# Project - High Level Design

# On

CI/CD for Java
Spring Energy app

**Course Name:** DevOps

*Institution Name:*
**Medicaps University**
**Datagami Skill Based Course**

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

| Sr no | Student Name | Enrolment Number |
|---|---|---|
| 1. | KHUSHI KUDADE | EN22CS304034 |
| 2. | SARTHAK KASAT | EN22CS306048 |
| 3. | RISHABH GUPTA | EN22CS304054 |
| 4. | SAKSHI TIWARI | EN22CS304055 |
| 5. | MOHAMMAD ZAID KHAN | EN22EE301011 |

***Group Name**:07D11*

***Project Number** :D0-33*

***Industry Mentor Name**:*

***University Mentor Name:** prof. Shyam Patel*

***Academic Year:**2026*

# Table of Contents

## 1. Introduction

The High-Level Design (HLD) document provides a structured and comprehensive overview of the DevOps-based Automated Deployment Pipeline system. The purpose of this project is to integrate software development and IT operations into a unified, automated workflow that ensures rapid, reliable, and scalable deployment of applications.

In traditional software deployment models, development and operations teams worked in isolation, resulting in delayed releases, configuration mismatches, manual errors, and downtime. This project eliminates those issues by implementing DevOps principles such as Continuous Integration (CI), Continuous Deployment (CD), Infrastructure as Code (IaC), containerization, and orchestration.

The system integrates:

- Application development using Spring Boot
- Containerization using Docker
- CI/CD automation using GitHub Actions
- Container orchestration using Kubernetes
- Infrastructure provisioning using Terraform
- Cloud hosting using AWS

The objective is to build a production-ready automated pipeline that ensures:

- Faster release cycles
- High availability
- Scalability
- Reduced manual effort
- Improved reliability

This document focuses on the architectural and design aspects of the system rather than code-level implementation.

## 1.1 Scope of the Document

This document describes the high-level structure and design of the complete DevOps pipeline. It explains how different components interact and how the overall system is organized.

The scope includes:

- Overall system architecture
- Application structure and design
- CI/CD workflow
- Containerization strategy
- Orchestration mechanism
- Infrastructure provisioning
- Data handling design
- Security and performance considerations
- Scalability mechanisms

The document does not cover:

- Source code details
- Unit-level implementation
- Configuration secrets
- Production credentials

It is designed to give stakeholders a clear technical understanding of how the system operates at a macro level.

---

### 1.2 Intended Audience

This document is intended for:

**1. Project Mentors**

To evaluate architectural understanding and DevOps implementation.

**2. DevOps Engineers**

To understand system automation and infrastructure design.

**3. Cloud Engineers**

To review AWS and Terraform-based provisioning.

**4. Developers**

To understand deployment lifecycle and integration workflow.

**5. Academic Evaluators**

To assess practical application of DevOps concepts.

The document provides enough clarity for both technical and semi-technical audiences.

