Project - High Level Design

on

Dockerized Construction Python Flask Services

Course Name: DevOps

Medicaps University – Datagami Skill Based Course

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

Sr no	Student Name	Enrolment Number
1	SNEHA GUPTA	EN22CS306051
2	NAMAN SUNDRIYAL	EN22CS304042
3	SHUBH RATHOD	EN22CS304060
4	AKASH SINGH	EN22CS303009
5	HARSH NANDWAL	EN22CS304027
6	YASH DHIMOLE	EN22CS304066

Group Name: Group 12D11

Project Number: DO-38

Industry Mentor Name: Mr. Vaibhav Sir

University Mentor Name: Prof. Shyam Patel

Academic Year: 2025-26

Table of Contents

1. Introduction

The Dockerized Construction Python Flask Service is a backend web application developed to manage construction project data in a scalable, portable, and production-ready environment. The system is built using Python Flask as the web framework and PostgreSQL as the database. To ensure consistency across development and production environments, the application is containerized using Docker and orchestrated using Docker Compose.

The project demonstrates modern DevOps practices including containerization, cloud deployment on AWS EC2, image management using DockerHub, and version control using GitHub. The infrastructure can be created either manually using AWS Console or automatically using Terraform.

The primary objective of this system is to showcase industry-standard deployment practices while maintaining clean architecture and modular design.

1.1 Scope of the Document

This High-Level Design (HLD) document describes the architecture, components, workflows, integrations, and deployment strategy of the Dockerized Construction Flask Service.

The scope includes:

- Application architecture
- Container architecture
- AWS deployment model
- Database design
- API structure
- DevOps workflow
- Security and performance considerations

The scope excludes:

- Frontend design
- Low-level source code implementation
- Cost estimation of cloud infrastructure

This document focuses only on components relevant to the backend system and deployment architecture.

1.2 Intended Audience

This document is intended for:

• Faculty and project evaluators
• Backend developers
• DevOps engineers
• Cloud engineers
• Team members working on deployment

It provides architectural clarity and ensures alignment between system design and implementation.

1.3 System Overview

The system is a REST-based backend service designed to manage construction projects. Users can create, update, retrieve, and delete project records.

The system consists of:

• Flask Application Container
• PostgreSQL Database Container
• Docker Compose orchestration
• AWS EC2 hosting environment

When deployed in the cloud, the application runs inside Docker containers hosted on an AWS EC2 instance. The database runs in a separate container within the same Docker network.

The system ensures:

• Portability across environments
• Isolated container execution
• Secure communication between services
• Easy scalability

2. System Design

The system follows a layered and containerized architecture model. The architecture separates responsibilities into logical layers to improve maintainability and scalability.

The system can be viewed from two perspectives:

• Application Architecture
• Deployment Architecture

2.1 Application Design

The Flask application is structured using modular design principles.

Core modules include:

