Heart Disease Prediction MLOps Pipeline

End-to-End Machine Learning Operations Implementation

|  |
| --- |
| **GROUP 2** |
| **Sl.**  **No.** | **Name** | **BITS ID** |
| **1** | **Srikanta Kumar Sahoo** | **2024AA05098** |
| **2** | **Pradeep Upadhyay** | **2024AA05260** |
| **3** | **Mada Akhila** | **2024AA05982** |
| **4** | **Avirup Mondal** | **2024AB05119** |
| **5** | **Anirudh Sharma** | **2024AB05207** |

## Dataset: Heart Disease UCI Dataset

## Source: https://archive.ics.uci.edu/dataset/45/heart+disease

# Table of Contents

1. Executive Summary

2. Project Overview

3. System Architecture

4. Technology Stack

5. Implementation Details

6. ML Pipeline Components

7. Deployment Strategy

8. Monitoring & Observability

9. CI/CD Pipeline

10. Testing & Validation

11. Project Structure

12. Future Enhancements

13. Conclusion

# 1. Executive Summary

The Heart Disease Prediction MLOps Pipeline is a production-grade machine learning solution designed to predict heart disease risk using patient health metrics. This project demonstrates a complete end-to-end MLOps implementation, integrating modern DevOps practices with machine learning workflows.

## Key Achievements:

* Automated ML pipeline with cross-platform support (Windows, Linux, macOS)
* Dynamic model loading with auto-training capabilities
* Containerized deployment using Docker and Kubernetes
* Real-time monitoring with Prometheus metrics
* Experiment tracking using MLflow
* CI/CD automation with GitHub Actions
* RESTful API with FastAPI framework
* Interactive web UI for predictions

## Business Impact:

Early detection of heart disease can significantly reduce mortality rates. This automated pipeline enables healthcare providers to quickly assess patient risk levels, facilitating timely interventions and improving patient outcomes.

# 2. Project Overview

## 2.1 Problem Statement

Heart disease remains one of the leading causes of death globally. Early prediction and intervention are crucial for patient survival. Traditional manual assessment methods are time-consuming and prone to human error. This project addresses these challenges by providing an automated, scalable, and accurate prediction system.

## 2.2 Solution Approach

The solution implements a complete MLOps pipeline that automates the entire machine learning lifecycle from data acquisition to model deployment and monitoring. The system compares multiple algorithms (Logistic Regression and Random Forest) and automatically selects the best-performing model based on evaluation metrics.

