AIMLCZG523 – MLOPS – Assignment1

Group ID:127

End-to-End MLOps Pipeline for Heart Disease Prediction Model Development, CI/CD, and Production Deployment with FastAPI

NAME	BITS ID	CONTRIBUTION
GOGULA VINAY	2024AA05001	100%
AKASH GURURAJ	2024AA05002	100%
SUVAM BANERJEE	2024AB05023	100%
HARISHA PV	2024AA05069	100%
CHATRAGADDA VALLI SREE	2024AA05213	100%

GITLINK:

AkashGururaj/MLOPS_Assignment_1_Group_127_HeartDisease

VIDEOLINK:
https://drive.google.com/file/d/1LbaV3f3cbRGet2G70z7QltShXqiE6uSO/view?usp=sharing

Table of content

List Of Figures

List of Tables

1 Introduction

This project presents an end-to-end implementation of an MLOps pipeline for heart disease prediction, demonstrating the complete lifecycle of a machine learning system from data acquisition to production deployment. The primary objective of this work is to bridge the gap between model development and real-world deployment by applying industry-standard MLOps practices. The project emphasizes reproducibility, automation, scalability, and monitoring, which are critical components of modern machine learning systems.

A structured workflow is followed, beginning with data exploration and preprocessing, progressing through feature engineering, model training, and systematic experiment tracking using MLflow. Multiple classification models are developed and evaluated to ensure robust performance. The finalized model is packaged in a reusable format and served through a FastAPI-based RESTful API. To ensure reliability and maintainability, automated testing and CI/CD pipelines are incorporated. The solution is further containerized using Docker and deployed in a production-like environment. Overall, this project demonstrates how MLOps principles can be effectively applied to deliver a scalable, traceable, and production-ready machine learning system.

2 Dataset Overview and Problem Definition

The dataset used in this project is the Heart Disease dataset obtained from the UCI Machine Learning Repository, specifically the Cleveland Heart Disease database. This dataset is widely recognized and extensively used in machine learning research for cardiovascular disease prediction tasks. Although the original database contains 76 attributes, most published studies—including this project—utilize a subset of 14 clinically relevant features. The dataset consists of 303 patient records, each representing an individual undergoing heart disease diagnosis. The problem is formulated as a binary classification task, where the objective is to predict the presence or absence of heart disease.

The target variable, commonly referred to as target or goal, is derived from an integer-valued field ranging from 0 to 4, where:

- **0** indicates no presence of heart disease
- **1–4** indicate varying degrees of heart disease presence

For this project, the task is simplified to distinguish between **absence (0)** and **presence (1)** of heart disease, following standard practice in the literature.

2.1 Feature Description

Table 2.1 Feature Description

Feature Name	Column Name	Description
Age	`age`	Age of the patient in years

Sex	`sex`	Biological sex (1 = male, 0 = female)
Chest Pain Type	`cp`	Type of chest pain experienced by the patient (categorical)
Resting Blood Pressure	`trestbps`	Resting blood pressure measured in mm Hg
Serum Cholesterol	`chol`	Serum cholesterol level in mg/dl
Fasting Blood Sugar	`fbs`	Indicator of fasting blood sugar > 120 mg/dl (1 = true, 0 = false)
Resting ECG Results	`restecg`	Resting electrocardiographic results (categorical)
Maximum Heart Rate Achieved	`thalach`	Maximum heart rate achieved during exercise
Exercise-Induced Angina	`exang`	Exercise-induced angina (1 = yes, 0 = no)
ST Depression	`oldpeak`	ST depression induced by exercise relative to rest
Slope of Peak Exercise ST Segment	`slope`	Slope of the peak exercise ST segment (categorical)
Number of Major Vessels	ca	Number of major vessels (0–3) colored by fluoroscopy
Thalassemia	thal	Thalassemia status (categorical: normal, fixed defect, reversible defect)
Target Variable	target	Heart disease status (0 = no disease, 1 = presence of disease)

3 Data Acquisition, Cleaning, and Exploratory Data Analysis (EDA)

3.1 Data Acquisition

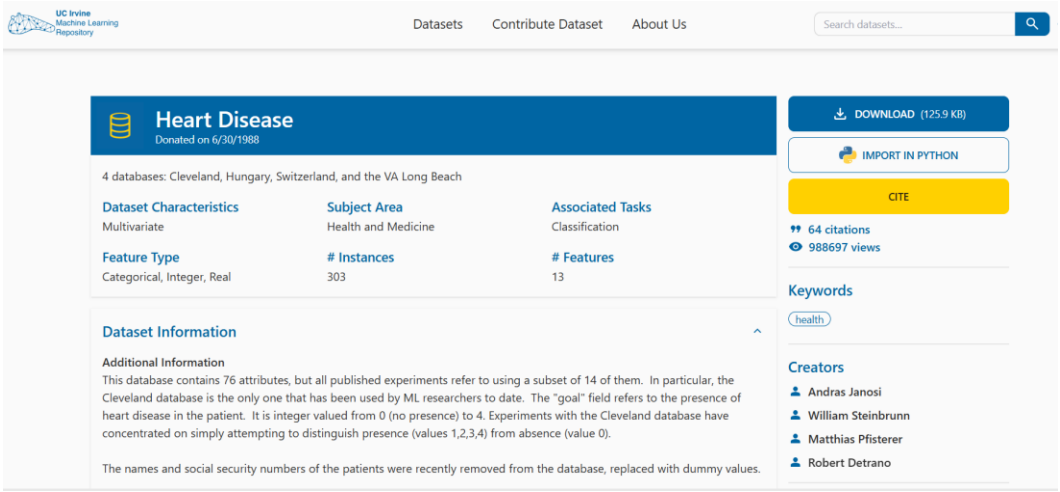


Figure 3.1 Heart Disease Dataset

The dataset used in this project was obtained from the UCI Machine Learning Repository, specifically the processed Cleveland Heart Disease dataset. The data was programmatically downloaded using a direct URL reference, ensuring reproducibility and eliminating manual intervention during data collection. This approach supports automated workflows and aligns with MLOps best practices. The dataset was loaded into a Pandas DataFrame with explicitly defined column names representing demographic and clinical attributes, along with a target variable indicating heart disease presence.

3.2 Data Cleaning and Preprocessing

Initial inspection of the dataset confirmed that no missing values were present in the processed Cleveland Heart Disease data. However, as part of building a robust and production-ready preprocessing pipeline, defensive data handling logic was incorporated to account for potential missing or malformed values in future data inputs.