---

### 1.3 System Overview

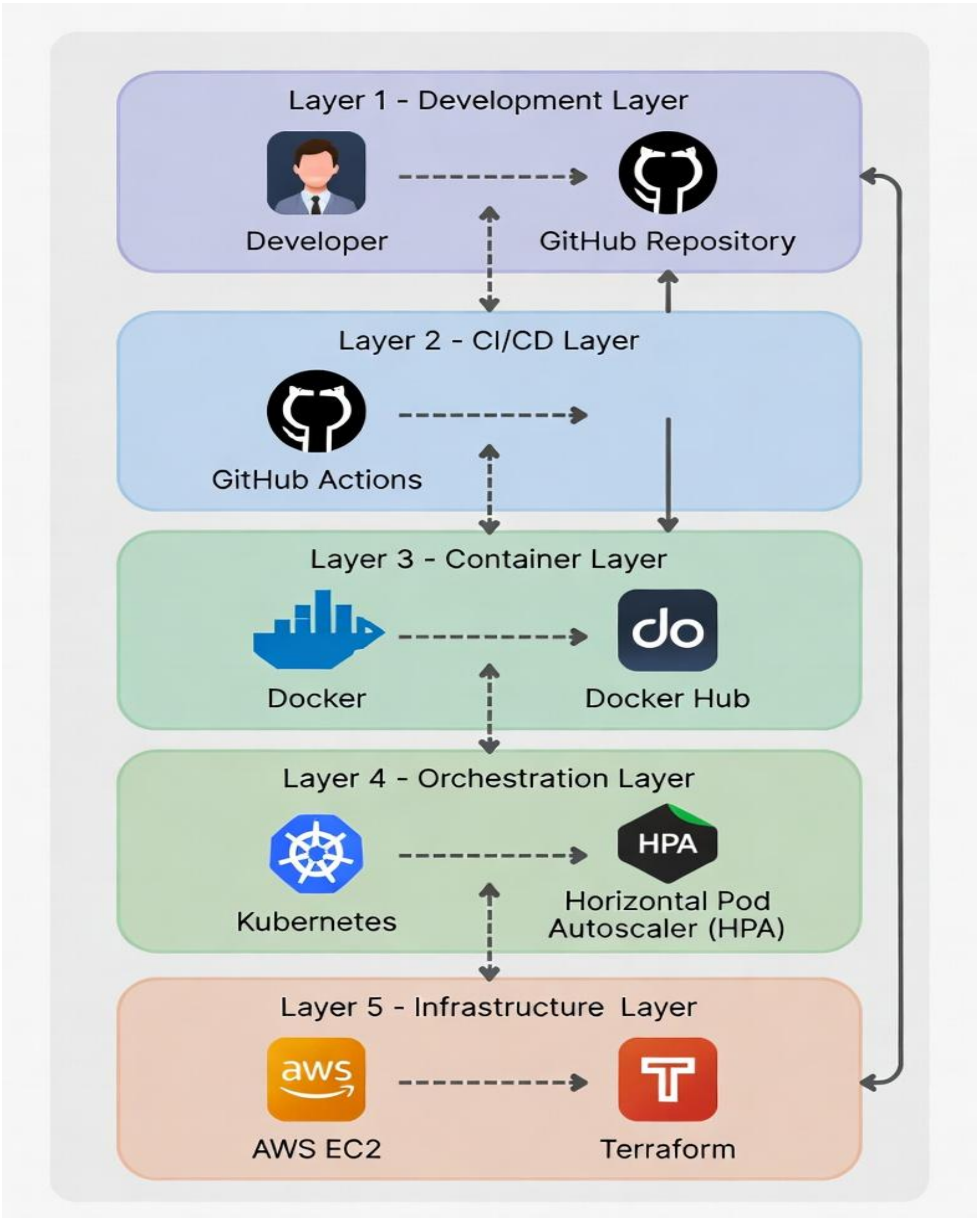
The system automates the complete lifecycle of application deployment from development to production.

**High-Level Working:**

1. Developer writes code and pushes it to GitHub.
2. GitHub Actions automatically triggers the CI/CD pipeline.
3. Maven builds the application and generates a JAR file.
4. Docker builds a container image of the application.
5. The image is pushed to Docker Hub.
6. Kubernetes pulls the latest image.
7. Pods are created or updated.
8. The application becomes accessible via Kubernetes Service.
9. Horizontal Pod Autoscaler adjusts replicas based on load.

This workflow ensures zero manual deployment steps.

## A) Architecture Diagram


Layer 1 - Development Layer
Developer
GitHub Repository
Layer 2 - CI/CD Layer
GitHub Actions
Layer 3 - Container Layer
Docker
Docker Hub
Layer 4 - Orchestration Layer
Kubernetes
HPA
Horizontal Pod Autoscaler (HPA)
Layer 5 - Infrastructure Layer
aws
AWS EC2
Terraform

## 2. System Design

The system design explains how different components are structured and interact logically. The architecture follows modular and layered principles to ensure maintainability and scalability.

Key architectural principles:

- Stateless application design
- Automation-driven deployment
- Infrastructure as Code
- Container-based packaging
- Horizontal scalability

---

### 2.1 Application Design

The application is developed using Java Spring Boot framework. It exposes REST APIs that handle HTTP requests and return JSON responses.

**Architectural Structure**

The application follows layered architecture:

1. Controller Layer
2. Service Layer
3. Repository Layer (future extension)

**Stateless Design**

The application does not store session information on the server. Each request is independent. This ensures:

- Easy scaling
- No session replication issues
- Compatibility with load balancers

**Externalized Configuration**

Environment variables are used to configure:

- Database connections

- Ports
- API keys

This ensures flexibility across environments.

