BITS ID	Name	Contribution
2024AA05041	Amit Kumar	100%
2024AA05042	Yogesh Agarwal	100%
2024AA05633	Yawar Abbas	100%

Heart Disease MLOps Solution: Step-by-Step Report with Code

Introduction & Project Overview

Purpose:
This project builds a full machine learning pipeline to predict heart disease, using the UCI Heart Disease dataset. The solution covers data download, cleaning, EDA, feature engineering, model training, experiment tracking, packaging, deployment, and monitoring.

Workflow Outline:

- Data download & cleaning
- Exploratory data analysis (EDA)
- Feature engineering
- Model training & evaluation
- Experiment tracking (MLflow)
- Model packaging & deployment (Docker, FastAPI, K8s)
- Monitoring & logging (Prometheus, Grafana)
- CI/CD automation (DVC, GitHub Actions)

Project Setup & Environment Checks

Cells 1–2: Project Initialization

Role:

- Sets up folders, checks Docker, and ensures a reproducible environment.

Key Code:

```
from pathlib import Path
import os

# Set up project root and folders
PROJECT_ROOT = Path("C:/.../mlops-assignment-final")
PROJECT_ROOT.mkdir(exist_ok=True)
os.chdir(PROJECT_ROOT)

# Print project structure
print(f"✅ PROJECT: {PROJECT_ROOT.absolute()}")
print("\n✅ FOLDERS:")
print(os.listdir("."))
```

Why:
Ensures all files are organized and the environment is ready for reproducible work.

Data Acquisition & Cleaning

Cells 3–5: Downloading and Preparing Data

Role:

- Downloads the UCI Heart Disease dataset, assigns column names, handles missing values, and saves both raw and cleaned datasets.

Key Code

```
# Download dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data"
columns = ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg',
           'thalach', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'target']
df_raw = pd.read_csv(url, names=columns, na_values='?')
df_raw.to_csv("data/heart_disease_raw.csv", index=False)

# Clean data
df_clean = df_raw.copy()
df_clean['target'] = (df_clean['target'] > 0).astype(int)
# Impute missing values
imputer_num = SimpleImputer(strategy='median')
imputer_cat = SimpleImputer(strategy='most_frequent')
df_clean[numeric_features] = imputer_num.fit_transform(df_clean[numeric_features])
df_clean[categorical_features] = imputer_cat.fit_transform(df_clean[categorical_features])
df = df_clean.dropna()
df.to_csv("data/heart_disease_cleaned.csv", index=False)

```

Why:
Clean, well-structured data is essential for building reliable machine learning models.

Exploratory Data Analysis (EDA)

Cells 6–7: Data Exploration and Visualization

Role:

- Shows summary statistics and creates professional plots for class balance, age, cholesterol, heart rate, and a correlation heatmap.

Key Code:

```
# Class balance plot
sns.countplot(data=df, x='target')
plt.title('Heart Disease Distribution (0=No, 1=Yes)')
plt.savefig('data/task1_eda_plots.png')

# Correlation heatmap
corr_matrix = df.corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Feature Correlation Matrix')
plt.savefig('data/task1_correlation_heatmap.png')
```

Why:
EDA helps us understand the data, spot patterns, and decide which features might be important for prediction.

Feature Engineering & Preprocessing

Cells 8–10: Building the Feature Pipeline

Role:

- Splits data into train/test, builds a preprocessing pipeline (imputation, scaling, encoding), and transforms the data.

Key Code:

```
from sklearn.pipeline import Pipeline

# Split data
X = df.drop('target', axis=1)
y = df['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y)

# Preprocessing pipeline
preprocessor = ColumnTransformer([
    ('num', Pipeline([
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ]), numeric_features),
    ('cat', Pipeline([
        ('imputer', SimpleImputer(strategy='most_frequent')),
        ('onehot', OneHotEncoder(drop='first', sparse_output=False,
handle_unknown='ignore'))
    ]), categorical_features)
])
X_train_processed = preprocessor.fit_transform(X_train)
```

Why:
Consistent preprocessing ensures the model sees data in the same format during training and prediction.

Model Training & Evaluation

Cells 11–13: Training and Comparing Models

Role:

- Trains Logistic Regression and Random Forest models, tunes hyperparameters, and compares their performance.

Key Code:

```python
# Logistic Regression
lr_model = LogisticRegression(random_state=42, max_iter=1000)
lr_model.fit(X_train_processed, y_train)
y_pred_lr = lr_model.predict(X_test_processed)
print("Accuracy:", accuracy_score(y_test, y_pred_lr))

# Random Forest with GridSearchCV
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [5, 10, None],
    'min_samples_split': [2, 5]
}
rf = RandomForestClassifier(random_state=42)
grid_search = GridSearchCV(rf, param_grid, cv=5, scoring='roc_auc')
```

Why:
Comparing models helps us pick the best one for deployment.

Experiment Tracking with MLflow

Cells 14–19: Logging Experiments

Role:

- Sets up MLflow, logs all model parameters, metrics, artifacts, and provides a summary table of experiments.

