Project - High Level Design

on

Design and implement a professional CI/CD pipeline for a a Go (Golang) Logistics application

Course Name: DevOps

Institution Name: Medicaps University – Datagami Skill Based Course

Sr no	Student Name	Enrolment Number
1	SUHANI MOGRA	EN22CS301989
2	VAIBHAV SHRIVASTAVA	EN22CS3011051
3	TAMANNA KATARIYA	EN22CS3011013
4	NAMAN PATIDAR	EN22ME303013
5	SNEHA ARAS	EN22CS301964

Group Name : 05D9

Project Number : 05

Industry Mentor Name: Mr. VAIBHAV

University Mentor Name:Dr. Ritesh Joshis

Academic Year : 2025-26

Table of Contents

1. Introduction

- This document describes the High Level Design (HLD) of the Cloud-Based Logistics Application with CI/CD Pipeline**.**
- The project demonstrates automated software delivery using Spring Boot, Docker, GitHub Actions, and AWS EC2.
- The main objective is to show how a modern DevOps pipeline works — from development to cloud deployment — with minimal manual intervention.

1.1 Scope of the Document

The scope of this document includes:

- Overall system architecture
- Application deployment design
- CI/CD workflow
- Docker containerization approach
- AWS EC2 hosting model
- High-level security and performance aspects

This document does **not** include low-level coding or database implementation details.

1.2 Intended Audience

This document is intended for:

- Faculty members and project evaluators
- Development team members
- DevOps learners
- System administrators

1.3 System Overview

The system is a Spring Boot–based logistics web application deployed using Docker and hosted on AWS EC2.

The development workflow is:

- Code written in VS Code
- Source control using GitHub
- Automated build and image creation using GitHub Actions
- Docker image stored in Docker Hub
- AWS EC2 pulls the image and runs Tomcat inside Docker

Users access the application through the EC2 public IP address.

The application exposes REST endpoints to verify system status and health.

2. System Design

This section describes the overall architecture and working flow of the Logistics Application. The system is designed using DevOps principles where application development, testing, containerization, and deployment are fully automated.

The architecture follows a layered model:

- Presentation Layer (User Browser)
- Application Layer (Spring Boot)
- Web Server Layer (Tomcat)
- Container Layer (Docker)
- CI/CD Layer (GitHub Actions)
- Cloud Infrastructure Layer (AWS EC2)

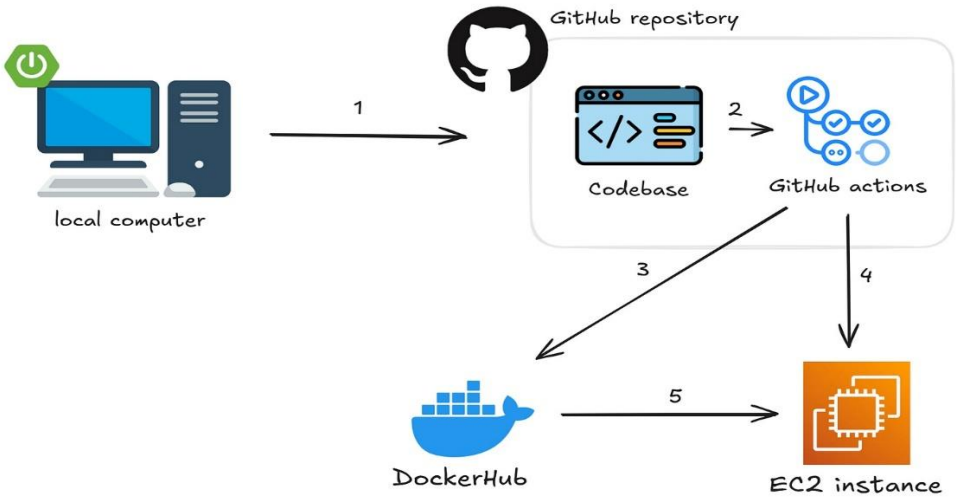
2.1 Application Design

The application is developed using Java Spring Boot and exposes REST APIs.

Key characteristics:

- Stateless REST-based service
- Packaged as WAR file
- Deployed on Apache Tomcat
- Runs inside Docker container
- Hosted on AWS EC2

The controller layer handles HTTP requests and returns responses.



