21IPE333P – PBL-1

BLOOD BANK MANAGEMEBT SYSTEM USING DOCKER , GIT HUB for VERSION CONTROL AND AWS EC2

DEEPESH N **CSE AIML (Z1)**

RA2211026010254

https://github.com/gh-Deepesh-N/Dev-ops-practice-PBL1

Introduction

This project is a Blood Bank Management System deployed on an AWS EC2 instance using Docker & Docker-Compose for containerization. It includes a Flask-based backend, a PostgreSQL database, and an Apache-powered frontend.

Infrastructure & Technology Stack

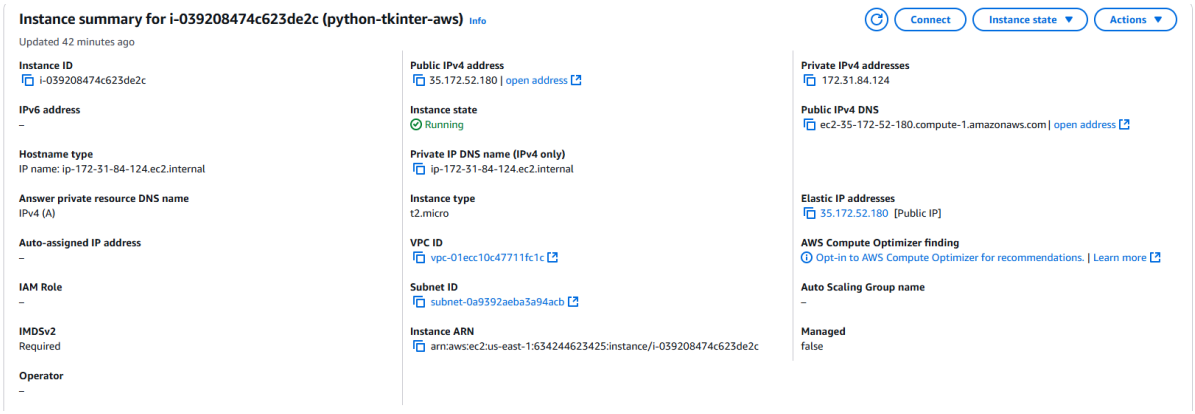
- Frontend → HTML, CSS, Bootstrap, hosted on Apache (httpd Docker container)
- Backend → Flask (Python), hosted in a Docker container
- Database → PostgreSQL, running inside a Docker containers
- Deployment Environment → AWS EC2 Instance
- Orchestration → Docker Compose
- Version Control → GitHub
- Security → Configured Security Groups for EC2
- Networking → Containers communicate via Docker Network

Setup & Deployment Process

Step 1: AWS EC2 Instance Setup

1. Launched an EC2 Instance (t2.micro in free tier).
2. Configured Security Group:
    - Port 22 → For SSH access.
    - Port 80 → For frontend (Apache).
    - Port 5050 → For backend (Flask).
    - Port 5432 → For PostgreSQL (internal container communication).

3. Connected via SSH using VS Code Remote SSH Extension.



Step 2: Installed Required Software

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y docker docker-compose git
```

Step 3: Database Setup (PostgreSQL)

1. Created a PostgreSQL Docker container with persistent storage.
2. Initialized the Blood Bank Database:

```
CREATE DATABASE BloodBankManagement_Normalized;
CREATE TABLE blood_donor (
    bd_id SERIAL PRIMARY KEY,
    bd_name TEXT,
    bd_age INTEGER,
    bd_Bgroup TEXT,
    bd_sex TEXT
);
```

```
docker exec -it postgres-container psql -U postgres -d BloodBankManagement_Normalized
```

Step 4: Backend Setup (Flask API)

1. Wrote app.py for CRUD operations.
2. Configured Flask to connect to PostgreSQL inside Docker**:**
3. Created a Dockerfile for Flask backend
4. Built and ran the backend container:

```
def connect_db():
    return psycopg2.connect(
        dbname="BloodBankManagement_Normalized",
        user="postgres",
        password="db@12db",
        host="db",  # Resolves to the database container
        port="5432"
    )
```

```
FROM python:3.9
WORKDIR /app
COPY . /app/
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000
CMD ["python3", "app.py"]
```

