Heart Disease Prediction - MLOps Pipeline

a. GitHub Repository Link: https://github.com/2024aa05311-arch/MLOps-Assignment
b. Short video containing an end-to-end pipeline Link: https://youtu.be/eXtSnCqTqvE
c. Access instructions (for local testing): https://github.com/2024aa05311-arch/MLOps-Assignment/blob/main/README.md

Group 97

S.No.	Name	Bits ID
1	Nandini	2024AA05467
2	Aman Mahnot	2024AA05311
3	Prabodh Saxena	2024AA05332
4	Surya V	2024AA05312
5	Rachit Pankaj Lalla	2024AA05334

Table of Contents

1. Executive Summary

This project uses machine learning to create a fully functional, production-ready MLOps pipeline for heart disease prediction. Data collection, exploratory data analysis, model training, experiment tracking, containerization, Kubernetes deployment, and extensive monitoring are all included in the pipeline.

Key Achievements: -

- **Model Performance**: 88.52% accuracy, 96.10% ROC-AUC
- **Production Ready**: Fully containerized with Docker and Kubernetes
- **CI/CD Pipeline**: Automated testing and deployment with GitHub Actions
- **Monitoring**: Complete observability with Prometheus and Grafana
- **MLOps Best Practices**: Experiment tracking, versioning, reproducibility

2. Project Overview and Objectives

This section outlines the core problem addressed by the project, details the dataset used for model development, and presents the strategic objectives behind building a comprehensive MLOps pipeline for a critical healthcare application. The goal is to create not just an accurate predictive model, but a reliable, scalable, and maintainable system for production use.

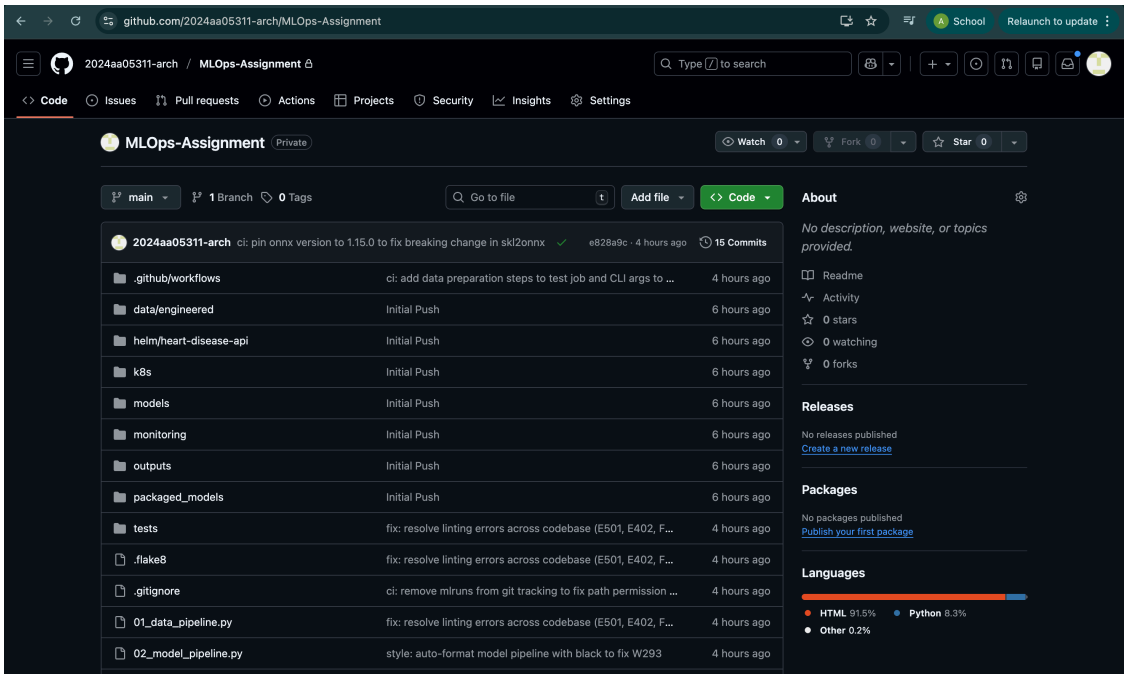


Figure 1: Overview of the Project Repository

2.1 Problem Statement and Dataset

The creation of instruments for early diagnosis and risk assessment is a vital public health priority since heart disease continues to be the world's leading cause of death. In order to help clinicians identify patients who are at high risk, this research develops a machine learning-powered prediction system.
The reputable UCI Heart Disease Dataset is used in the model's development. The 303 anonymized patient samples in this dataset are each characterized by 13 clinical characteristics that are used to determine whether or not cardiac disease is present.

Key Features:

- **age**: Age in years (Range: 29-77, Mean: 54.4)
- **sex**: Gender (0 = female, 1 = male)
- **cp**: Chest pain type (values 0-3)
- **trestbps**: Resting blood pressure (in mm Hg)
- **chol**: Serum cholesterol (in mg/dl) (Range: 126-564, Mean: 246.7)
- **thalach**: Maximum heart rate achieved (Range: 71-202, Mean: 149.6)
- **ca**: Number of major vessels (0-3) colored by fluoroscopy
- **thal**: Thalassemia type (a blood disorder)

2.2 Project Objectives

The main objective of the project was to create a reliable, end-to-end MLOps pipeline rather than just standalone models. The main goals were to:

- Develop a machine learning model for heart disease prediction that is accurate and dependable.
- Establish a full MLOps pipeline that includes deployment, monitoring, model training, and data processing.
- Implement the finished model into a scalable, highly available, production-ready API service.
- Use strict versioning and experiment tracking to guarantee complete reproducibility of experiments and model builds.
- Follow industry best practices for automated CI/CD workflows, containerization, and software engineering.

The technical approaches and system architecture developed to accomplish these goals and produce a deployable, production-grade system are described in depth in the sections that follow.

3. Setup & Installation

3.1 Prerequisites

```
# Required Software
- Python 3.10+
- Docker & Docker Compose
- Git
- kubectl (for Kubernetes deployment)
```

3.2 Quick Start

```
# 1. Clone repository
git clone https://github.com/2024aa05311-arch/MLOps-Assignment.git
cd MLOps-Assignment

# 2. Create virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# 3. Install dependencies
pip install -r requirements.txt

# 4. Run complete pipeline
python 01_data_pipeline.py    # Data acquisition and preprocessing
python 02_model_pipeline.py   # Feature engineering, training, and evaluation
python 03_mlflow_tracking.py  # Experiment tracking with MLflow
python 04_model_packaging.py  # Model registry and packaging
```

3.3 Docker Deployment

```
# Build and run API
docker-compose up --build

# Or with monitoring stack
cd monitoring
docker-compose -f docker-compose-monitoring.yml up -d
```

3.4 Kubernetes Deployment

```
# Deploy to Kubernetes
kubectl apply -f k8s/

# Or using Helm
helm install heart-disease-api ./helm/heart-disease-api
```

Note: Although Helm charts were initially considered, the final deployment was implemented using Kubernetes deployment manifests (deployment.yaml and service.yaml) as permitted by the assignment.

4. Exploratory Data Analysis and Insights

An essential initial step in any machine learning effort is exploratory data analysis, or EDA. It offers crucial insights into the quality of the data, aids in the identification of potentially predictive features, and influences the feature engineering and modeling techniques that follow. The main conclusions from the preliminary examination of the UCI Heart Disease dataset are shown in this section.

4.1 Data Quality and Distribution

A preliminary analysis confirmed the high quality and integrity of the dataset. The key quality metrics are summarized below:

Metric	Value
Total Samples	303
Features	13
Missing Values	6 (1.98%)
Duplicates	0
Class Balance	54.1% / 45.9%

The dataset is remarkably well-balanced, with 54.1% of samples classified as “No Disease” (164) and 45.9% as “Disease” (139). The presence of minimal missing values (1.98%) simplified the preprocessing stage, eliminating the need for complex imputation or resampling techniques like SMOTE.