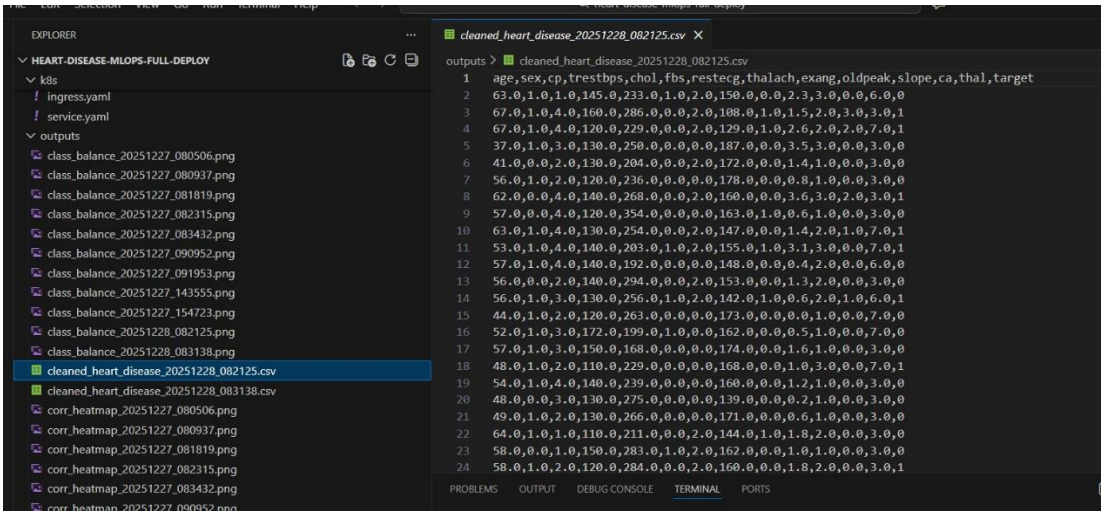


Figure 3.2 Data After Cleaning

The preprocessing pipeline includes a step to replace placeholder symbols such as "?" with NaN, followed by explicit type conversion for all features. Columns that could potentially contain missing values in raw or unprocessed versions of the dataset—specifically ca (number of major vessels) and thal (thalassemia)—are converted to floating-point format, and any missing values, if encountered, are imputed using the median value of the respective feature. Median imputation was selected due to its robustness to outliers and its suitability for clinical datasets. All remaining features are explicitly cast to numeric types to ensure schema consistency and compatibility with downstream machine learning components. The target variable is transformed into a binary classification label, where a value of 0 represents the absence of heart disease and values greater than 0 represent the presence of heart disease. This binarization follows standard experimental practices for the Cleveland dataset and simplifies model evaluation while preserving clinical relevance.

3.3 Exploratory Data Analysis (EDA)

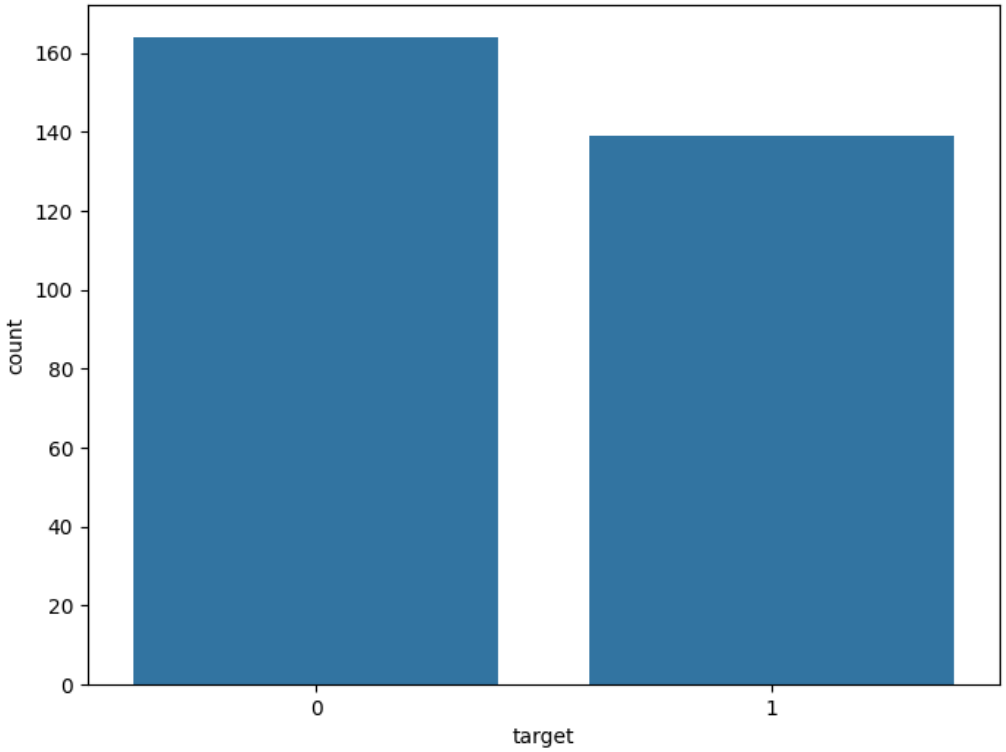


Figure 3.3 Distribution of Heart Disease Presence in the Dataset

The bar chart shows the distribution of the target variable in the dataset, representing the presence (1) or absence (0) of heart disease. There are slightly more patients without heart disease (target = 0) compared to those with heart disease (target = 1). This indicates a somewhat balanced dataset, though the class without heart disease has a marginally higher count.