```
docker build -t flask-app .
docker run -d -p 5050:5000 flask-app
```

Step 5: Frontend Setup (Apache & HTML)

1. Built the frontend using HTML, CSS & Bootstrap.
2. Updated API URL inside index.html:
3. Created Dockerfile for Apache-based frontend
4. Built and ran the frontend container

```
const apiUrl = "http://35.172.52.180:5050/donors";
```

```
FROM httpd:2.4
COPY . /usr/local/apache2/htdocs/
EXPOSE 80
```

```
docker build -t apache-frontend .
docker run -d -p 80:80 apache-frontend
```

Step 6: Docker Compose Setup (All Containers in One Command)

```
version: "3.8"

services:
  db:
    image: postgres
    environment:
      POSTGRES_DB: BloodBankManagement_Normalized
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: db@12db
    ports:
      - "5432:5432"
    restart: always

  backend:
    build: ./backend
    ports:
      - "5050:5000"
    depends_on:
      - db
    restart: always

  frontend:
    build: ./frontend
    ports:
      - "80:80"
    depends_on:
      - backend
    restart: always
```

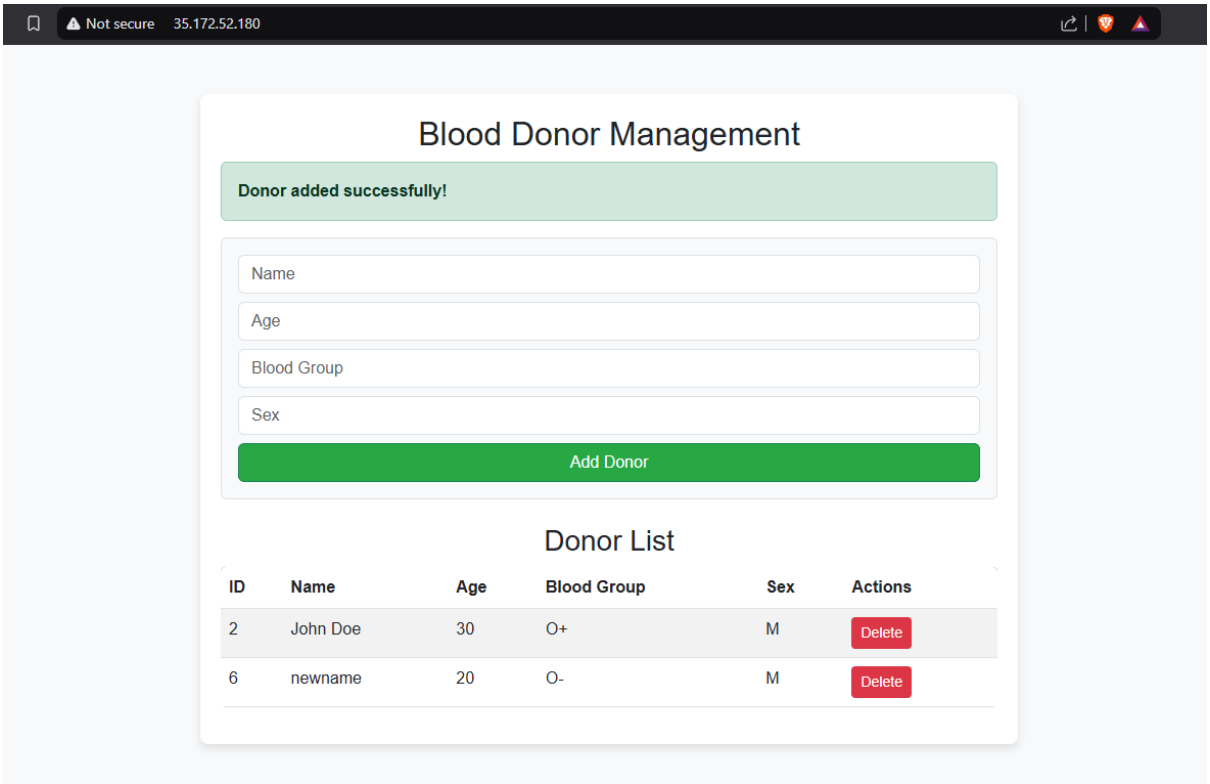
Step 7 : build docker - Started all services together:

```
sudo docker-compose up -d --build
```

Step 7: Testing & Verification

Testing Frontend

- **Open http://35.172.52.180 in browser.**



CRUD Operations

- **GET: Fetch donor list.**
- **POST: Add a donor.**
- **DELETE: Remove a donor.**

Docker Logs Check

