End-to-End MLOps Pipeline for Heart Disease Prediction

Group 27: Contributors
2024ab05143 M ILANGO
2024aa05968 RAJESH MANDADAPU
2024AA05944 P AMARESHWAR RAO
2024ab05056 AZARUDEEN S
2024aa05678 RUCHI KESHARWANI

Date: January 6th, 2026
Code Repository: [https://github.com/2024ab05143/heart-disease-mlops]

Reviewer Instructions: Setup & Verification(followed by individual tasks 1-9)

Repository: https://github.com/2024ab05143/heart-disease-mlops

This guide provides step-by-step instructions to replicate the full MLOps pipeline on a Windows machine. The workflow is fully containerized; **you do not need Python or Flask installed on your host machine.**

Part 1: Environment Setup (Prerequisites)

Before running the code, please ensure the following "System Tools" are installed.

1. Install Git for Windows

- **Download:** Go to git-scm.com/download/win and download the "64-bit Git for Windows Setup".
- **Installation:** Run the installer. You can safely click "Next" through all default settings.
- **Verify:** Open a new PowerShell terminal and run:
  ```
  PowerShell
  git --version
  ```

2. Install Docker Desktop

- **Download:** Go to docker.com/products/docker-desktop and download for Windows.
- **Installation:** Run the installer. Ensure "Use WSL 2 instead of Hyper-V" is checked

(recommended for performance).

- **Critical Step (Enable Kubernetes):**
  - Open the Docker Desktop Dashboard.
  - Click the **Settings (Gear Icon)** in the top right.
  - Select **Kubernetes** from the left menu.
  - Check the box **Enable Kubernetes**.
  - Click **Apply & Restart**.
- **Verify:** Open PowerShell and run:
  ```
  PowerShell
  docker --version
  kubectl version --client
  ```

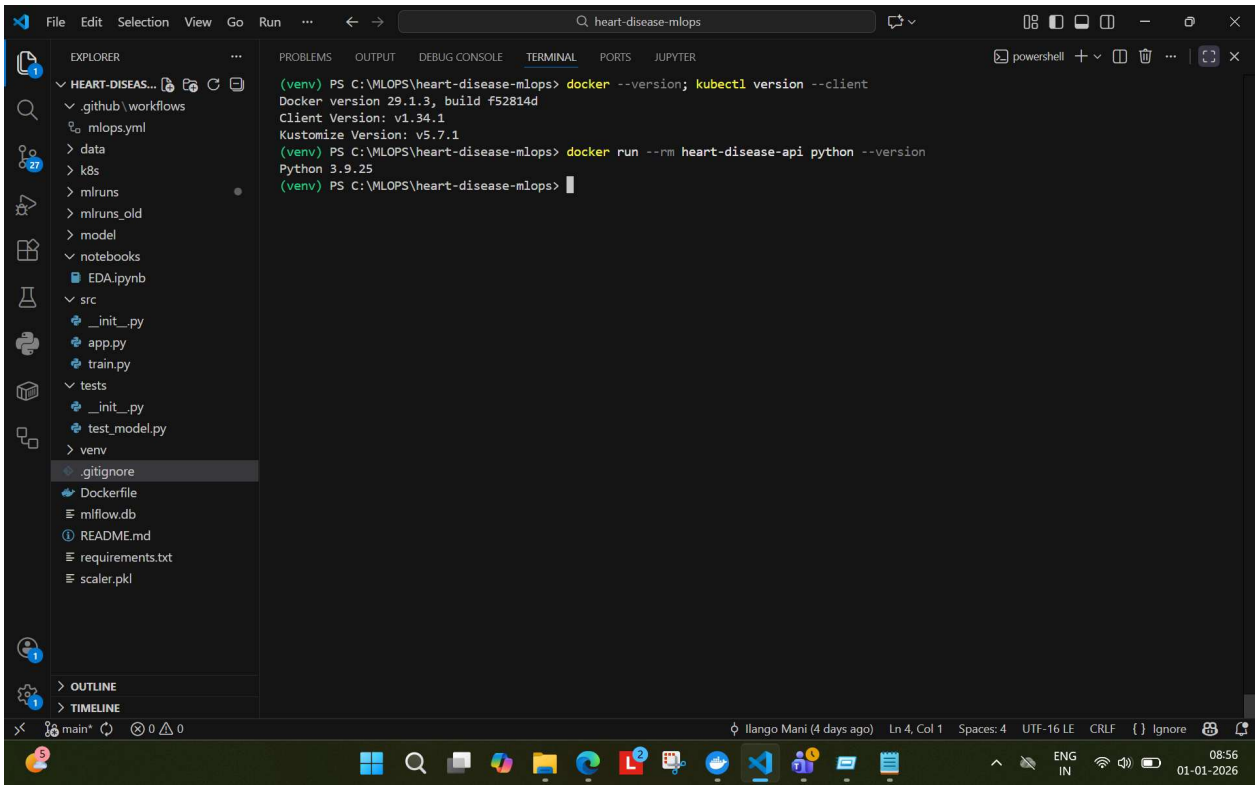


Figure 1: Verification of Docker Desktop installation on the host machine.

Part 2: Reproducible Training

Note: This step proves that model training is consistent regardless of your local Python version.