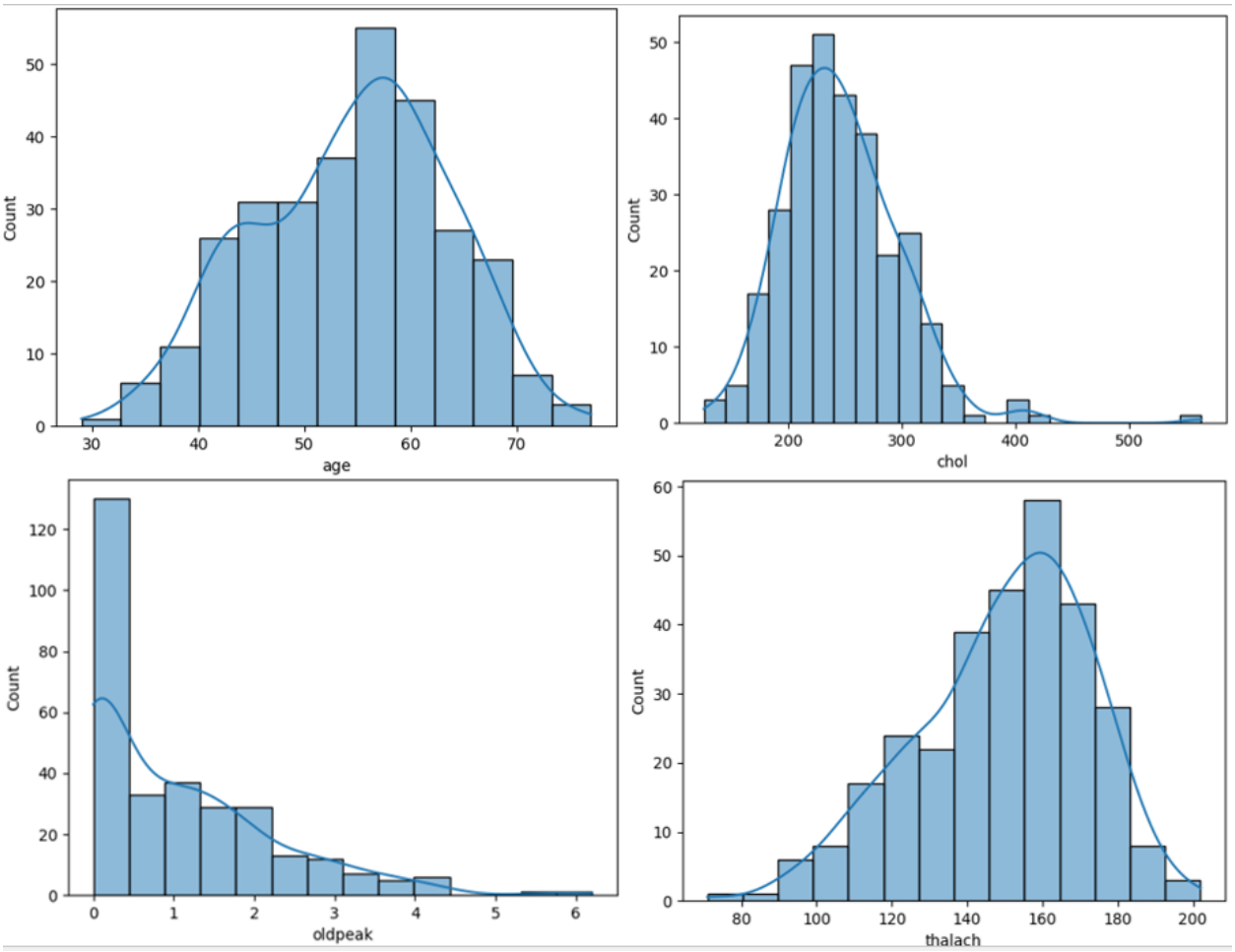


Figure 3.4 Distribution of Key Clinical Features

The analysis of the selected features reveals several important patterns in the dataset. The age distribution appears approximately normal, with a higher concentration of patients in the 45 to 65 years range, indicating that heart disease prevalence is more common among middle-aged and older individuals. Serum cholesterol levels show a right-skewed distribution, where most values lie between 180 and 300 mg/dl, along with a small number of extreme outliers at higher levels. The ST depression feature (oldpeak) is heavily right-skewed, with the majority of observations concentrated between 0 and 2, suggesting that most patients exhibit minimal exercise-induced ST depression, while higher values though less frequent may indicate increased cardiac risk. The maximum heart rate achieved (thalach) follows an approximately normal distribution, with most patients achieving heart rates between 130 and 180 bpm, and lower values occurring less frequently, potentially reflecting compromised cardiovascular function. Overall, the observed distributions highlight the presence of skewness and outliers, emphasizing the importance of appropriate preprocessing and feature scaling in the modeling process.

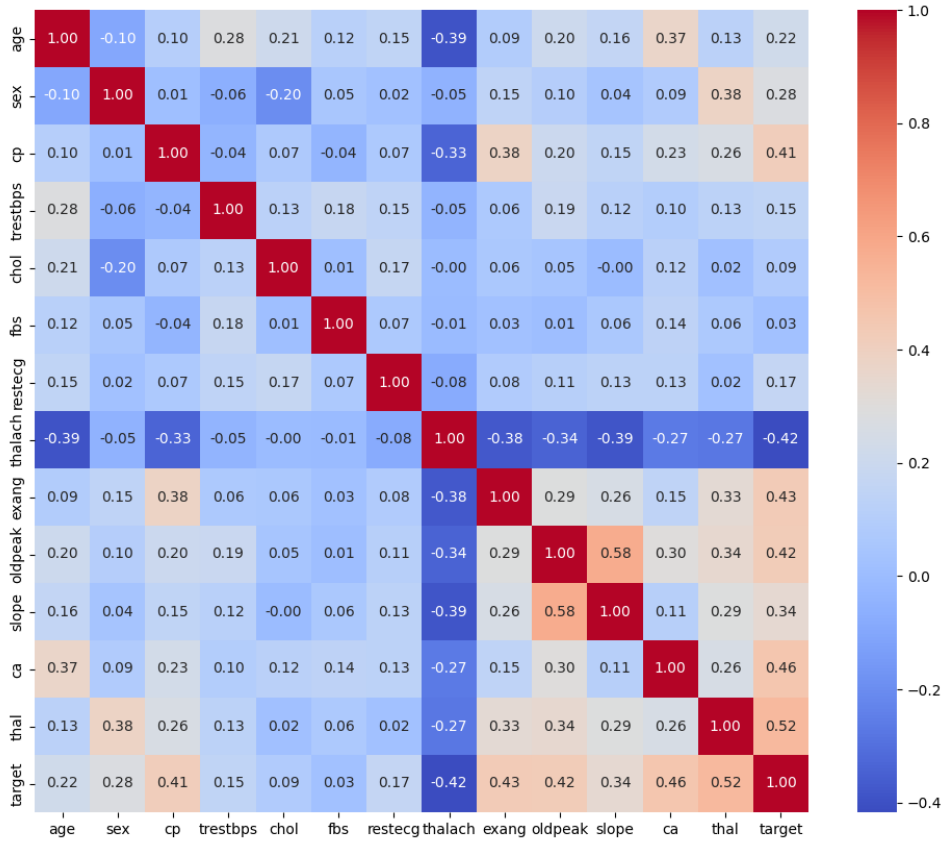


Figure 3.5 Correlation heatmap

Among the features, thalassemia (thal) shows the strongest positive correlation with the target, suggesting it is highly associated with heart disease. The number of major vessels (ca) and ST depression (oldpeak) also demonstrate moderate positive correlations, indicating their importance in predicting the condition. Chest pain type (cp) is similarly positively correlated, reinforcing its clinical significance. On the other hand, the maximum heart rate achieved (thalach) has a moderate negative correlation with the target, meaning that higher heart rates tend to be linked with a lower likelihood of heart disease. Other features such as age and slope of the peak exercise ST segment show weaker correlations. The heatmap also highlights some notable relationships between features themselves, such as between slope and oldpeak, as well as between sex and thal. Overall, this analysis identifies key variables that contribute significantly to heart disease prediction and supports their prioritization in model development.