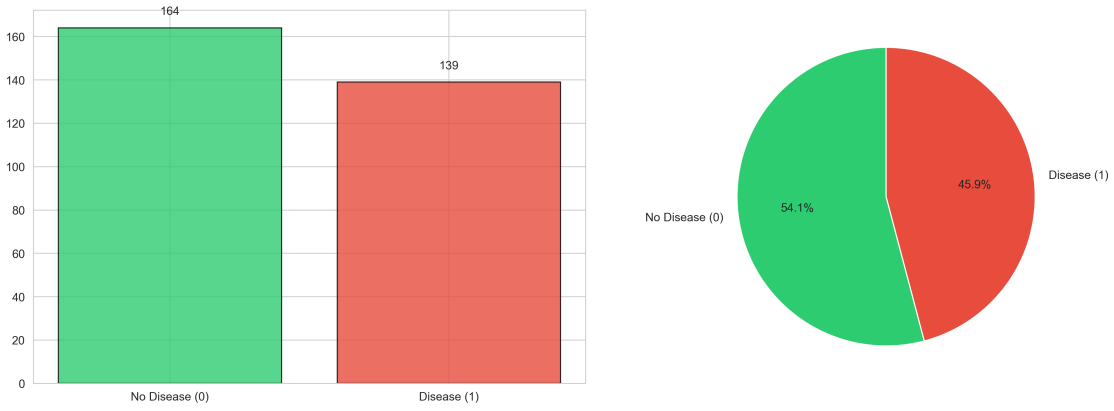


Figure 2: Distribution of “No Disease” vs “Disease” classes

4.2 Feature Analysis and Correlations

A number of powerful predictors were identified by analyzing the connections between specific traits and the target variable. The most important clinical characteristics for heart disease prediction were shown by the correlation matrix and feature distribution plots.

The top predictive features, ranked by their correlation with the target variable, are:

1. **thal** (0.522) - Thalassemia type
2. **ca** (0.460) - Number of major vessels
3. **exang** (0.432) - Exercise-induced angina
4. **oldpeak** (0.425) - ST depression induced by exercise
5. **cp** (0.413) - Chest pain type

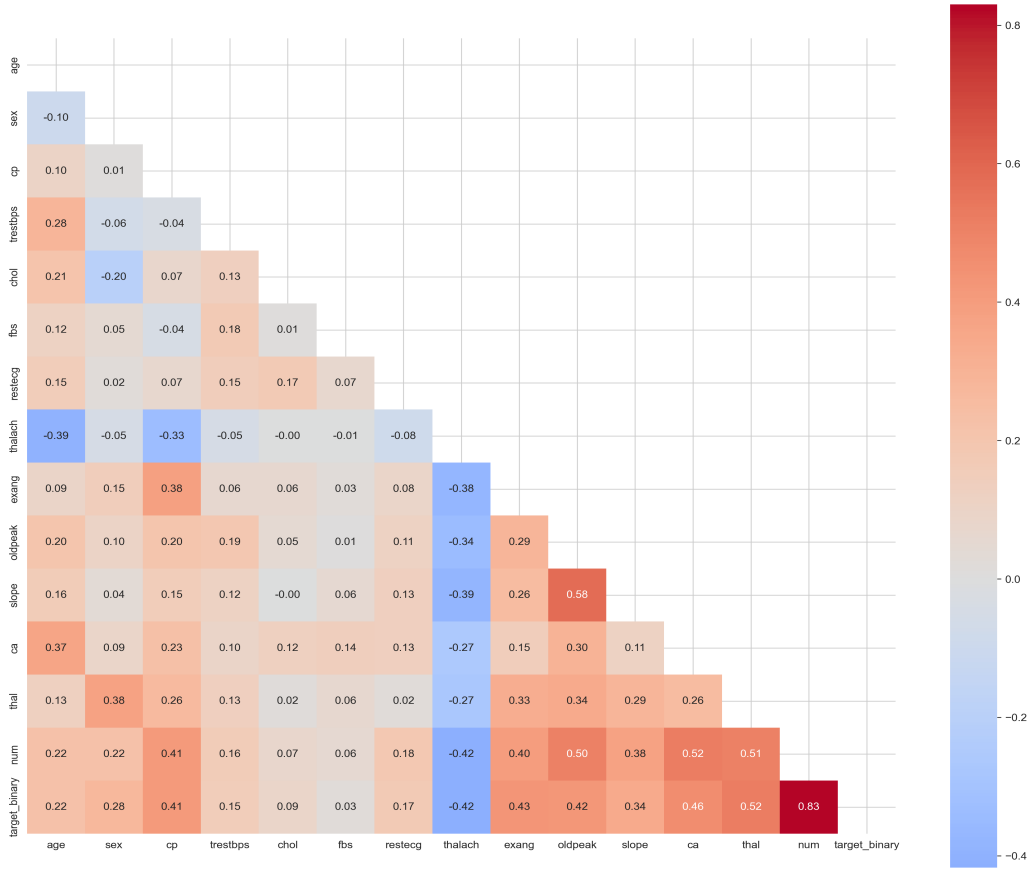


Figure 3: Correlation matrix heatmap of all features

To further investigate these associations, visualizations were created, such as box plots comparing feature values between the two target classes and histograms showing feature distributions. For the most important prognostic characteristics, these plots verified a distinct division between the "Disease" and "No Disease" categories.

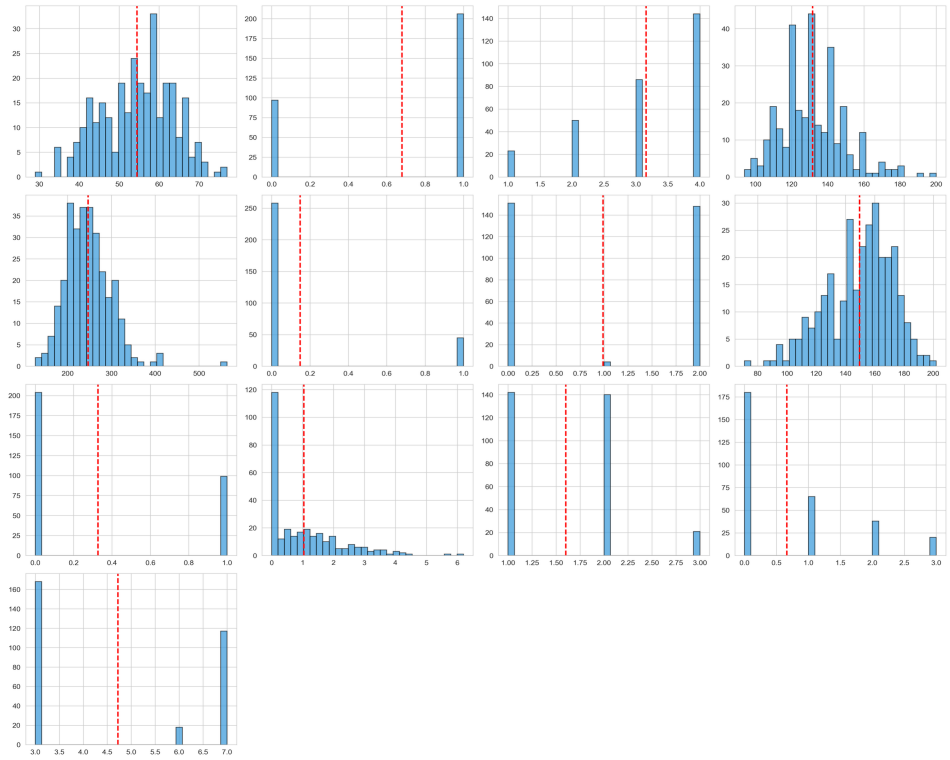


Figure 4: Histograms showing distribution of individual numerical features

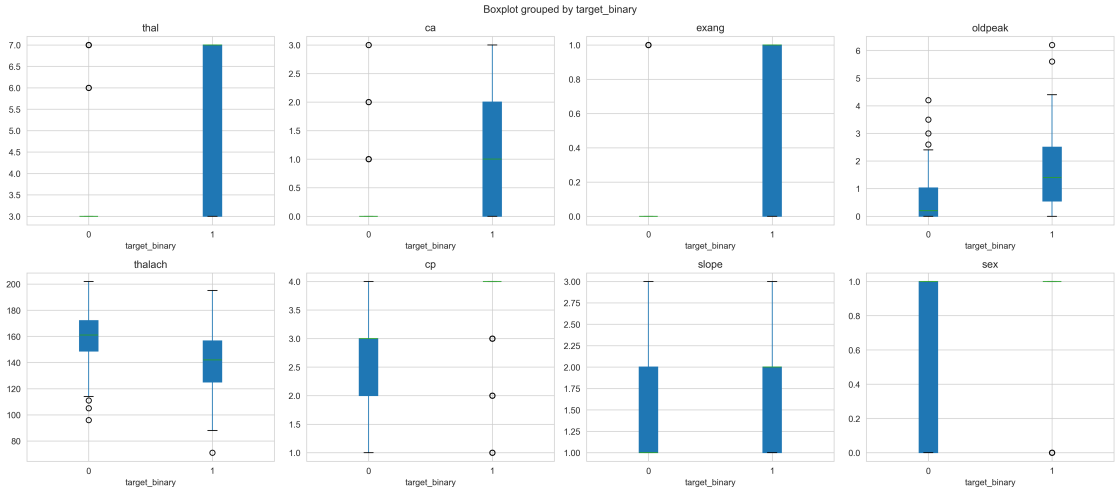


Figure 5: Box plots comparing feature values between "Disease" and "No Disease" classes