## 2.3 Dataset

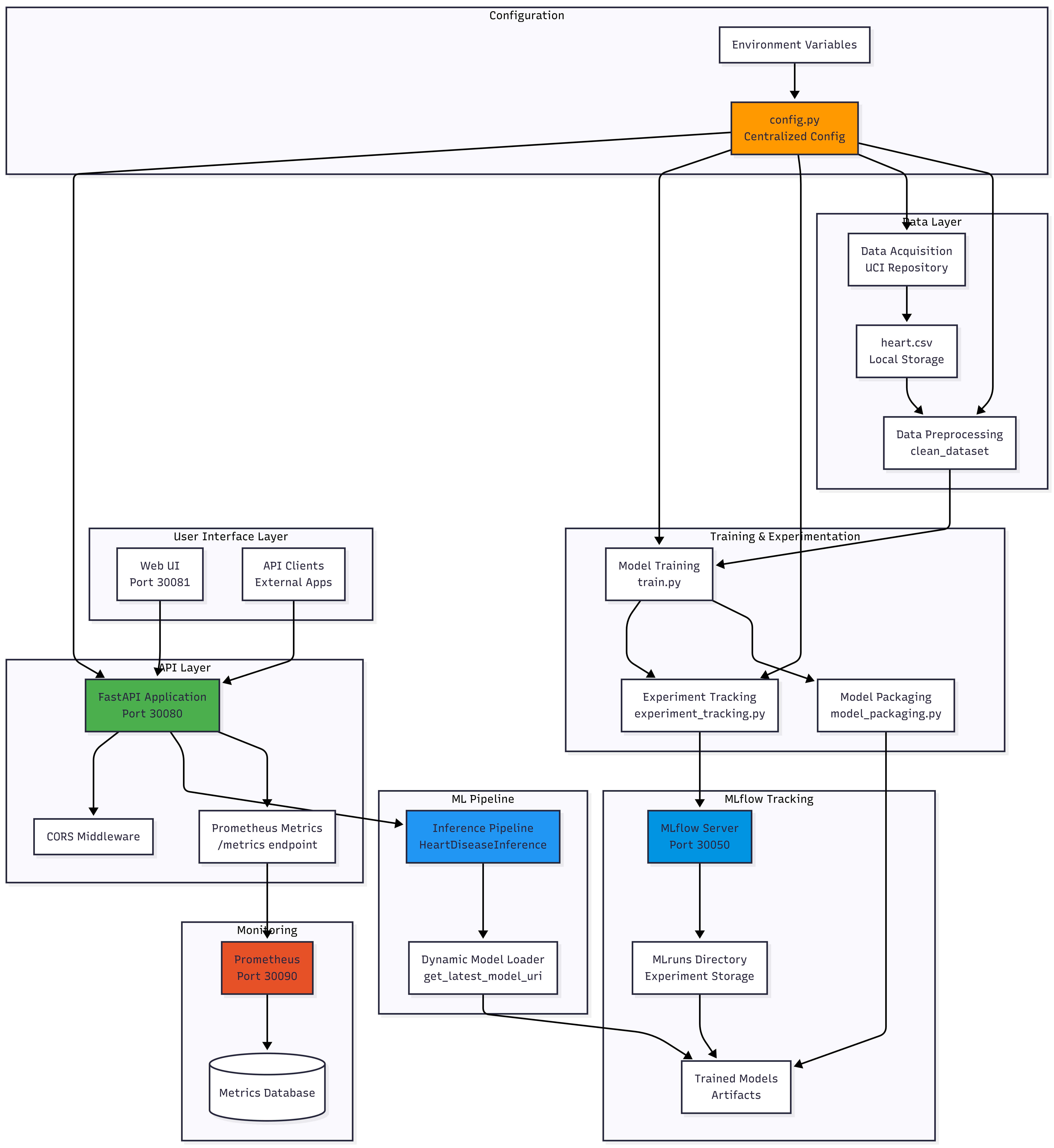
The project uses the UCI Heart Disease dataset, which contains 13 clinical features including:

* Age: Patient age in years
* Sex: Gender (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 electrocardiographic 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 (0-2)
* CA: Number of major vessels colored by fluoroscopy (0-3)
* Thal: Thalassemia (0-3)

# 3. System Architecture

## 3.1 High-Level Architecture

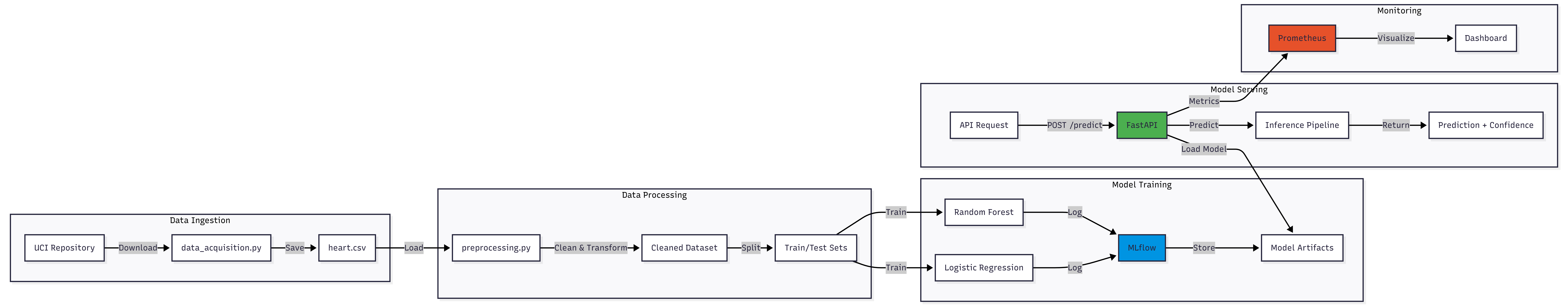
The system follows a microservices architecture with the following components:



* Data Layer: UCI dataset acquisition and storage
* Processing Layer: Data preprocessing and feature engineering
* Training Layer: Model training with hyperparameter tuning
* Tracking Layer: MLflow experiment tracking and model registry
* API Layer: FastAPI REST endpoints for predictions
* UI Layer: Web interface for user interactions
* Monitoring Layer: Prometheus metrics collection
* Orchestration Layer: Kubernetes deployment and scaling

## 3.2 Data Flow

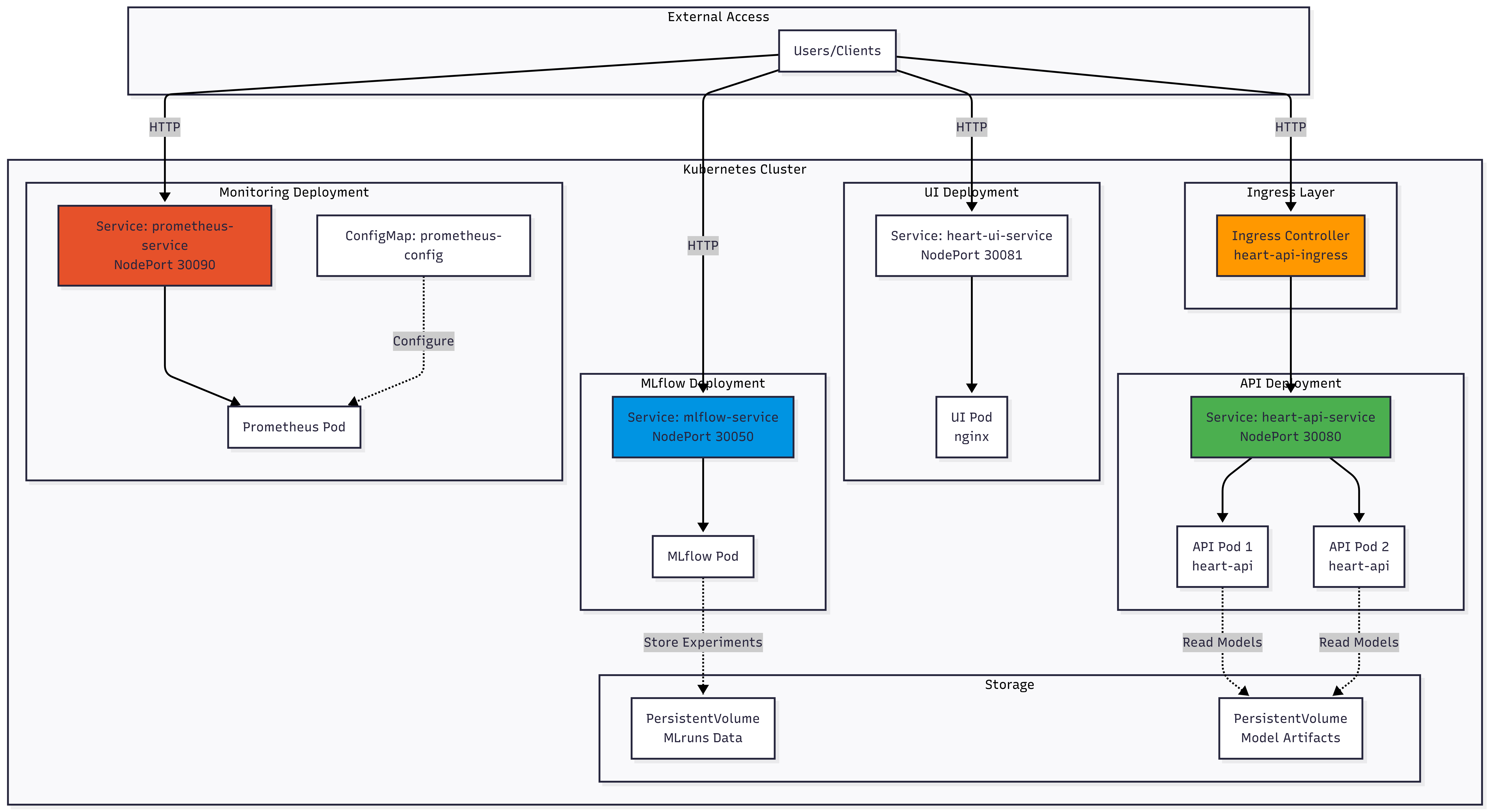
The data flows through the following stages:



1. Data Acquisition → Downloads dataset from UCI repository
2. Preprocessing → Cleans data, handles missing values, encodes categorical features
3. Model Training → Trains multiple models with cross-validation
4. Experiment Tracking → Logs metrics, parameters, and artifacts to MLflow
5. Model Selection → Selects best model based on performance metrics
6. Model Packaging → Serializes model for deployment
7. API Deployment → Serves model via REST API
8. Prediction → Accepts patient data and returns risk assessment

## 3.3 Deployment Architecture

The application supports multiple deployment strategies:



* Local Development: Direct Python execution with virtual environment
* Docker Deployment: Containerized API and UI services
* Kubernetes Deployment: Orchestrated multi-pod deployment with auto-scaling
* Cloud Deployment: AWS deployment via GitHub Actions CI/CD

# 4. Technology Stack

## 4.1 Core Technologies

|  |  |  |
| --- | --- | --- |
| Category | Technology | Purpose |
| Programming Language | Python 3.9 | Core development language |
| ML Framework | Scikit-Learn | Model training and evaluation |
| Data Processing | Pandas, NumPy | Data manipulation and analysis |
| API Framework | FastAPI | RESTful API development |
| Experiment Tracking | MLflow | Model versioning and tracking |
| Containerization | Docker | Application containerization |
| Orchestration | Kubernetes | Container orchestration |
| Monitoring | Prometheus | Metrics collection and monitoring |
| CI/CD | GitHub Actions | Automated testing and deployment |
| Testing | Pytest | Unit and integration testing |
| Code Quality | Flake8 | Linting and code standards |
| Cloud Platform | AWS ECR/App Runner | Cloud deployment target |

## 4.2 Machine Learning Algorithms

Logistic Regression:

* Linear classification algorithm
* Fast training and inference
* Interpretable coefficients
* Baseline model for comparison

Random Forest Classifier:

* Ensemble learning method
* Hyperparameter tuning with GridSearchCV
* Handles non-linear relationships
* Feature importance analysis

# 5. Implementation Details

## 5.1 Project Structure

The project follows a modular structure:

heart-disease-mlops-final/  
├── .github/workflows/ # CI/CD pipeline definitions  
│ ├── ci.yml # Continuous Integration  
│ └── cd.yml # Continuous Deployment  
├── data/ # Dataset storage  
├── docs/ # Documentation files  
│ ├── images/ # Architecture diagrams  
│ ├── videos/ # Demo recordings  
│ └── \*.md # Deployment guides  
├── k8s/ # Kubernetes manifests  
│ ├── deployment.yaml # API deployment  
│ ├── service.yaml # Service definitions  
│ ├── ui-deployment.yaml  
│ ├── mlflow-deployment.yaml  
│ └── monitoring.yaml # Prometheus configuration  
├── models/ # Serialized models  
├── mlruns/ # MLflow experiment data  
├── notebooks/ # Jupyter notebooks for EDA  
├── report/ # Generated reports  
├── src/ # Source code  
│ ├── app.py # FastAPI application  
│ ├── config.py # Configuration management  
│ ├── data\_acquisition.py  
│ ├── preprocessing.py  
│ ├── train.py # Model training  
│ ├── experiment\_tracking.py  
│ ├── model\_packaging.py  
│ ├── model\_utils.py  
│ └── inference\_pipeline.py  
├── tests/ # Unit tests  
│ ├── test\_api.py  
│ ├── test\_data.py  
│ └── test\_model.py  
├── ui/ # Web UI files  
│ └── index.html  
├── Dockerfile # API container definition  
├── Dockerfile.ui # UI container definition  
├── requirements.txt # Python dependencies  
└── run\_local\_pipeline.py # Pipeline orchestration script

## 5.2 Configuration Management

The project uses a centralized configuration system (config.py) that manages all settings including paths, hyperparameters, API settings, and environment-specific configurations. This approach ensures consistency across different environments and simplifies deployment.

## 5.3 Dynamic Model Loading

The API implements intelligent model discovery that automatically finds and loads the latest trained model from the MLflow tracking directory. If no model is found, the system triggers auto-training on startup, ensuring the API is always ready to serve predictions.