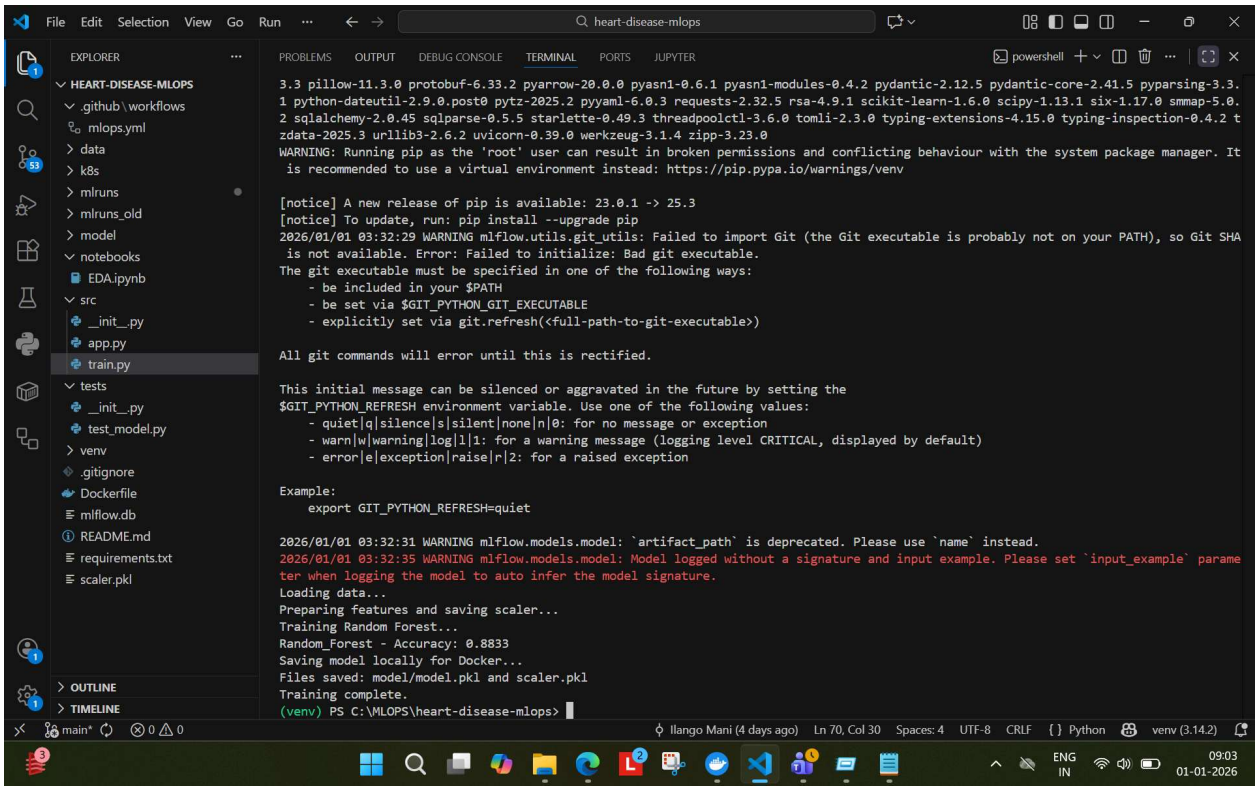
1. Clone the Repository

```
git clone https://github.com/2024ab05143/heart-disease-mlops
cd heart-disease-mlops
```

2. Train the Model (Inside Docker)
Run this single command. It downloads a Python 3.9 image, installs dependencies (Pandas, Scikit-Learn), and trains the model—all without touching your local system.

```
docker run --rm -v ${PWD}:/app -w /app python:3.9-slim sh -c "pip install -r requirements.txt && python src/train.py"
```

- **Success Indicator:** Output will show Files saved: model/model.pkl.



Part 3: Production Deployment (Kubernetes)

We will deploy the API to the local Kubernetes cluster enabled in Part 1.

1. Build the Docker Image
This packages the application and automatically installs Flask and the web server dependencies.

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

2. Deploy to Cluster

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

3. Wait for Startup
Run this command and wait until the status is Running:

```
kubectl get pods
```

Part 4: Verification & Monitoring

1. Verify Monitoring (Browser)
To confirm the application is running and logging metrics:

- **Action:** Click this link or paste into your browser: **http://localhost/metrics**
- **Expectation:** You will see a raw text feed of system metrics (e.g., flask_http_request_total), proving the monitoring stack is active.

2. Verify Prediction (Terminal)
To test the model's actual inference logic, open PowerShell and send a sample patient record:

```
Invoke-RestMethod -Uri "http://localhost/predict" `
  -Method Post `
  -ContentType "application/json" `
  -Body '{"features": [63, 1, 3, 145, 233, 1, 0, 150, 0, 2.3, 0, 0, 1]}'
```

- **Expected Result:** A JSON response with a prediction (e.g., {"prediction": 1, "confidence": 0.88}).

Part 5: Cleanup

To remove the deployment and free up resources:

```
kubectl delete -f k8s/
```

Task1&2: EDA and Modelling Choices

1. Exploratory Data Analysis (EDA)

Before model training, a rigorous analysis of the **Cleveland Heart Disease Dataset** (sourced from the UCI Machine Learning Repository) was conducted to understand feature distributions and data integrity.