```
sudo docker logs backend-container
```

```
ubuntu@ip-172-31-84-124:/opt/docker-setup$ sudo docker logs docker-setup_backend_1
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.20.0.3:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 887-480-225
106.51.168.131 - - [22/Feb/2025 03:02:55] "GET /donors HTTP/1.1" 200 -
106.51.168.131 - - [22/Feb/2025 03:03:14] "OPTIONS /donors HTTP/1.1" 200 -
106.51.168.131 - - [22/Feb/2025 03:03:15] "POST /donors HTTP/1.1" 201 -
106.51.168.131 - - [22/Feb/2025 03:03:15] "GET /donors HTTP/1.1" 200 -
106.51.168.131 - - [22/Feb/2025 03:03:21] "OPTIONS /donors/3 HTTP/1.1" 200 -
106.51.168.131 - - [22/Feb/2025 03:03:22] "DELETE /donors/3 HTTP/1.1" 200 -
106.51.168.131 - - [22/Feb/2025 03:03:22] "GET /donors HTTP/1.1" 200 -
```

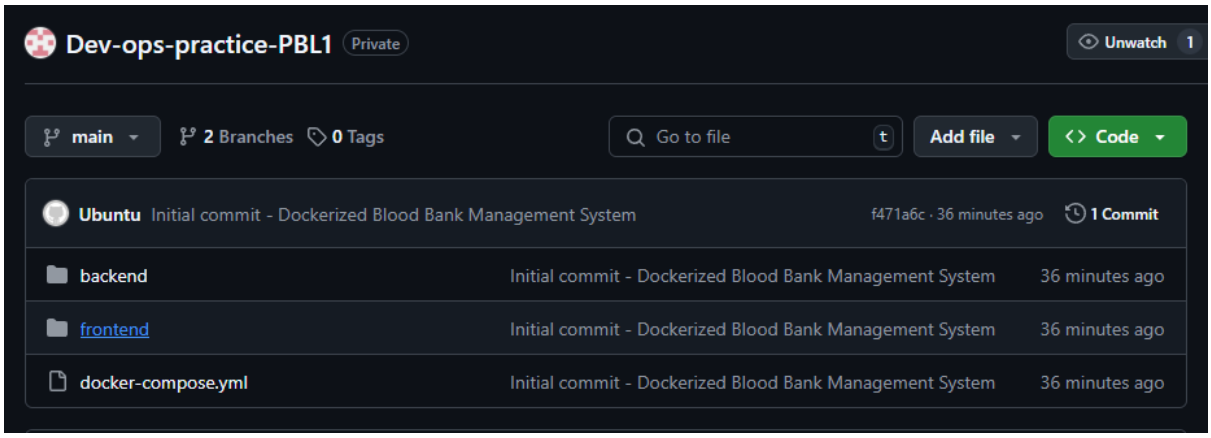
sudo docker ps – containers running

```
ubuntu@ip-172-31-84-124:/opt/docker-setup$ sudo docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED       STATUS         PORTS                                       NAMES
b3736bd12bf7   docker-setup_frontend   "httpd-foreground"       32 hours ago   Up 50 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp           docker-setup_frontend_1
d5879ab719a5   docker-setup_backend    "python3 app.py"        32 hours ago   Up 50 minutes   0.0.0.0:5050->5000/tcp, :::5050->5000/tcp   docker-setup_backend_1
0e4b16121ae9   postgres:16             "docker-entrypoint.s…"   32 hours ago   Up 50 minutes   0.0.0.0:5432->5432/tcp, :::5432->5432/tcp   docker-setup_db_1
```

Step 8: GitHub Version Control

1. **Set up Git authentication (Personal Access Token required).**
2. **Added all files & committed changes:**

