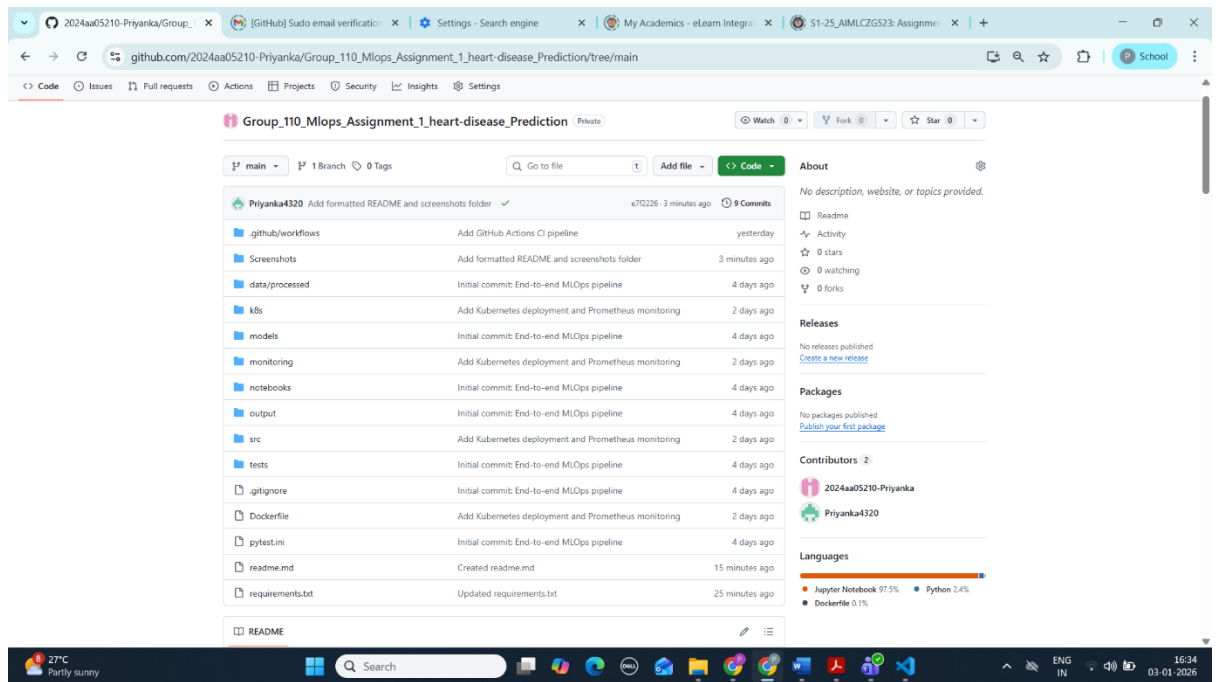
MLOps Workflow for Heart Disease Prediction



12. Repository Structure

The GitHub repository contains:

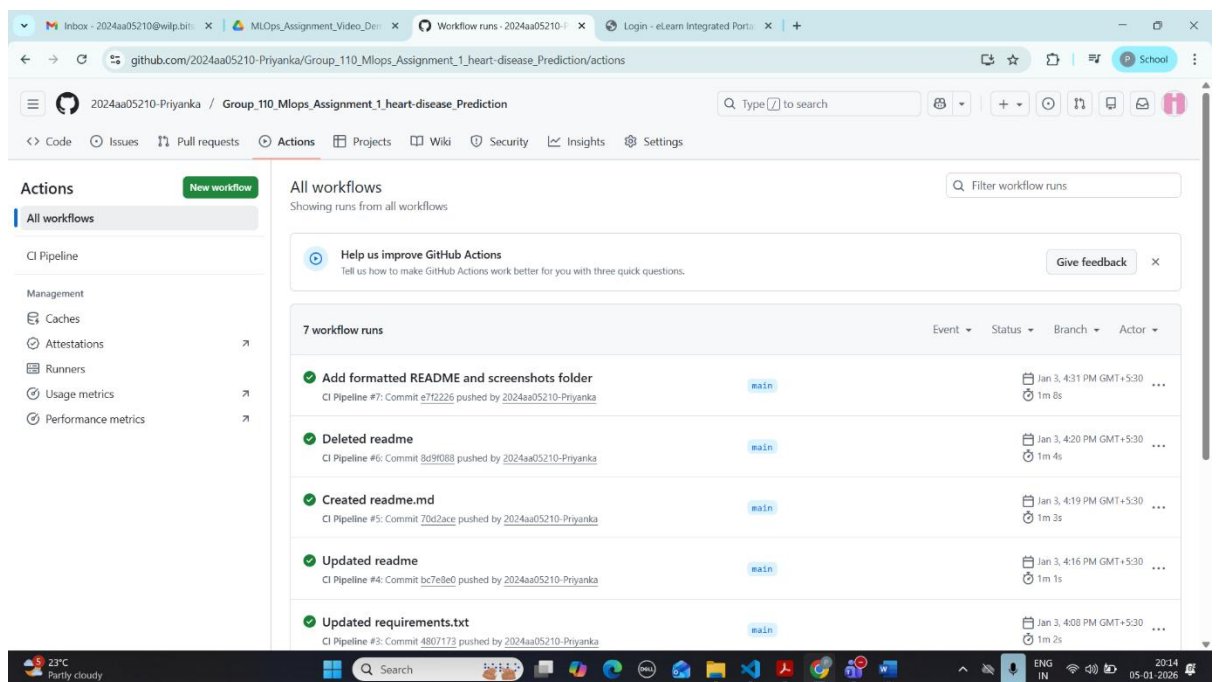
- Source code (src/)
- Jupyter notebooks (notebooks/)
- Unit tests (tests/)
- Dockerfile
- Kubernetes manifests
- Prometheus configuration
- CI/CD workflow
- Final report and screenshots



13. Screenshots and Verification

The following screenshots are included in the screenshots/ folder and referenced in the report:

- CI pipeline success (GitHub Actions)



- ML Flow for tracking

The screenshot shows the MLflow web interface at the URL `127.0.0.1:5000/#/experiments/381544595184043901/models/m-130e8bb690c34e0fa16ecd5b7c1f3ead`. The page displays details for a model named 'model' (Not registered). The 'Overview' tab is active, showing a description, metrics, and parameters.

Metrics (2)

Metric	Dataset	Source run	Value
accuracy	-	Random Forest (Cola...	0.901639
roc_auc	-	Random Forest (Cola...	0.954545

Parameters (2)

Parameter	Value

About this logged model

- Created at: 3 minutes ago
- Created by: PRIYANKA R
- Status: Ready
- Model ID: m-130e8bb690c34e0fa16ec...
- Source run: Random Forest (Cola-trained)
- Source run ID: Se0812d7696e485b834c4b5...
- Logged from: .log_experiments.py