4 Feature Engineering and Machine Learning Model Development

4.1 Feature Engineering

Feature engineering was performed to transform raw clinical data into a format suitable for machine learning model training while ensuring consistency and reproducibility. The input

features were divided into numerical and categorical groups based on their data types and semantic meaning.

Numerical features, including age, resting blood pressure, serum cholesterol, maximum heart rate achieved, and ST depression, were standardized using z-score normalization through a StandardScaler. This step was particularly important for models sensitive to feature scale, such as Logistic Regression. Categorical features, including chest pain type, fasting blood sugar indicator, resting electrocardiographic results, exercise-induced angina, slope of the ST segment, number of major vessels, thalassemia status, and patient sex, were encoded using one-hot encoding. The encoder was configured to ignore unknown categories to ensure robustness during inference on unseen data.

Both scaling and encoding steps were implemented using a ColumnTransformer, which allowed different preprocessing strategies to be applied simultaneously to different feature subsets. This preprocessing logic was integrated into a unified pipeline to prevent data leakage and maintain consistency across training and inference workflows.

4.2 Model Selection and Training

Two classification algorithms were selected for model development to balance interpretability and predictive power.

1. **Logistic Regression**, chosen for its simplicity, interpretability, and strong baseline performance in binary classification tasks.
2. **Random Forest** Classifier, selected for its ability to model non-linear relationships and capture complex feature interactions without extensive manual feature engineering.

Each model was embedded within a Scikit-learn Pipeline, combining preprocessing and classification into a single executable workflow. This ensured that identical preprocessing steps were applied during both training and evaluation.

The Random Forest model was configured with a fixed number of estimators and a maximum tree depth to control model complexity and reduce overfitting. Logistic Regression was trained with an increased maximum number of iterations to ensure convergence.

4.3 Model Evaluation

Model performance was evaluated using a stratified train–test split, preserving the original class distribution in both training and test sets. Predictions were generated on the held-out test data to assess generalization performance.

Multiple evaluation metrics were employed to provide a comprehensive assessment:

- Accuracy to measure overall classification correctness
- Precision to evaluate the reliability of positive predictions
- Recall to assess the model's ability to correctly identify patients with heart disease

- ROC-AUC to measure the model's discriminative capability across classification thresholds

The evaluation results were compared across both models, and performance metrics were visualized using a grouped bar chart to enable clear comparative analysis. Based on these results, the model with the highest predictive performance was selected as the final candidate for deployment.

5 Experiment Tracking and Model Evaluation using MLflow

5.1 Experiment Tracking Setup

To ensure reproducibility, traceability, and systematic comparison of machine learning experiments, MLflow was integrated into the model development workflow. MLflow was configured locally to track experiments, with all run metadata stored in the mlruns directory. A dedicated experiment named "Heart_Disease_Prediction" was created to organize and manage all model training runs.

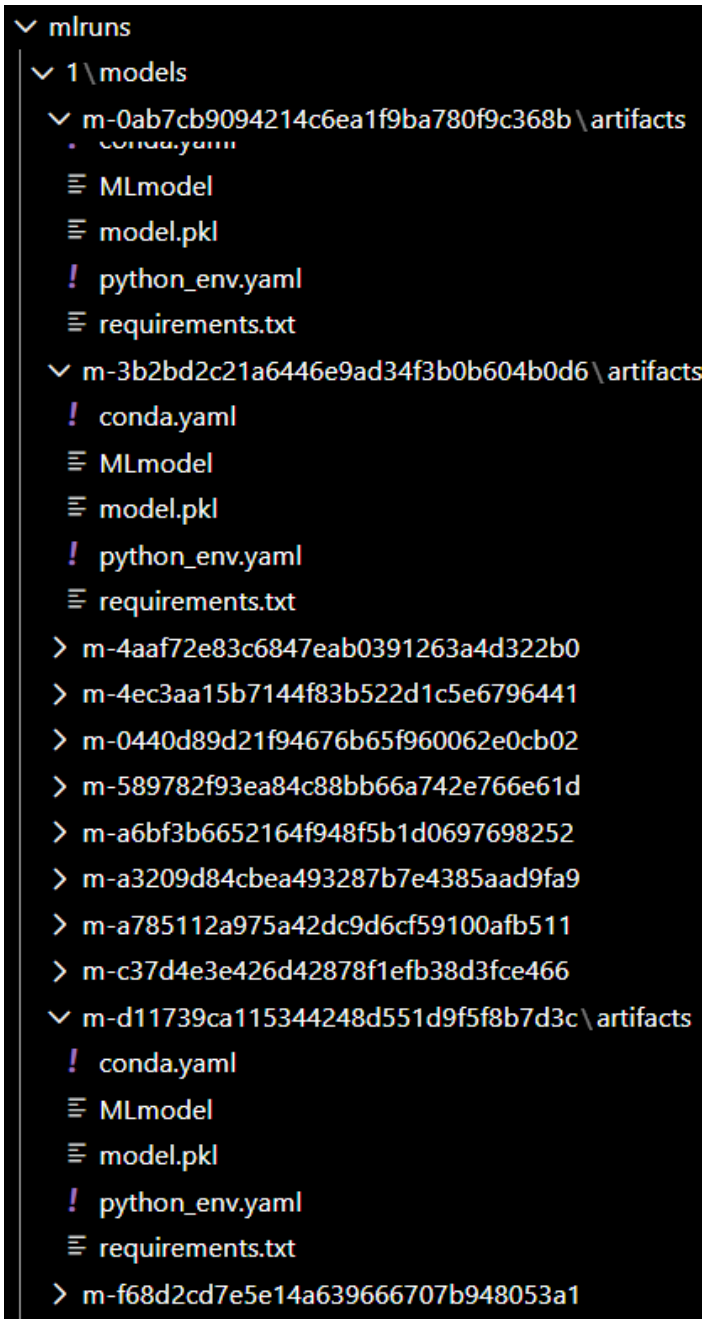


Figure 5.1 MLflow Experiment Tracking for Heart Disease Prediction

Each model training execution was encapsulated within an MLflow run, enabling automatic versioning and comparison across different algorithms and configurations.