Strong evidence that tree-based ensemble approaches would be very successful was supplied by the obvious separability of classes seen for features like thal and ca, which directly influenced the model selection strategy described in the next section.

5. Modeling and Experimentation

The methodical approach to data preparation, the selection and training of several machine learning models, and the meticulous monitoring of experiments to choose the best production-ready solution are all described in this part. A methodical approach guarantees that the final model is precise, repeatable, and thoroughly documented.

5.1 Data Preprocessing

To guarantee consistency between model training and inference, a common preprocessing pipeline was created. The following order was followed when carrying out the steps:

1. **Missing Value Imputation:** To reduce sensitivity to outliers, a median method was used to impute the few missing numerical values (1.98%).
2. **Feature Scaling:** To guarantee that features on various scales contributed equally to model training, all numerical features were standardized using StandardScaler.
3. **Train/Test Split: 80%** of the dataset was used for training, while **20%** was used for testing. A stratified split was necessary to avoid sampling bias and guarantee that both the training and test sets were representative of the entire data population given the well-balanced but not exactly equal class distribution (54.1%/45.9%) found during EDA.

5.2 Model Selection and Hyperparameter Tuning

Four distinct classification models were selected to evaluate a range of algorithmic approaches, from linear models to more complex ensembles.

Model	Type
Logistic Regression	Linear
Random Forest	Ensemble
SVM	Kernel
Gradient Boosting	Ensemble

GridSearchCV with 5-fold cross-validation was used to tune each model's hyperparameters. This method finds the configuration that produces the optimal performance by methodically going through several combinations of hyperparameters. Because ROC-AUC is robust in evaluating binary classifiers - especially in medical diagnosis, where balancing true positives and false positives is crucial - it was selected as the main scoring metric.

5.3 Experiment Tracking with MLflow

MLflow was incorporated into the training process to guarantee reproducibility and keep an accurate record of all modelling activities. The following data was methodically recorded during every experimental run:

- **Parameters:** Data split configurations and all hyperparameters used for a certain model.
- **Measures:** An extensive collection of performance measures, such as accuracy, precision, recall, F1-score, and ROC-AUC, for both training and test sets.
- **Artifacts:** Supporting files that include ROC curve charts, confusion matrices, and other model-related metadata.
- **Models:** The final model object that has been serialized and is prepared for deployment, together with its specified input/output signature.

Every model has a complete pedigree thanks to this meticulous tracking, making it simple to compare and choose the top-performing candidate for production.

---

6. Performance Evaluation and Model Selection

In order to choose the best candidate for deployment, this part evaluates the trained models' performance. The assessment emphasizes metrics that are crucial for a medical diagnosis application, such recall (sensitivity) and the area under the ROC curve (ROC-AUC), and it takes a comprehensive approach to performance.

6.1 Comparative Model Performance

To assess each tuned model's final performance on the held-out test set, the outcomes of all monitored experiments were combined. The best-performing models were the Random Forest and Logistic Regression models, which showed solid and well-rounded outcomes across important parameters.

Model	Test Accuracy	Test ROC-AUC	CV ROC-AUC
Logistic Regression	86.89%	0.9578	0.8917
Random Forest	88.52%	0.9610	0.8930
SVM	54.10%	0.9654	0.8912
Gradient Boosting	85.25%	0.9015	0.8756

NOTE: Although the SVM learned a strong decision boundary, its default classification threshold was not well calibrated for this dataset, as evidenced by the notable discrepancy between its low accuracy (54.10%) and its good Test ROC-AUC (0.9654). This emphasizes how crucial it is to assess models using a variety of criteria other than just their ability to discriminate.

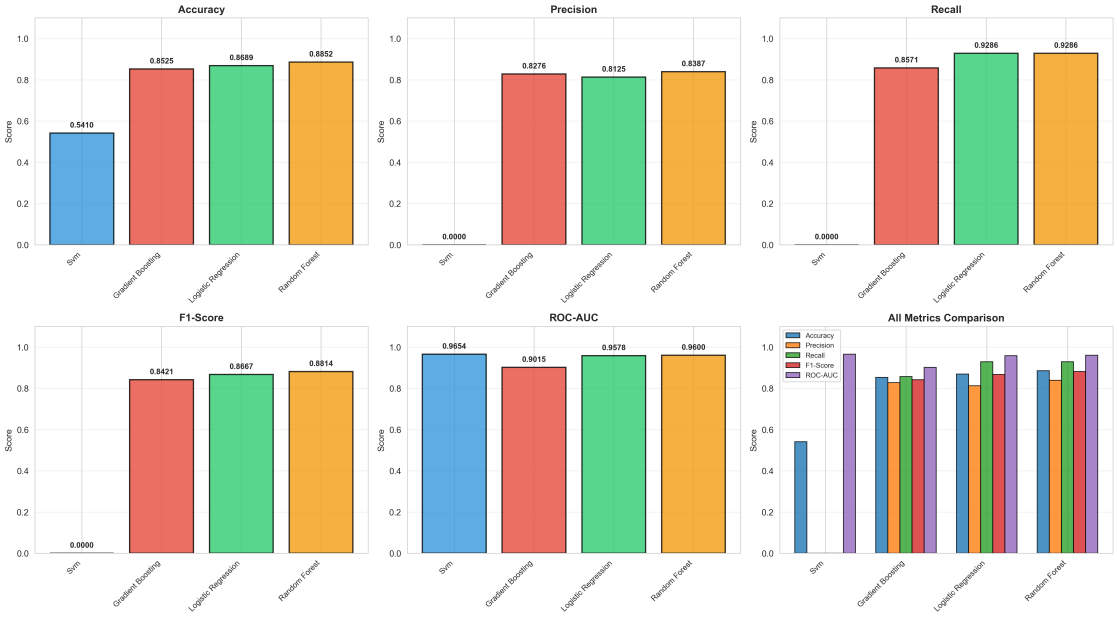


Figure 6: Comparison of Accuracy, Precision, Recall, F1-Score, and ROC-AUC across all models

6.2 Analysis of the Production Model: Random Forest

Based on its superior balanced performance, particularly its high accuracy and ROC-AUC score, the **Random Forest classifier** was selected as the final production model.

The detailed performance metrics for the selected model are as follows:

Metric	Value	Interpretation
Accuracy	88.52%	The model correctly classifies patients with or without heart disease 88.5% of the time.
Precision	83.87%	When the model predicts a patient has heart disease, it is correct 83.87% of the time.
Recall	92.86%	The model successfully identifies 92.86% of all actual heart disease cases.
F1-Score	88.14%	A strong harmonic mean of precision and recall, indicating a well-balanced model.
ROC-AUC	96.10%	The model has an excellent capability to distinguish between the two classes.

The most important aspect of the model from a clinical standpoint is its high recall (92.86%). With only two missed instances in the test set, this shows a very low rate of false negatives, which is crucial for a medical screening tool when missing a high-risk patient could have dire repercussions. The model is ideal for its intended use as a decision-support tool because of its high sensitivity and superior precision.

