Project - High Level Design

On

Dockerized Healthcare Python Flask Service

Course Name: DevOps Fundamentals

Institution Name: Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr no	Student Name	Enrolment Number
01.	Vanshika Vyas	EN22CS3011062
02.	Tulsi Chouhan	EN22CS3011038
03.	Ujjawal Goswami	EN22CS3011043
04.	Swayam Mishra	EN22CS3011009
05.	Mukund Mishra	EN22ME304064

Group Name:02D9

Project Number:DO-02

Industry Mentor Name:Mr Vaibhav

University Mentor Name:Dr. Ritesh Joshi

Academic Year:2025-2026

Table of Contents

1. Introduction

The Dockerized Healthcare Python Flask Service is a web-based healthcare application designed to manage and process healthcare-related data such as patient records. The application is developed using the **Python Flask framework** for backend services and a simple **HTML/CSS-based frontend** for user interaction.

The primary objective of this project is to **containerize the healthcare application using Docker industry best practices**, orchestrate the application using **Docker Compose**, and deploy it on a **cloud-based AWS EC2 Linux server**. The system ensures portability, consistency, and reliability across different environments without dependency on local machine configurations.

This project focuses on **containerization and deployment**, not on CI/CD or Kubernetes-based orchestration.

The solution ensures:

Portability across environments (Local and Cloud)

Optimized Docker images using multi-stage builds

Isolated and secure runtime environment

Easy deployment and management using Docker Compose

Cloud-based deployment using AWS EC2

1.1 Scope of the Document

This document provides a High-Level Design (HLD) of the Dockerized Healthcare Python Flask Service. It includes:

Overall system architecture

Application and component design

Docker containerization strategy

Docker Compose-based orchestration

AWS EC2 deployment architecture

Database design using SQLite

Security, performance, and scalability considerations

1.2 Intended Audience

This document is intended for:

Software Developers

DevOps Students

Cloud Engineers

System Architects

Faculty and Project Evaluators

Deployment and Operations Teams

1.3 System Overview

The Healthcare Management System is a **containerized web application** consisting of the following layers:

1. Frontend Layer

Built using HTML and CSS

Provides user interface for healthcare data operations

Sends HTTP requests to the backend Flask application

2. Backend Layer

Developed using Python Flask

Handles business logic and request processing

Interacts with SQLite database

3. Database Layer

SQLite database

Lightweight, file-based database

Suitable for small-scale healthcare applications

4. Containerization Layer

Application packaged using a **multi-stage Dockerfile**

Reduces image size and improves security

Ensures consistency across environments

5. Orchestration Layer

Docker Compose used to manage application container

Single-command deployment and management

6. Cloud Deployment Layer

AWS EC2 Linux instance

Deployed using Terraform

Public access via EC2 public IP

2. System Design

The system follows a **container-based architecture** where the Flask application runs inside a Docker container on an AWS EC2 instance. Docker Compose is used to manage the application lifecycle.

The architecture is designed to be:

Simple

Portable

Secure

Easy to deploy and manage

2.1 Application Design

The application follows a **three-tier architecture**:

1. Presentation Layer

HTML and CSS-based frontend

Stateless UI

Communicates with backend via HTTP

2. Application Layer (Backend)

Python Flask framework

REST-style endpoints

Handles:

Request validation

Business logic

Database operations

Runs inside a Docker container

3. Data Layer

SQLite database

Embedded database stored as a file

Connected directly to Flask application

2.2 Process Flow

Step 1: User Interaction

User accesses the application using a web browser by entering the EC2 public IP.

Step 2: Frontend Request

The frontend sends an HTTP request to the Flask backend.

Step 3: Backend Processing

Flask:

Receives the request

Applies business logic

Interacts with SQLite database

Step 4: Database Operation

SQLite performs required CRUD operations.

Step 5: Response Handling

Flask sends the response back to the frontend.

Step 6: User Output

Updated data is displayed to the user in the browser.

2.3 Information Flow

The system follows a simple and structured information flow:

User Browser
↓
Frontend (HTML/CSS)
↓
AWS EC2 Server
↓
Docker Engine
↓
Flask Application (Docker Container)

↓
SQLite Database
↓
Response to User

2.4 Components Design

1. Flask Application Container

Hosts backend logic

Runs using Flask development server

Built using multi-stage Dockerfile

Exposes application port (9000)

2. SQLite Database

File-based database

Stored inside Docker volume

Provides persistent storage

3. Docker Engine

Container runtime

Ensures isolation and portability

4. Docker Compose

Orchestrates application container

Manages ports, restart policies, and volumes

5. AWS EC2 Instance

Linux-based cloud server

Hosts Docker and Docker Compose

Provides public access to application

2.5 Key Design Considerations

1. Container Optimization

Multi-stage Docker build

Minimal Python base image

Reduced image size

2. Security

Docker container isolation

AWS Security Groups for controlled access

SSH key-based authentication

3. Portability

Same Docker image runs locally and on cloud

No environment-specific dependency

4. Maintainability

Modular code structure

Easy start/stop using Docker Compose

5. **Reliability**

Container restart policies

Cloud-based hosting ensures availability

2.6 API Catalogue

1. Get All Patients

Endpoint: GET /patients

Description: Fetches all patient records

2. Get Patient by ID

Endpoint: GET /patients/{id}

Description: Retrieves patient details

4. Add New Patient

Endpoint: POST /patients

Description: Adds a new patient record

4. Update Patient

Endpoint: PUT /patients/{id}

Description: Updates patient data

5. Delete Patient

Endpoint: DELETE /patients/{id}

Description: Deletes a patient record

3. Data Design

3.1 Data Model

Patient Table

patient_id

name

Age

disease

3.2 Data Access Mechanism

Flask interacts with SQLite using SQL queries

Supports CRUD operations:

Create

Read

Update

Delete

3.3 Data Storage and Persistence

SQLite database stored inside Docker volume

Ensures data persistence even after container restart

3.4 Data Migration

Currently not implemented

Future migration possible to MySQL or PostgreSQL

4. Interfaces

User Interface

Web browser interface

System Interface

Flask REST APIs

Cloud Interface

AWS EC2 public IP access

5. State and Session Management

Application is stateless
Each request is processed independently
No session persistence implemented

6. Caching

Caching not implemented
Can be added in future using Redis

7. Non-Functional Requirements

7.1 Security Aspects

Docker container isolation

AWS Security Groups

SSH-based secure access

7.2 Performance Aspects

Lightweight Flask application

SQLite provides fast local access

7.3 Scalability Aspects

Vertical scaling using larger EC2 instance

Horizontal scaling possible with multiple containers

7.4 Reliability Aspects

Docker ensures consistent runtime

AWS EC2 provides high availability

8. References

Python Flask Documentation

Docker Documentation

Docker Compose Documentation

AWS EC2 Documentation

Terraform Documentation

SQLite Documentation

Final Architecture Summary

User Browser
↓
Internet
↓
AWS EC2 Instance
↓
Docker Container
↓
Flask Application
↓
SQLite Database

Conclusion

The Dockerized Healthcare Python Flask Service demonstrates a modern container-based deployment approach using Docker and Docker Compose. The application is successfully deployed on AWS EC2 using Terraform, ensuring portability, reliability, and ease of deployment. This High-Level Design clearly outlines the system architecture, components, workflows, and key design decisions aligned with real-world DevOps practices.