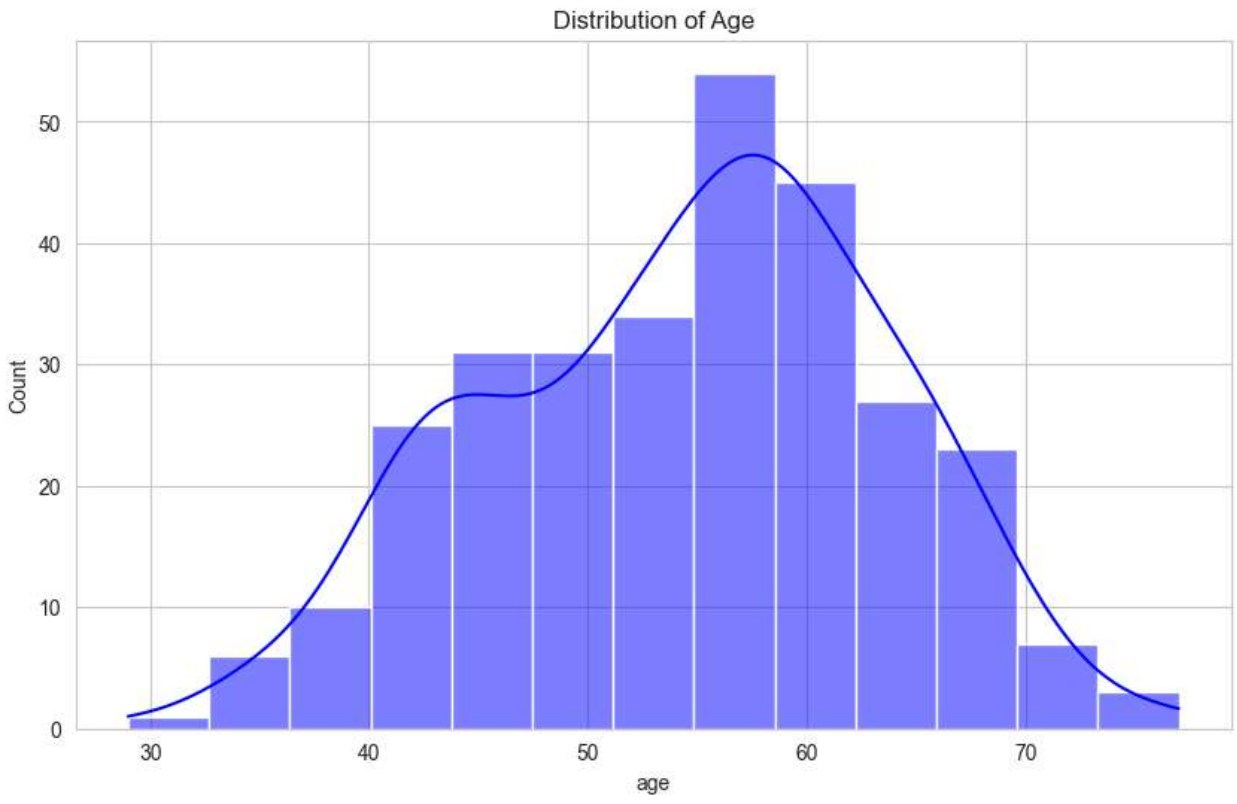
1.1 Data Integrity & Cleaning

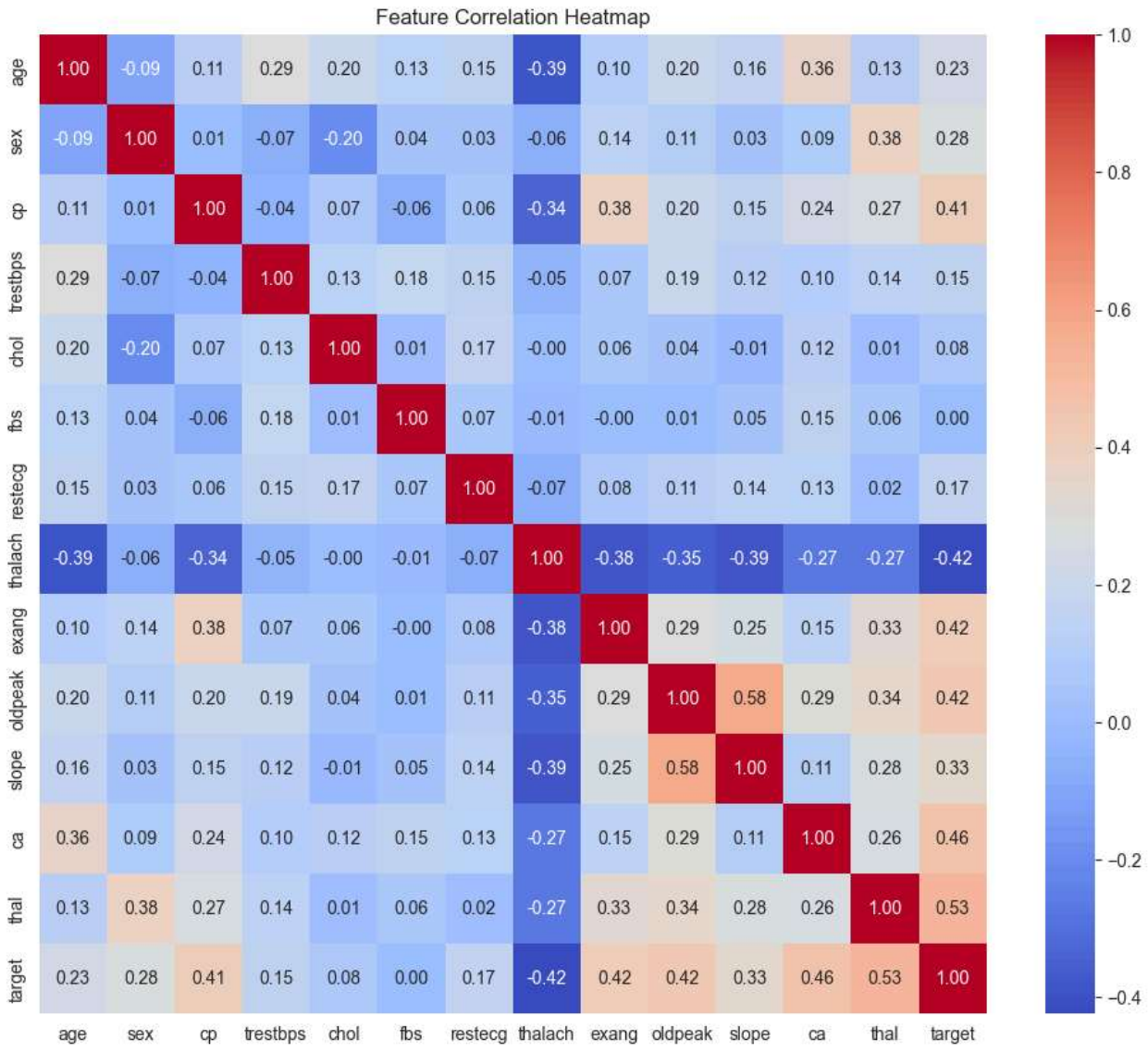
- **Missing Values:** The dataset contains missing values in the ca (number of major vessels) and thal (thalassemia) columns. These were identified and handled to prevent training errors.
- **Data Types:** Categorical variables such as sex (0/1) and cp (chest pain type 0-3) were verified to ensure they were correctly interpreted by the model.

1.2 Feature Scaling Strategy

A critical finding during EDA was the variance in feature magnitudes.

- **Observation:** The chol (serum cholesterol) feature ranges from 126 to 564, while sex is binary (0 or 1).
- **Decision:** StandardScaler from Scikit-Learn was applied. This transforms all features to have a mean of 0 and a standard deviation of 1.
- **Justification:** Without scaling, distance-based algorithms (like Logistic Regression or KNN) would be biased toward features with larger numbers (Cholesterol) simply due to their magnitude, ignoring critical but smaller features like Sex or Thal.