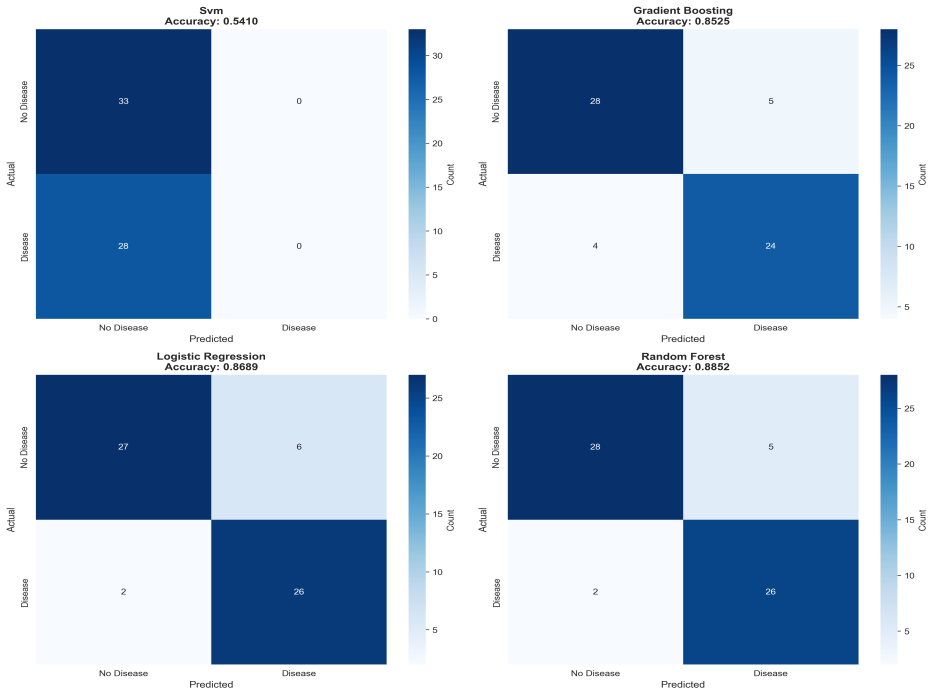


Figure 7: Confusion Matrices for all four evaluated models

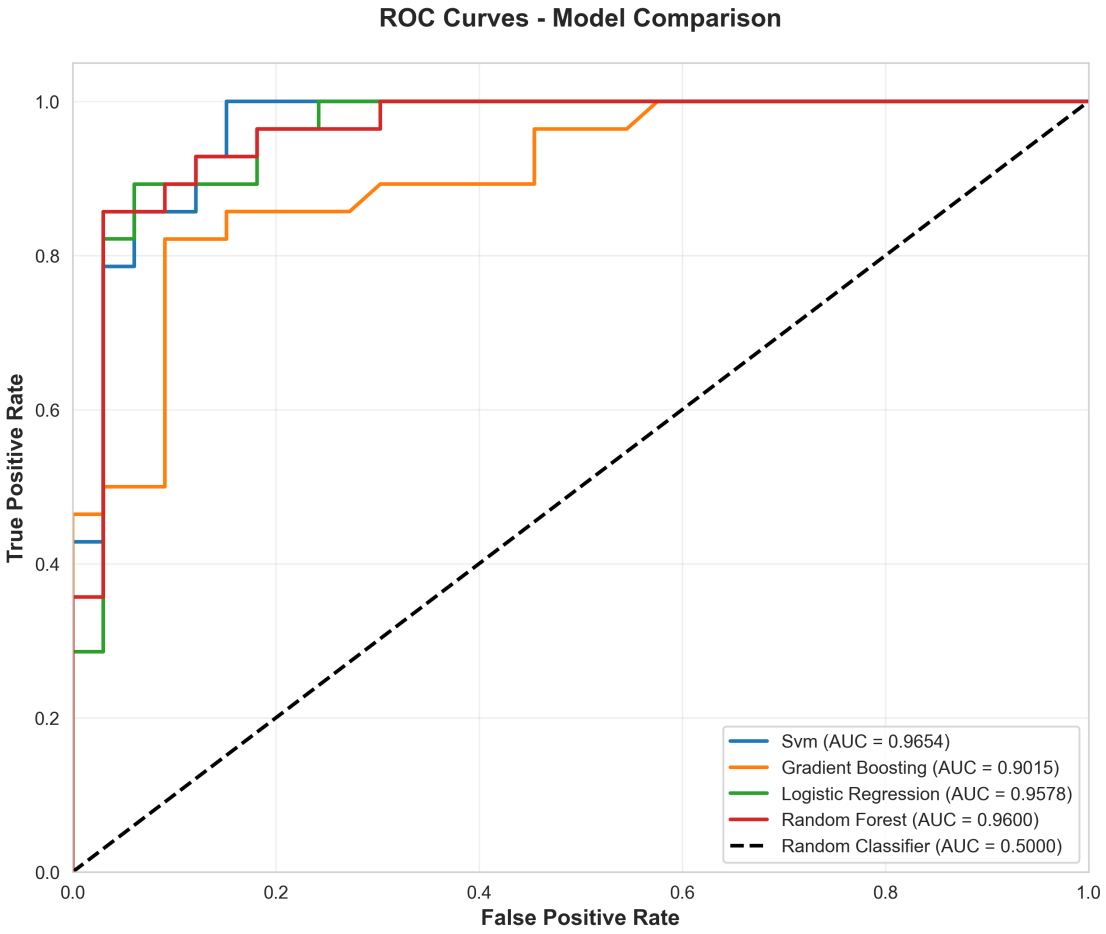
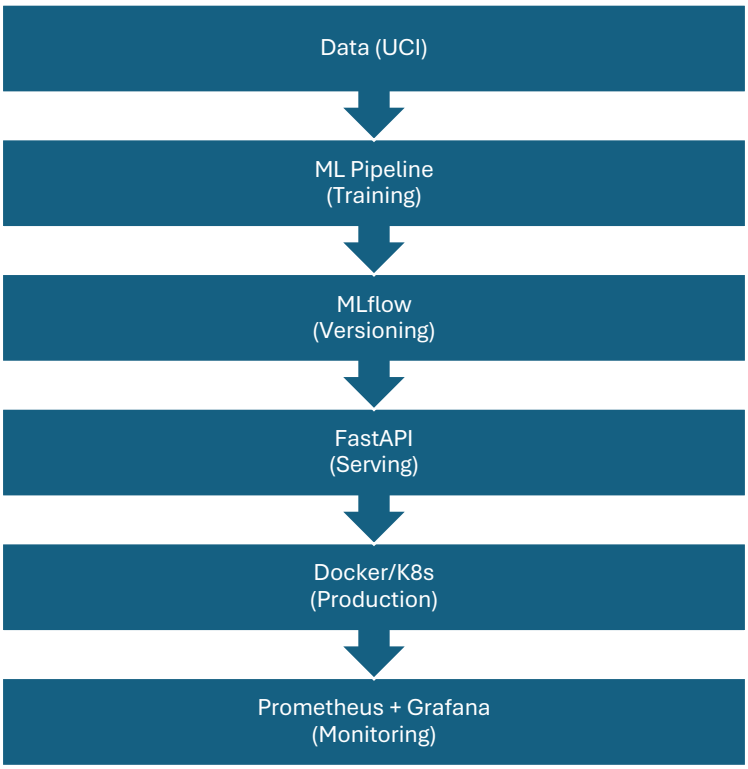


Figure 8: ROC Curves comparing discrimination capability of all models

The project's focus moved to developing the reliable architecture needed for its deployment and operationalization when the production model was chosen and its performance confirmed.