**Why Spring Boot?**

- Rapid setup
- Embedded server
- Production-ready configuration
- Microservices-friendly
- Easy Docker packaging

---

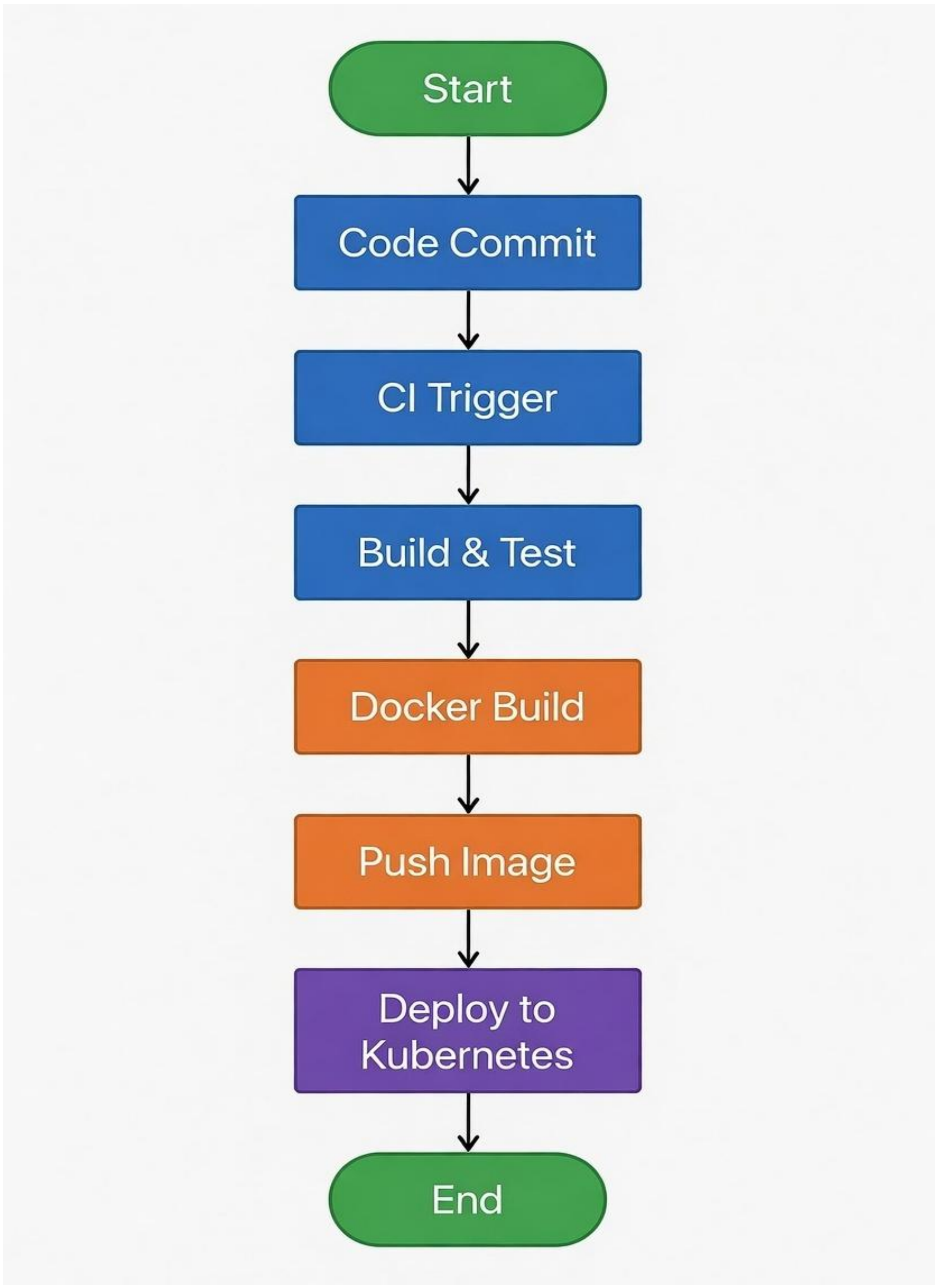
### 2.2 Process Flow

The system follows an automated deployment workflow.