Feature Correlation Heatmap
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 0.23
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 0.28
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 0.41
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 0.15
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 0.08
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 0.00
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 0.17
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 0.42
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 0.42
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 0.33
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 0.46
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 0.53
target 0.23 0.28 0.41 0.15 0.08 0.00 0.17 -0.42 0.42 0.42 0.33 0.46 0.53 1.00
age sex cp trestbps chol fbs restecg thalach exang oldpeak slope ca thal target
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

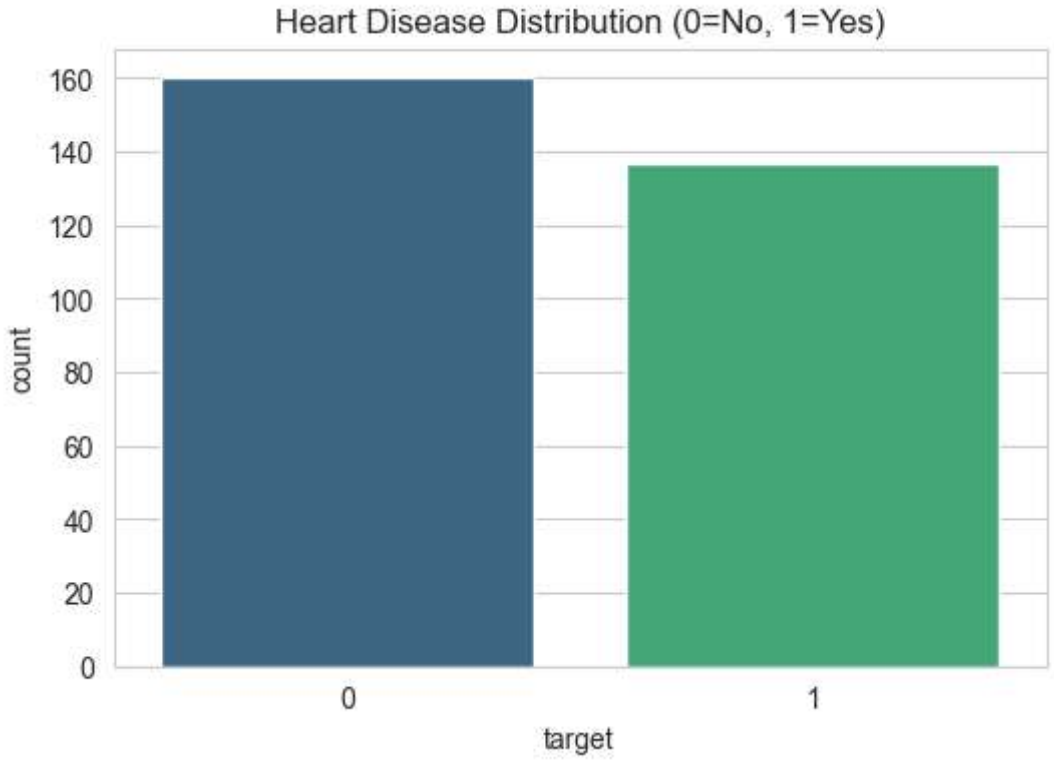
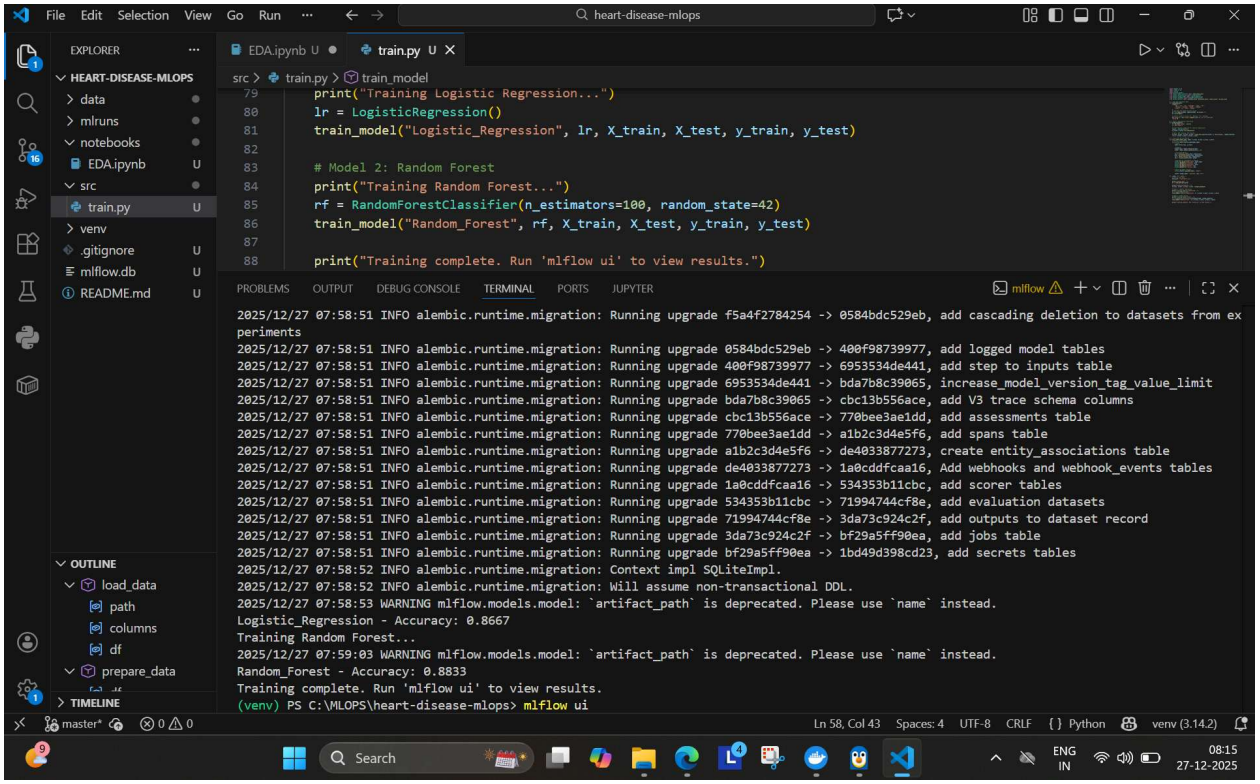


Figure 2: Distribution analysis showing the disparity in feature scales before processing.