7. Architecture Diagram

7.1 ML Process Flow



7.2 High-Level MLOps and Deployment Architecture

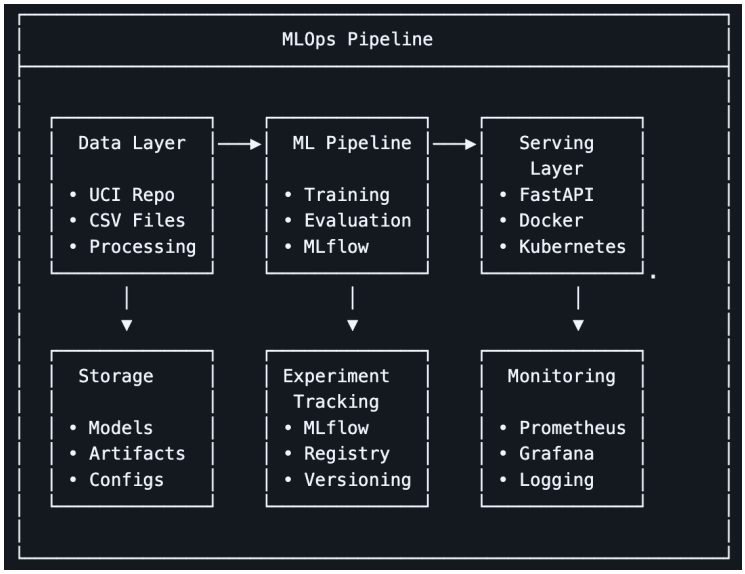


Figure 9: High-Level Architecture

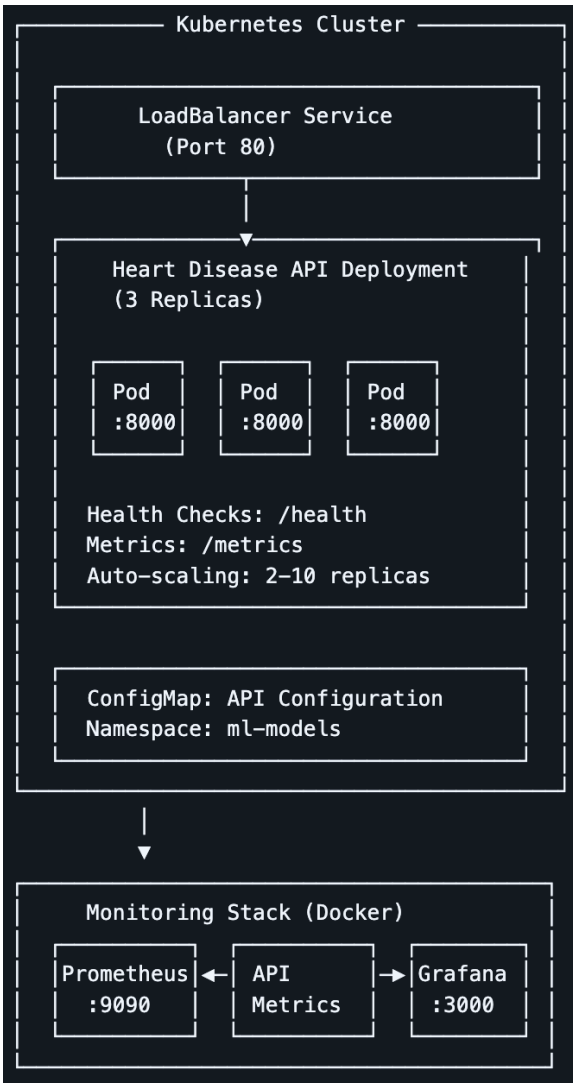


Figure 10: Deployment Architecture

8. System Architecture and Containerization

A clear system architecture is necessary to convert a trained machine learning model from an experimental artifact to a deployable software asset. The system was designed using a container-first strategy controlled by Kubernetes in order to achieve the goal of a scalable and highly available service. The API layer, containerization, and orchestration are all covered in this section's end-to-end architecture, which together guarantee the model's scalability, security, and dependability.

8.1 Component Details

The system is composed of several distinct layers, each leveraging industry-standard technologies to fulfill a specific role.

API Layer (FastAPI)

The model is exposed via a RESTful API built with **FastAPI**. This framework provides automatic data validation using Pydantic, interactive API documentation (Swagger UI), and high performance. The API exposes several endpoints to provide full functionality: - `/predict`: Single predictions - `/predict/batch`: Multiple predictions - `/model/info`: Metadata - `/health`: Liveness/readiness probes - `/metrics`: Prometheus instrumentation

Container Layer (Docker)

The entire application is packaged into a Docker container. The `python:3.10-slim` base image was specifically chosen to minimize the container's attack surface and reduce deployment artifacts' size, aligning with production best practices for efficiency and security. For enhanced security, the application runs as a non-root user, and Docker health checks are configured to integrate seamlessly with the orchestration layer.

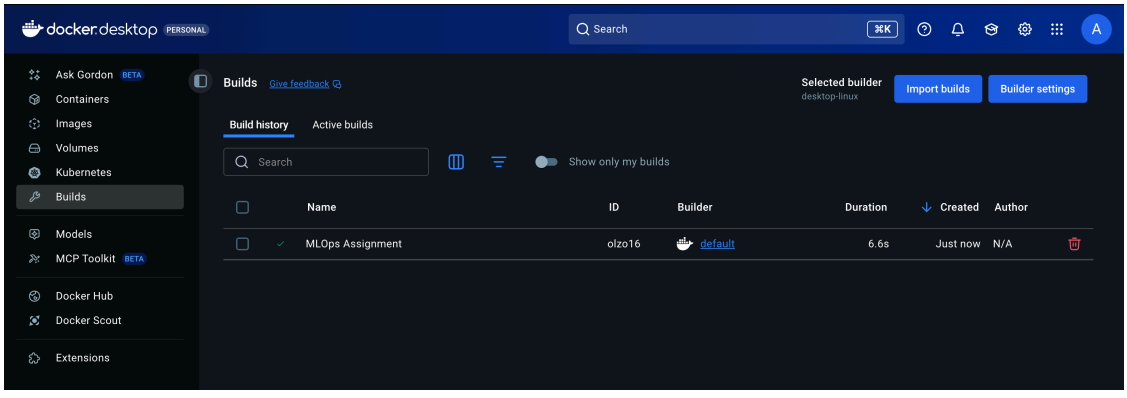


Figure 11: Docker Build Process

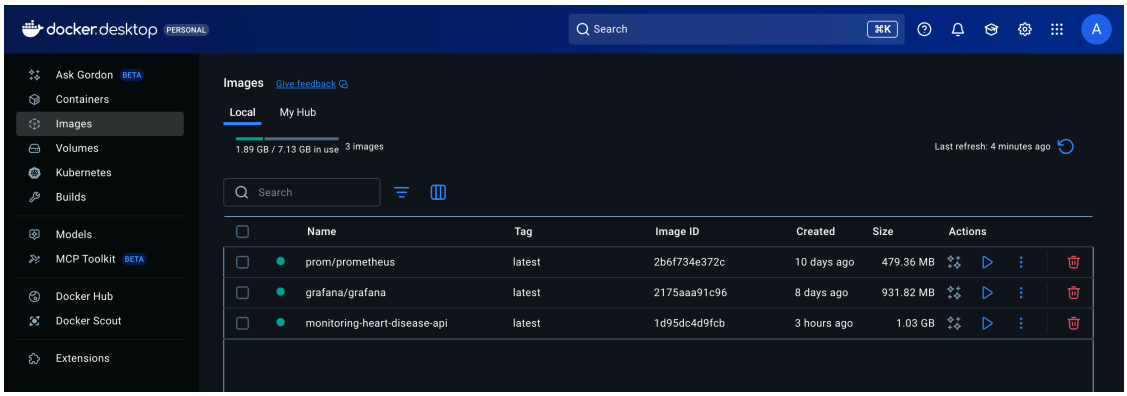


Figure 12: Docker Image Details

Orchestration (Kubernetes)

Kubernetes is responsible for managing the containerized application. **Three** replicas are set up in the deployment to guarantee fault tolerance and high availability. By dividing requests among the available pods, a LoadBalancer service makes the API accessible to outside traffic. A **Horizontal Pod Autoscaler (HPA)** is set up to automatically scale between 2 and 10 pods depending on a threshold of 70% CPU and 80% memory utilization in order to manage varying workloads. To guarantee steady cluster operation, resource restrictions are established at one CPU and one gigabyte of RAM per pod.

8.2 High-Level Architecture Diagram

The system is designed with a multi-layered approach for robustness and scalability, as depicted in the architecture diagram below.

