HEART DISEASE PREDICTION - MLOPS ASSIGNMENT SUBMISSION

Group No: 75

Student Name	ID	Contribution
SAI SUDHEER KOLAGATLA	2024aa05637	100%
RHYTHM ASHOK GOYAL	2023ac05438	100%
ISHAN SITARAM MAHADULE	2024aa05655	100%
TEJASWINI PANDEY	2024aa05465	100%
RAJ PATEL	2024aa05158	100%

1. Project Overview

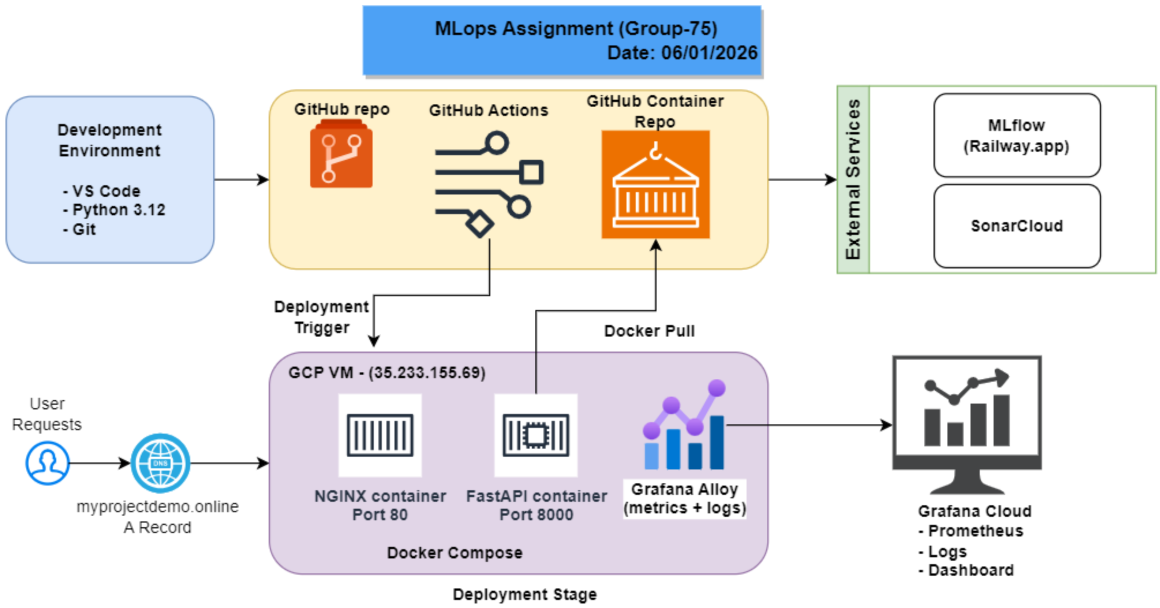
This is our MLOps course assignment where we built an end-to-end machine learning pipeline for predicting heart disease. The main goal was to learn how to take a machine learning model from development all the way to production, covering everything from data acquisition to monitoring.

We created a binary classifier that predicts whether a patient has heart disease based on 13 clinical features from the UCI Heart Disease dataset. The model is deployed as a REST API on a GCP VM, and we set up proper CI/CD, monitoring, and code quality checks.

The project demonstrates the complete MLOps lifecycle:

- Data acquisition and exploratory analysis
- Feature engineering and model training
- Experiment tracking
- Model packaging and containerization
- CI/CD automation
- Cloud deployment
- Production monitoring

Architecture



2. Live URLs and Resources

Here are all the live URLs for our project:

Resource	URL
Live API	http://myprojectdemo.online
API Documentation (Swagger)	http://myprojectdemo.online/docs
MLflow Experiments	https://mlflow-tracking-production-53fb.up.railway.app/
Grafana Dashboard	https://group75mlops.grafana.net/d/suwdlv9/group75-assignment
SonarCloud Analysis	https://sonarcloud.io/project/overview?id=sudheer628_Group75-MLops-Assignment
GitHub Repository	https://github.com/sudheer628/group75-mlops-assignment
Container Registry	ghcr.io/sudheer628/group75-mlops-assignment/heart-disease-api

GCP VM Details:

- IP Address: 35.233.155.69
- Domain: myprojectdemo.online

3. Tech Stack

Here's what we used to build this project:

Programming and ML:

- Python 3.12
- scikit-learn for machine learning
- pandas and numpy for data processing
- FastAPI for the REST API

Infrastructure:

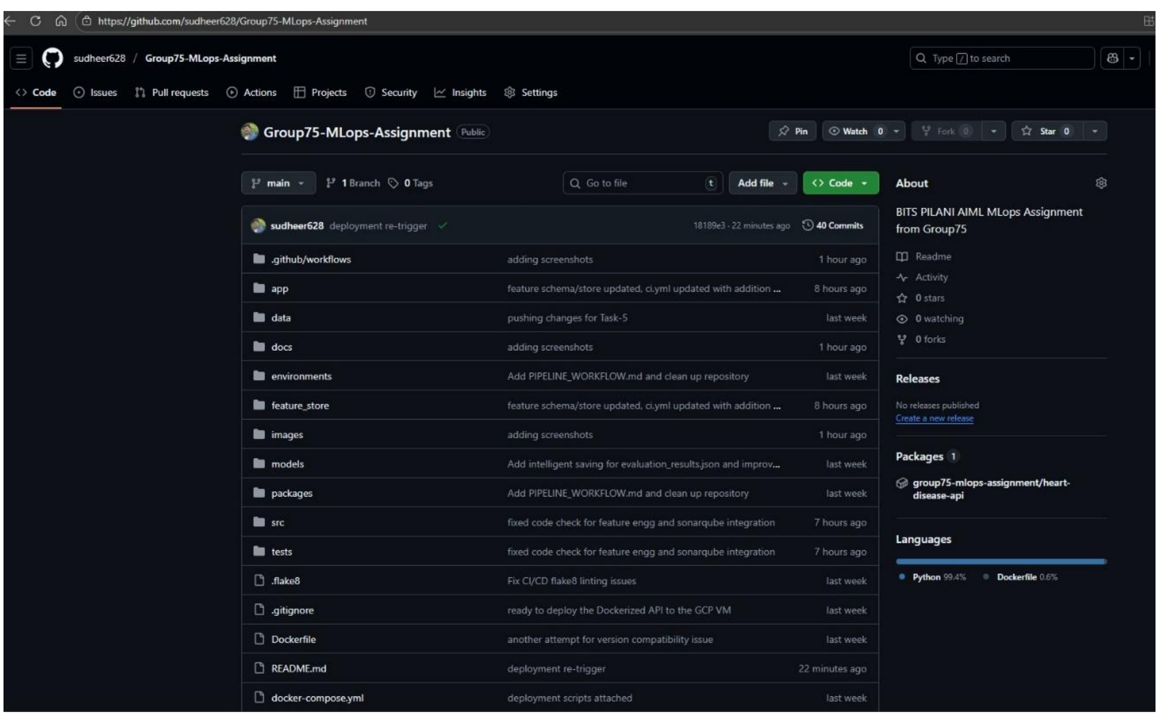
- Docker and Docker Compose for containerization
- Nginx as reverse proxy
- GCP VM (Google Cloud Platform) for hosting