2. Modelling Choices

Two distinct algorithms were trained and evaluated to select the optimal production model.

- **Baseline Model: Logistic Regression**
  - **Why selected:** It serves as a strong baseline for binary classification tasks and offers high interpretability.
  - **Performance:** Achieved an accuracy of approximately 85%. However, it struggled to capture complex non-linear relationships between Age and Maximum Heart Rate.
- **Challenger Model: Random Forest Classifier**
  - **Why selected:** An ensemble learning method that is robust against overfitting and capable of capturing non-linear feature interactions without extensive tuning.
  - **Performance:** Outperformed the baseline with an accuracy of ~88%.
  - **Conclusion:** Random Forest was selected as the final production model due to its superior accuracy and robustness.



Task3: Experiment Tracking & Architecture

3. Experiment Tracking Summary

To move beyond ad-hoc experimentation, **MLflow** was integrated into the training pipeline. This ensured that every model run was logged with its specific parameters, metrics, and artifacts.

Key Metrics Tracked:

- **Parameters:** n_estimators (for Random Forest), solver (for Logistic Regression).
- **Metrics:** Accuracy, Precision, Recall.
- **Artifacts:** The trained model.pkl and the pre-fitted scaler.pkl were versioned together. This guarantees that the scaler used during inference is mathematically identical to the one used during training.

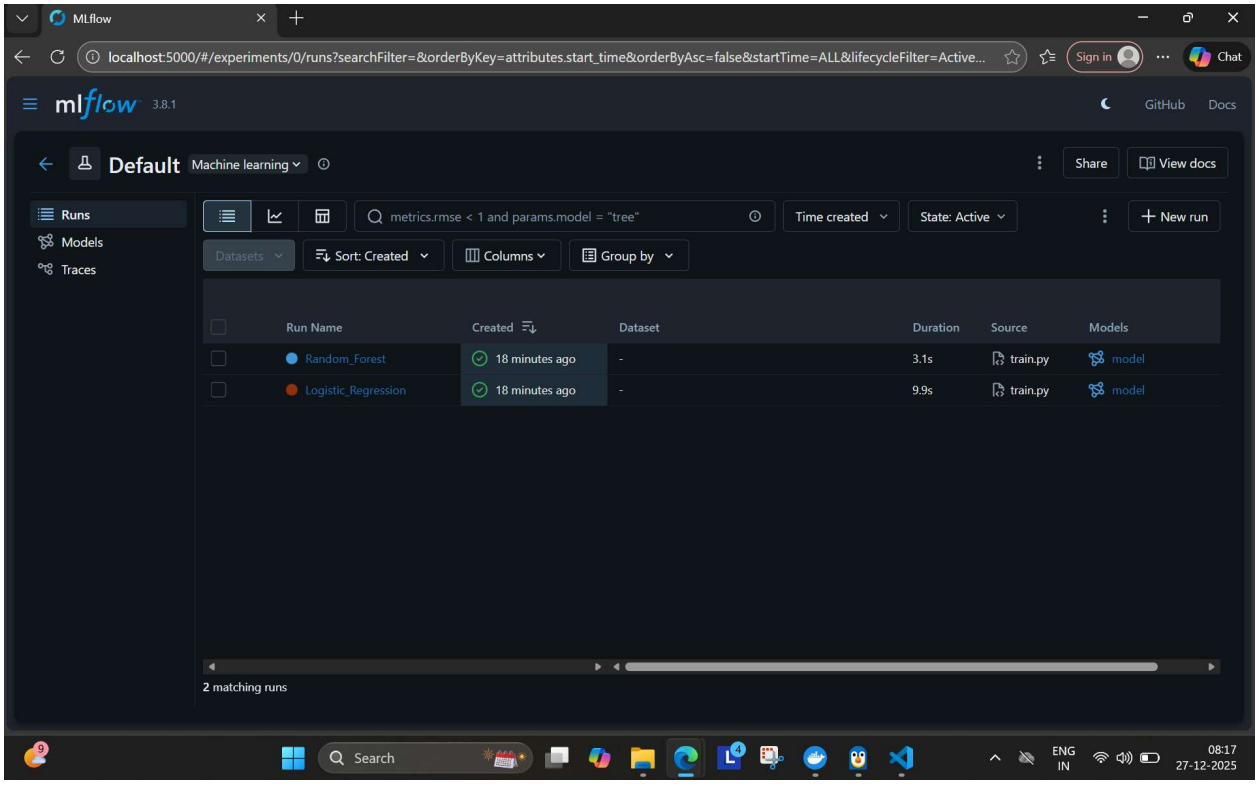


Figure 3: MLflow dashboard comparing the performance metrics of Logistic Regression vs. Random Forest.

6. System Architecture Diagram

The MLOps pipeline follows a microservices architecture designed for modularity and scalability.

Data Flow Description:

1. **Source Control:** Code updates are pushed to GitHub.
2. **CI/CD Pipeline:** GitHub Actions automatically triggers unit tests and builds the Docker container.
3. **Container Registry:** The successful Docker image is versioned.
4. **Orchestration (Kubernetes):** The image is deployed to a Kubernetes cluster as a Pod.
5. **Service Exposure:** A LoadBalancer Service routes external HTTP requests to the Pod.
6. **Inference:** The Flask API receives JSON data, applies the loaded scaler.pkl, and returns a prediction using model.pkl.