# 

# 6. ML Pipeline Components

## 6.1 Data Acquisition

Automatically downloads the UCI Heart Disease dataset and stores it locally. Includes error handling and validation to ensure data integrity.

## 6.2 Data Preprocessing

* Missing Value Imputation: Handles missing data using appropriate strategies
* Feature Encoding: Converts categorical variables to numerical format
* Data Validation: Ensures data quality and consistency
* Feature Scaling: Standardizes features using StandardScaler
* Train-Test Split: Separates data for training and evaluation

## 6.3 Model Training

The training pipeline includes:

* Pipeline Architecture: Combines preprocessing and model training
* Hyperparameter Tuning: GridSearchCV for Random Forest optimization
* Cross-Validation: 5-fold CV for robust performance estimation
* Model Comparison: Evaluates multiple algorithms side-by-side
* Best Model Selection: Automatically selects top performer

## 6.4 Experiment Tracking

MLflow integration provides comprehensive experiment tracking including:

* Metrics Logging: Accuracy, Precision, Recall, F1-Score
* Parameter Tracking: Model hyperparameters and configurations
* Artifact Storage: Model files, confusion matrices, ROC curves
* Model Registry: Versioned model storage and retrieval
* UI Dashboard: Visual experiment comparison at http://localhost:5000

## 6.5 Model Evaluation Metrics

|  |  |
| --- | --- |
| Metric | Description |
| Accuracy | Overall correctness of predictions |
| Precision | Proportion of positive predictions that are correct |
| Recall | Proportion of actual positives correctly identified |
| F1-Score | Harmonic mean of precision and recall |
| ROC-AUC | Area under the receiver operating characteristic curve |
| Confusion Matrix | Visual representation of prediction performance |

# 7. Deployment Strategy

## 7.1 Local Development

For local development and testing:

1. Clone repository and install dependencies
2. Run pipeline: python run\_local\_pipeline.py
3. Start API: uvicorn src.app:app --reload
4. Access UI: Open ui/index.html in browser
5. View MLflow: mlflow ui (port 5000)

## 7.2 Docker Deployment

Containerized deployment process:

1. Build API image: docker build -t heart-api .
2. Build UI image: docker build -f Dockerfile.ui -t heart-ui .
3. Run containers with docker-compose or manually
4. Access services via exposed ports

## 7.3 Kubernetes Deployment

Production-grade orchestration:

* Multi-pod deployment with replica sets
* Service discovery and load balancing
* NodePort services for external access
* Health checks (liveness and readiness probes)
* Resource limits and requests
* Horizontal pod autoscaling capability
* Persistent volumes for model storage
* ConfigMaps and Secrets for configuration

Access Points:

* API Documentation: http://localhost:30080/docs
* Web UI: http://localhost:30081
* MLflow UI: http://localhost:30050
* Prometheus: http://localhost:30090

## 7.4 Cloud Deployment (AWS)

Automated cloud deployment via GitHub Actions:

* Code push triggers CI/CD pipeline
* Docker image built and pushed to Amazon ECR
* Deployment to AWS App Runner or ECS
* Automatic rollback on failure

# 8. Monitoring & Observability

## 8.1 Prometheus Metrics

The API exposes Prometheus metrics for monitoring:

* api\_requests\_total: Counter for total API requests
* prediction\_latency: Histogram of prediction response times
* model\_version: Gauge tracking current model version
* Health endpoint: /health for liveness checks
* Metrics endpoint: /metrics for Prometheus scraping

To populate the dashboard with metrics, generate API traffic by making prediction requests via the web UI (http://localhost:30081) or directly via API calls to http://localhost:30080/predict. The dashboard will automatically display metrics as requests are processed.

Generating Test Data:

* Storage: emptyDir volume (ephemeral, resets on pod restart)
* Service Type: NodePort on port 30300 for external access
* Datasource: Auto-configured to connect to prometheus-service:9090
* Kubernetes Resources: grafana.yaml contains deployment, service, and ConfigMaps

Configuration:

* Auto-refresh: Dashboard updates every 5 seconds
* Dashboard: Navigate to Dashboards → Browse → "Heart Disease API Metrics"
* Default Credentials: admin / admin
* URL: http://localhost:30300

Access Information:

* Cumulative Requests: Historical view of request growth and usage trends
* API Health Status: Real-time availability indicator using up{job="heart-api"} metric
* Total API Requests: Gauge displaying cumulative request count with color-coded thresholds
* API Request Rate: Time series graph showing requests per second using rate(api\_requests\_total[1m])

Dashboard Components:

* Historical data analysis and trend identification
* Kubernetes-native deployment with ConfigMap-based setup
* Auto-provisioned Prometheus datasource configuration
* Pre-configured dashboard with essential API monitoring panels
* Real-time metric visualization with 5-second auto-refresh

Grafana provides a powerful visualization layer on top of Prometheus, offering rich, interactive dashboards for monitoring API metrics. The integration enables real-time visibility into system health, performance trends, and usage patterns through customizable, auto-refreshing dashboards.

## 8.2 Grafana Dashboards

## 8.3 Logging Strategy

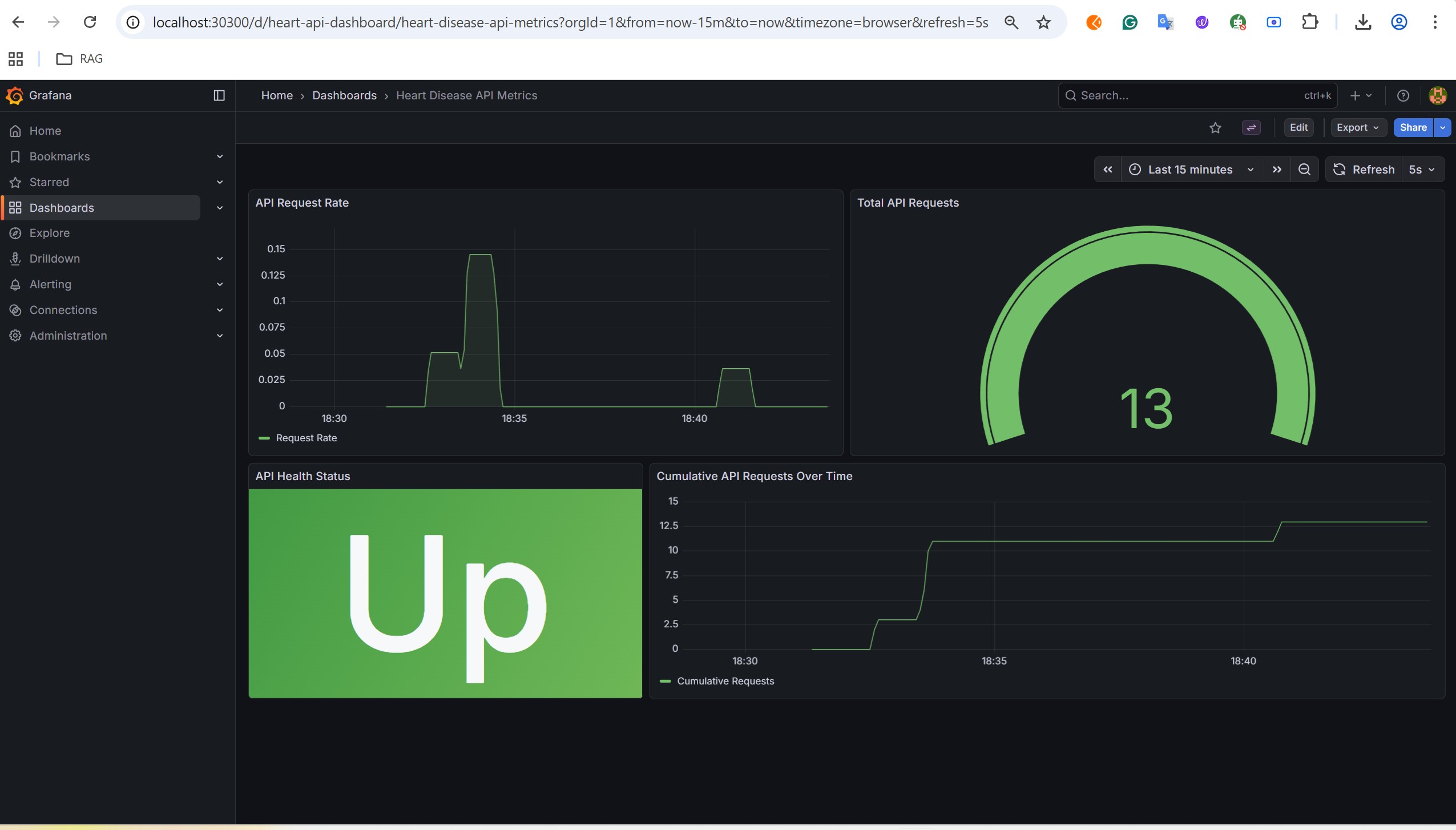
Comprehensive logging implementation:

* Structured logging with configurable levels
* Request/response logging for debugging
* Error tracking with stack traces
* Performance metrics logging
* Centralized log aggregation ready

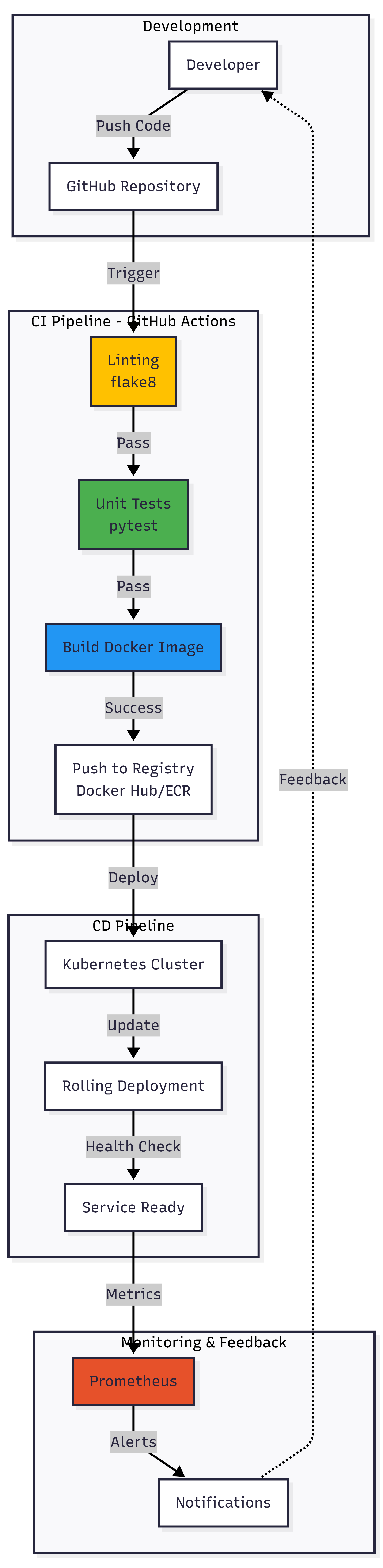
## 8.4 Health Checks

Multiple health check mechanisms:

* API health endpoint: Returns service status
* Kubernetes liveness probe: Restarts unhealthy pods
* Kubernetes readiness probe: Controls traffic routing
* Model availability check: Ensures model is loaded



# 9. CI/CD Pipeline



## 

## 9.1 Continuous Integration (ci.yml)

Automated quality checks on every push/PR:

1. Environment Setup: Python 3.9 installation
2. Dependency Installation: pip install -r requirements.txt
3. Code Linting: Flake8 checks for PEP 8 compliance
4. Unit Tests: Pytest execution with coverage
5. Integration Tests: End-to-end pipeline validation
6. Test Reporting: Coverage reports and test results

## 9.2 Continuous Deployment (cd.yml)

Automated deployment pipeline:

1. Trigger: Runs after successful CI on main branch
2. AWS Authentication: Configure AWS credentials
3. Docker Build: Create production image
4. ECR Push: Upload image to Amazon ECR
5. Deployment: Deploy to AWS App Runner/ECS
6. Verification: Health check validation

## 9.3 GitHub Actions Workflow

The CI/CD pipeline provides:

* Automated testing on every commit
* Fast feedback on code quality issues
* Consistent build and deployment process
* Reduced manual deployment errors
* Version tracking and rollback capability
* Parallel job execution for speed

**10. Testing & Validation**

## 10.1 Unit Tests

Comprehensive test coverage:

* test\_data.py: Data acquisition and preprocessing validation
* test\_model.py: Model training and evaluation tests
* test\_api.py: API endpoint functionality tests
* Fixtures: Reusable test data and configurations
* Mocking: Isolated component testing