CI/CD and Code Quality:

- GitHub Actions for automation
- GitHub Container Registry (GHCR) for Docker images
- SonarCloud for static code analysis

Monitoring and Tracking:

- MLflow on Railway for experiment tracking
- Grafana Cloud for monitoring dashboards
- Grafana Alloy for metrics and log collection
- Prometheus metrics



4. Task 1: Data Acquisition and EDA

Script: src/data_acquisition_eda.py

For the first task, we downloaded the UCI Heart Disease dataset and performed exploratory data analysis.

Dataset Details:

- Source: UCI Machine Learning Repository (ID: 45)
- Samples: 303 patients
- Features: 13 clinical features + 1 target variable
- Target: Binary classification (0 = no heart disease, 1 = heart disease)

Features in the dataset:

- age - Age in years
- sex - Sex (1 = male, 0 = female)
- cp - Chest pain type (0-3)
- trestbps - Resting blood pressure (mm Hg)
- chol - Serum cholesterol (mg/dl)
- fbs - Fasting blood sugar > 120 mg/dl (1 = true, 0 = false)
- restecg - Resting ECG results (0-2)
- thalach - Maximum heart rate achieved
- exang - Exercise induced angina (1 = yes, 0 = no)
- oldpeak - ST depression induced by exercise
- slope - Slope of peak exercise ST segment
- ca - Number of major vessels colored by fluoroscopy (0-3)
- thal - Thalassemia (1 = normal, 2 = fixed defect, 3 = reversible defect)

What we did:

- Loaded dataset using ucimlrepo library
- Checked for missing values and data quality
- Analyzed target distribution (found it reasonably balanced)
- Examined feature statistics by target class
- Saved processed data to data/processed/ directory

The data quality assessment showed no missing values and no duplicate rows, which made preprocessing straightforward.

5. Task 2: Feature Engineering and Model Training

Script: src/feature_engineering.py

In this task, we engineered new features and trained multiple machine learning models.

Feature Engineering:

We created 6 new engineered features to improve model performance:

- age_risk - Age-based risk category (higher age = higher risk)
- bp_category - Blood pressure category based on trestbps
- heart_rate_reserve - Difference between max heart rate and resting
- cholesterol_ratio - Cholesterol relative to age
- cardiac_index - Combined cardiac health indicator
- exercise_capacity - Exercise tolerance score

Models Trained:
We trained and compared 4 different models:

- Logistic Regression
- Random Forest
- Gradient Boosting
- Support Vector Machine (SVM)

Training Process:

- Split data into 80% training and 20% test sets
- Used stratified sampling to maintain class balance
- Applied RobustScaler for feature scaling
- Performed 5-fold cross-validation
- Did hyperparameter tuning using GridSearchCV

Results:
After evaluation, Logistic Regression performed best with:

- ROC-AUC: 0.96
- Accuracy: ~85%
- Good balance between precision and recall

We saved all trained models to the models/ directory, with the best model saved as best_model.joblib.

6. Task 3: Experiment Tracking with MLflow

Script: src/experiment_tracking.py

For experiment tracking, we used MLflow hosted on Railway (a cloud platform).

MLflow Dashboard: https://mlflow-tracking-production-53fb.up.railway.app/

What we tracked:

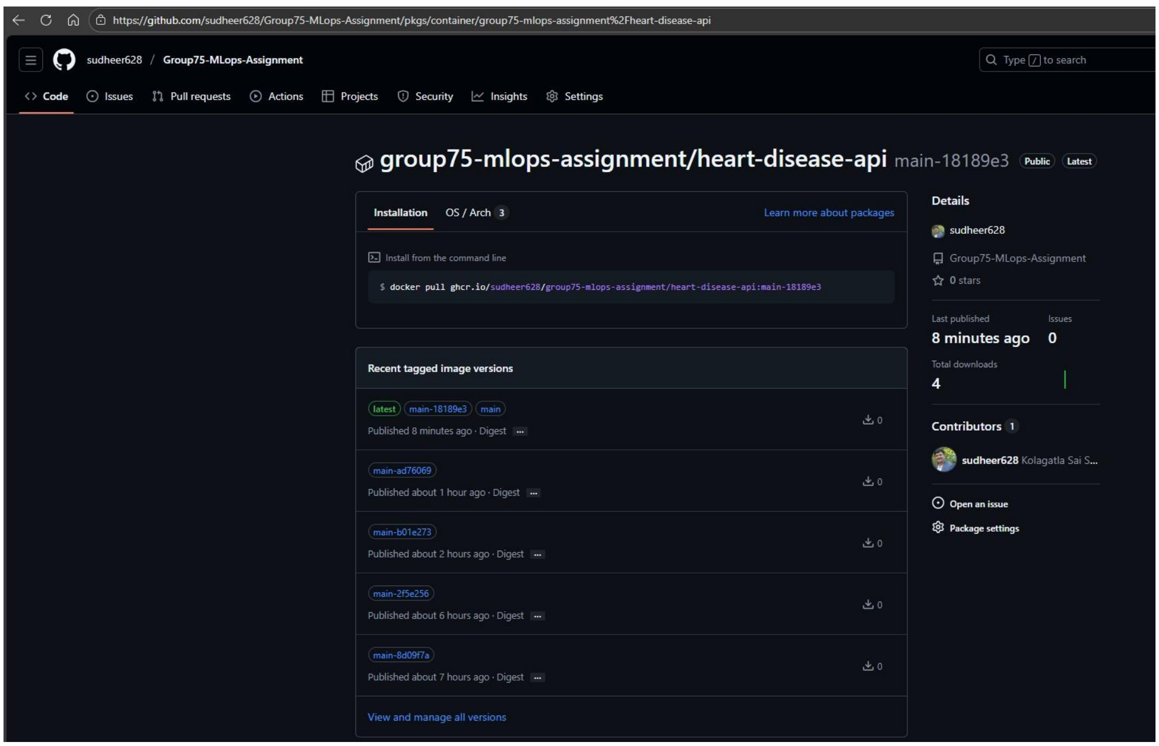
- Model parameters (hyperparameters for each model)
- Training metrics (accuracy, precision, recall, F1, ROC-AUC)
- Model artifacts (trained model files)
- Feature importance plots
- Confusion matrices