```
git add .
git commit -m "Initial commit with working setup"
git push -u origin main
```



Key Highlights & Learning Outcomes

✅ Full-stack Deployment: Set up database, backend, and frontend.
✅ Containerization with Docker: Ensured seamless portability.
✅ Networking & Security: Configured AWS security groups and Docker networking.
✅ Automated Setup with Docker-Compose: Simplified multi-container orchestration.
✅ Version Control with GitHub: Maintained project history.

LUNG DISEASE PREDICTION WITH CHAT BOT INTEGRATION USING DOCKER, KUBERNETES with CI/CD PIPELINE

Github - https://github.com/gh-Deepesh-N/lung-predict

Introduction

This project is a Flask-based AI-powered lung disease prediction system that analyzes X-ray images and detects potential lung diseases using TensorFlow. The application is built with DevOps best practices, ensuring automation, scalability, and reliability through Docker, Kubernetes, and GitHub Actions CI/CD.

The project follows the DevOps lifecycle, ensuring continuous integration, delivery, and deployment (CI/CD). The infrastructure is containerized using Docker, orchestrated using Kubernetes (Minikube), and automated using GitHub Actions. This enables:

- Automated Testing & Deployment
- Scalable & Fault-Tolerant Architecture
- Easy Integration & Portability with Containers
- Infrastructure as Code (IaC) for Deployment Management

Infrastructure & Tech Stack

Workflow

GitHub Actions → Docker Build → DockerHub → Kubernetes Deployment → Service Exposure

Category	Technology Used
Backend	Flask (Python)
Machine Learning	TensorFlow, Keras
Containerization	Docker
Orchestration	Kubernetes (Minikube)
CI/CD	GitHub Actions
Storage	DockerHub (Container Registry)
Version Control	Git & GitHub

Setup & Process

Step 1: Clone the Repository

```
git clone https://github.com/your-repo/lung-predict.git
cd lung-predict
```

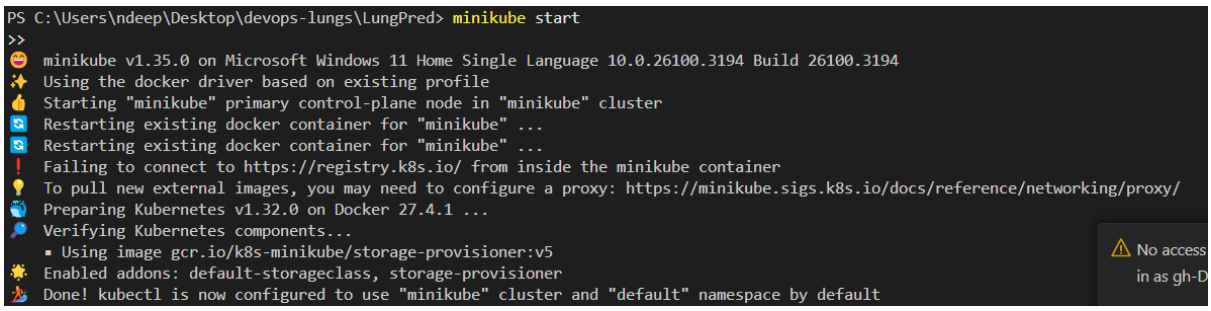
Step 2: Setting Up Virtual Environment - Since the project is based on Flask & TensorFlow, we create a virtual environment for dependency management.

Step 3: Containerizing with Docker

```
docker build -t lung-predict:v1 .
```

```
docker run -p 5000:5000 lung-predict:v1
```

Step 4: Setting Up Kubernetes (Minikube)




```
PS C:\Users\ndeep\Desktop\devops-lungs\LungPred> kubectl get nodes
>>
NAME       STATUS   ROLES           AGE    VERSION
minikube   Ready    control-plane   2d22h   v1.32.0
```

Step 5: Deploying to Kubernetes

```
kubectl apply -f deployment.yml
```

```
PS C:\Users\ndeep\Desktop\devops-lungs\LungPred> kubectl get pods
NAME                              READY   STATUS    RESTARTS      AGE
lung-predict-547d6d4587-2mm5x     1/1     Running   2 (47s ago)   2d11h
lung-predict-547d6d4587-vf556     1/1     Running   2 (47s ago)   2d11h
```