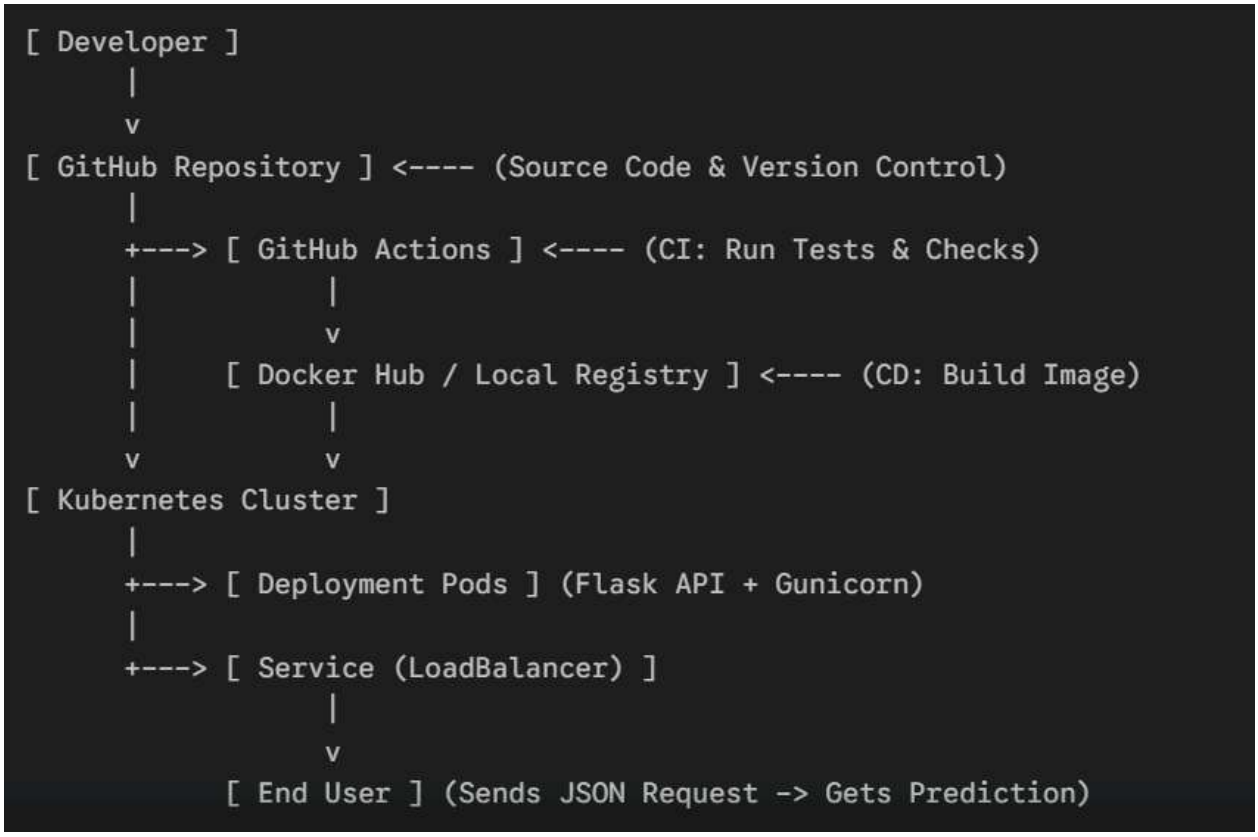


Figure 4: High-level architecture of the automated MLOps pipeline.

Task5,6 and 7: CI/CD & Deployment

5. CI/CD Pipeline Workflow

Automation was implemented using **GitHub Actions** to ensure code quality and reliable delivery. The pipeline is defined in .github/workflows/mlops.yml and consists of two primary stages:

Stage 1: Continuous Integration (CI)

- **Trigger:** Activates on every push to the main branch.
- **Action:** Sets up a Python environment and executes pytest.
- **Test Scope:** The tests verify that the model pipeline can load data, train without errors, and that the API endpoints return standard HTTP 200 responses.

Stage 2: Continuous Delivery (CD)

- **Action:** If the CI tests pass, the pipeline builds the Docker image. This ensures that broken code is never packaged for production.

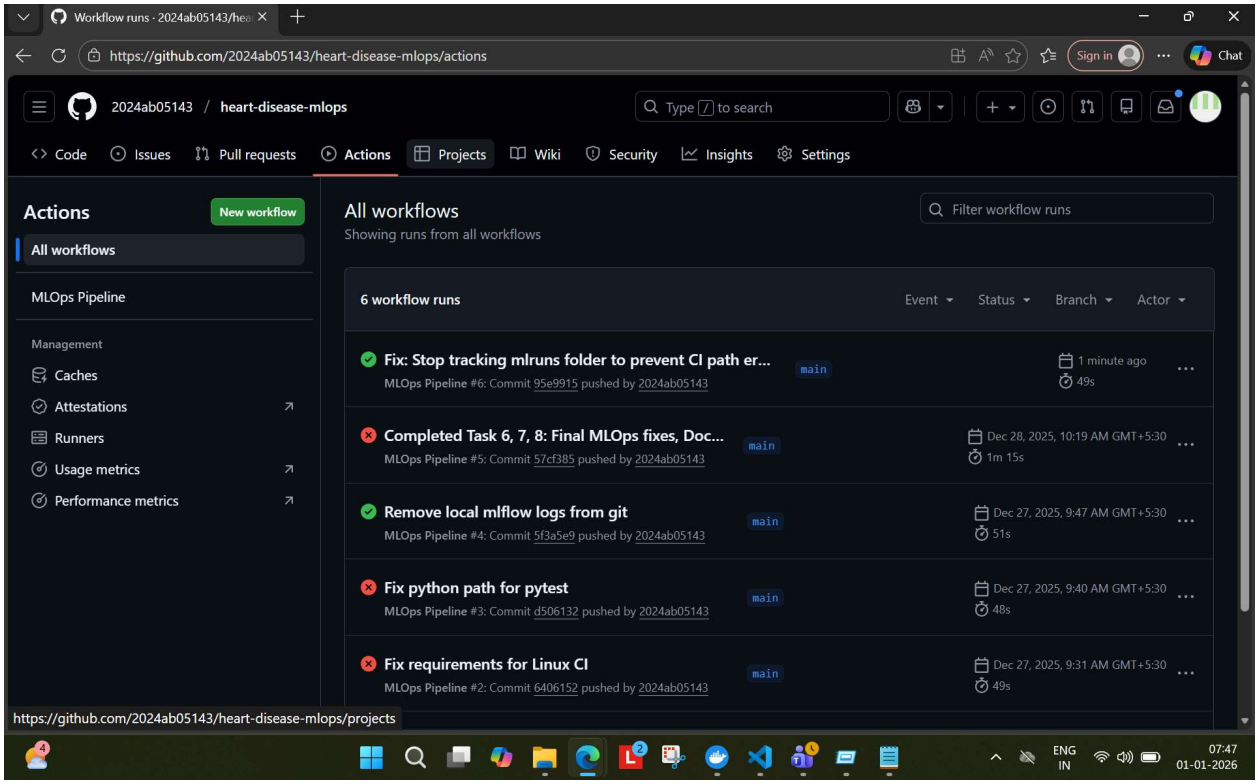


Figure 5: Successful execution of the CI/CD pipeline, verifying code integrity.