**Step-by-Step Detailed Flow:**

1. Code Commit
   The developer pushes changes to GitHub main branch.
2. Trigger Event
   GitHub Actions detects the push event.
3. Build Stage
   Maven compiles code and runs tests.
4. Packaging Stage
   JAR file is generated.
5. Docker Build Stage
   Docker image is created using Dockerfile.
6. Docker Push Stage
   Image is uploaded to Docker Hub.
7. Deployment Stage
   Kubernetes pulls updated image.
8. Rolling Update
   New pods replace old pods gradually.

## B)CI/CD Flowchart Diagram Structure



### 2.3 Information Flow

Information flow explains data movement within the system.

**External Flow:**

User → Browser → Kubernetes Service → Pod → Application → Response

**Internal Flow:**

GitHub → CI/CD → Docker Hub → Kubernetes Cluster

All communication happens via HTTP/HTTPS.

The system isolates application logic from infrastructure, ensuring clean separation of concerns.

---

### 2.4 Components Design

Each component has a specific role:

**1. Spring Boot Application**

Handles API requests and business logic.

**2. GitHub Repository**

Stores source code and tracks versions.

**3. GitHub Actions**

Automates build and deployment tasks.

**4. Docker**

Packages application with dependencies.

**5. Docker Hub**

Stores container images.

**6. Kubernetes**

Manages container deployment and scaling.

**7. Terraform**

Provisions AWS infrastructure.

**8. AWS EC2**

Provides compute resources for Kubernetes.

This modular separation increases system reliability.

---

### 2.5 Key Design Considerations

**Scalability**

Enabled through Kubernetes Horizontal Pod Autoscaler.

**High Availability**

Multiple pod replicas ensure uptime.

**Portability**

Docker allows environment independence.

**Automation**

CI/CD eliminates manual deployment.

**Reproducibility**

Terraform ensures identical infrastructure every time.

**Maintainability**

Modular architecture allows easy updates.

---

### 2.6 API Catalogue

| **Method** | **Endpoint** | **Description** |
|---|---|---|
| GET | / | Returns welcome message |

Future expansion may include:

- Health check endpoint
- User management APIs
- Monitoring endpoint

## 3. Data Design

The Data Design section defines how data is structured, accessed, managed, secured, and migrated within the DevOps-based automated deployment system. Even though the current implementation is lightweight and primarily stateless, the architecture is designed to support full database integration in the future without significant modification.

The data design ensures consistency, scalability, integrity, and security. It separates business logic from data storage logic to maintain modularity and flexibility. This separation allows the application to scale independently from the data layer.

The system handles different types of data including:

- Application data
- Deployment metadata
- CI/CD build information
- Docker image versions
- Logs and monitoring data

The design supports both present requirements and future expansion.

---

### 3.1 Data Model

The data model defines how information is logically organized within the system. Although the current implementation exposes a basic REST endpoint, the system is designed to accommodate structured entities for future enhancements.

**Proposed Logical Entities**

**1. User Entity**

Attributes:

- user_id (Primary Key)
- name
- email
- role
- created_at

- updated_at

This entity will manage authentication and authorization if implemented in future versions.

---

**2. Deployment Entity**

Attributes:

- deployment_id
- version
- docker_image_tag
- deployment_time
- environment
- status

This entity stores details of each deployment performed through the CI/CD pipeline.

---

**3. Build Information Entity**

Attributes:

- build_id
- commit_hash
- build_status
- build_timestamp
- artifact_location

This entity maintains build history and traceability.

---

**4. Application Logs Entity**

Attributes:

- log_id
- log_level

- message
- timestamp
- source

This entity stores runtime logs for monitoring and auditing.

### 3.2 Data Access Mechanism

The system follows a layered architecture to manage data access efficiently. The data access mechanism ensures separation between business logic and database operations.

The application structure includes:

- Controller Layer
- Service Layer
- Repository Layer

The Repository Layer interacts directly with the database. In future implementation, Spring Data JPA with Hibernate ORM can be used to perform CRUD operations.

The data access mechanism ensures:

- Transaction management
- Data consistency
- Secure database connectivity
- Abstraction from database engine

Database configuration details will be stored in:

- application.properties file
- Environment variables
- Kubernetes Secrets

Sensitive credentials will never be hardcoded in the source code.

**Suitable Diagram – Layered Data Access Architecture**

Draw vertical layers:

Controller Layer
↓
Service Layer
↓
Repository Layer
↓
Database

This shows clear separation of responsibilities.