```
PS C:\Users\ndeep\Desktop\devops-lungs\LungPred> kubectl get services
NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes            ClusterIP   10.96.0.1        <none>        443/TCP    2d23h
lung-predict-service  ClusterIP   10.102.114.227   <none>        5000/TCP   2d22h
```

Step 6: Automating Deployment with GitHub Actions CI/CD

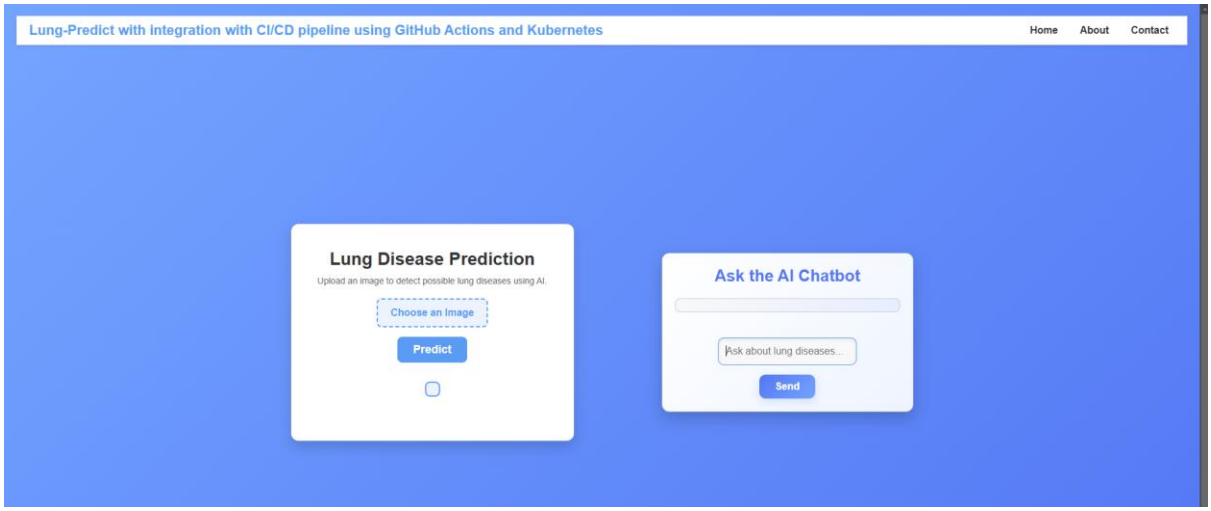
```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v2

      - name: Set Up Python Environment
        uses: actions/setup-python@v3
        with:
          python-version: '3.11'

      - name: Install Dependencies
        run: |
          pip install --upgrade pip
          pip install -r requirements.txt

      - name: Build and Push Docker Image
        run: |
          docker build -t dockerdeepesh7/lung-predict:v1 .
          docker push dockerdeepesh7/lung-predict:v1