Datasets used: None

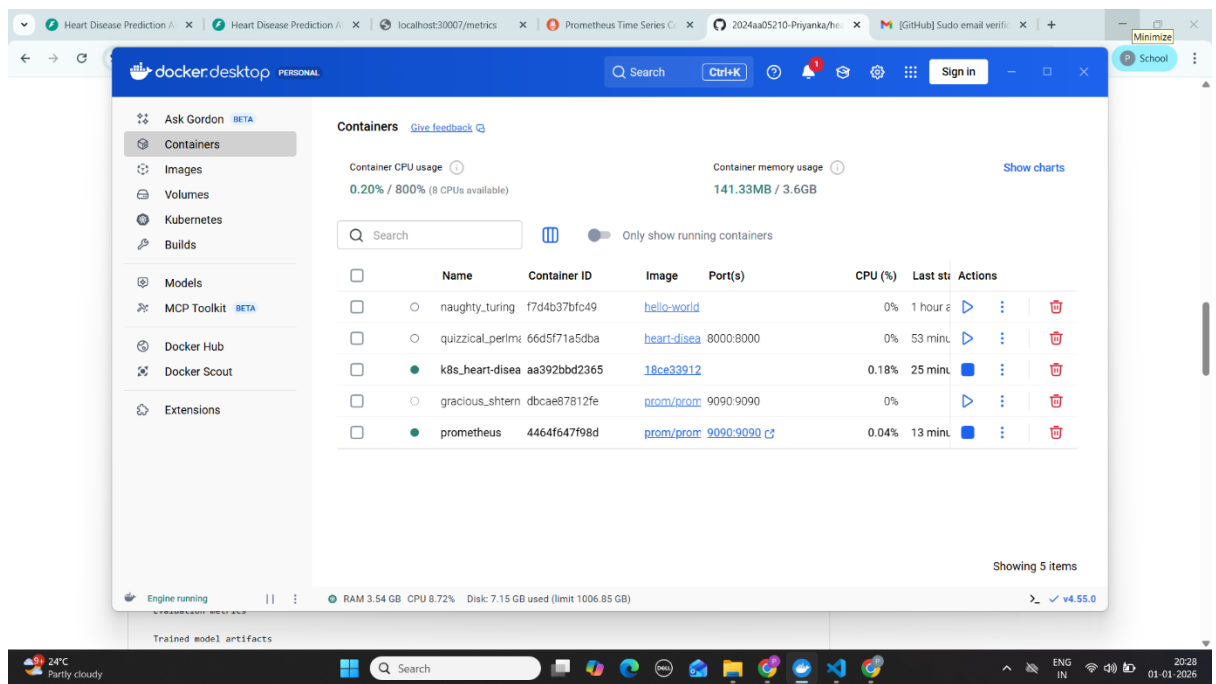
Model versions: None

- Docker image build

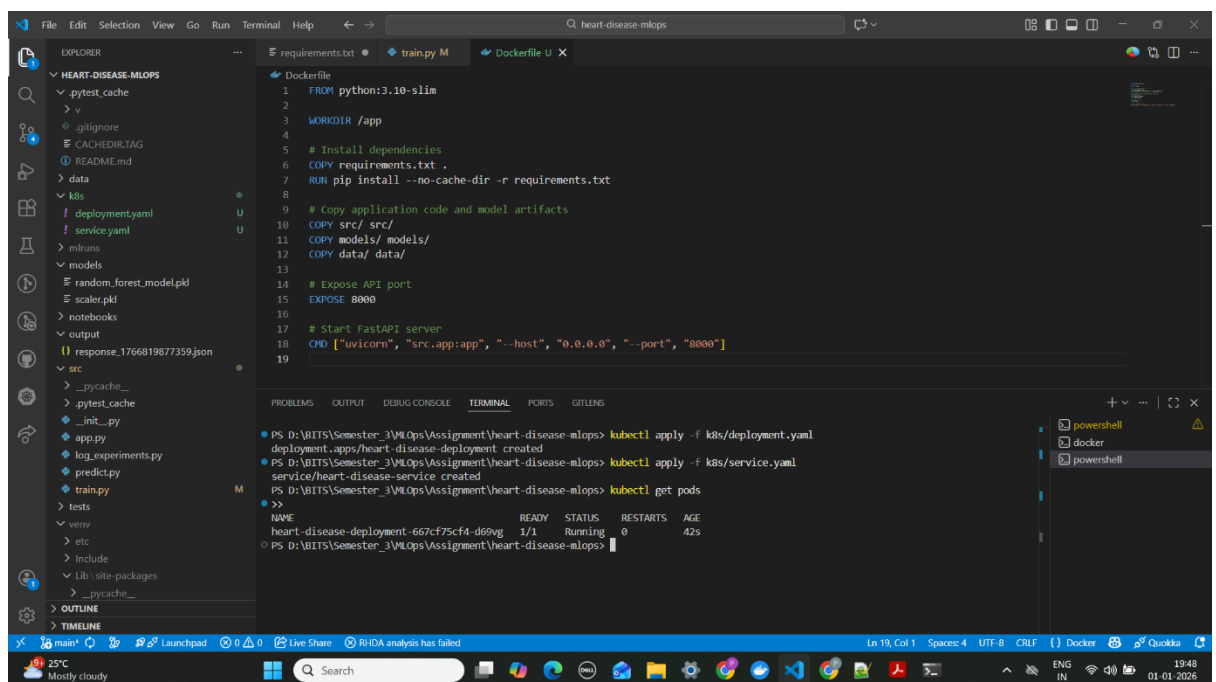
The screenshot shows a VS Code editor with a Dockerfile build process. The terminal output displays the build progress, including the installation of dependencies and the creation of the Docker image.

```

PS D:\BITS\Semester_3\MLOps\Assignment\heart-disease-mlops> docker build -t heart-disease-api .
[+] Building 145.6s (12/12) FINISHED
-> [internal] load build definition from Dockerfile
-> [internal] load metadata for docker.io/library/python:3.10-slim
-> [internal] load .dockerignore
-> [internal] load build context
-> [1/7] FROM docker.io/library/python:3.10-slim@sha256:7b68a5fa7cf6d28b4cedb1dc9a134fdd394fe27edbc4c2519756c01d21df2313
-> sha256:7da4424a113245eb185ea22f2512ecb36f80ca1d0547c64b117f28495d3c3e5 250B / 250B
-> sha256:9c27bc7ba63d1ac690daefc68302197d3ab9a91fc50e19f447cd57eda92d87c 13.82MB / 13.82MB
-> sha256:02d7611c4ea219af91448a4720bdba03657d3bc035cfe12774af85daa6aff 29.78MB / 29.78MB
-> sha256:8719e52fa1374bdc2694370a1c607c8172809c2ebbc0e9d9ab1a9ca86763 1.29MB / 1.29MB