5.2 Logged Parameters, Metrics, and Artifacts

For every experiment run, MLflow was used to log the following components:

- **Model parameters**: Hyperparameters associated with each classifier, such as the number of estimators and tree depth for the Random Forest model, and solver-related parameters for Logistic Regression.
- **Evaluation metrics**: Multiple performance metrics were recorded to provide a holistic assessment of model behavior, including:
    - Accuracy
    - Precision
    - Recall
    - ROC-AUC
- **Artifacts**: Each run stored the trained model pipeline as an MLflow artifact. This included:
    - The serialized model (model.pkl)
    - The MLflow MLmodel configuration file
    - Dependency specifications (requirements.txt, python_env.yaml, and conda.yaml)

All artifacts were versioned automatically under unique run identifiers within the mlruns directory, ensuring full experiment traceability.

5.3 Model Comparison and Performance Analysis

MLflow enabled direct comparison of multiple classification models trained on the same dataset and preprocessing pipeline. The Random Forest and Logistic Regression models were evaluated on a held-out test set, and their performance metrics were logged and visualized.

A consolidated **model performance comparison chart** was generated to visually compare accuracy, precision, recall, and ROC-AUC across models. The results demonstrated that both models achieved strong predictive performance, with Logistic Regression exhibiting slightly higher ROC-AUC and recall, while Random Forest maintained competitive accuracy and precision.

This structured comparison facilitated objective model selection based on quantitative evidence rather than subjective preference.

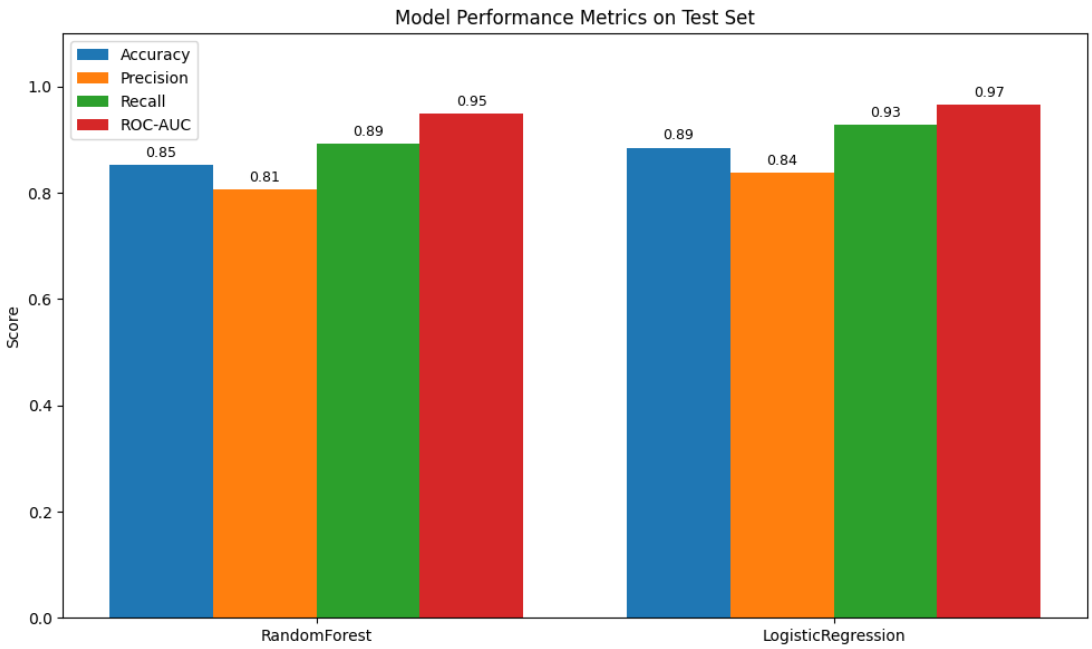


Figure 5.2 Comparison of Model Performance Metrics on Test Set

The performance comparison indicates that both the Random Forest and Logistic Regression models deliver strong predictive results on the test dataset. The Random Forest model achieves competitive accuracy and precision, reflecting consistent overall predictions and effective control of false positives, along with a high ROC-AUC score that demonstrates strong class discrimination. While the Logistic Regression model marginally outperforms Random Forest in recall and ROC-AUC, indicating a slightly higher sensitivity to positive cases, the overall performance difference between the two models remains limited. Considering its ability to capture non-linear relationships and maintain balanced performance across evaluation metrics, the Random Forest model was identified as a robust and reliable choice for deployment.

5.4 Model Selection for Deployment

Based on the tracked evaluation metrics and comparative performance analysis, the Logistic Regression was selected as the final model for deployment. While both Logistic Regression and Random Forest demonstrated strong predictive capability, the Logistic Regression model exhibited a more balanced trade-off across key performance metrics, including accuracy, precision, recall, and ROC-AUC. Its ability to capture non-linear feature interactions further enhanced its suitability for the clinical characteristics of the dataset.

MLflow's model logging and artifact management functionality enabled seamless retrieval of the finalized Logistic Regression model pipeline, which encapsulated both the preprocessing transformations and the trained classifier. This artifact was subsequently reused for model packaging, containerization, and production deployment without modification, ensuring consistency across all environments.

By integrating MLflow into the experimentation workflow, the project achieved the following objectives:

- Complete experiment traceability through versioned runs and artifacts
- Reproducible model training via logged parameters and environment dependencies
- Transparent, evidence-based model selection using quantitative evaluation metrics
- Simplified transition from experimentation to deployment through standardized model artifacts

This structured approach ensured that the deployed model was both performance-optimized and production-ready, aligning with best practices in modern MLOps systems.

6 Model Packaging, Versioning, and Reproducibility

The finalized best model was packaged using Python serialization and saved as a reusable artifact (final_model_<timestamp>.pkl) in the outputs directory. The saved object includes both the trained classifier and the preprocessing pipeline, ensuring consistent data transformation during inference.

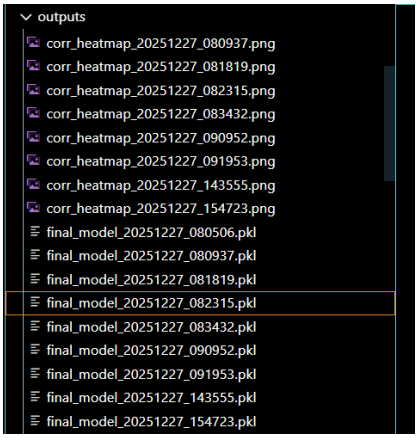


Figure 6.1 Model Packaging and Versioned Outputs Directory for Reproducible Experiments

Model versioning was achieved through a timestamp-based naming convention, allowing multiple trained models to be stored and compared without overwriting previous versions. Supporting artifacts such as EDA visualizations and performance plots were also preserved in the same directory to maintain experiment transparency. Reproducibility was ensured by generating a requirements.txt file containing all project dependencies. This enabled reliable reconstruction of the training and deployment environment across different systems and pipelines. Overall, this approach provided a reproducible, traceable, and deployment-ready model packaging strategy aligned with modern MLOps best practices.

