Project - High Level Design on Logi-Track

Course Name: DevOps

Institution Name: Medi-caps University – Datagami Skill Based Course

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

Sr no	Student Name	Enrollment Number
1.	Vasu Joshi	EN22EL301063
2.	Yash Kag	EN22CS3011103
3.	Yash Parashar	EN22CS3011106
4.	Yashee Verma	EN22CS3011115
5.	Yashraj Singh Sisodiya	EN22CS3011120

Group Name: Group 12D10

Project Number: DO-25

Industry Mentor Name:

University Mentor Name: Prof. Avnesh Joshi

Academic Year: 2025-26

Table of Contents

1. Introduction:

1.1 Scope of the document:

This document describes the system design of the **Logistics CI/CD (Continuous Integration & Continuous Deployment)** project. It explains architecture, components, data flow, processes, interfaces, non-functional requirements, and design decisions for a robust CI/CD pipeline that builds, tests, and deploys a Python Logistics application using Docker and AWS EC2.
The main focus is on system behavior, integration flow between components, and deployment strategy.

1.2 Intended Audience:

This document is intended for:

- Developers building or enhancing the Logistics CI/CD pipeline
- DevOps engineers responsible for deployment & automation
- System architects reviewing design decisions
- QA engineers understanding automated testing requirements
- Project managers tracking implementation and system behavior

1.3 System Overview:

The Logistics CI/CD project integrates:

✓ A Python-based logistics application

✓ Automated GitHub Actions workflows

✓ Docker for containerization

✓ AWS EC2 for deployment

The system automates every step from code commit to production deployment.