---

### 3.3 Data Retention Policies

Data retention policies define how long different types of data are stored in the system. Proper retention ensures storage optimization and compliance with best practices.

The system may implement the following retention strategies:

- Application logs retained for 30 days
- CI/CD logs stored in GitHub as per repository policy
- Docker images versioned using tags
- Old unused Docker images periodically removed
- AWS EC2 snapshots used for backup
- Terraform state files maintained securely

Retention policies help in:

- Reducing storage costs
- Improving system performance
- Maintaining audit trails
- Ensuring traceability of deployments

Log rotation strategies may also be implemented in Kubernetes to prevent excessive disk usage.

### 3.4 Data Migration

Data migration ensures smooth system upgrades and infrastructure changes without data loss or downtime.

The system supports safe migration strategies including:

- Rolling updates in Kubernetes
- Blue-Green deployment strategy
- Version-controlled database scripts
- Backup before deployment
- Terraform state versioning

Rolling updates allow new application versions to be deployed gradually while old versions are terminated only after new pods become stable. This ensures zero downtime.

If a database is integrated in future, schema migration tools such as Flyway or Liquibase can be used to manage version-controlled database updates.

Terraform maintains infrastructure state files, ensuring reproducibility and safe infrastructure changes.

Data migration strategy ensures:

- No data loss
- No service interruption
- Backward compatibility
- Safe rollback capability

## 4. Interfaces

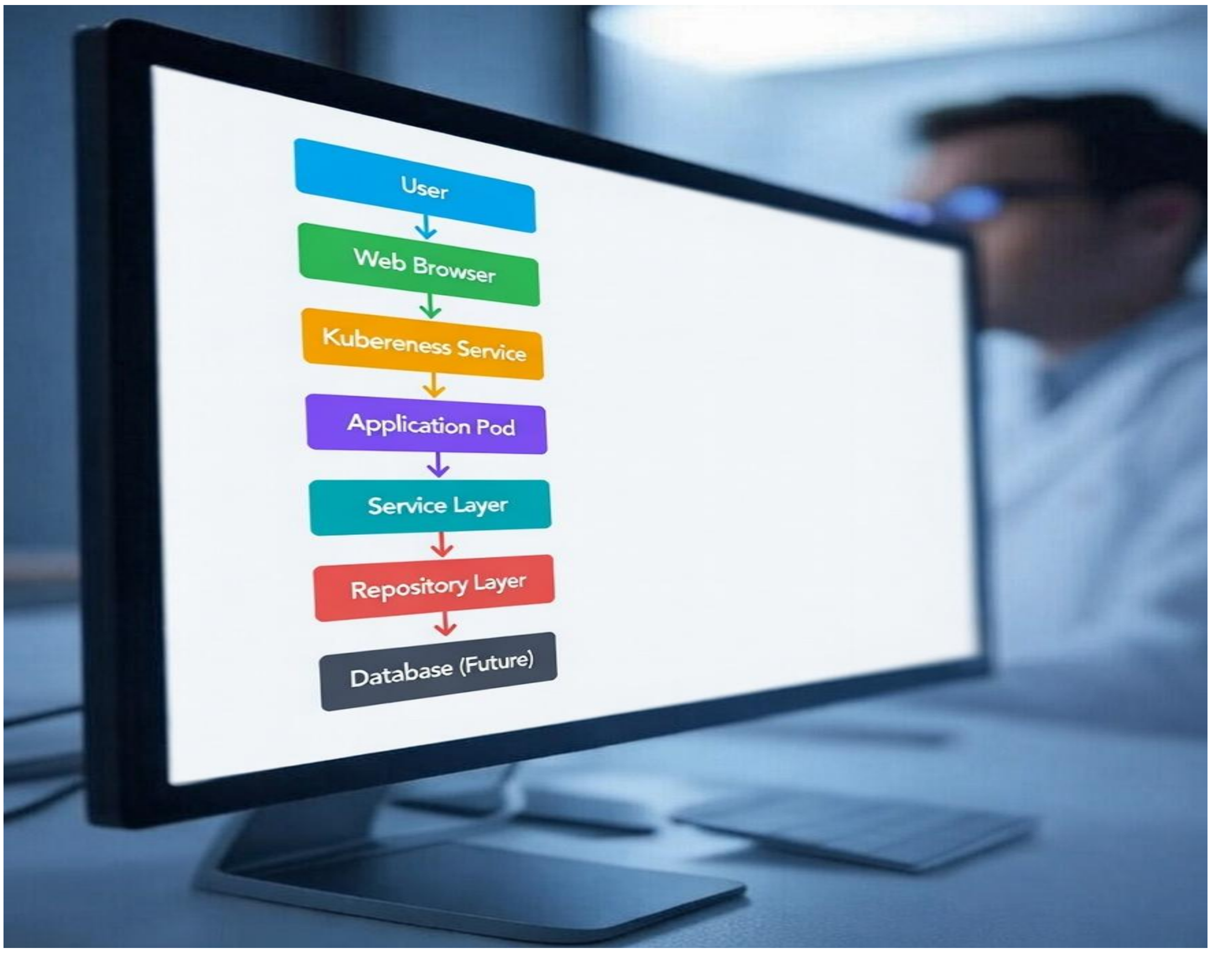
The Interfaces section describes how different components of the system communicate with each other and with external users. Proper interface design ensures smooth interaction, modularity, and secure data exchange within the DevOps-based automated deployment pipeline.