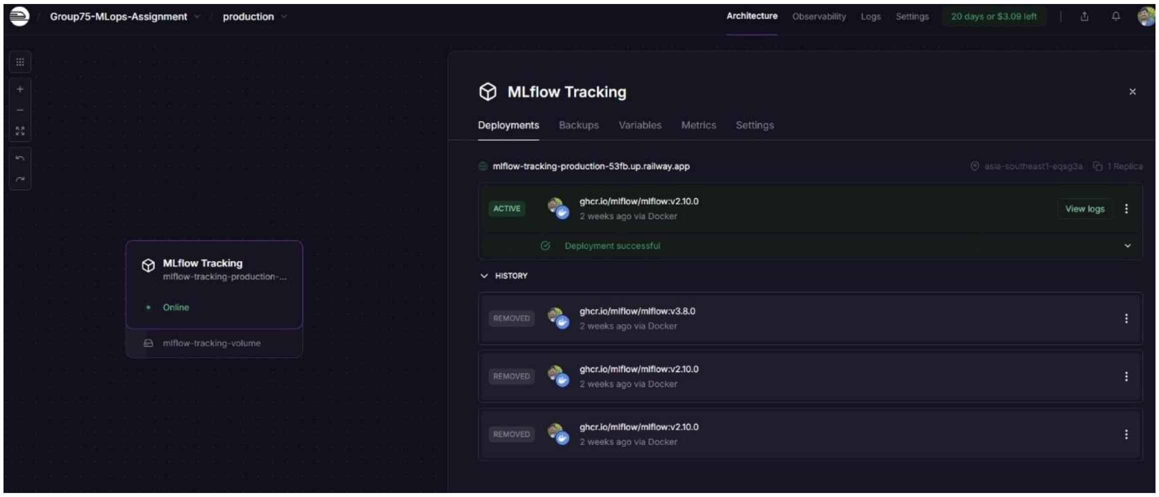
Why Railway:
We chose Railway to host MLflow because:

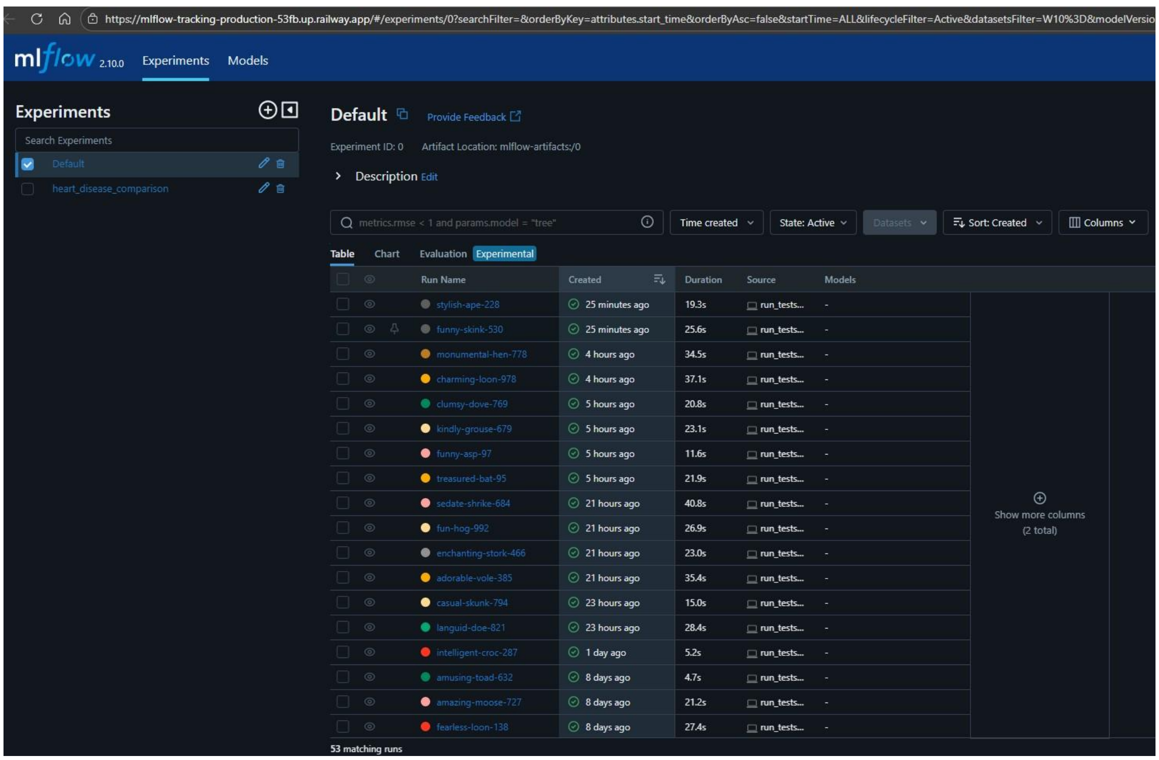
- Free tier available for small projects
- Easy deployment from GitHub
- Persistent storage for experiments
- Accessible from anywhere (not just local)

The experiment tracking helps us:

- Compare different model versions
- Reproduce experiments
- Track what parameters worked best
- Share results with team members







7. Task 4: Model Packaging

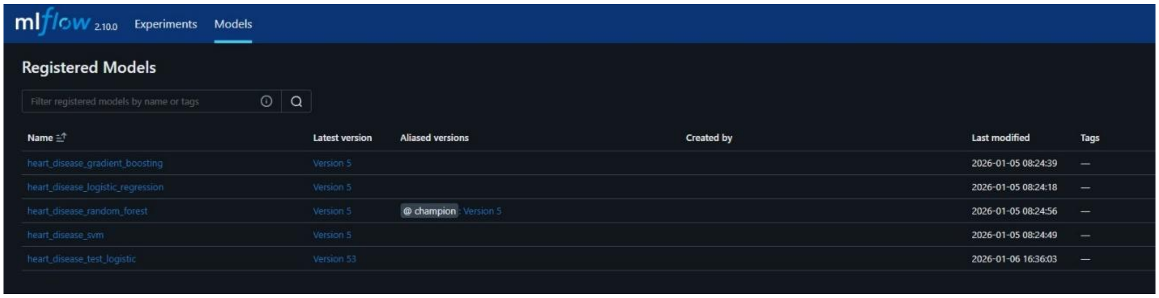
Script: src/model_packaging.py

In this task, we packaged the trained model with all its dependencies for deployment.

What we packaged:

- Trained model file (best_model.joblib)
- Feature names and schemas
- Preprocessing pipeline
- Model metadata (version, training date, metrics)

The packaging script creates a self-contained package that can be deployed anywhere. The model is also uploaded to MLflow's model registry for versioning.