## 10.2 Integration Tests

End-to-end validation:

* Full pipeline execution test
* API request/response validation
* Model loading and inference test
* MLflow integration verification
* Docker container health checks

## 10.3 Model Validation

Ensuring model quality:

* Cross-validation for robust performance estimation
* Holdout test set evaluation
* Confusion matrix analysis
* ROC curve and AUC calculation
* Performance comparison across algorithms
* Threshold tuning for optimal predictions

# 11. API Documentation

## 11.1 Endpoints

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Endpoint | Description | Response |
| GET | / | Health check | Status message |
| POST | /predict | Heart disease prediction | Prediction result |
| GET | /metrics | Prometheus metrics | Metrics data |
| GET | /docs | API documentation | Swagger UI |

## 11.2 Request Schema

The /predict endpoint accepts JSON with the following fields:

{  
 "age": 63,  
 "sex": 1,  
 "cp": 3,  
 "trestbps": 145,  
 "chol": 233,  
 "fbs": 1,  
 "restecg": 0,  
 "thalach": 150,  
 "exang": 0,  
 "oldpeak": 2.3,  
 "slope": 0,  
 "ca": 0,  
 "thal": 1  
}

## 11.3 Response Schema

Prediction response format:

{  
 "prediction": 1,  
 "probability": 0.85,  
 "risk\_level": "High",  
 "model\_version": "v1.0"  
}

# 12. Conclusion

The Heart Disease Prediction MLOps Pipeline demonstrates a comprehensive implementation of modern machine learning operations practices. The project successfully integrates data science, software engineering, and DevOps principles to create a production-ready system.

## Key Takeaways:

* Automation: Complete pipeline automation from data to deployment
* Scalability: Kubernetes-based architecture supports horizontal scaling
* Reliability: Comprehensive testing and monitoring ensure system stability
* Maintainability: Modular design and clear documentation facilitate updates
* Reproducibility: Version control and experiment tracking enable reproducible results
* Production-Ready: CI/CD pipeline and cloud deployment capabilities

This implementation serves as a template for building MLOps pipelines in healthcare and other domains. The modular architecture and comprehensive documentation make it easy to adapt for different use cases and requirements.

## Project Links:

GitHub Repository: https://github.com/ssrikantasahoo/heart-disease-mlops-final

Documentation: See docs/ folder for detailed guides

API Documentation: http://localhost:30080/docs (when running)

# Appendix A: Installation Guide

Prerequisites:

* Python 3.9 or higher
* Docker Desktop (with Kubernetes enabled)
* Git
* 4GB RAM minimum (8GB recommended)
* Internet connection for dataset download

Installation Steps:

# Clone repository  
git clone https://github.com/ssrikantasahoo/heart-disease-mlops-final.git  
cd heart-disease-mlops-final  
  
# Create virtual environment  
python -m venv venv  
  
# Activate virtual environment  
# Windows:  
.\venv\Scripts\activate  
# Linux/Mac:  
source venv/bin/activate  
  
# Install dependencies  
pip install -r requirements.txt  
  
# Run pipeline  
python run\_local\_pipeline.py  
  
# Start API  
uvicorn src.app:app --reload  
  
# Access UI  
# Open ui/index.html in browser

# Appendix B: Troubleshooting

Issue: Model not found error

Solution: Run python run\_local\_pipeline.py to train models first

Issue: Port already in use

Solution: Change port in config.py or kill process using the port

Issue: Docker image pull error

Solution: Ensure imagePullPolicy: Never in k8s/deployment.yaml

Issue: Kubernetes pods not starting

Solution: Check logs with kubectl logs <pod-name>

Issue: MLflow UI not accessible

Solution: Verify MLflow service is running: kubectl get svc

# Appendix C: References

* UCI Heart Disease Dataset: https://archive.ics.uci.edu/ml/datasets/heart+disease
* FastAPI Documentation: https://fastapi.tiangolo.com/
* MLflow Documentation: https://mlflow.org/docs/latest/index.html
* Kubernetes Documentation: https://kubernetes.io/docs/
* Prometheus Documentation: https://prometheus.io/docs/
* Scikit-Learn Documentation: https://scikit-learn.org/stable/
* Docker Documentation: https://docs.docker.com/