The system interfaces are broadly categorized into:

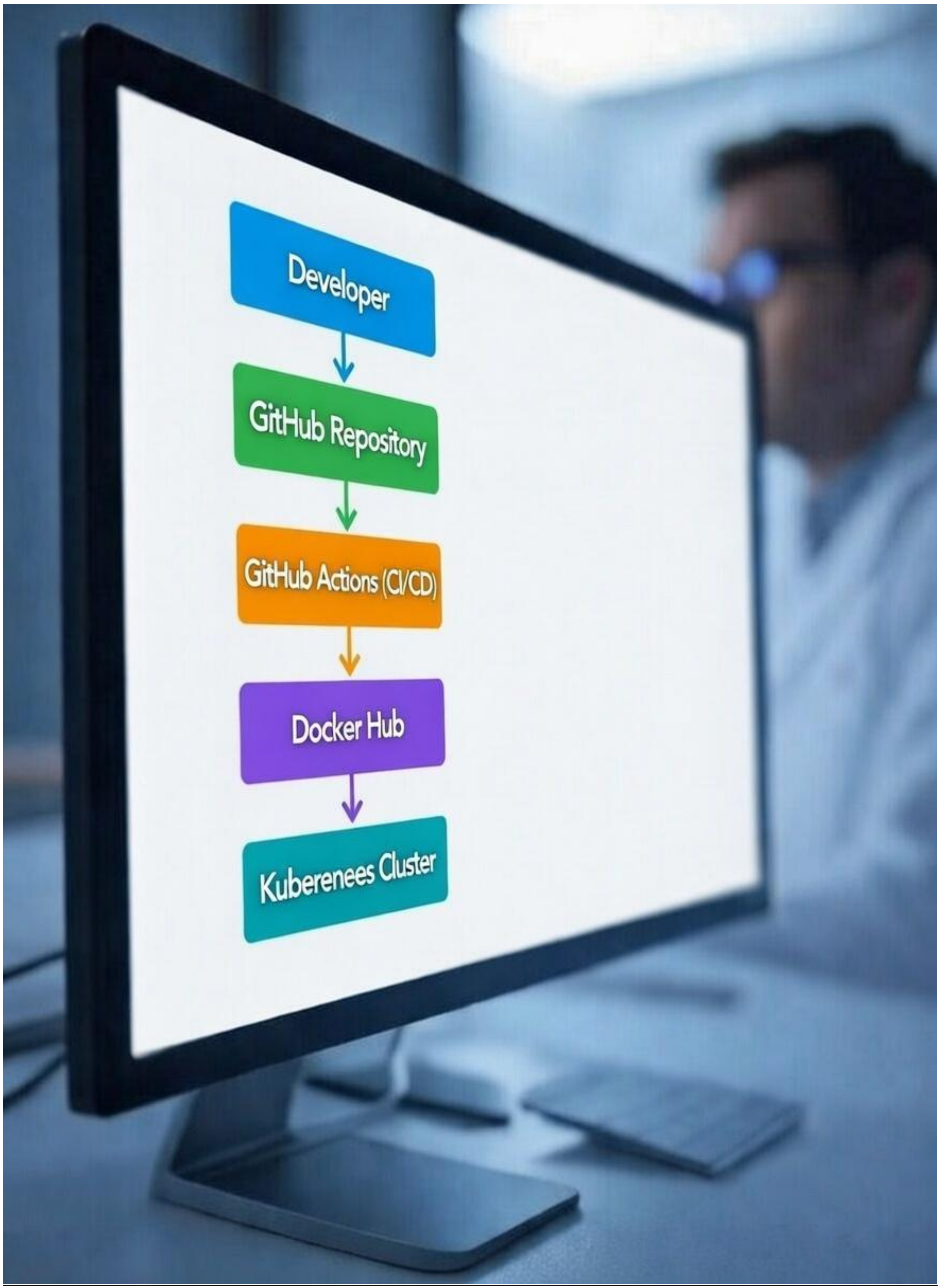
- External Interfaces
- Internal Interfaces
- CI/CD and Infrastructure Interfaces