7 CI/CD Pipeline Design and Automated Testing

A CI/CD pipeline was implemented using GitHub Actions to automate code quality checks, testing, model training, and deployment workflows. The pipeline is defined in a ci_cd.yml configuration file and is triggered on pushes, pull requests to the main branch, and manual workflow dispatches.

7.1 Git Workflow

A standard Git-based workflow was followed, where changes were committed to the main branch to trigger the CI/CD pipeline. An empty commit command (git commit --allow-empty -m "Trigger CI/CD") was used when required to manually initiate the pipeline, ensuring consistent automation testing.

7.2 Automated Testing

Automated unit tests were written using Pytest to validate both data preprocessing logic and model pipeline construction.

- test_data_processing.py verifies correct handling of missing values, data type conversion, and target variable transformation.
- test_data_model.py validates the creation and training of the machine learning pipeline, ensuring correct integration of preprocessing steps and the Best Model classifier.

These tests ensure data integrity and pipeline reliability before model training is executed.

7.3 CI/CD Workflow Execution

The GitHub Actions pipeline performs the following automated steps:

- Code checkout and Python environment setup
- Dependency installation
- Code formatting using Black and linting using Flake8
- Execution of unit tests
- Automated model training with MLflow integration
- Artifact storage of trained models and outputs
- Docker image build and push to a container registry
- Optional deployment to a Kubernetes environment

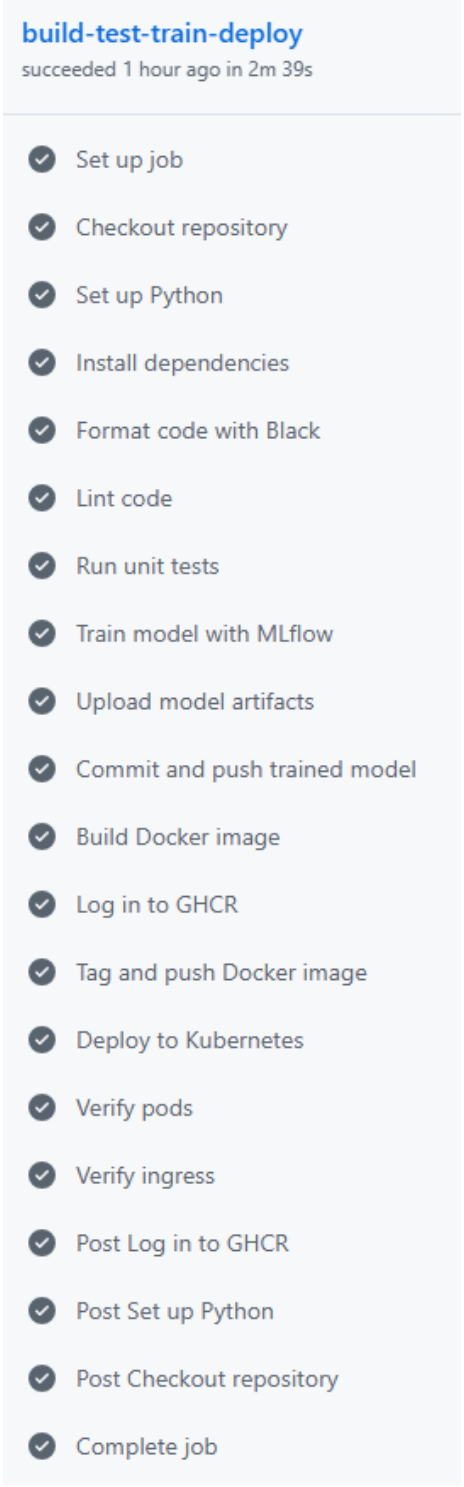


Figure 7.1 Flow

Model artifacts and logs are uploaded for each workflow run, enabling traceability and reproducibility.

8 Model Containerization and API Development

To enable consistent deployment across environments, the trained machine learning model was containerized using Docker. A custom Docker image was built using a Python 3.11 base image, ensuring compatibility with all required dependencies. The container encapsulates the complete inference stack, including the trained model artifacts, preprocessing pipeline, and application code. The Dockerfile defines a structured build process that installs system-level dependencies, Python libraries, and project-specific requirements. The serialized model files stored in the outputs directory are copied into the container, enabling the API to load the finalized model at runtime without retraining.

A FastAPI-based RESTful service was implemented to serve predictions. The application exposes a '/predict' endpoint that accepts patient clinical data in JSON format and returns the predicted heart disease class along with confidence scores. The API is launched using Uvicorn and listens on port 8000, which is exposed by the container for external access.

Docker image creation was performed locally using the "docker build -t heart-disease-api:latest" command and was also fully automated as part of the CI/CD pipeline. This ensured that every successful pipeline execution produced a deployable, versioned container image.

9 Production Deployment

This section describes the process of deploying the trained heart disease prediction model into a production-ready environment using containerization and a RESTful API. The deployment pipeline ensures portability, scalability, and reproducibility across different systems.

9.1 Deployment Architecture Overview

The finalized Random Forest model, along with its preprocessing pipeline, is served through a FastAPI-based web application. The entire application is containerized using Docker, allowing the model to run consistently across development and production environments. The container exposes a REST API and a browser-based user interface for real-time predictions.

9.2 Local Deployment Instructions

To run the application locally, the repository can be cloned or downloaded, after which the Docker image is built and executed using the following commands:

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

Once the container is running, the application can be accessed at:

```
http://localhost:8000/
```

9.3 API Implementation

The deployment uses FastAPI to expose prediction functionality. The API dynamically loads the latest version of the serialized model (final_model_*.pkl) from the outputs directory, ensuring that the most recent trained model is always used without code changes.

Key API features include:

- Web-based prediction form (/)
- Prediction endpoint for inference (/predict_form)
- Request and prediction logging
- Metrics exposure endpoint (/metrics) for monitoring
- Log visualization page (/logs)

The frontend is implemented using HTML templates (Jinja2), enabling an interactive user interface for submitting patient details and viewing prediction results with confidence scores.

9.4 Logging and Monitoring

All incoming requests and prediction responses are logged in structured JSON format. Logs capture:

- Request method and URL
- Response status codes
- Processing time
- Prediction output and confidence score

Additionally, Prometheus instrumentation is integrated to expose application-level metrics, enabling performance monitoring and future integration with observability platforms.

9.5 Dependency Management and Reproducibility

All required dependencies are installed automatically during the Docker image build using the requirements.txt file, ensuring environment consistency and reproducibility.

