# Technical Report: End-to-End MLOps Pipeline on Azure AKS

**Project:** Heart Disease Prediction System

**Date:** January 05, 2026

**Status:** Deployed & Verified

## 1. Contributors

| **Name** | **Student ID** | **Role/Contribution** |
| --- | --- | --- |
| Gaurav Sharma | 2024AB05112 | Pipeline Orchestration & DevOps |
| HARSHITHA S | 2024AA05282 | Frontend Development (Django) |
| NA.R. YANGYESWAR | 2024AA05285 | Backend API (FastAPI) |
| PRATIMA CHAUHAN | 2024AB05324 | ML Model Integration |
| HATHI SHIVANI NIREN | 2024AB05014 | Infrastructure Management |

## 2. Table of Contents

1. [Executive Summary](https://www.google.com/search?q=%233-executive-summary)
2. [System Architecture & Microservices](https://www.google.com/search?q=%234-system-architecture--microservices)
3. [CI/CD Pipeline Architecture](https://www.google.com/search?q=%235-cicd-pipeline-architecture)
4. [Deployment & Infrastructure Specifications](https://www.google.com/search?q=%236-deployment--infrastructure-specifications)
5. [Testing, Verification & Monitoring](https://www.google.com/search?q=%237-testing-verification--monitoring)

## 3. Executive Summary

This project demonstrates a robust, production-ready **MLOps Lifecycle** for a Heart Disease Prediction System. By decoupling the user interface from the inference engine, the system ensures high scalability and modularity.

Key technical highlights include:

* **Containerized Architecture:** Leveraging Docker for consistency across development and production.
* **Orchestration:** Using Azure Kubernetes Service (AKS) to manage load balancing and self-healing pods.
* **Automated Lifecycle:** A GitHub Actions pipeline that handles code linting, image builds, and infrastructure power-management to reduce cloud expenditure.

## 4. System Architecture & Microservices

The application follows a distributed architecture where the UI and the Model Logic exist in separate namespaces but communicate over a private network.

### 4.1 Frontend Layer (Django)

* **Technology:** Python 3.9+, Django Framework.
* **Functionality:** Handles session management, form validation, and triggers HTTP POST requests to the backend.
* **Security:** Communicates with the backend using internal Kubernetes DNS (e.g., http://backend-service:8000), ensuring the API is never exposed directly to the public internet.

### 4.2 Backend Layer (FastAPI)

* **Technology:** FastAPI, Uvicorn, Scikit-Learn.
* **Functionality:** Exposes a /predict endpoint. Upon receiving data, it performs feature engineering (matching the training pipeline) and generates a prediction using a serialized .pkl model.
* **Performance:** FastAPI's asynchronous nature allows it to handle multiple concurrent inference requests with minimal latency.

### 4.3 Data Flow Diagram

graph LR  
 A[User Browser] -- HTTP/Port 80 --> B[Azure Load Balancer]  
 B --> C[Django Pod]  
 C -- Internal REST API --> D[FastAPI Pod]  
 D -- Local Load --> E[ML Model File]  
 E -- Result --> D  
 D -- JSON --> C  
 C -- Rendered HTML --> A

## 5. CI/CD Pipeline Architecture

The automation strategy focuses on **Efficiency** and **Traceability**.

### 5.1 The "Smart Build" Logic

To optimize GitHub Actions runner time and Azure Container Registry (ACR) storage:

* **Path Filtering:** The pipeline uses paths filters to detect changes. If only the backend/ code is modified, the frontend/ build job is skipped.
* **Layer Caching:** Docker layer caching is utilized to speed up repeated builds.

### 5.2 Workflow Stages

1. **Linting & Testing:** Validates Python code standards (PEP8).
2. **Containerization:** \* Builds images using Dockerfile.
   * Tags: latest and ${GITHUB\_SHA} (for precise version tracking).
3. **Infrastructure Pre-flight:** \* The workflow queries the Azure CLI to check clusterState.
   * If Stopped, it executes az aks start and waits for the "Succeeded" state before proceeding.
4. **Manifest Patching:** Uses envsubst or Kustomize to inject the specific Commit SHA into the deployment.yaml file, ensuring the cluster pulls the exact version just built.

## 6. Deployment & Infrastructure Specifications

### 6.1 Azure Cloud Environment

* **Resource Group:** HeartDiseaseRG (Centralized management for AKS, ACR, and Networking).
* **AKS Cluster:** HeartDiseaseCluster
  + **Node Pool:** Standard\_DS2\_v2 (Balanced CPU/Memory for ML workloads).
  + **Autoscaling:** Enabled to handle traffic spikes.

### 6.2 Kubernetes Resource Management

* **Service (Frontend):** Type LoadBalancer with a fixed DNS label heart-disease-2024ab05112.
* **Service (Backend):** Type ClusterIP (Internal only).
* **Resource Limits:** \* CPU: 250m (Request) / 500m (Limit)
  + Memory: 512Mi (Request) / 1Gi (Limit)

## 7. Testing, Verification & Monitoring

### 7.1 Deployment Verification (CLI)

Once the pipeline completes, the following commands are used to verify cluster health:

# Check if pods are running  
kubectl get pods -l app=heart-disease  
  
# Check the public endpoint  
kubectl get svc heart-disease-frontend-service

### 7.2 Functional Acceptance Test

To confirm the MLOps pipeline is serving the correct model:

1. **Access:** Open http://heart-disease-2024ab05112.eastus.cloudapp.azure.com.
2. **Input:** Enter high-risk parameters (Age: 65, Chol: 300, etc.).
3. **Output:** Verify that the backend returns a classification result with a probability score (e.g., Risk: High | Confidence: 88%).

### 7.3 Log Monitoring

Logs are aggregated via standard output and can be viewed using:

kubectl logs -f deployment/heart-disease-backend

**End of Report**