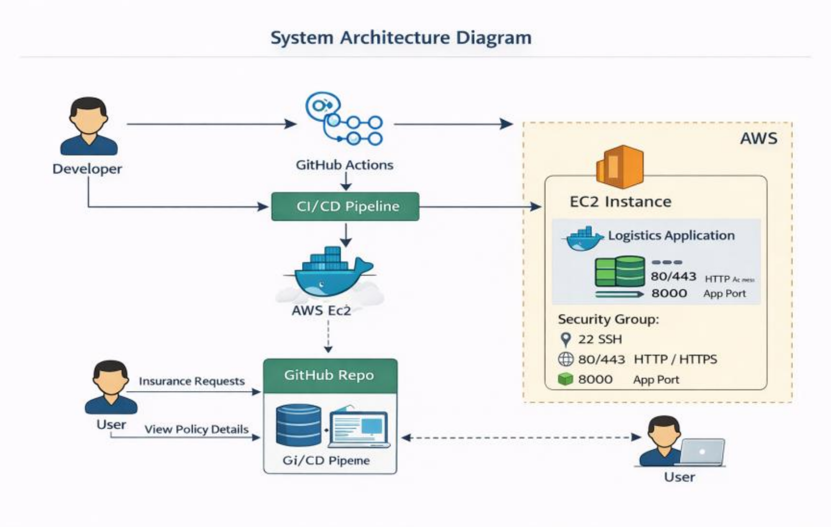

Fig 1.1: CI/CD pipeline for logistics application

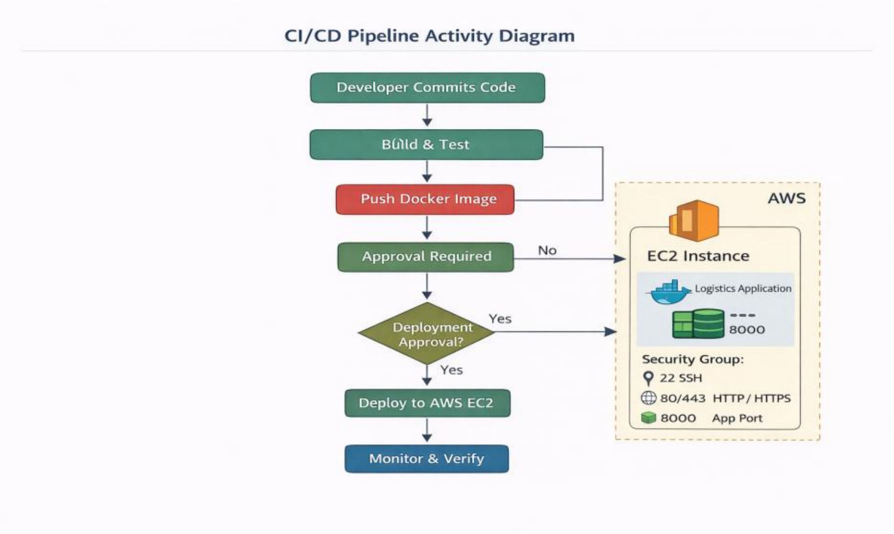

Fig 1.2: CI/CD pipeline activity diagram

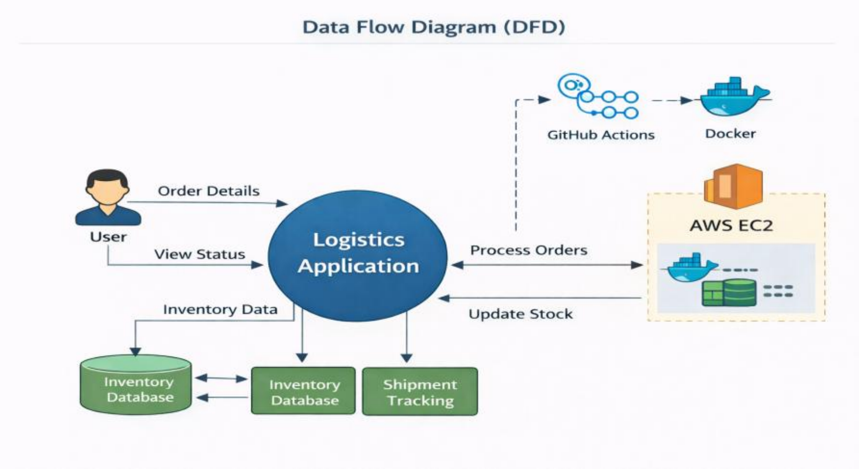

Fig 1.3: Data flow diagram for logistics system

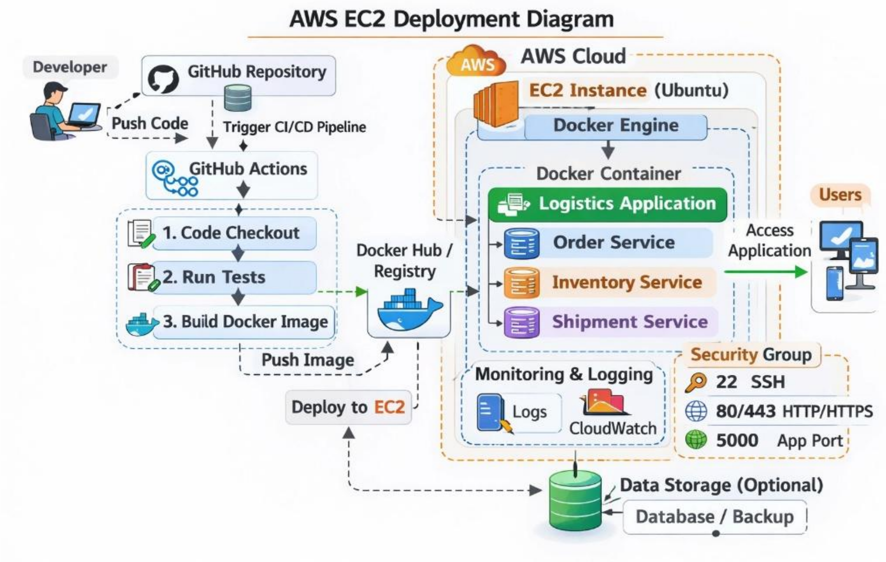

Fig 1.4: AWS EC2 logistics application deployment diagram

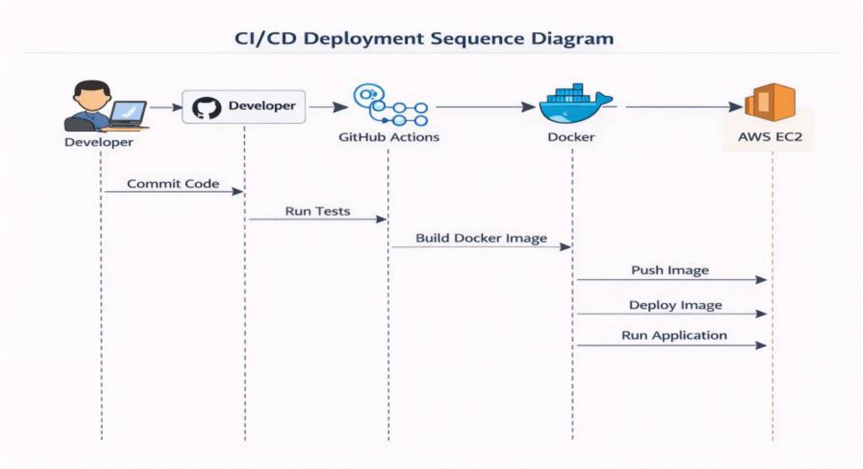

Fig 1.5: CI/CD pipeline sequence diagram

2. System Design:

2.1 Application Design:

The application is a **Python service** that simulates logistics processes.

- Core logic is implemented in Python modules
- Services are containerized using Docker
- The app runs inside a Docker container on AWS EC2

This design ensures:

- Environment consistency
- Easy scalability
- Portable deployments

2.2 Process Flow:

The process can be described in sequential steps:

1. Developer writes code
2. Pushes to GitHub
3. GitHub Actions triggers pipeline
4. Code is built + tested
5. Docker image is created
6. Image is deployed on EC2
7. App runs in a container
8. Logging + monitoring

2.3 Information Flow:

Information in the system flows as:

Developer → GitHub Repo → GitHub Actions → Docker CLI → AWS EC2 Instance → Application

Key steps involve:

- Source code transfer
- Test results feedback

- Deployment instructions
- Application access logs

2.4 Components Design:

Component	Purpose
GitHub Repo	Source code storage
GitHub Actions	CI/CD automation
Docker Daemon	Build and run containers
AWS EC2	Deployment server
Python App	Core business logic execution

2.5 Key Design Considerations:

- **Modularity:** Code separated into testable units
- **Automation:** Zero manual deployment for reproducibility
- **Scalability:** Containerized environment enables growth
- **Portability:** Docker ensures environment consistency
- **Cloud readiness:** EC2 as deployment target

2.6 API Catalogue:

Since the project is primarily backend logistics, all entrypoints are routed through Python code and exposed on container ports.