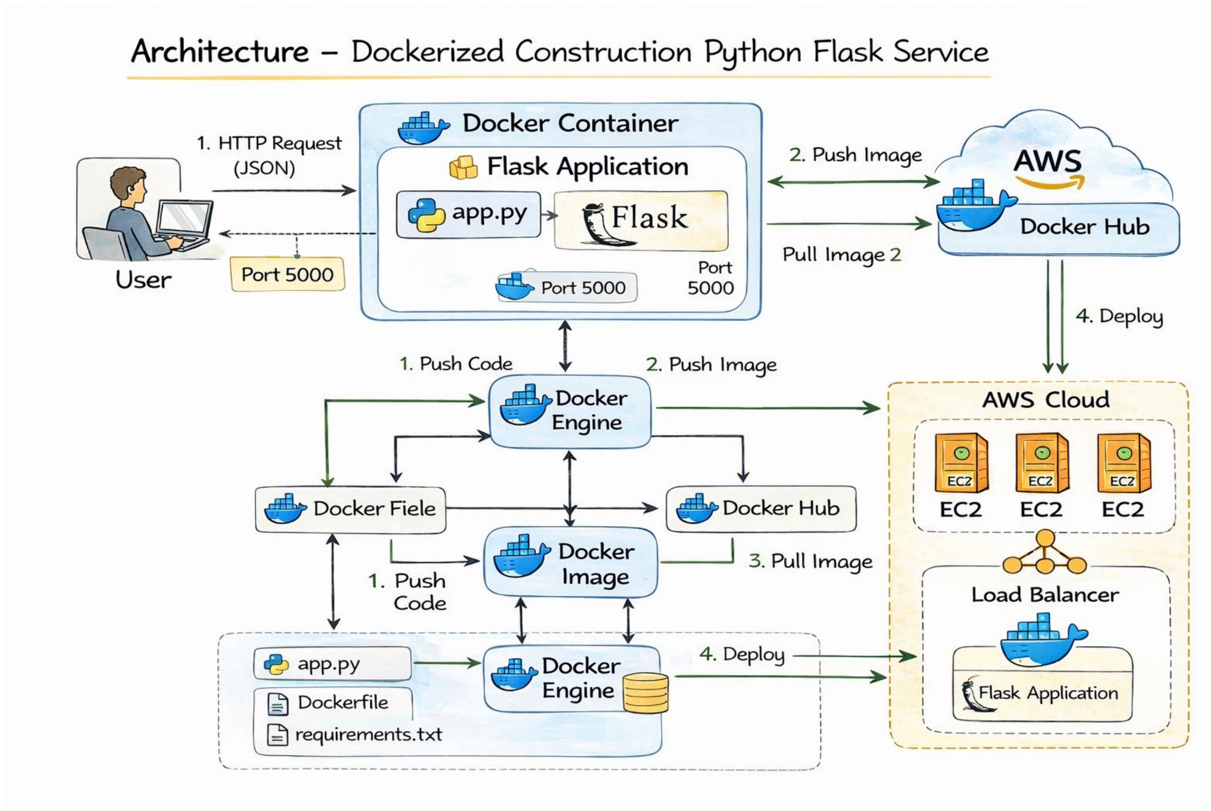
• app.py – Entry point of the application
• models.py – Defines database schema using SQLAlchemy
• routes.py – Contains API endpoint definitions
• config.py – Manages environment configurations
• requirements.txt – Lists application dependencies

The application uses Gunicorn as a production-grade WSGI server to handle concurrent requests efficiently.

Key application characteristics:

• Stateless API design
• RESTful endpoints
• JSON-based request/response
• ORM-based database interaction
• Environment variable configuration

This modular approach ensures clean separation of concerns and improved maintainability.



2.2 Process Flow

The system processes requests in a structured workflow.

Project Creation Flow:

1. Client sends POST request to /projects.
2. Flask route validates request payload.
3. Business logic processes the data.
4. SQLAlchemy inserts data into PostgreSQL.
5. Success response returned to client.

Deployment Flow:

1. Developer pushes code to GitHub.
2. Dockerfile builds image.
3. Image is pushed to DockerHub.
4. AWS EC2 pulls image.
5. Docker Compose starts containers.
6. Application becomes publicly accessible.

2.3 Information Flow

Information flows securely between components.

• Client sends HTTP request
• Docker forwards request to Flask container
• Flask processes request
• Flask queries PostgreSQL container
• Database returns result
• Flask sends JSON response

When deployed on AWS:

Internet → EC2 Instance → Docker → Flask → Database

Docker networking ensures internal communication is isolated and secure.

2.4 Components Design

The system contains multiple integrated components.

Flask Application Container
Handles API processing and business logic.

PostgreSQL Database Container
Stores persistent project data.

Dockerfile
Defines container build instructions using multi-stage builds.

Docker Compose
Manages multi-container orchestration and networking.

GitHub Repository
Stores source code and maintains version control.

DockerHub
Hosts container images for deployment.

AWS EC2
Provides virtual machine infrastructure to run containers.