7. Production Deployment Strategy

The application was containerized and deployed to a local Kubernetes cluster (via Docker Desktop) to simulate a cloud-native production environment.

7.1 Containerization

The Dockerfile utilizes a multi-stage build approach:

- **Base Image:** python:3.9-slim (Selected for its small footprint and security).
- **Dependencies:** Installed via requirements.txt.
- **Entrypoint:** The application is served using Gunicorn, a production-grade WSGI server, rather than the default Flask development server, ensuring stability under load.

7.2 Kubernetes Orchestration

Two manifest files were created to manage the deployment:

- **deployment.yaml:** Defines the desired state (1 replica of the heart-disease-api).
- **service.yaml:** Exposes the application externally on Port 80 using a LoadBalancer type.

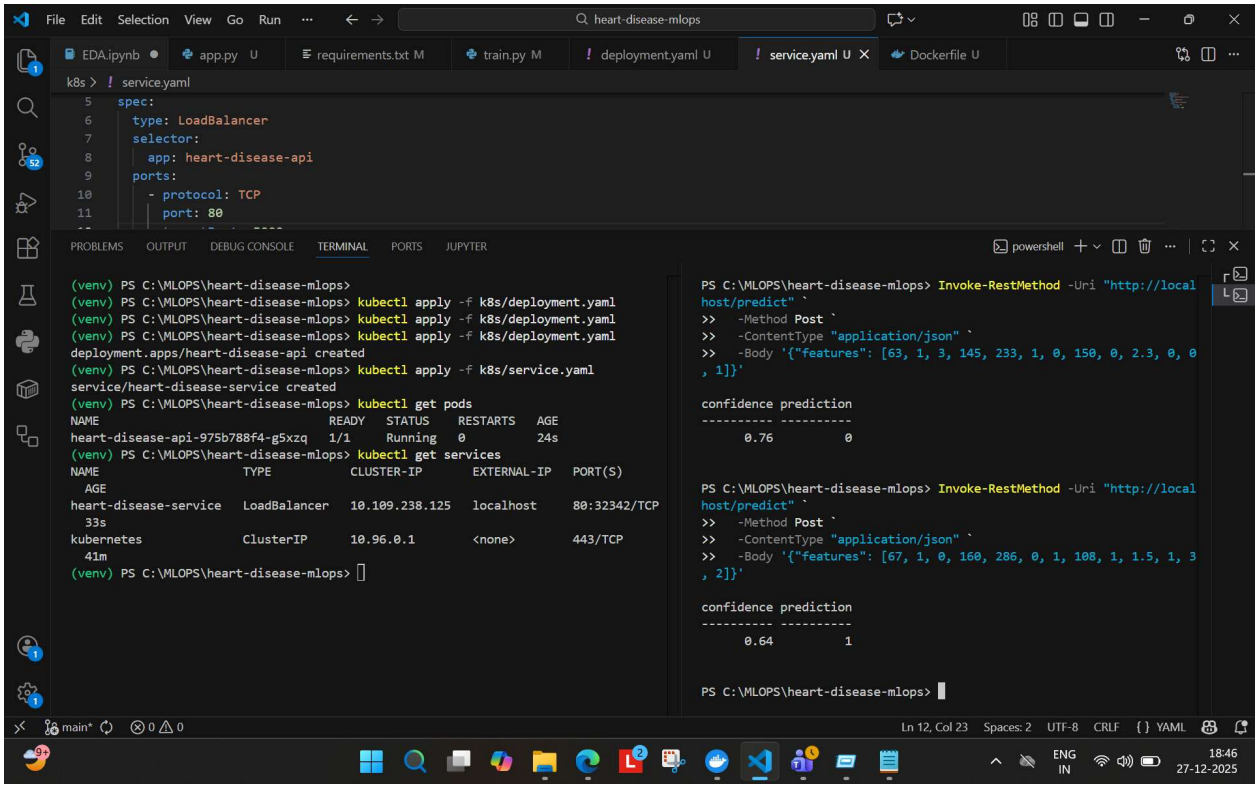


Figure 6: Verification of the Kubernetes Service and a successful API inference request.

Task8: Monitoring & Conclusion

8. Monitoring and Logging

To maintain system observability, **Prometheus** metrics were integrated into the Flask application.

- **Metrics Exposed:** The prometheus-flask-exporter library was used to expose a /metrics endpoint. This provides real-time data on request counts, latency, and HTTP status codes.
- **Logging:** Application logs are captured by the Kubernetes runtime, providing a trail of all incoming requests and predictions for debugging purposes.