8. Task 5: CI/CD Pipeline

Location: .github/workflows/

We set up GitHub Actions to automate testing, building, and deployment. We have 5 workflow files:

ci.yml - Main CI Pipeline

- Runs on every push to main/develop
- Linting with flake8, black, isort
- Feature store validation
- Unit tests on Python 3.11 and 3.12
- SonarCloud analysis

container-build.yml - Container Build

- Triggers after CI passes
- Builds Docker image
- Tests the container

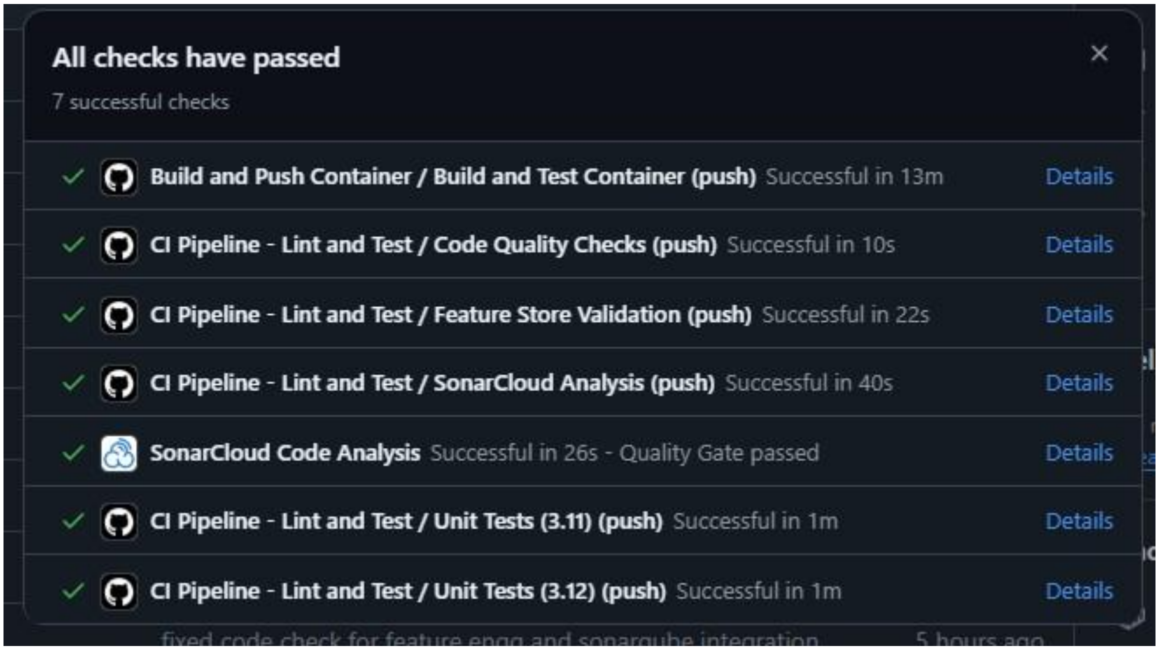
Pushes to GitHub Container Registry

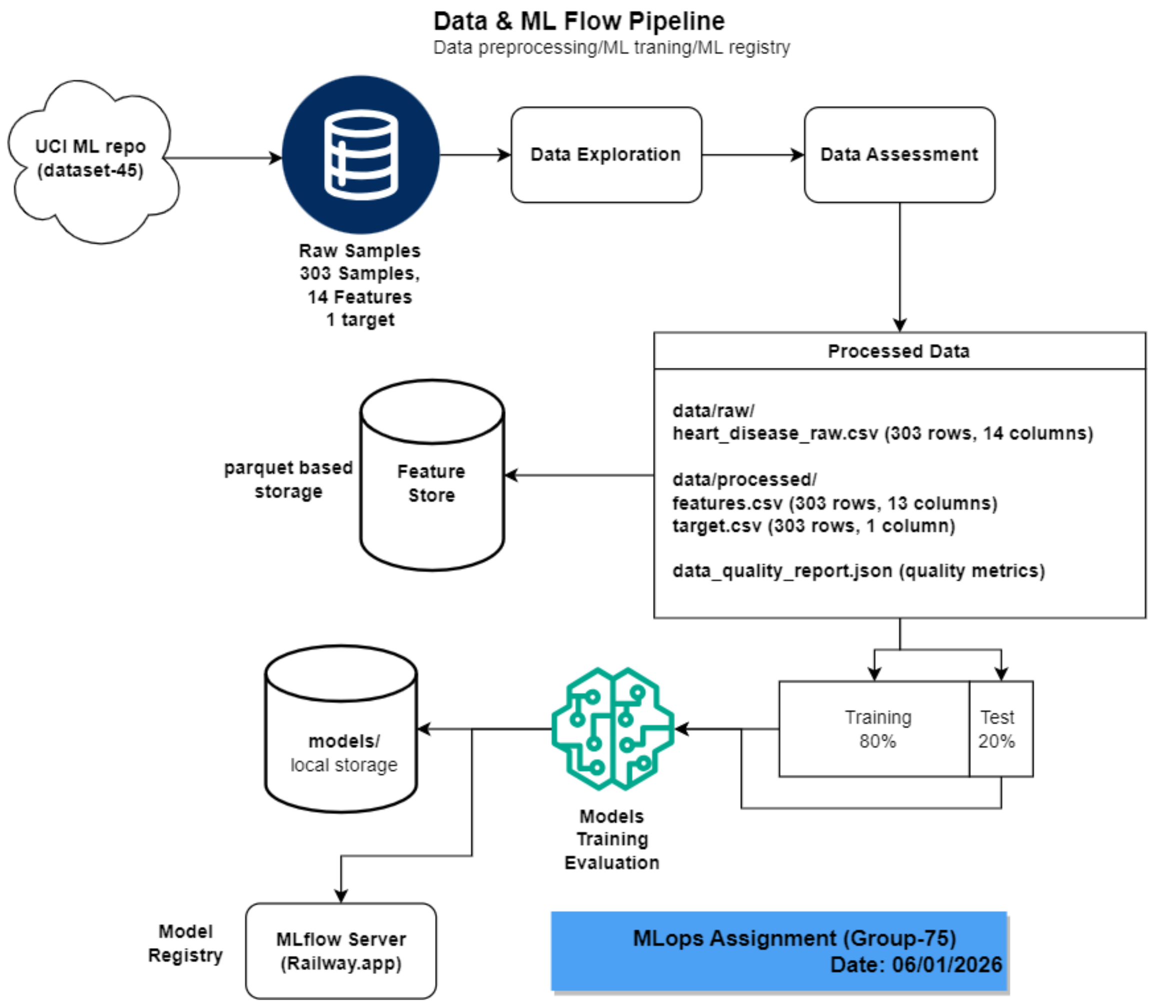
- deploy.yml - Deploy to GCP
- Triggers after container build
- SSHs into GCP VM
- Pulls latest code and Docker image
- Restarts containers

Runs health check

- pr-validation.yml - PR Validation
- Quick checks for pull requests
- Faster feedback for developers
- model-training.yml - Model Training
- Manual trigger or weekly schedule
- Retrains models and uploads to MLflow

Pipeline Flow:





Total deployment time: About 6-10 minutes from push to production.