  deploy:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - name: Apply Kubernetes Deployment
        run: kubectl apply -f deployment.yml
```

Step 7: Port Forwarding for Local Access

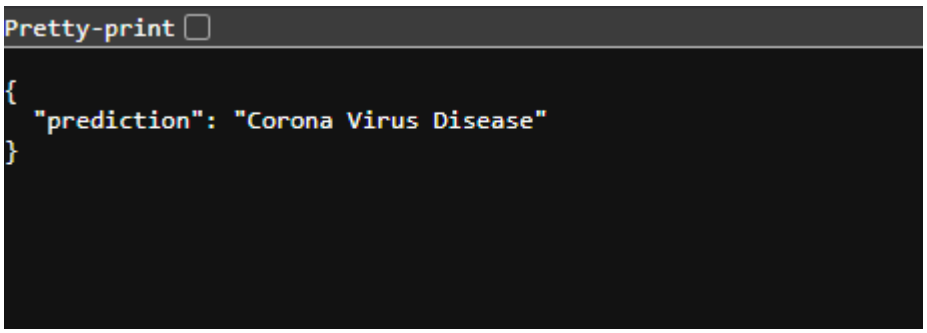
```
kubectl port-forward service/lung-predict-service 5000:5000
```

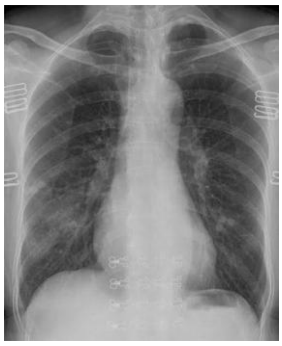
Step 8 : Final Output and Demo

Home Page

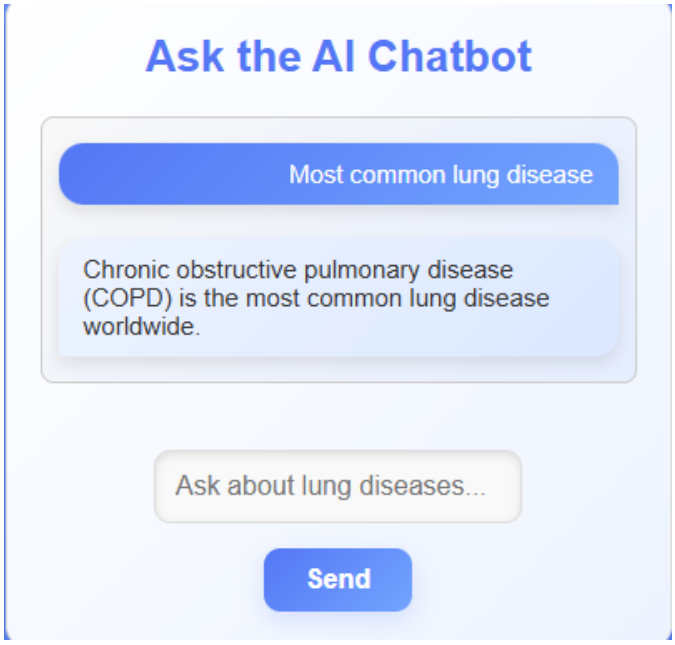


ML model output:

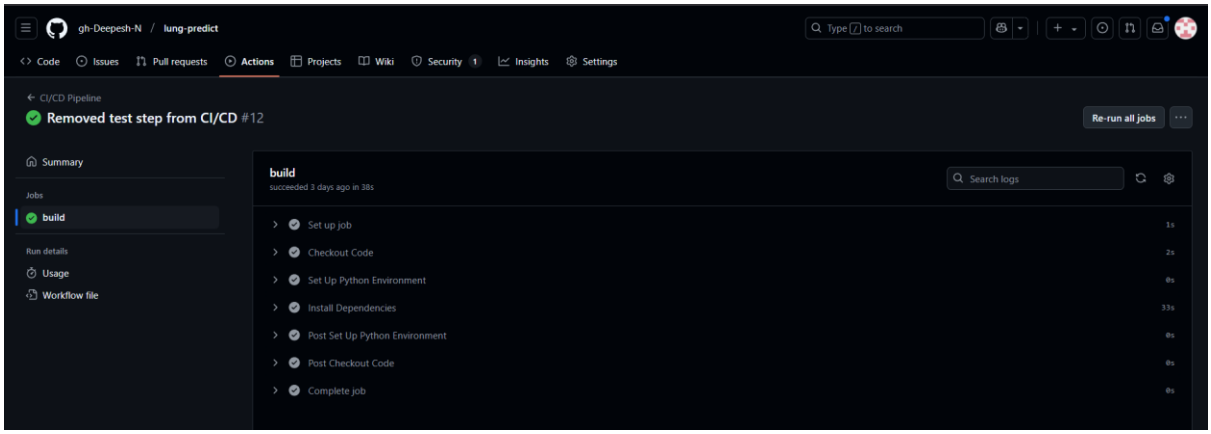




Chat Bot Working:



CI/CD working:



Key Highlights & Learning Outcomes

DevOps Practices Implemented

- Containerization: Docker ensures a consistent runtime environment.
- Orchestration: Kubernetes manages scaling & deployment.
- CI/CD Pipeline: GitHub Actions automates testing & deployment.
- Infrastructure as Code (IaC): Kubernetes manifests (deployment.yml) define deployment configurations.
- Portability: Works locally on Minikube and can be extended to cloud Kubernetes clusters.
- Fault Tolerance: Kubernetes automatically restarts failing containers (CrashLoopBackOff recovery).

Learning Outcomes

- Understanding DevOps Tools – Learned how Docker, Kubernetes, and CI/CD pipelines integrate into ML workflows.
- End-to-End Deployment – From local testing to automated deployments, the project follows best DevOps practices.
- Scalability & Automation – Instead of manually deploying models, the system automatically updates and scales using Kubernetes.
- Bridging ML & DevOps – AI models can be packaged, deployed, and scaled in production-ready environments.

Conclusion:

This project successfully integrates Machine Learning with DevOps to create an AI-powered, scalable, and automated Lung Disease Prediction System. The Flask API, TensorFlow model, Docker containers, Kubernetes orchestration, and GitHub Actions CI/CD pipeline work together to enable a fully automated, fault-tolerant, and production-ready AI system.

This project demonstrates how DevOps principles can be applied to AI applications for real-world deployment and automation**.**