API Endpoint	Method	Description
/	GET	Root endpoint, returns "Service running"
/orders	GET / POST	Get or create order (if implemented)
/health	GET	Return deployment health

3. Data Design:

3.1 Data Model:

Since the logistics app is a prototype, it does not include a full database model. Typical model elements could include:

- Order
- Shipment
- Inventory

(Data persistence could be enhanced in future with PostgreSQL, MongoDB, etc.)

3.2 Data Access Mechanism:

Currently:

✓ Data is stored in application memory (no database)

Future deployment should include:

✓ Dedicated DB service

✓ ORM integration (SQLAlchemy, Django ORM)

3.3 Data Retention Policies:

Since persistent storage is not yet fully implemented:

- Data retains only as long as app is running
- Changes are not persistent on restart

For production:

✓ Set auto-backup

✓ Log retention policies

3.4 Data Migration:

Not applicable in the current scope since no DB is present.

Future migrations could include:

✓ Schema migration tools (Alembic, Flyway)

4. Interfaces:

Interfaces used are:

- REST endpoints for the application
- GitHub API through Actions
- Docker CLI via workflow
- SSH into EC2 for deployment scripts

5. State and Session Management:

Since the application is stateless (no sessions), a recommended future improvement:

✓ Session handling via Redis / cache

✓ JWT tokens for authentication

6. Caching:

No caching is implemented currently.

Future implementations may use:

✓ Redis / Memcached

7. Non-Functional Requirements:

7.1 Security Aspects:

- Only SSH keys should be allowed for EC2 deployment
- ii.Use least privileged IAM roles
- iii.GitHub secrets for sensitive tokens

7.2 Performance Aspects:

- Docker ensures minimal deployment overhead
- EC2 instance size should be optimized based on traffic
- Monitoring autoscaling can be implemented later

8. References:

The design is based on:

✓ …logistics-ci-cd repository
✓ GitHub Actions docs
✓ AWS EC2 deployment best practices
✓ Docker official guidelines