-> extracting sha256:02d7611c4ea219af91448a4720bdba03657d3bc035cfe12774af85daa6aff 1.0s
-> extracting sha256:8719e52fa1374bdc2694370a1c607c8172809c2ebbc0e9d9ab1a9ca86763 0.1s
-> extracting sha256:9c27bc7ba63d1ac690daefc68302197d3ab9a91fc50e19f447cd57eda92d87c 0.6s
-> extracting sha256:7da4424a113245eb185ea22f2512ecb36f80ca1d0547c64b117f28495d3c3e5 0.0s
-> [2/7] WORKDIR /app
-> [3/7] COPY requirements.txt .
-> [4/7] RUN pip install --no-cache-dir -r requirements.txt
-> [5/7] COPY src/ src/
-> [6/7] COPY models/ models/
-> [7/7] COPY data/ data/
-> exporting to image
-> exporting manifest sha256:38a011d71590ca35f911d47f3a80066dac33e301b438c3da6ca76936f890e92 0.0s
-> exporting config sha256:77609262103d3ca97fa621bcf28b40b0638fec81ef575f0aa26b210ecd9546 0.0s
-> exporting attestation manifest sha256:310e2cd9e8bdaa4993717349290a37b1ed3e5afa92259f3960b359213b1aaf6 0.0s
-> exporting manifest list sha256:bc0b40ee7b7596032c20f7cbb1cd650b0ae36354653c3207ac2d9925f1ba0d 0.0s
-> naming to docker.io/library/heart-disease-api:latest
-> unpacking to docker.io/library/heart-disease-api:latest 13.4s
PS D:\BITS\Semester_3\MLOps\Assignment\heart-disease-mlops>
  