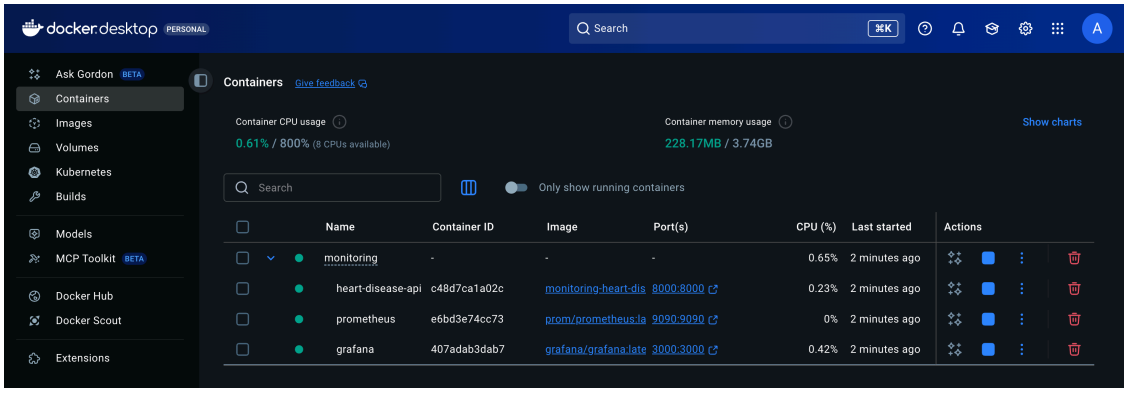


Figure 13: Containerized Service Architecture

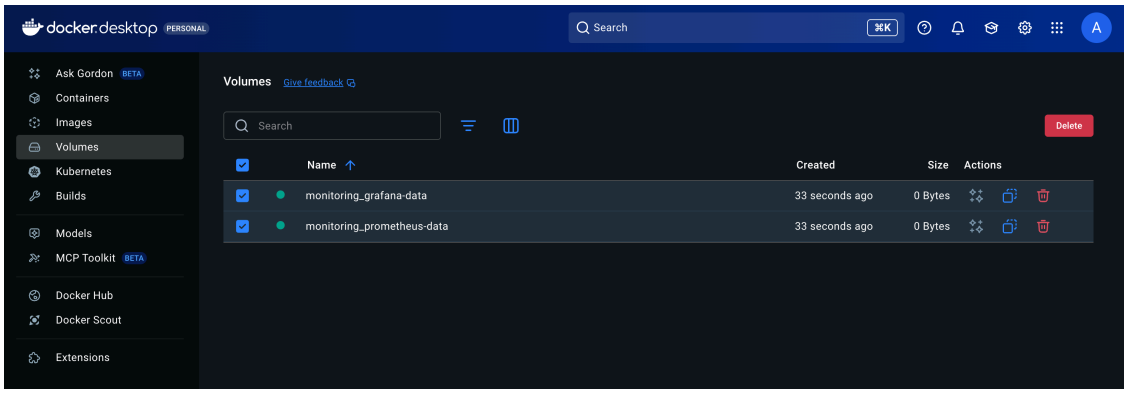


Figure 14: Docker Volume Configuration

The automated CI/CD pipeline in charge of developing, testing, and delivering the service has a strong basis thanks to this design, which offers a clear division of responsibilities.

---

9. CI/CD, Deployment, and Monitoring

The automated and operational features of the MLOps pipeline are described in detail in this last technical part. It covers the setup of the live production environment, the monitoring stack used to sustain system performance and health over time, and the CI/CD workflow that guarantees code quality and automates deployments.

9.1 Automated CI/CD with GitHub Actions

GitHub Actions was used to create a full Continuous Integration and Continuous Deployment (CI/CD) pipeline. Pushes to the main branch, pull requests, or manual dispatch all automatically start the procedure. There are five sequential jobs in pipeline:

1. **Lint:** Uses flake8 and black to check Python code for formatting problems and style consistency.
2. **Test:** Uses Pytest to run a set of 31 unit tests and confirms that code coverage reaches the project's 70% requirement.
3. **Train:** All candidate models are trained using the whole machine learning pipeline, and the top-performing model is packaged for deployment.
4. **Deploy:** Creates deployment-ready documentation, packages and uploads important assets, such as the finished model and evaluation reports.
5. **Notify:** Provides visibility into both successes and failures by reporting the pipeline run's final state.