Key Code:

```python
import mlflow
mlflow.set_tracking_uri("file:./mlruns")
mlflow.set_experiment("Heart_Disease_Experiments")

with mlflow.start_run(run_name="Logistic_Regression_Baseline"):
    mlflow.log_params({'model_type': 'LogisticRegression', 'max_iter': 1000})
    mlflow.log_metrics({'accuracy': accuracy_score(y_test, y_pred_lr)})
    mlflow.sklearn.log_model(lr_model, "model")
```

Why:
MLflow makes it easy to track, compare, and reproduce experiments, which is essential for MLOps.

Model Packaging & Deployment

Cells 20–26: Production Pipeline and API

Role:

- Packages the best model and preprocessing pipeline, creates a FastAPI app, and generates Dockerfiles for deployment.

Key Code:

```python
# Save full pipeline
from sklearn.pipeline import Pipeline
import joblib
full_pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', best_rf)
])
joblib.dump(full_pipeline, "models/heart_disease_pipeline_prod.joblib")

# FastAPI app (main.py)
from fastapi import FastAPI
import joblib
import pandas as pd

app = FastAPI()
model = joblib.load("models/heart_disease_pipeline_prod.joblib")

@app.post("/predict")
def predict(data: dict):
```

Why:
Containerization and APIs make it easy to deploy the model anywhere and serve predictions reliably.

CI/CD Automation & Monitoring

Cells 27–49: Automation, Monitoring, and Reporting

Role:

- Sets up DVC for data/model versioning, GitHub Actions for CI/CD, Kubernetes manifests for deployment, and Prometheus/Grafana for monitoring.

Key Code:

```yaml
# .github/workflows/mlops.yml (GitHub Actions)
name: MLOps CI/CD Pipeline
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  test-train-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python 3.11
        uses: actions/setup-python@v4
```

Why:
Automation and monitoring are key to reliable, maintainable, and observable ML systems in production.

Docker: Containerizing the ML Model API

Role:
Docker packages the ML model and API (FastAPI/Flask) into a portable container. This ensures the solution runs identically on any system.

Key Steps:

- Write a Dockerfile to install dependencies and copy code/model files.
- Expose the /predict endpoint for inference.

Sample Dockerfile:

Dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8080"]
```

How to Build & Run:

```
docker build -t heart-disease-api .
docker run -p 8080:8080 heart-disease-api
```

Kubernetes: Production Deployment & Orchestration

Role:
Kubernetes (K8s) manages deployment, scaling, and reliability of the containerized API. It can run locally (Minikube, Docker Desktop) or on cloud (GKE, EKS, AKS).

Key Steps:

- Write a deployment manifest or Helm chart.
- Expose the API via LoadBalancer or Ingress.

Sample Deployment Manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: heart-disease-api
spec:
 replicas: 2
 selector:
```

```
  matchLabels:
    app: heart-disease-api
 template:
  metadata:
    labels:
      app: heart-disease-api
  spec:
    containers:
    - name: api
      image: heart-disease-api:latest
      ports:
      - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
 name: heart-disease-service
spec:
 type: LoadBalancer
 selector:
  app: heart-disease-api
 ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

Grafana: Monitoring & Visualization

Role:
Grafana visualizes metrics collected from the API, such as request rates, errors, and latency. It connects to Prometheus as a data source.

Key Steps:

- Install Grafana and Prometheus (often via Helm charts).
- Import a dashboard JSON to visualize API metrics.

Sample Grafana Dashboard Setup:

- Add Prometheus as a data source.
- Import dashboard JSON (provided in your repo).
- View real-time metrics for API health.

Loki: Centralized Logging

Role:
Loki collects and indexes logs from the API containers, making it easy to search and analyze logs in Grafana.

Key Steps:

- Install Loki and configure it as a Grafana data source.
- Ensure API logs are written to stdout/stderr for collection.

Sample Logging Code (Python/FastAPI):

```python
import logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("api_logger")

@app.post("/predict")
def predict(data: dict):
    logger.info(f"Received prediction request: {data}")
    # ...rest of code...
```

Prometheus: Metrics Collection

Role:
Prometheus scrapes metrics from the API (using /metrics endpoint) and stores them for visualization in Grafana.

Key Steps:

- Add Prometheus client code to expose metrics.
- Configure Prometheus to scrape the API.

Sample Prometheus Metrics Code:

```python
from prometheus_client import Counter, generate_latest
REQUEST_COUNT = Counter('api_requests_total', 'Total API requests', ['method', 'endpoint', 'status'])

@app.middleware("http")
async def prometheus_middleware(request, call_next):
    response = await call_next(request)
    REQUEST_COUNT.labels(request.method, request.url.path, response.status_code).inc()
    return response

@app.get("/metrics")
def metrics():
    return Response(generate_latest(), media_type="text/plain")
```

Prometheus Config Example:

```yaml
scrape_configs:
  - job_name: 'heart-disease-api'
    static_configs:
      - targets: ['heart-disease-service:8080']
```

Integration Workflow

1. **Build Docker image** for the API.
2. **Deploy on Kubernetes** using manifests/Helm charts.
3. **Expose metrics** and logs from the API.
4. **Prometheus** scrapes metrics; **Loki** collects logs.
5. **Grafana** visualizes both metrics and logs for monitoring.