Terraform (Optional)
Automates AWS infrastructure provisioning.

Each component works together to form a complete DevOps workflow.

2.5 Key Design Considerations

The system follows industry best practices to ensure reliability and security.

Important design considerations include:

• Multi-stage Docker builds for optimized image size
• Non-root container execution for security
• Environment variable configuration
• Persistent Docker volumes for database
• Isolated Docker network
• Stateless architecture
• Cloud deployment readiness

These considerations improve system maintainability and scalability.

3. Data Design

The system uses a relational database model to ensure structured and consistent data storage.

3.1 Data Model

The primary entity is Project.

Attributes include:

• id – Primary key
• name – Project name
• budget – Project budget
• status – Project status (Planning, In Progress, Completed)
• created_at – Timestamp

The relational model ensures data integrity and validation constraints.

3.2 Data Access Mechanism

SQLAlchemy ORM is used for database interaction.

Advantages include:

• Abstraction of SQL queries
• Object-oriented interaction
• Automatic query generation
• Reduced risk of SQL injection

The ORM simplifies database management and improves maintainability.

3.3 Data Retention Policies

Project records are retained until deleted by authorized users.

Database persistence is maintained using Docker volumes, ensuring data remains intact even if containers restart.

Logs can be extended to centralized logging systems in future implementations.

3.4 Data Migration

Flask-Migrate is used to manage schema changes.

Migration process includes:

- Generating migration scripts
- Applying schema upgrades
- Version control of database schema

This allows safe database evolution without data loss.

4. Interfaces

The system provides the following interfaces:

- REST API Interface (HTTP/JSON)
- Docker Container Interface
- AWS Cloud Infrastructure Interface

External systems interact via HTTP requests, while internal services communicate through Docker networking.

5. State and Session Management

The application follows a stateless architecture.

Each request:

• Is processed independently
• Does not rely on server-side session storage
• Can be handled by any container instance

This design supports horizontal scaling and cloud deployment.

6. Caching

Currently, caching is not implemented.

However, future enhancements may include:

• Redis-based caching
• API response caching
• Database query caching

Caching can improve read performance in high-traffic environments.

7. Non-Functional Requirements

The system must satisfy several quality attributes.

• Portability

Portability ensures that the application can run consistently across different environments. By using Docker containers, the system can be deployed on local machines, testing servers, or cloud platforms like AWS without configuration changes.

• Scalability

Scalability allows the system to handle increased user load efficiently. The stateless Flask architecture and container-based deployment enable horizontal scaling by running multiple instances of the application.

• Security

Security ensures protection of application data and infrastructure. The system uses non-root Docker containers, environment-based configuration, controlled port exposure, and secure cloud settings to minimize vulnerabilities.

• Reliability

Reliability ensures consistent and uninterrupted system operation. Docker containers provide isolated execution, while database persistence through volumes ensures data safety even during restarts.

• Maintainability

Maintainability refers to how easily the system can be updated or modified. The modular Flask structure, version control via GitHub, and clear container configuration improve system maintainability.

• Performance

Performance ensures efficient response handling and resource utilization. Gunicorn enables concurrent request processing, while optimized Docker images and database indexing improve overall system speed.

Docker ensures environment consistency, while AWS ensures scalability.

7.1 Security Aspects

Security measures implemented include:

- Non-root container execution
- Environment variable-based secret management
- Secure Docker networking
- Controlled port exposure
- Firewall configuration on AWS
- Dependency version pinning

These measures reduce security risks in production environments.

7.2 Performance Aspects

Performance optimization strategies include:

• Gunicorn multi-worker support
• Lightweight Docker image
• Efficient database indexing
• Stateless design for scalability
• Cloud resource scalability

These techniques ensure the application performs efficiently under load.

8. References

• Docker Official Documentation
• Flask Official Documentation
• SQLAlchemy Documentation
• AWS EC2 Documentation
• GitHub Documentation
• DockerHub Documentation
• Terraform Documentation