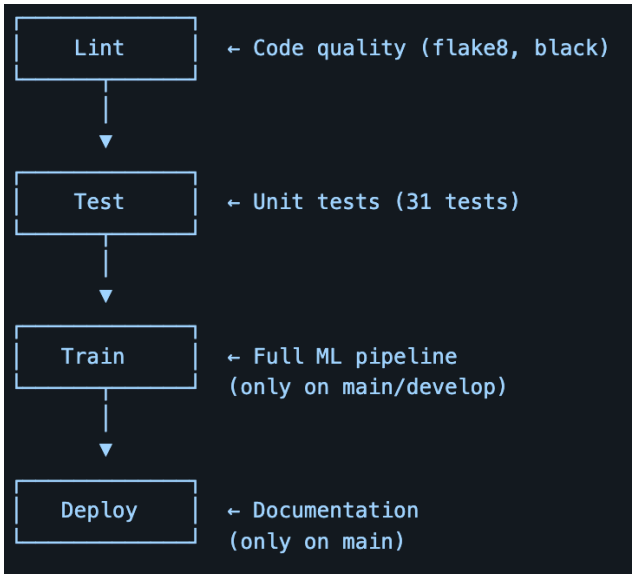


Figure 15: Pipeline Stages

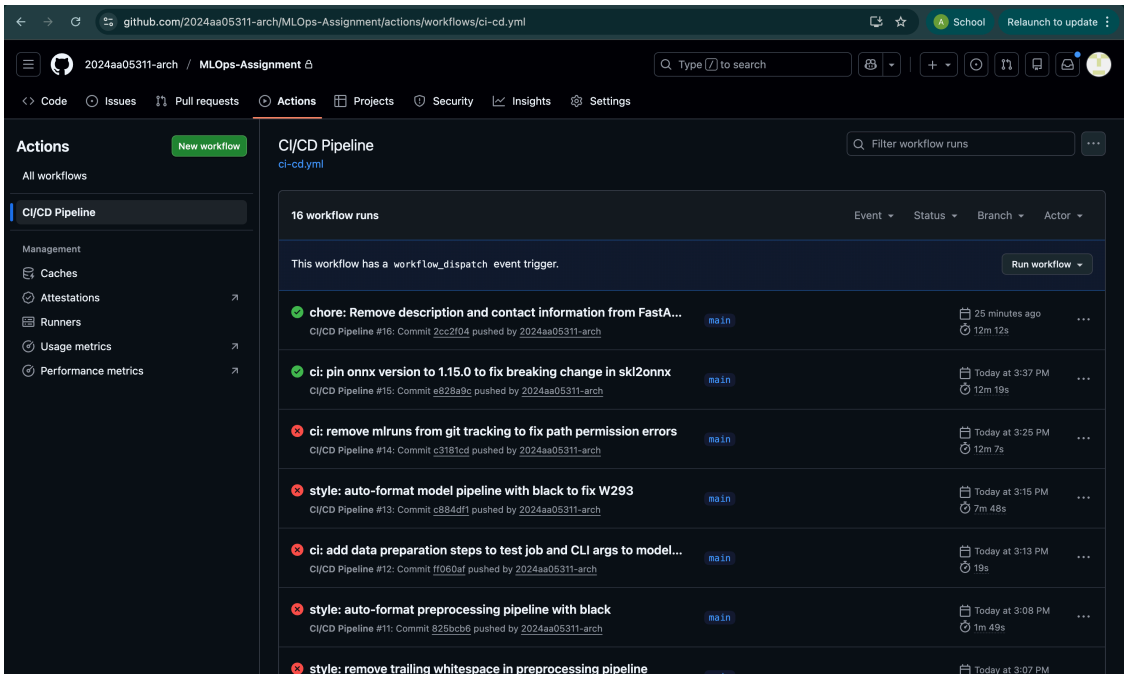


Figure 16: GitHub Actions History

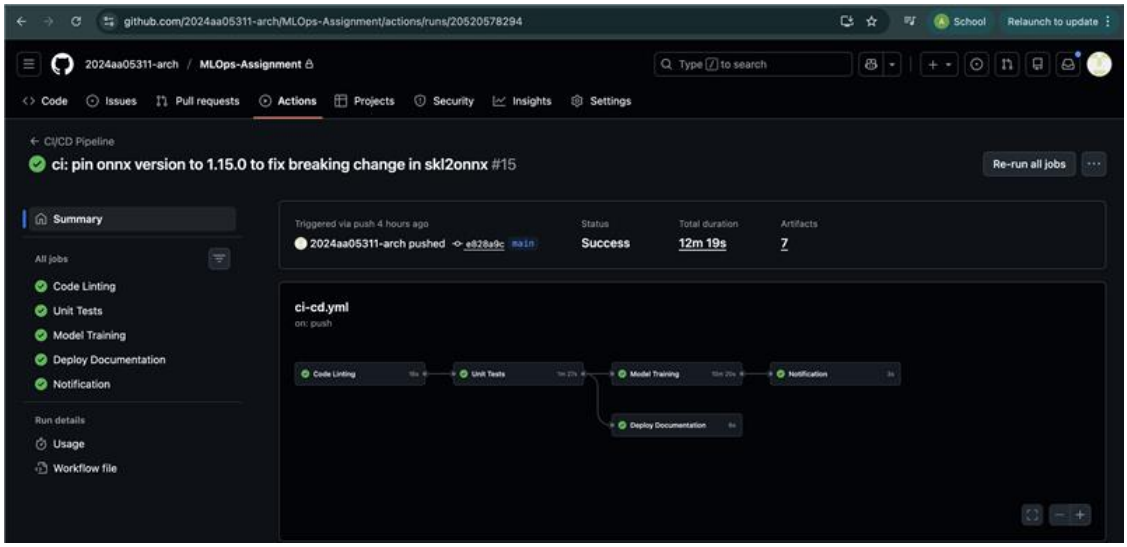


Figure 17: CI/CD Pipeline Flow Diagram

9.2 Live Deployment and API

To guarantee high availability and load dispersion, the application is set up to operate three replicas of the API pod on a Kubernetes cluster. The application is exposed externally through a LoadBalancer service.

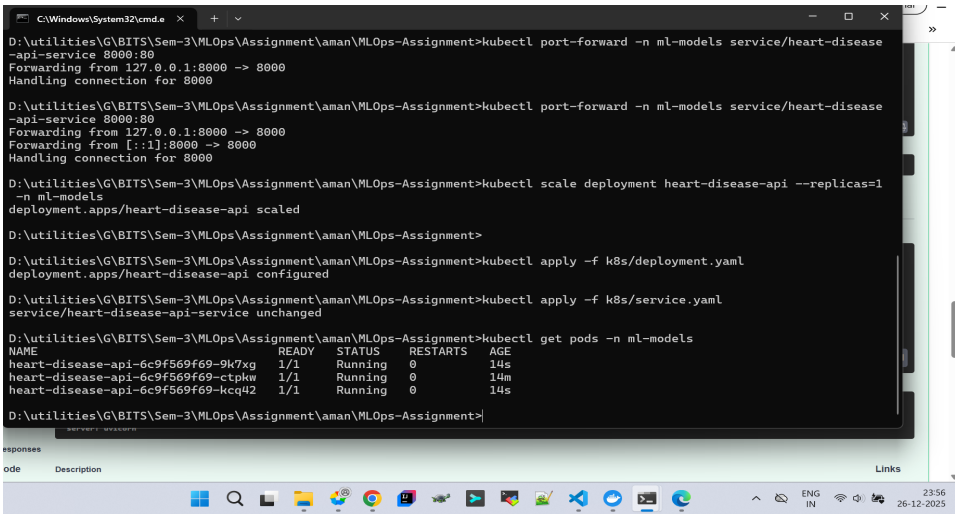


Figure 18: Kubernetes Pods Status

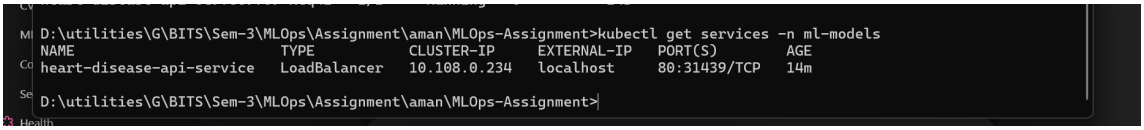


Figure 19: Kubernetes Exposed Services

Once deployed, the API is accessible and provides interactive documentation via its Swagger UI, allowing users to easily test the prediction endpoints.