```

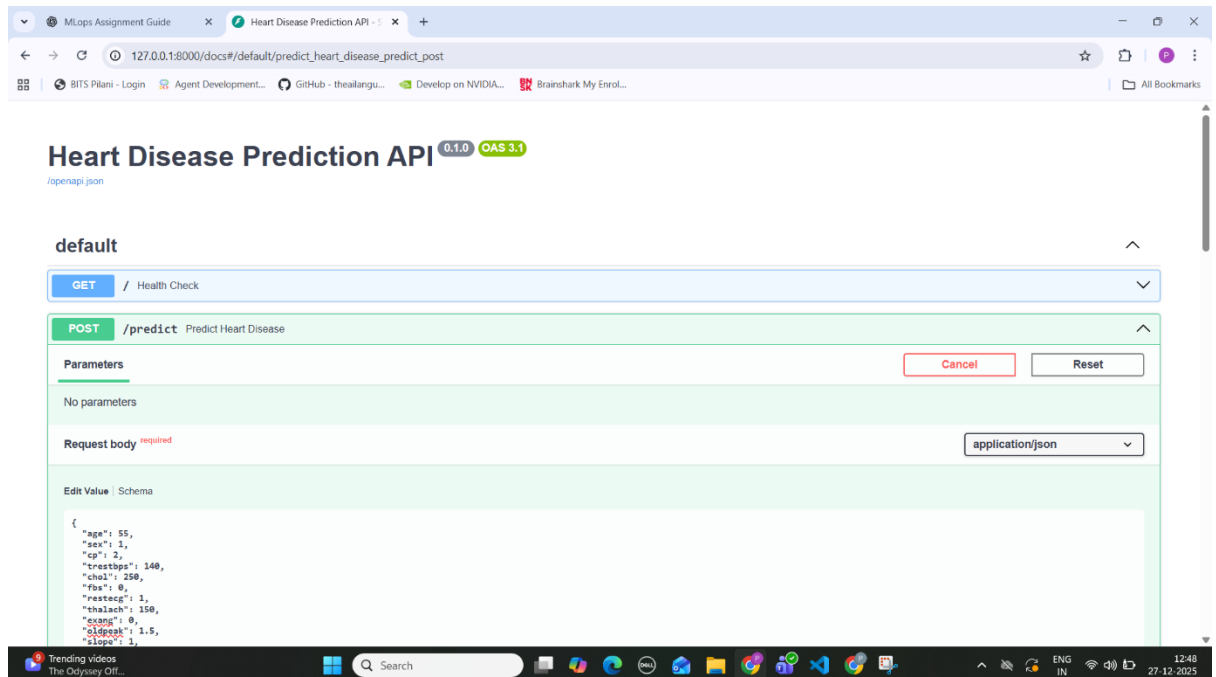


- **Kubernetes pods and services**

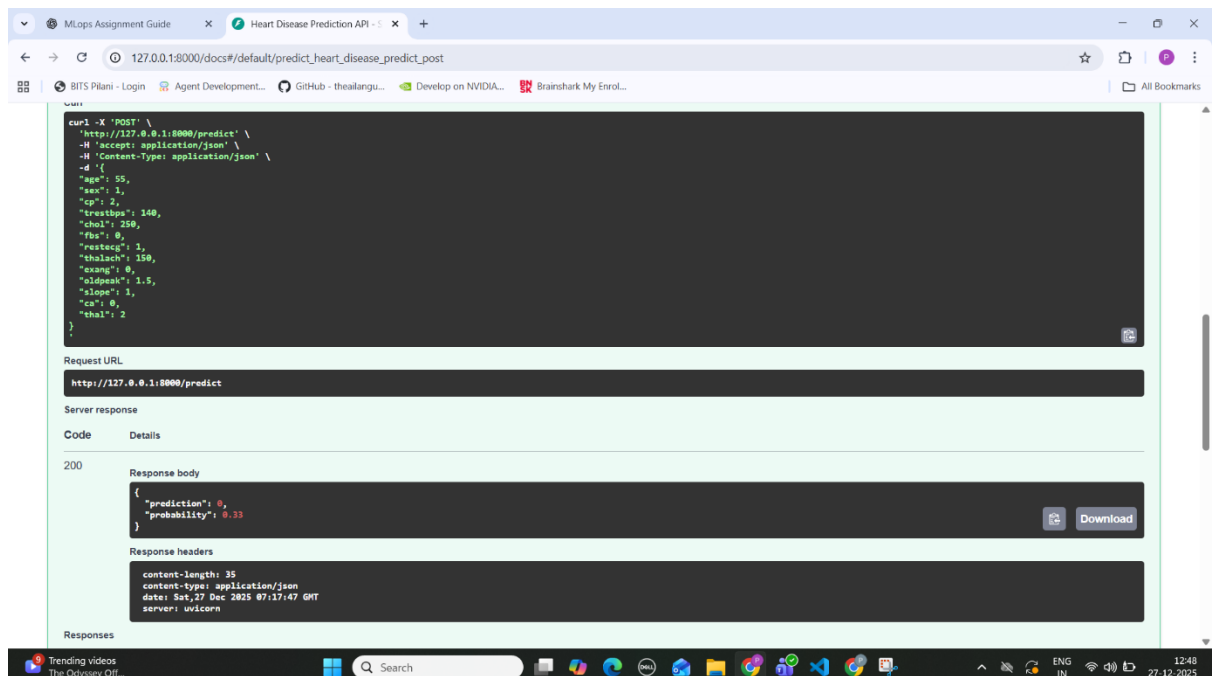


- Swagger UI for deployed API

Request:



Response:



- Metrics endpoint output

```
# TYPE http_requests_total counter
http_requests_total{endpoint="/metrics",method="GET"} 3.0
http_requests_total{endpoint="/predict",method="POST"} 2.0
# HELP http_requests_created Total HTTP requests
# TYPE http_requests_created gauge
http_requests_created{endpoint="/metrics",method="GET"} 1.7672780063360786e+09
http_requests_created{endpoint="/predict",method="POST"} 1.7672780379128466e+09
# HELP http_request_latency_seconds Request latency
# TYPE http_request_latency_seconds histogram
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.005"} 0.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.01"} 1.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.025"} 2.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.05"} 2.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.075"} 2.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.1"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.25"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.5"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="0.75"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="1.0"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="2.5"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="5.0"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="7.5"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="10.0"} 3.0
http_request_latency_seconds_bucket{endpoint="/metrics",le="+Inf"} 3.0
http_request_latency_seconds_count{endpoint="/metrics"} 3.0
http_request_latency_seconds_sum{endpoint="/metrics"} 0.12123250961303711
http_request_latency_seconds_bucket{endpoint="/predict",le="0.005"} 0.0
http_request_latency_seconds_bucket{endpoint="/predict",le="0.01"} 0.0
http_request_latency_seconds_bucket{endpoint="/predict",le="0.025"} 0.0
http_request_latency_seconds_bucket{endpoint="/predict",le="0.05"} 0.0
http_request_latency_seconds_bucket{endpoint="/predict",le="0.075"} 0.0
http_request_latency_seconds_bucket{endpoint="/predict",le="0.1"} 0.0
http_request_latency_seconds_bucket{endpoint="/predict",le="0.25"} 1.0
http_request_latency_seconds_bucket{endpoint="/predict",le="0.5"} 1.0
http_request_latency_seconds_bucket{endpoint="/predict",le="0.75"} 1.0
http_request_latency_seconds_bucket{endpoint="/predict",le="1.0"} 1.0
http_request_latency_seconds_bucket{endpoint="/predict",le="2.5"} 1.0
http_request_latency_seconds_bucket{endpoint="/predict",le="5.0"} 2.0
http_request_latency_seconds_bucket{endpoint="/predict",le="7.5"} 2.0
http_request_latency_seconds_bucket{endpoint="/predict",le="10.0"} 2.0
http_request_latency_seconds_bucket{endpoint="/predict",le="+Inf"} 2.0
http_request_latency_seconds_count{endpoint="/predict"} 2.0
http_request_latency_seconds_sum{endpoint="/predict"} 2.869148969562686
# HELP http_request_latency_seconds_created Request latency
# TYPE http_request_latency_seconds_created gauge
http_request_latency_seconds_created{endpoint="/metrics"} 1.767278006336158e+09
http_request_latency_seconds_created{endpoint="/predict"} 1.76727803791292e+09
```