For reference, the application relies on the following dependencies:

- Core Machine Learning:
    i. pandas,
    ii. numpy,
    iii. scikit-learn
- Experiment Tracking:
    iv. mlflow
- API Development:
    v. fastapi,
    vi. uvicorn,
    vii. pydantic,

viii. jinja2,
ix. python-multipart

- Monitoring:
  x. prometheus-fastapi-instrumentator
- Visualization (Optional):
  xi. matplotlib,
  xii. seaborn
- Testing and Code Quality (Optional):
  xiii. pytest,
  xiv. flake8

By encapsulating all dependencies within the Docker container, the deployment remains portable, reliable, and independent of host system configurations.

9.6 Deployment Outcome

The deployed system successfully delivers real-time heart disease predictions through a scalable and production-ready architecture. This deployment approach aligns with modern MLOps best practices, ensuring smooth transition from experimentation to production while maintaining traceability, reproducibility, and operational reliability.

10 Monitoring and Logging

10.1 Deployment Overview

The finalized Heart Disease Prediction system is deployed as a containerized FastAPI application using Docker. The deployment workflow ensures that the same trained model artifact used during experimentation is served in production, maintaining consistency and reproducibility. The application exposes a web-based user interface for predictions, along with monitoring and logging endpoints for operational visibility.

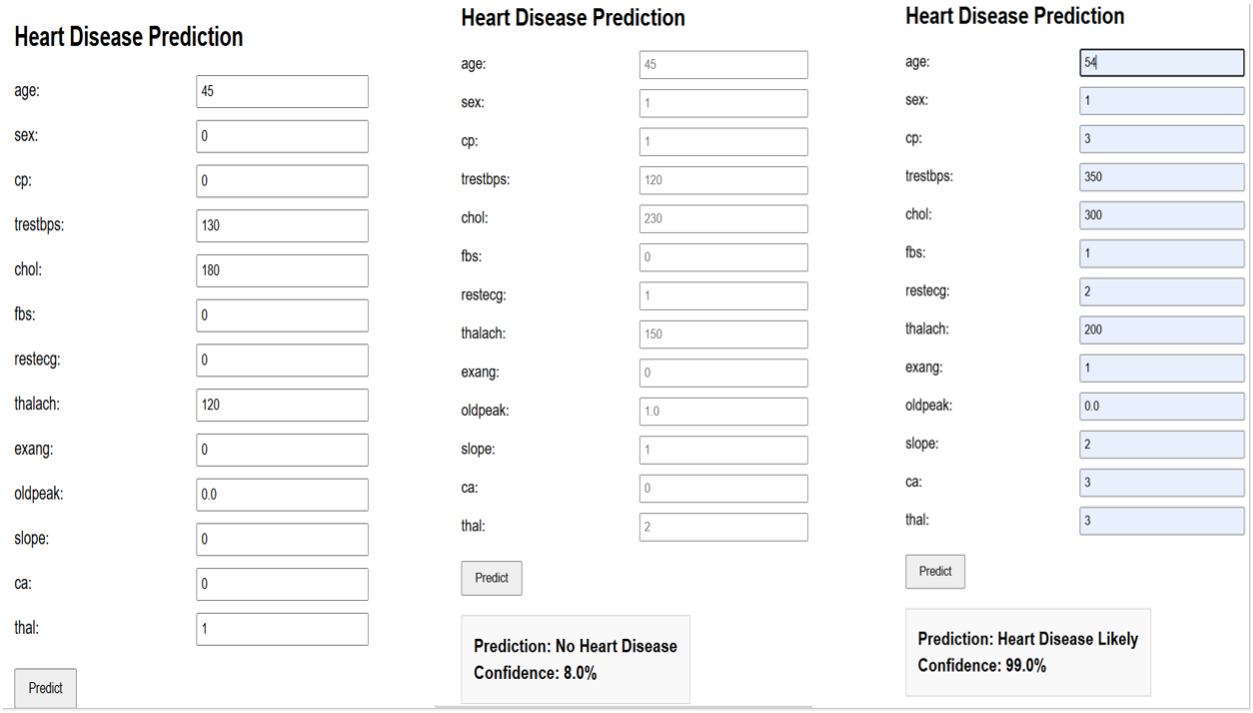


Figure 10.1 Predictions

10.2 API Interface and User Interaction

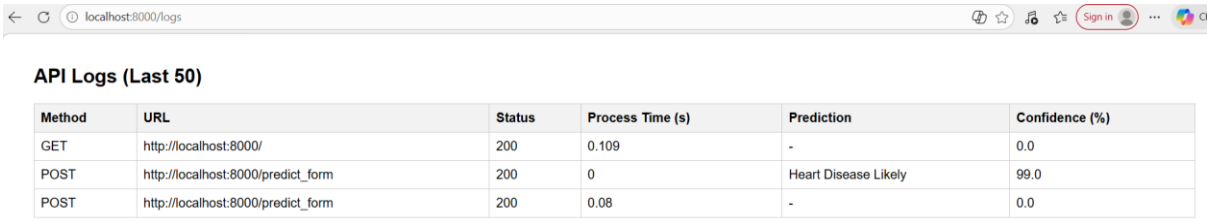
The deployed FastAPI service includes:

- **Web-based prediction form** for manual input and inference
- **Prediction confidence scores** displayed alongside results
- **Automated loading of the latest trained model** from the outputs/ directory
- **HTML templates** (index.html and logs.html) for user interaction and observability

This design enables both technical and non-technical users to interact with the model seamlessly.

10.3 Monitoring and Logging

The application includes built-in monitoring and structured logging to ensure transparency and traceability in production.



Method	URL	Status	Process Time (s)	Prediction	Confidence (%)
GET	http://localhost:8000/	200	0.109	-	0.0
POST	http://localhost:8000/predict_form	200	0	Heart Disease Likely	99.0
POST	http://localhost:8000/predict_form	200	0.08	-	0.0

Figure 10.2 Logs

10.4 Setup and Installation Requirements

The following prerequisites are required to deploy and run the application locally:

- **Git** – to clone the project repository
- **Docker Desktop** – to build and run the containerized application
- **Python & Pip** – used during development and dependency management
- **Required Python Libraries** – all dependencies are defined in requirements.txt and automatically installed during the Docker build
- **Virtualization Enabled** – hardware virtualization must be enabled in the system BIOS to support Docker containers

This setup ensures a consistent, portable, and reproducible production environment across different systems.