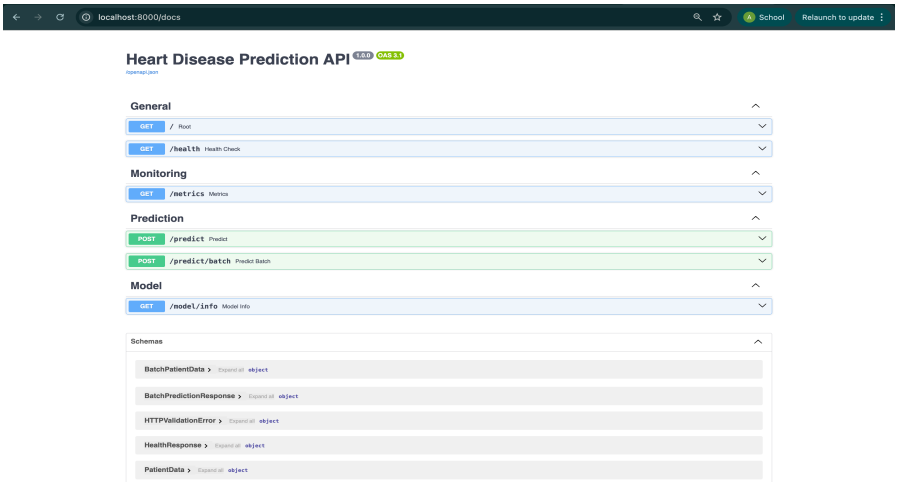


Figure 20: Swagger UI for Manual Testing

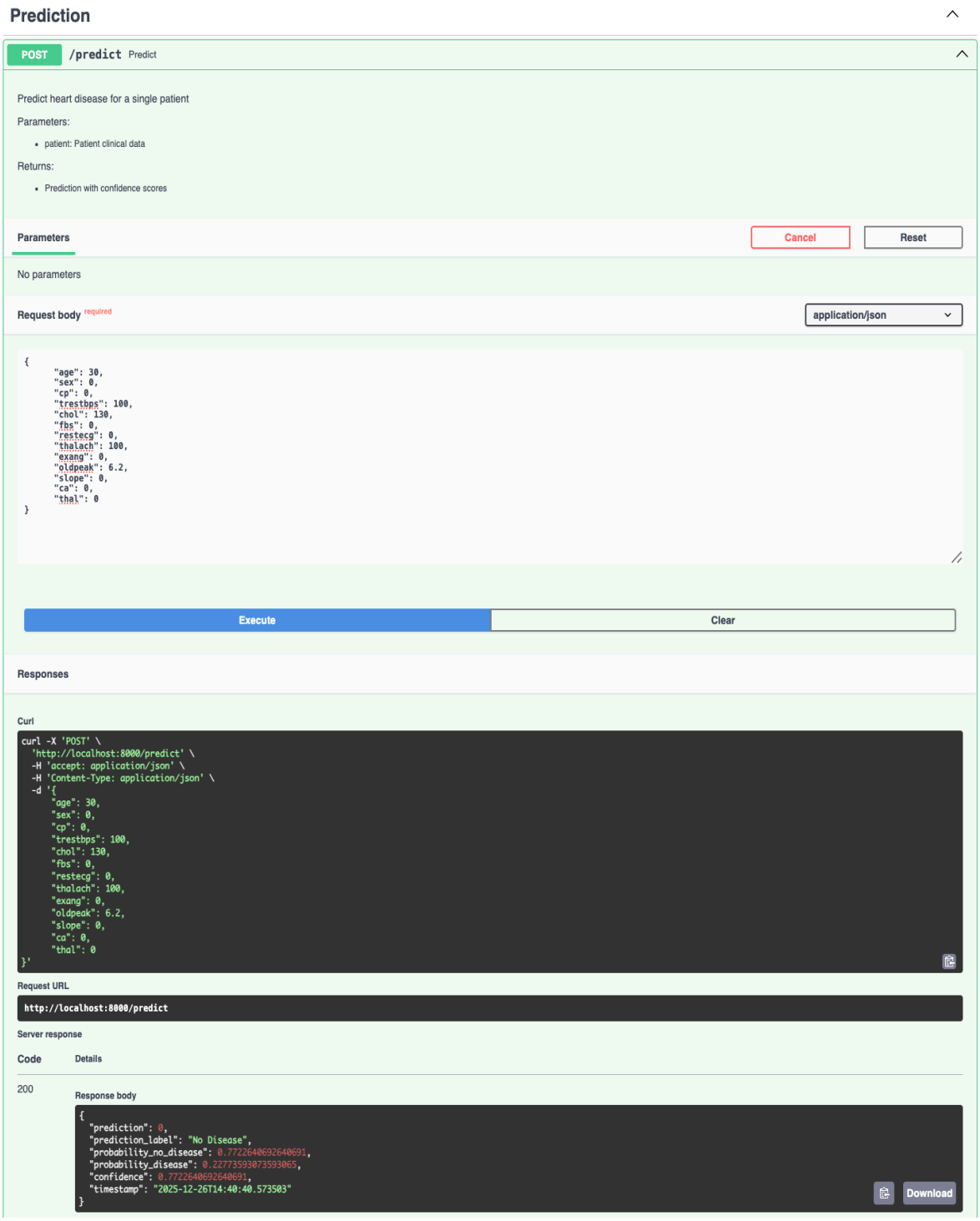


Figure 21: Successful Single Prediction Request

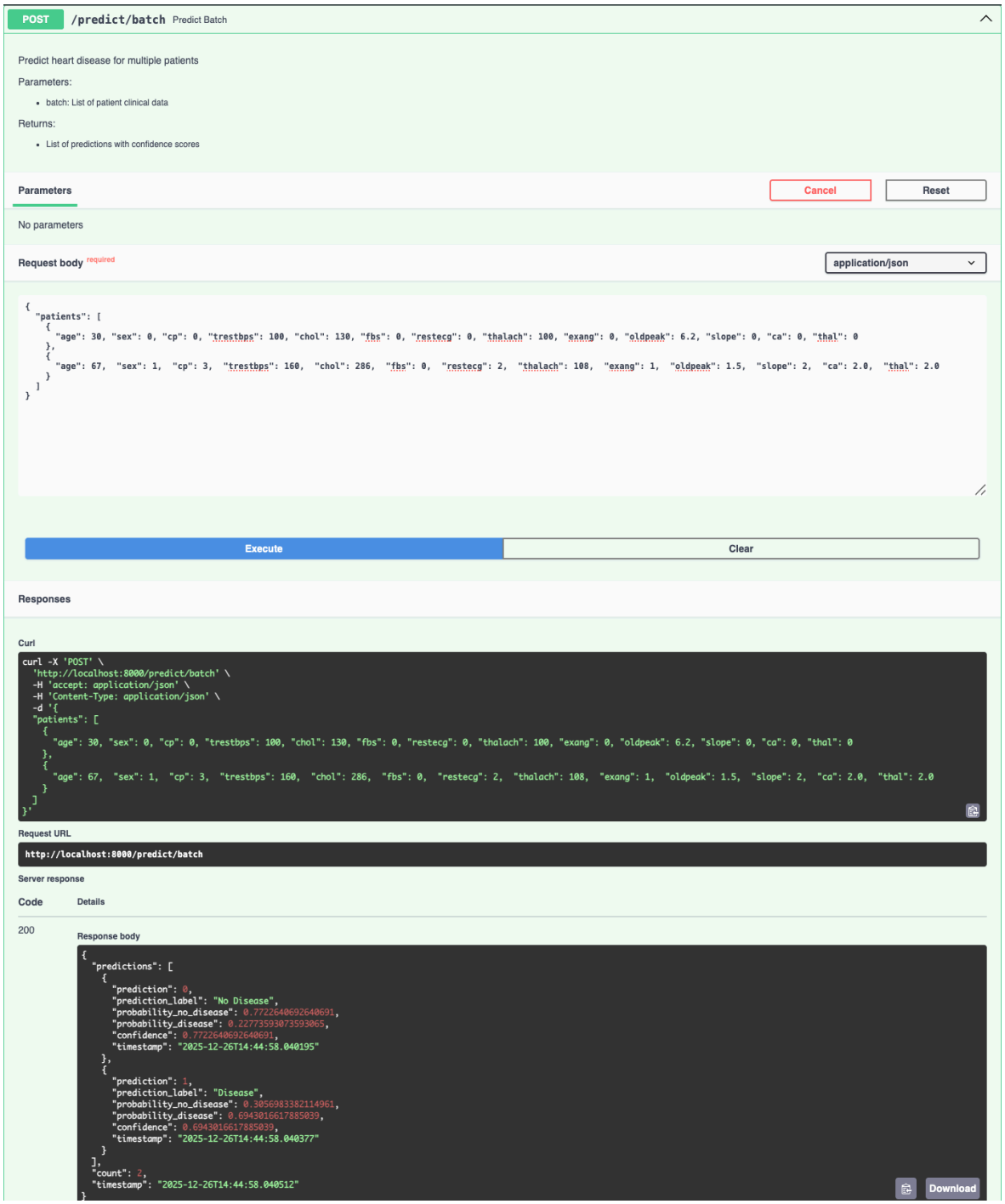


Figure 22: Batch Prediction Execution

9.3 Monitoring and Observability

Prometheus and **Grafana** make together a strong monitoring stack that offers comprehensive observability into the functionality and health of the deployed service. Prometheus gathers time-series data by scraping the /metrics endpoint that the FastAPI application is instrumented to expose.

Key metrics collected are categorized as follows:

- **Request Metrics:** `api_requests_total`, `api_request_latency_seconds`, `active_requests`
- **Prediction Metrics:** `predictions_total`, `prediction_confidence`
- **System Metrics:** `model_loaded`

Real-time monitoring of request rates, latency, prediction distributions, and general system health is made possible by the visualization of this data in Grafana dashboards.

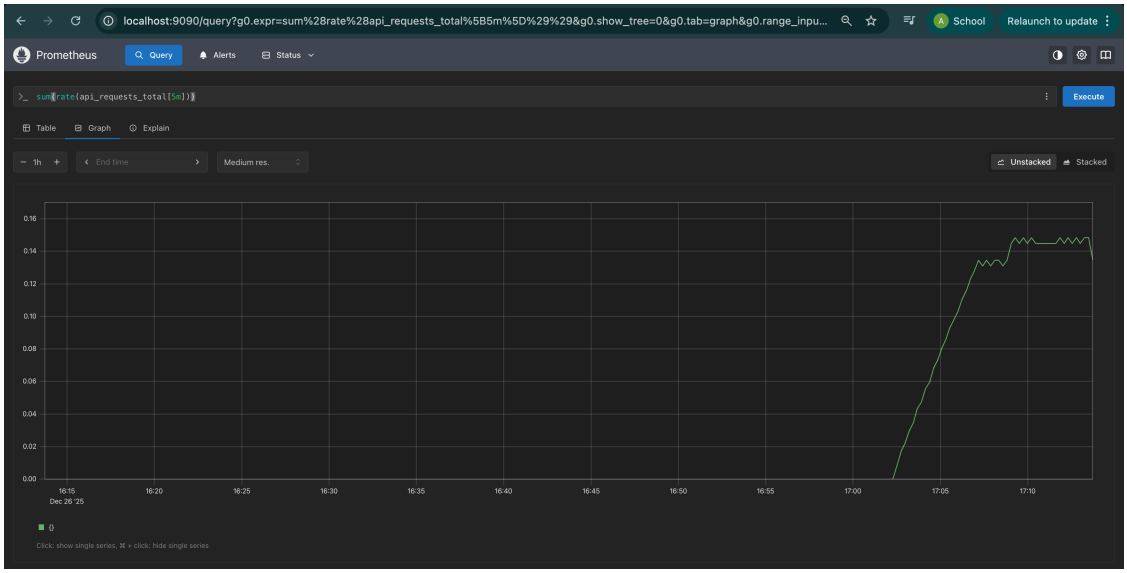


Figure 23: Prometheus Metrics Dashboard

Conclusion

By providing a fully automated, deployed, and monitored machine learning service from initial data analysis to production-grade operations, this completes the end-to-end MLOps lifecycle.