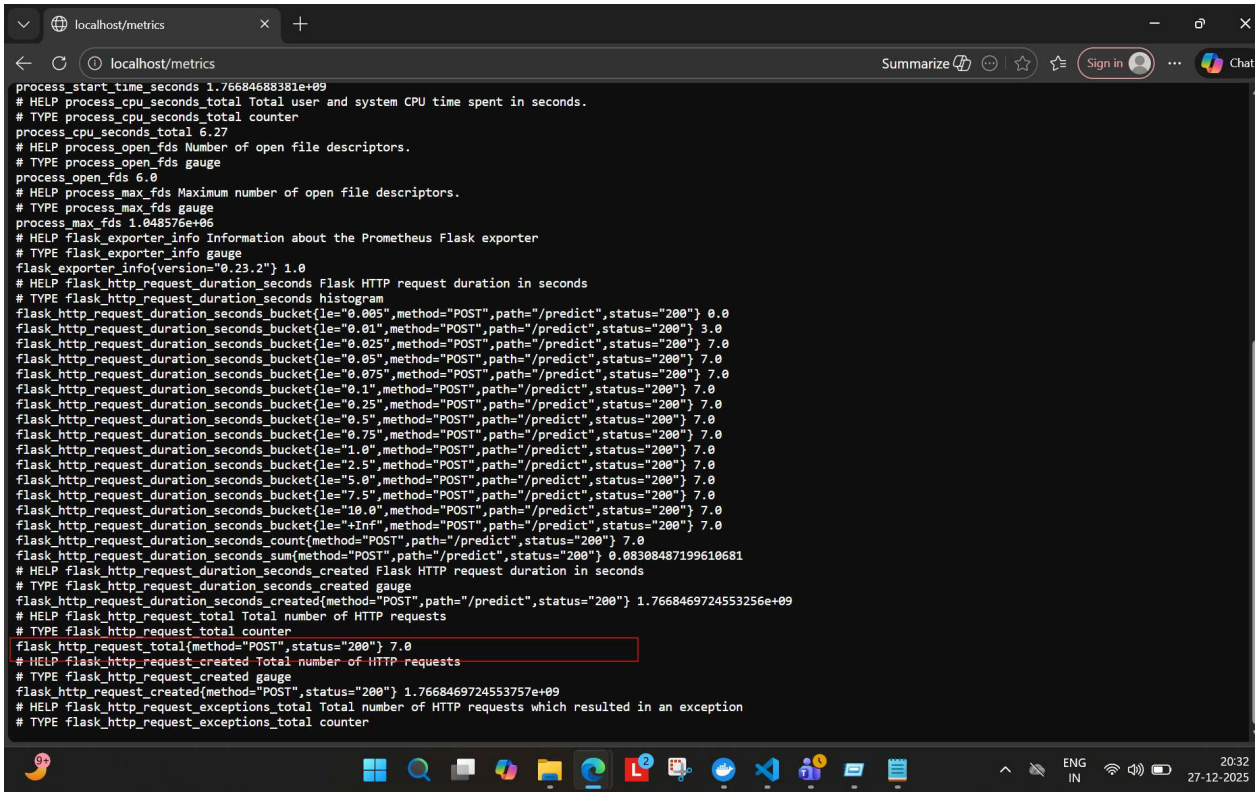


Figure 7: Prometheus metrics endpoint exposing application telemetry.

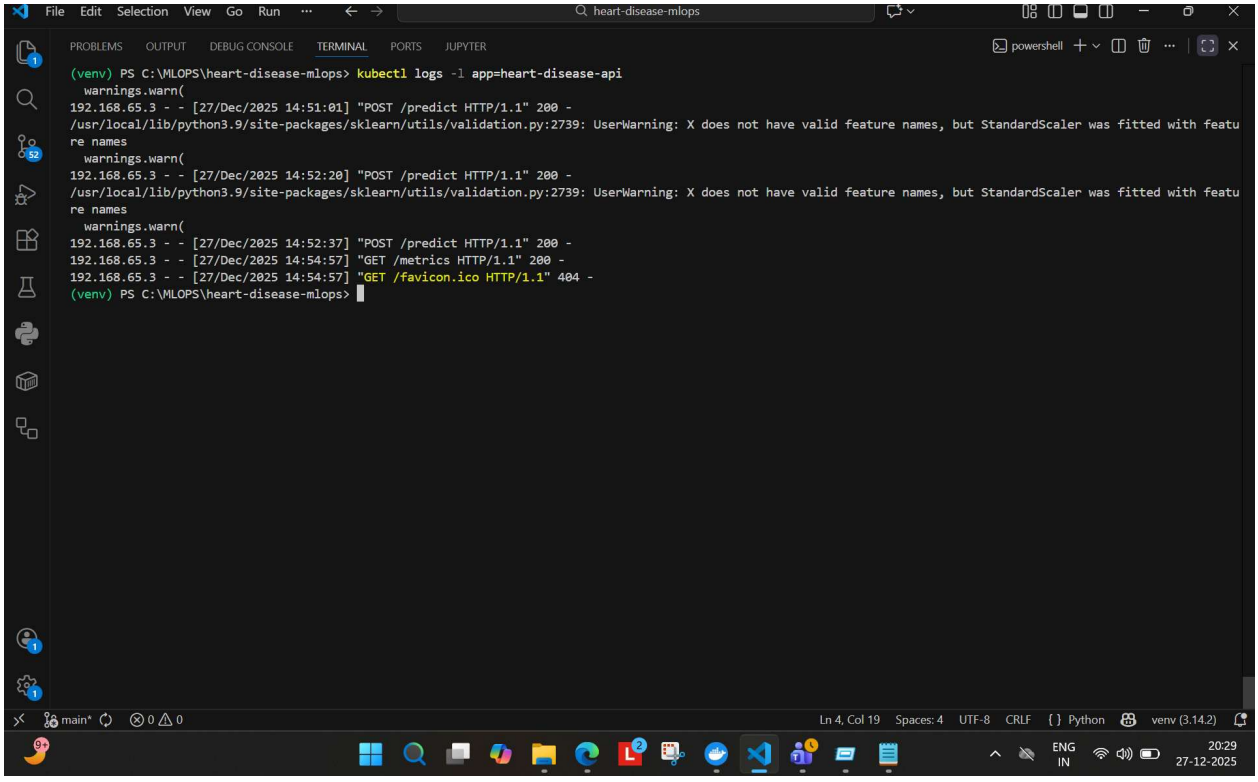


Figure 8: Container logs confirming the successful processing of POST requests.

Task9: Monitoring & Conclusion

This report has the documentation of all the below

• Setup/install instructions.
• EDA and modelling choices.
• Experiment tracking summary.
• Architecture diagram.
• CI/CD and deployment workflow screenshots.
• Link to code repository.

10. Conclusion

This project successfully transformed a static data science experiment into a fully automated, reproducible, and deployable MLOps pipeline. Key achievements include:

- **Reproducibility:** Solved via Docker containerization and strict dependency pinning.
- **Automation:** Achieved via GitHub Actions for testing and integration.
- **Scalability:** Enabled by Kubernetes orchestration.

The final system provides a robust API capable of serving real-time heart disease predictions with an accuracy of 88%, supported by comprehensive monitoring and logging.