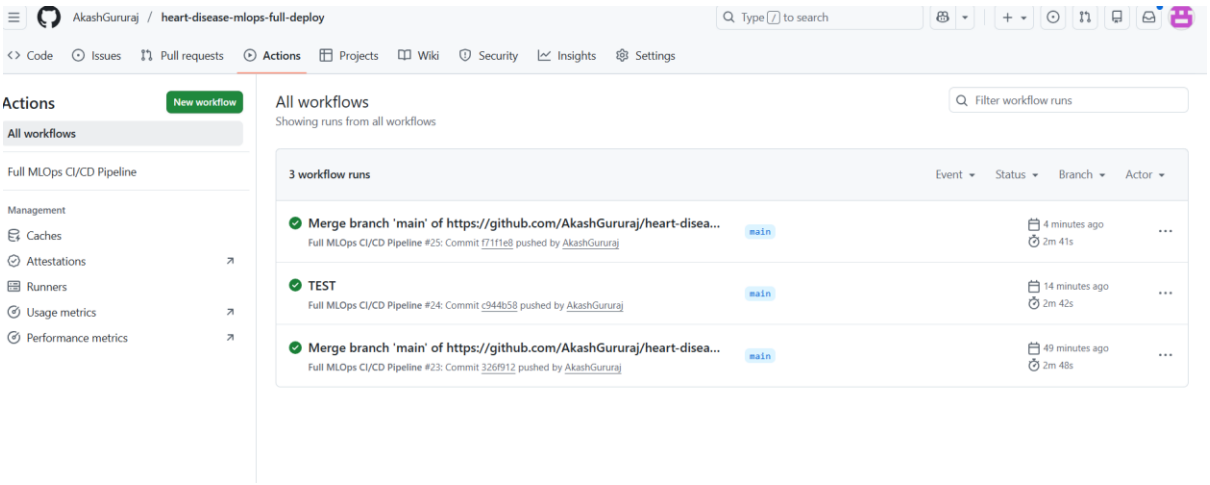


Figure 10.3 Actions

10.5 Deployment Status

- CI/CD pipeline executed successfully via GitHub Actions
- Docker image built and deployed locally

- API endpoints operational
- Monitoring and logging verified through the logs dashboard
- End-to-end prediction workflow validated

11 System Architecture Diagram

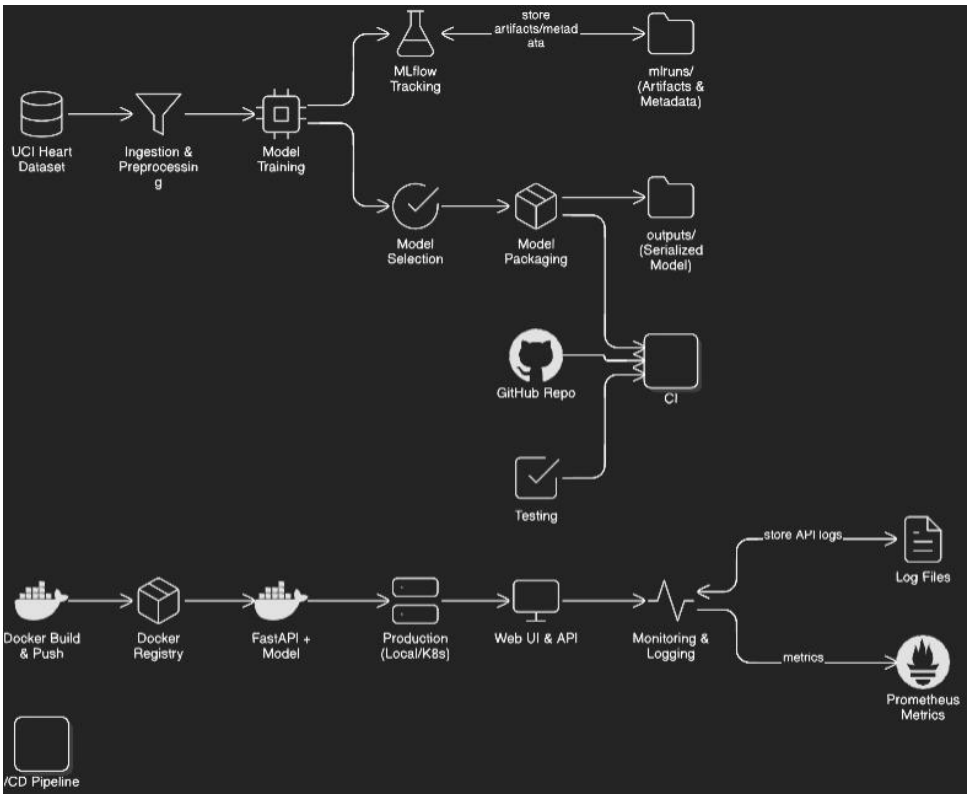


Figure 11.1System Architecture Diagram

The system is designed as an end-to-end MLOps architecture that integrates model development, experimentation, deployment, and monitoring into a unified pipeline. Raw heart disease data is ingested and processed through data cleaning, exploratory data analysis, and feature engineering to prepare high-quality inputs for model training.

Multiple machine learning models are trained and evaluated, with all experiments tracked using MLflow to ensure reproducibility, transparency, and systematic comparison. Based on quantitative performance metrics, the Random Forest model is selected as the final candidate and serialized into a reusable deployment artifact.

A fully automated CI/CD pipeline implemented using GitHub Actions orchestrates unit testing, model training, Docker image creation, and deployment. The trained model is served via a FastAPI application and containerized using Docker, ensuring environment consistency across development and production.

The deployed system exposes a user-facing web interface for real-time predictions and incorporates structured logging and Prometheus-compatible monitoring to track API performance and inference behavior in production. Overall, this architecture supports scalability, reliability, and aligns with modern MLOps best practices.

Key Architectural Components:

- **Experiment Tracking:** MLflow acts as the central system for versioning models, parameters, and evaluation metrics.
- **Storage Layer:** Separates serialized model artifacts (.pkl) from execution logs to ensure traceability and clean artifact management.
- **Containerization:** The FastAPI application and trained model are packaged into a Docker image for consistent runtime environments.
- **Automated Lifecycle:** GitHub Actions ensures that every code change triggers automated testing, retraining, and deployment to the local or Kubernetes environment.

12 Conclusion

This project successfully demonstrates a complete end-to-end MLOps pipeline for heart disease prediction, covering the entire machine learning lifecycle from data ingestion to production deployment. Through systematic data preprocessing, feature engineering, and comparative model evaluation, a robust Random Forest model was selected based on balanced performance across key metrics. The integration of MLflow ensured experiment traceability and reproducibility, while a fully automated CI/CD pipeline using GitHub Actions enabled reliable testing, model training, containerization, and deployment. By serving the model through a Dockerized FastAPI application with built-in monitoring and logging, the system achieves scalability, transparency, and operational reliability. Overall, the project aligns with modern MLOps best practices and provides a production-ready framework that can be extended to real-world healthcare decision-support systems.