9. Task 6: Containerization

Files: Dockerfile, docker-compose.yml, nginx.conf

We containerized the application using Docker for consistent deployments.

Docker Setup:

- FastAPI application runs in one container (port 8000)
- Nginx runs in another container as reverse proxy (port 80)
- Both containers are orchestrated with Docker Compose

Dockerfile highlights:

- Based on Python 3.12 slim image
- Installs only production dependencies

- Downloads model from MLflow on startup
- Exposes port 8000 for the API

docker-compose.yml:

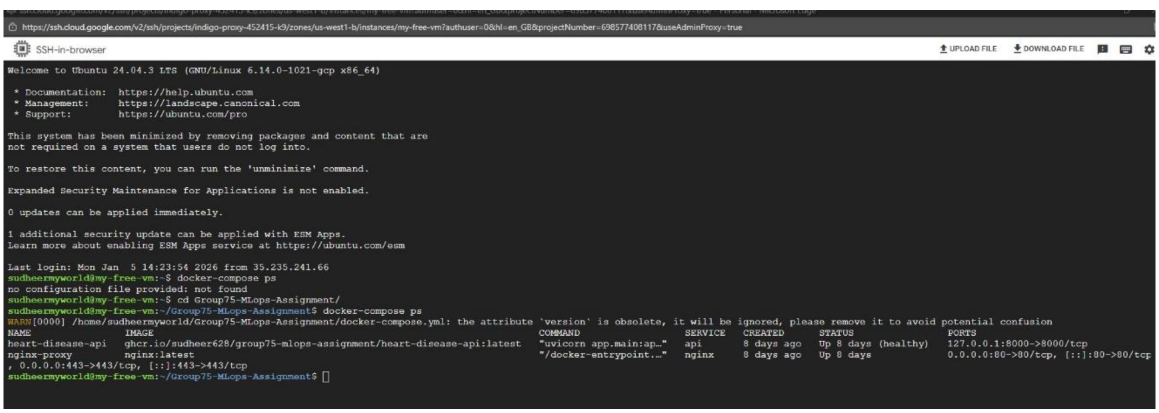
- Defines two services: api and nginx
- Sets up internal network between containers
- Configures health checks
- Handles automatic restarts

Nginx Configuration:

- Routes external traffic to FastAPI
- Handles HTTP on port 80
- Proxies requests to internal port 8000

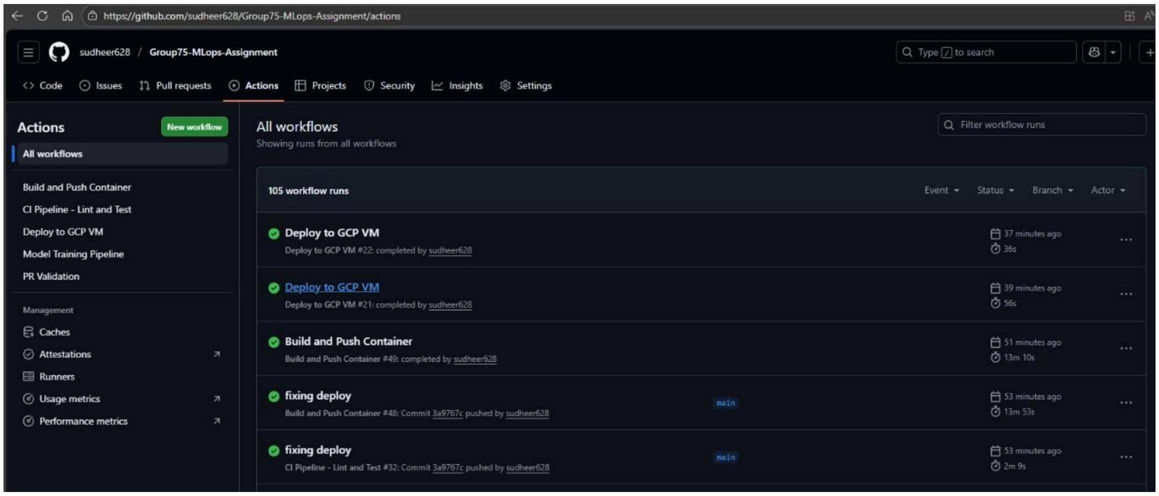
To run locally with Docker:

```
docker-compose up -d
```



10. Task 7: Cloud Deployment

So far, we have finished the ML work flow jobs (CI CD)



For deployment, We have chosen Google Cloud Platform VM.

GCP VM Setup:

- VM Name: my-free-vm
- IP Address: 35.233.155.69
- Domain: myprojectdemo.online
- OS: Ubuntu/Debian

Initial Setup Steps:

1. Created GCP VM instance
2. Installed Docker and Docker Compose
3. Cloned the repository
4. Configured firewall to allow HTTP traffic
5. Set up DNS A record pointing domain to VM IP

Automated Deployment:
We set up GitHub Actions to automatically deploy when we push to main:

1. GitHub Actions SSHs into the VM
2. Pulls latest code from GitHub
3. Pulls latest Docker image from GHCR
4. Restarts containers with docker-compose
5. Runs health check to verify deployment

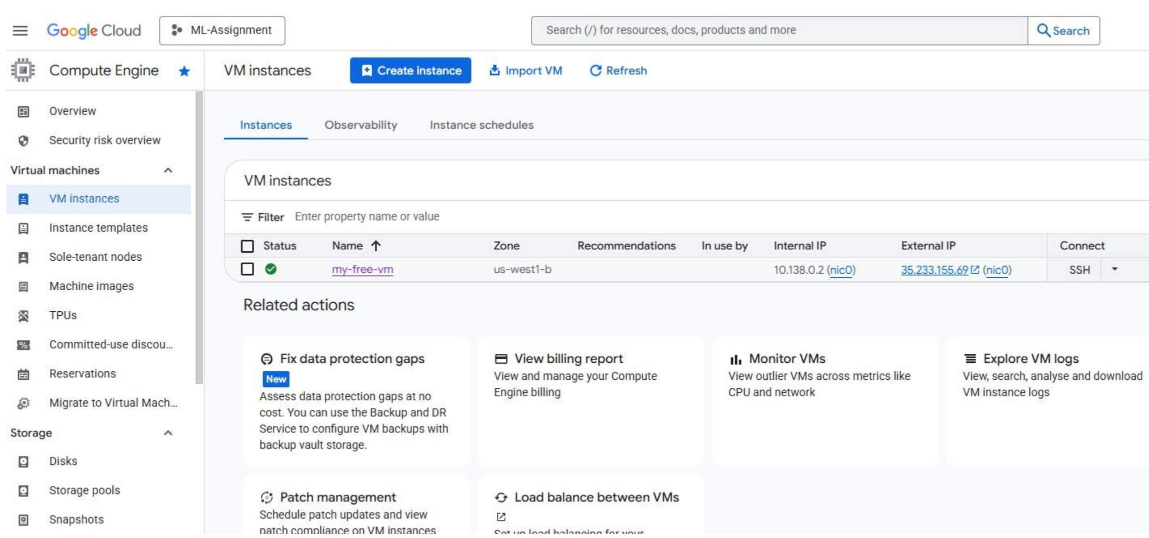
GitHub Secrets Required:

- GCP_VM_HOST: 35.233.155.69
- GCP_VM_USER: <username>
- GCP_VM_SSH_KEY: Private SSH key for authentication

SSH Key Setup:
We generated an SSH key pair and added the public key to GCP VM metadata (via GCP Console -> Compute Engine -> Metadata -> SSH Keys). The private key is stored as a GitHub secret.

Request Flow:

```
User Request
     |
     v
myprojectdemo.online (DNS)
     |
     v
35.233.155.69 (GCP VM)
     |
     v
Nginx Container (Port 80)
     |
     v
FastAPI Container (Port 8000)
     |
     v
Response
```



11. Task 8: Monitoring and Logging

We set up Grafana Cloud for monitoring and logging.

Grafana Dashboard: https://group75mlops.grafana.net/

Tools Used:

- Grafana Cloud: Hosted monitoring platform
- Grafana Alloy: Agent running on GCP VM for collecting metrics and logs
- Prometheus: For metrics storage
- Loki: For log aggregation

Metrics Collected:

- heart_disease_predictions_total: Count of predictions by result
- heart_disease_prediction_latency_seconds: How long predictions take
- api_requests_total: Total API requests by endpoint
- api_errors_total: Error counts
- heart_disease_model_loaded: Whether model is loaded (0/1)

Logs Collected:

- FastAPI application logs
- Nginx access and error logs
- Docker container logs

Architecture:

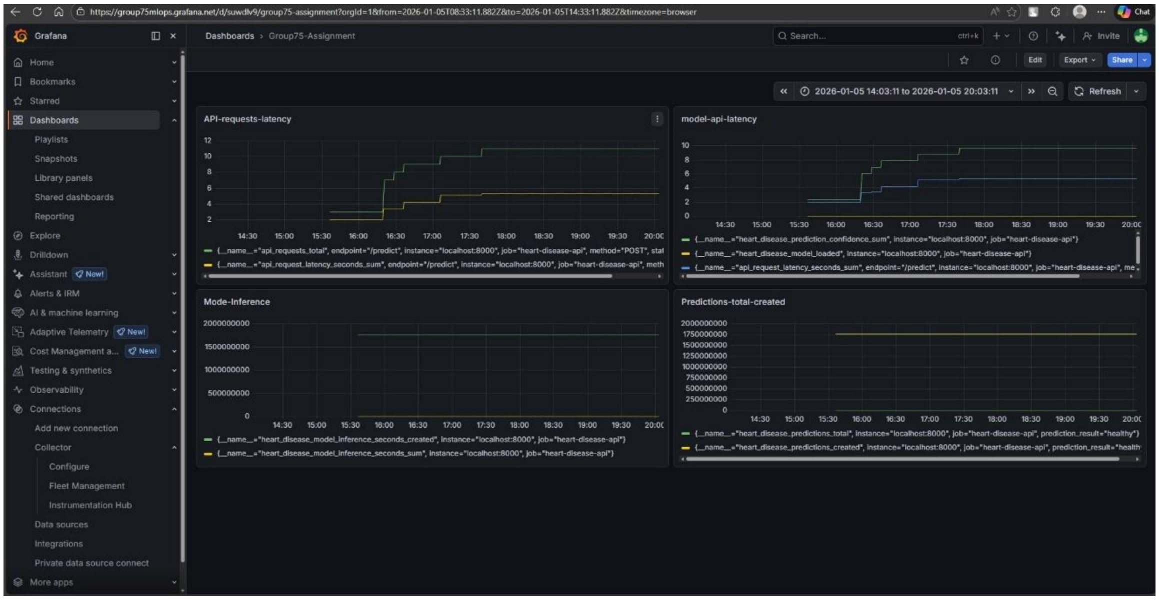
```
GCP VM
  |
  +-- FastAPI Container (/metrics endpoint)
  |
  +-- Nginx Container (logs)
  |
  +-- Grafana Alloy (collects and sends to cloud)
        |
        v
Grafana Cloud
  +-- Prometheus (metrics)
  +-- Loki (logs)
  +-- Dashboards
```

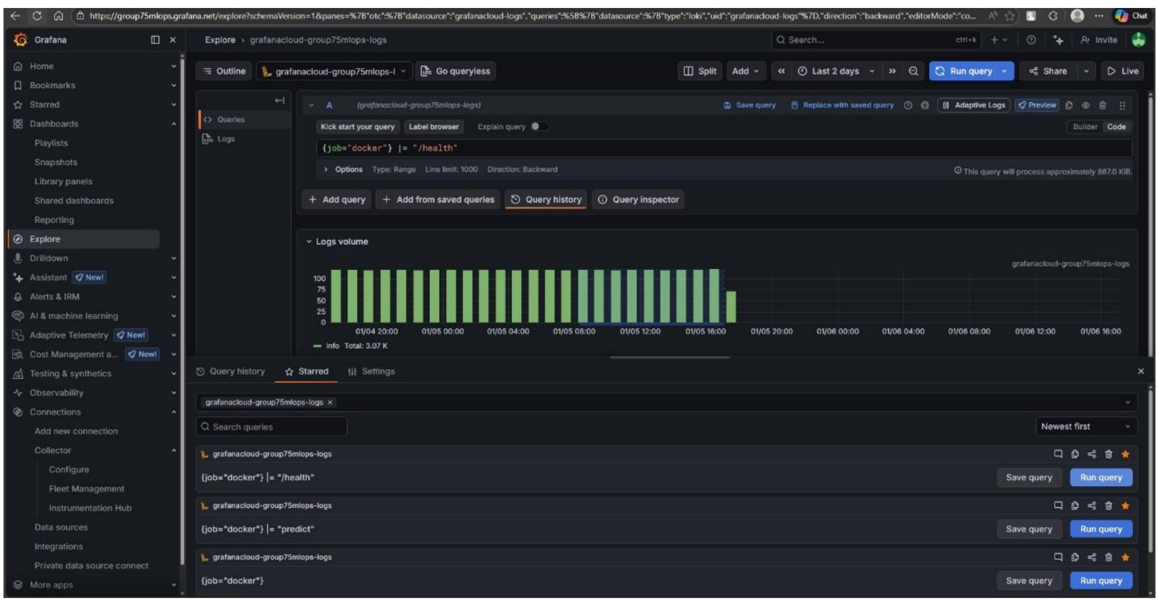
Alloy Configuration:
The Grafana Alloy agent runs as a systemd service on the VM. It:

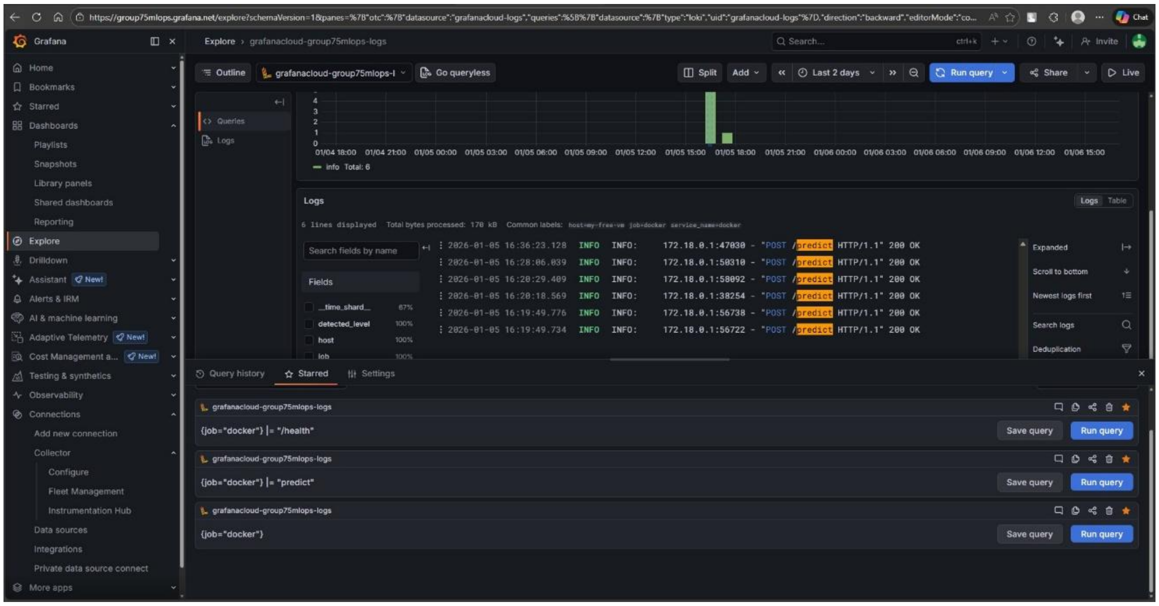
- Scrapes /metrics endpoint every 15 seconds
- Collects Docker container logs
- Sends everything to Grafana Cloud

Dashboards Created:

1. Heart Disease API Monitoring - Shows predictions, latency, errors
2. Logs Dashboard - Live log stream and error filtering







12. Code Quality with SonarCloud

SonarCloud Dashboard: https://sonarcloud.io/project/overview?id=sudheer628_Group75-MLops-Assignment

We integrated SonarCloud for static code analysis to catch bugs and code smells automatically.

Setup Steps:

1. Created SonarCloud account (Free plan) via GitHub login
2. Imported the repository
3. Generated authentication token
4. Added SONAR_TOKEN to GitHub secrets
5. Created sonar-project.properties configuration
6. Added SonarCloud job to CI pipeline

What SonarCloud Checks:

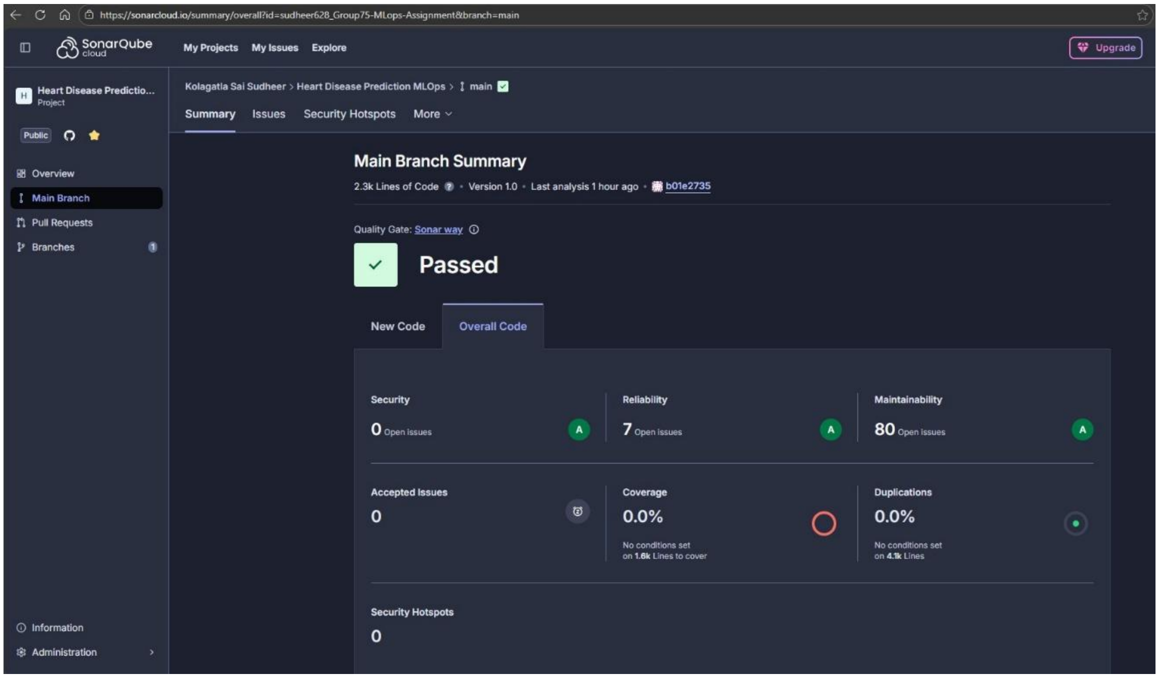
- Bugs and potential bugs
- Security vulnerabilities
- Code smells (maintainability issues)
- Code duplication
- Technical debt

Configuration (sonar-project.properties):

```
sonar.projectKey=sudheer628_Group75-MLops-Assignment
sonar.organization=sudheer628
sonar.sources=src,app
```

```
sonar.tests=tests
sonar.python.version=3.12
```

The analysis runs automatically on every push to main as part of the CI pipeline.