2.2 Process Flow

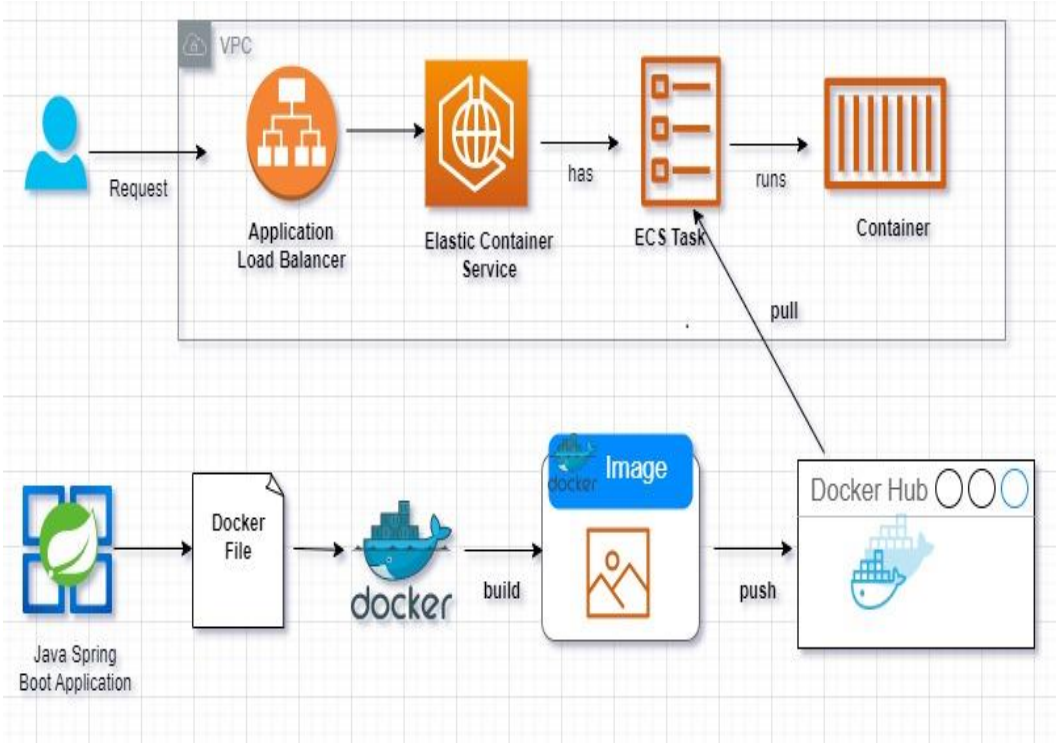
The overall process flow is divided into two parts:

CI/CD Flow

1. Developer pushes code to GitHub
2. GitHub Actions triggers automatically
3. Maven builds WAR file
4. Unit tests are executed
5. Docker image is created
6. Image is pushed to Docker Hub

Deployment Flow

1. AWS EC2 pulls Docker image
2. Docker runs Tomcat container
3. WAR is deployed as ROOT application
4. Application becomes accessible through EC2 public IP



2.3 Information Flow

User → EC2 Public IP → Docker Container → Tomcat → Spring Controller → Response

- User sends HTTP request from browser
- Request reaches AWS EC2 public IP
- Docker forwards request to Tomcat
- Tomcat routes request to Spring Boot application
- Controller processes request
- Response is returned to user

2.4 Components Design

- VS Code –Used for coding and testing application locally.
- GitHub – Stores application source code.
- GitHub Actions – Automates build, test, Docker image creation, and push.
- Docker Hub – Image storage
- Docker – Runs application containers
- Tomcat – Application server hosting Spring Boot WAR
- AWS EC2 – Cloud server hosting application.

2.5 Key Design Considerations

- Automated deployment
- Portable containers
- Cloud hosting
- Secure credentials (GitHub Secrets)
- Stateless application

2.6 API Catalogue

Endpoint	Method	Purpose
/	GET	App status
/ health	GET	Health check

3. Data Model

In the present implementation:

- The application does not store data in a database.
- It only returns static responses for status and health endpoints.

Future enhancement may include integration of a relational database such as MySQL or PostgreSQL.

4. Interfaces

The Logistics Application interacts with users and system components through well-defined interfaces.

External Interfaces

- **Web Browser:** Used by users to access the application via EC2 public IP.
- **REST API Interface:** HTTP-based endpoints exposed by Spring Boot.
- **Docker CLI:** Used to build, run, and manage containers.
- **GitHub Web Interface:** Used to manage source code and CI/CD pipelines.
- **AWS EC2 Console:** Used to manage cloud infrastructure.

Internal Interfaces

- **Spring Boot ↔ Tomcat:** Application deployed as WAR inside Tomcat.
- **Tomcat ↔ Docker Engine:** Tomcat runs inside Docker container.
- **GitHub Actions ↔ Docker Hub:** CI/CD pipeline pushes Docker images.
- **Docker Engine ↔ AWS EC2 OS:** Container runtime on cloud server.

5. State and Session Management

The application follows a **stateless architecture**.

- Each HTTP request is processed independently.
- No user session is maintained on the server.
- No server-side session storage is used.

Benefits of stateless design:

- Improved scalability
- Simplified deployment
- Better fault tolerance

Future enhancements may include session management if authentication is added.

6. Non-Functional Requirements

This section defines system quality attributes such as security and performance.

6.1 Security Aspects

The following security measures are implemented:

- AWS EC2 Security Group allows only:
  - Port 22 for SSH access
  - Port 8080 for application access
- Docker containers provide isolation from host system.
- Docker Hub credentials are stored securely using GitHub Secrets.
- No passwords are hardcoded in application or pipeline files.
- Only authorized users can access GitHub repository and CI/CD pipeline.

6.2 Performance Aspects

Performance is optimized using:

- Lightweight Spring Boot application
- Docker containerization for fast startup
- Apache Tomcat for efficient request handling
- Automated CI/CD to reduce deployment time

Observed application startup time is approximately 3–5 seconds.

7. References

1. Spring Boot Documentation
2. Docker Documentation
3. GitHub Actions Documentation
4. AWS EC2 Documentation
5. Apache Tomcat Documentation
6. Maven Documentation