- Prometheus dashboard

Prometheus

Query Alerts Status

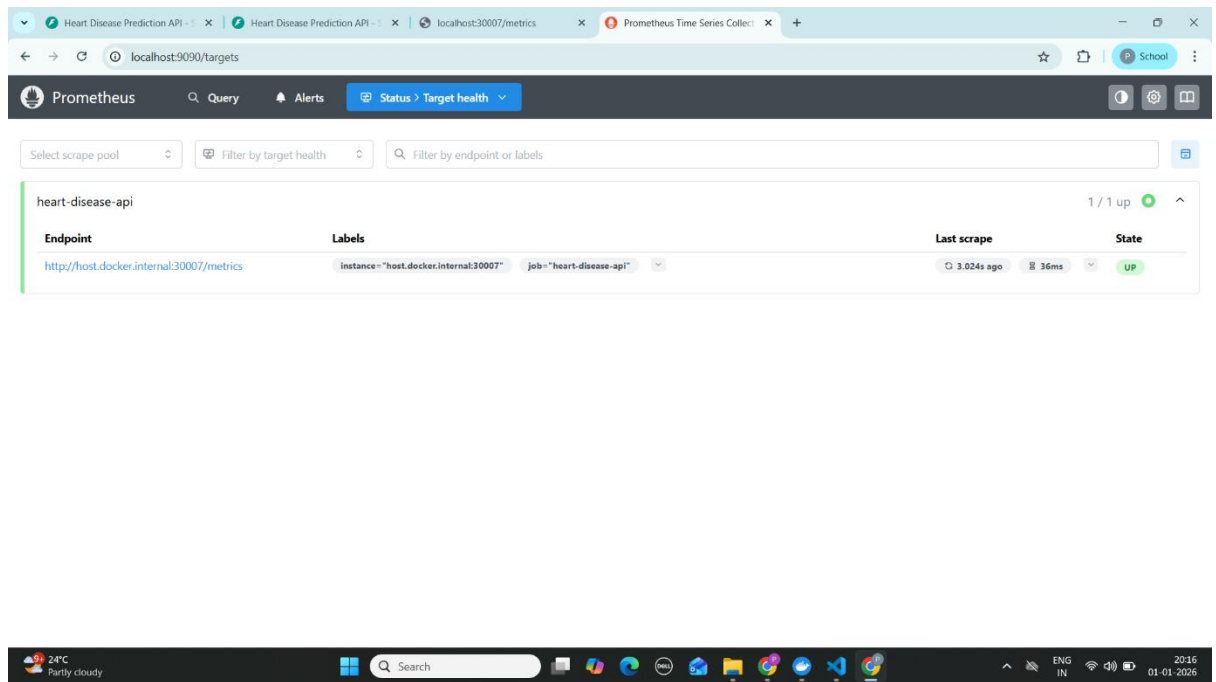
Query: `_ http_requests_total` [Execute]

Table Graph Explain

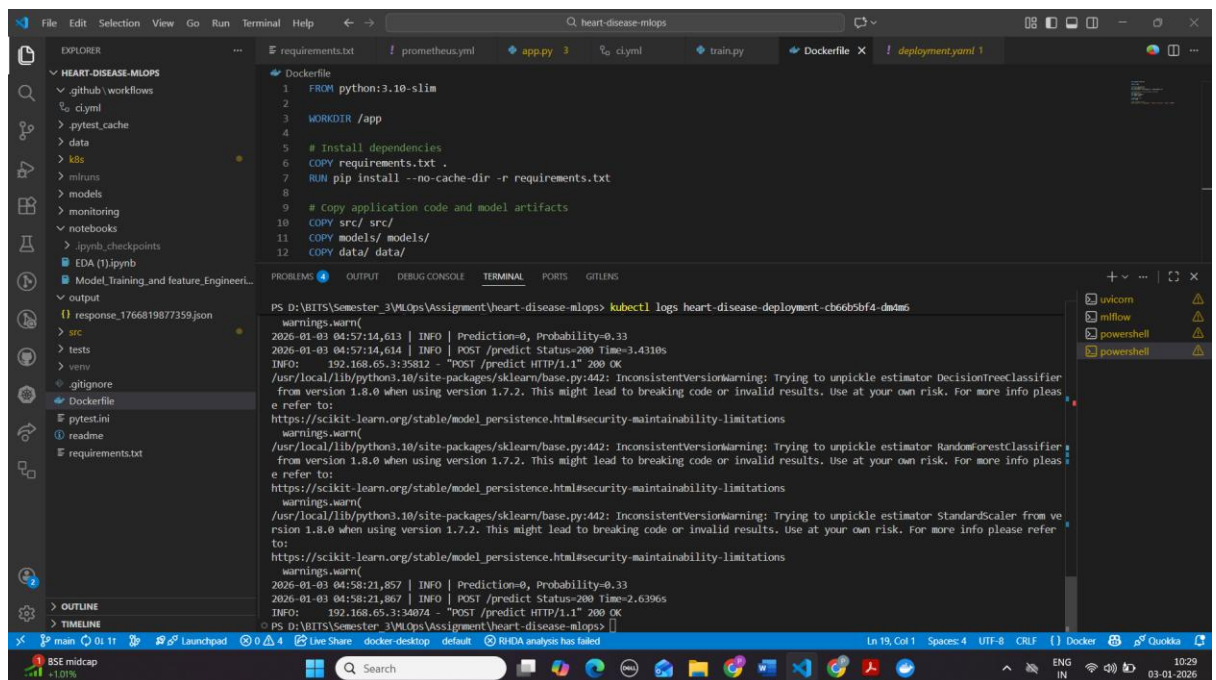
Evaluation time: [] Load time: 24ms Result series: 2