13. Feature Store

Script: src/feature_store.py

We built a simple Parquet-based feature store to solve the training/serving skew problem.

The Problem:
Feature engineering logic was duplicated in two places:

- src/feature_engineering.py (for training)
- app/prediction.py (for inference)

If someone updates one but forgets the other, the model behaves differently in production than during training.

The Solution:
We created a centralized feature store that:

- Defines all features in one place
- Computes features consistently
- Validates feature schemas
- Stores features as Parquet files

Directory Structure:

```
feature_store/
  features/    - Parquet files with computed features
  schemas/     - JSON schema definitions
  metadata/    - Metadata about saved feature sets
```

CI Integration:
The feature store validation runs in CI pipeline:

```
python src/feature_store.py --validate
```

This ensures training and inference always use the same feature logic.

14. API Testing

The API is live at http://myprojectdemo.online

Endpoints:

- GET /health - Health check
- GET /docs - Swagger documentation
- GET /metrics - Prometheus metrics
- GET /model/info - Model information
- POST /predict - Make predictions

Health Check:

```
curl http://myprojectdemo.online/health
```

Make a Prediction (Low Risk - No Heart Disease):

```
curl -X POST http://myprojectdemo.online/predict -H "Content-Type:
application/json" -d "{\"age\": 35, \"sex\": 0, \"cp\": 0,
\"trestbps\": 110, \"chol\": 180, \"fbs\": 0, \"restecg\": 0,
\"thalach\": 175, \"exang\": 0, \"oldpeak\": 0.0, \"slope\": 2, \"ca\":
0, \"thal\": 2}"
```

Make a Prediction (High Risk - Heart Disease):

```
curl -X POST http://myprojectdemo.online/predict -H "Content-Type:
application/json" -d "{\"age\": 65, \"sex\": 1, \"cp\": 3,
\"trestbps\": 160, \"chol\": 300, \"fbs\": 1, \"restecg\": 2,
```

```
\"thalach\": 100, \"exang\": 1, \"oldpeak\": 3.5, \"slope\": 0, \"ca\":
3, \"thal\": 3}"
```

Sample Response:

```
{
  "prediction": 1,
  "confidence": 0.85,
  "probabilities": [0.15, 0.85],
  "risk_level": "High"
}
```

Input Features:

- age: Patient age in years
- sex: 1 = male, 0 = female
- cp: Chest pain type (0-3)
- trestbps: Resting blood pressure
- chol: Serum cholesterol
- fbs: Fasting blood sugar > 120 mg/dl
- restecg: Resting ECG results
- thalach: Maximum heart rate
- exang: Exercise induced angina
- oldpeak: ST depression
- slope: Slope of ST segment
- ca: Number of major vessels (0-3)
- thal: Thalassemia type

15. Project Structure

```
Group75-MLops-Assignment/
|
+-- app/                          # FastAPI application
|   +-- main.py                   # API endpoints
|   +-- prediction.py             # Prediction logic
|   +-- models.py                 # Pydantic models
|   +-- metrics.py                # Prometheus metrics
|   +-- model_loader.py           # Model loading from MLflow
```

```
|   +-- config.py                   # Configuration
|
+-- src/                            # ML pipeline scripts
|   +-- data_acquisition_eda.py     # Task 1: Data acquisition
|   +-- feature_engineering.py      # Task 2: Feature engineering
|   +-- experiment_tracking.py      # Task 3: MLflow tracking
|   +-- model_packaging.py          # Task 4: Model packaging
|   +-- feature_store.py            # Feature store
|
+-- tests/                          # Unit tests
|   +-- test_task1.py
|   +-- test_task2.py
|   +-- test_task3.py
|   +-- test_task4.py
|   +-- test_feature_store.py
|
+-- .github/workflows/              # CI/CD pipelines
|   +-- ci.yml
|   +-- container-build.yml
|   +-- deploy.yml
|   +-- pr-validation.yml
|   +-- model-training.yml
|
+-- data/                           # Data files
|   +-- raw/                        # Raw dataset
|   +-- processed/                  # Processed data
|
+-- models/                         # Trained models
|   +-- best_model.joblib
|   +-- evaluation_results.json
|   +-- feature_names.json
|
+-- feature_store/                  # Feature store
|   +-- features/
|   +-- schemas/
|   +-- metadata/
|
+-- docs/                           # Documentation
|   +-- SETUP.md
|   +-- PIPELINE_WORKFLOW.md
|   +-- DEPLOYMENT-PLAN.md
|   +-- MONITORING-PLAN.md
|   +-- SONARQUBE.md
|
+-- images/                         # Screenshots and diagrams
|
```

```
+-- Dockerfile                      # Container definition
+-- docker-compose.yml              # Container orchestration
+-- nginx.conf                      # Nginx configuration
+-- requirements.txt                # Python dependencies
+-- requirements-api.txt            # API-specific dependencies
+-- sonar-project.properties        # SonarCloud config
+-- run_tests.py                    # Test runner
+-- README.md                       # Project readme
```

16. How to Run Locally

Prerequisites:

- Python 3.12
- Conda (recommended) or pip
- Docker (optional, for containerized run)

Setup Steps:

1. Clone the repository:

```
git clone https://github.com/sudheer628/group75-mlops-assignment.git
cd group75-mlops-assignment
```

2. Create conda environment:

```
conda create -n myenv python=3.12
conda activate myenv
```

3. Install dependencies:

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

4. Run tests to verify setup:

```
python run_tests.py
```

5. Run individual tasks:

```
python src/data_acquisition_eda.py      # Task 1
python src/feature_engineering.py       # Task 2
python src/experiment_tracking.py       # Task 3
python src/model_packaging.py           # Task 4
```

6. Run with Docker (optional):

```
docker-compose up -d
```

The API will be available at http://localhost:8000

17. Conclusion

In this assignment, we successfully built an end-to-end MLOps pipeline for heart disease prediction. We covered all the major aspects of taking a machine learning model from development to production:

What we accomplished:

- Built a binary classifier with 96% ROC-AUC score
- Set up automated CI/CD with GitHub Actions
- Containerized the application with Docker
- Deployed to GCP VM with automated deployments
- Implemented monitoring with Grafana Cloud
- Added code quality checks with SonarCloud
- Created a feature store for training/inference consistency

Key learnings:

- MLOps is about more than just training models - deployment, monitoring, and automation are equally important
- CI/CD pipelines save a lot of manual work and reduce errors
- Containerization makes deployments consistent across environments
- Monitoring helps catch issues before users report them
- Feature stores prevent training/serving skew