http_requests_total{endpoint="/metrics",instance="host.docker.internal:30007",job="heart-disease-api",method="GET"}	9
http_requests_total{endpoint="/predict",instance="host.docker.internal:30007",job="heart-disease-api",method="POST"}	2

+ Add query



- **Kubernetes logs output**



These screenshots serve as evidence of successful CI/CD, deployment, and monitoring.

14. Summary of End-to-End Flow

The complete workflow followed in this project is summarized below:

1. Data ingestion and preprocessing
2. Exploratory data analysis
3. Model training and evaluation
4. Experiment tracking with MLflow
5. Continuous integration using GitHub Actions
6. API development using FastAPI
7. Containerization using Docker
8. Deployment on Kubernetes
9. Monitoring using Prometheus
10. Logging using Kubernetes logs

15. Conclusion

This project successfully demonstrates a complete MLOps pipeline that transforms a machine learning model into a production-ready service. The implementation emphasizes automation, reproducibility, deployment, and monitoring, aligning with real-world industry practices. Future enhancements may include cloud-based deployment, model retraining automation, and advanced monitoring dashboards.

APPENDIX: Setup and Installation Instructions

This section describes the steps required to set up the project environment and reproduce the results.

System Requirements

- Python 3.9 or higher
- Docker Desktop (with Kubernetes enabled)
- Git
- kubectl
- Internet access for dependency installation

Repository Setup

Clone the repository:

```
git clone <repository-url>
```

```
cd heart-disease-mlops
```

Python Environment Setup

Create and activate a virtual environment:

Windows

```
python -m venv venv
```

```
venv\Scripts\activate
```

Linux / macOS

```
python3 -m venv venv
```

```
source venv/bin/activate
```

Install dependencies:

```
pip install --upgrade pip
```

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

Run API:

```
uvicorn src.app:app --reload
```

Start MLflow UI

```
python -m mlflow ui
```

Access at: <http://127.0.0.1:5000>

Reproducible Training

The entire training process is packaged into a script.

Run Training

```
python src/train.py
```

This will:

- Load processed data
- Train models
- Select the best model
- Save model artifacts to models/

Model Inference

Inference logic is encapsulated in predict.py.

Example Usage:

```
from src.predict import predict
```

```
sample_input = {
```

```
    "age": 55,
```

```
    "sex": 1,
```

```
    "cp": 2,
```

```
    "trestbps": 140,
```

```
    "chol": 250,
```

```
    "fbs": 0,
```

```
    "restecg": 1,
```

```
    "thalach": 150,
```

```
    "exang": 0,
```

```
    "oldpeak": 1.5,
```

```
    "slope": 1,
```

```
    "ca": 0,
```

```
    "thal": 2
```

```
}
```

```
result = predict(sample_input)
```

```
print(result)
```

Testing

Unit tests validate: Model artifacts existence

Prediction output format and probability range

Run Tests:

pytest

CI/CD Pipeline

Implemented using GitHub Actions

Automatically:

- Installs dependencies
- Runs training script
- Executes unit tests
- Ensures reliability on every code push

Model Serving (FastAPI)

The trained model is served as a REST API.

Start API Locally

uvicorn src.app:app --reload

Access:

Health check: <http://127.0.0.1:8000/>

Swagger UI: <http://127.0.0.1:8000/docs>

Prediction Endpoint

Method: POST

Endpoint: /predict

Input: JSON payload

Output: Prediction + probability

Docker and Kubernetes Setup

Ensure Docker Desktop is running and Kubernetes is enabled:

- Docker Desktop → Settings → Kubernetes → Enable Kubernetes

Verify:

```
docker --version
```

```
docker build -t heart-disease-api .
```

```
docker run -p 8000:8000 heart-disease-api
```

```
kubectl get nodes
```