External interfaces allow users to access the application through REST APIs using HTTP/HTTPS protocols. Internal interfaces manage communication between application layers such as Controller, Service, and Repository. CI/CD interfaces connect GitHub, Docker, and Kubernetes to automate build and deployment processes.

The system follows REST-based communication, stateless request handling, and secure credential management to maintain reliability and scalability.



## Side Flow (CI/CD Automation)



## 5. State and Session Management

State and Session Management defines how user information and application state are handled during interactions with the system. In this DevOps-based automated deployment pipeline, the application follows a stateless architecture.

In a stateless system, the server does not store any client session data between requests. Each request from the client contains all the necessary information required for processing. This design is essential for cloud-native and containerized applications because it enables easy scaling and high availability.

When a user sends a request, Kubernetes routes it to any available pod. Since no session data is stored in memory, any pod can handle the request without dependency on previous interactions. This ensures load balancing efficiency and horizontal scalability.

**Key Characteristics**

- No server-side session storage
- Each request is independent
- No session replication required
- Suitable for distributed systems
- Supports auto-scaling

**Advantages of Stateless Design**

- Easy horizontal scaling
- Improved fault tolerance
- Better performance in distributed environments
- Simplified load balancing
- Faster recovery in case of pod failure

If session management is required in the future (for authentication or user login), it can be implemented using:

- JWT (JSON Web Tokens)
- OAuth2 authentication
- Redis-based distributed session store

## 6. Caching

Caching is used to improve system performance by temporarily storing frequently accessed data in memory so that repeated requests can be served faster. In the current implementation, caching is not enabled because the application is lightweight and stateless. However, the architecture supports future integration of caching mechanisms.

Caching reduces response time, decreases server load, and improves overall system scalability. In cloud-native systems running on Kubernetes, caching plays an important role when handling high traffic.

**Possible Future Caching Strategies**

- In-memory caching within the application
- Redis distributed caching
- API response caching
- Kubernetes sidecar caching

If implemented, caching would store frequently requested data in memory instead of processing it repeatedly. This improves performance and reduces CPU utilization.

**Benefits of Caching**

- Faster response time
- Reduced backend processing
- Improved scalability
- Better user experience

## 7. Non-Functional Requirements

Non-functional requirements define system quality attributes such as reliability, scalability, security, and performance. These requirements ensure that the system operates efficiently under different conditions.

The DevOps-based deployment pipeline is designed to meet the following non-functional goals:

- High availability
- Scalability

- Security
- Reliability
- Maintainability
- Automation

These characteristics make the system production-ready and enterprise capable.

---

### 7.1 Security Aspects

Security is implemented across multiple layers of the architecture to protect application data, infrastructure, and deployment pipelines.

**Application Level Security**

- HTTPS communication
- Input validation
- Dependency vulnerability checks

**CI/CD Security**

- GitHub Secrets for storing credentials
- Restricted branch access
- Controlled pipeline permissions

**Container Security**

- Minimal base Docker images
- Image versioning and tagging
- Vulnerability scanning

**Kubernetes Security**

- Role-Based Access Control (RBAC)
- Namespace isolation
- Kubernetes Secrets management

**Cloud Security (AWS)**

- IAM roles and policies

## 7.2 Performance Aspects

Performance ensures that the system responds quickly and efficiently under varying workloads.

The system achieves performance optimization through:

- Horizontal Pod Autoscaling
- Load balancing across pods
- Stateless architecture
- Container resource isolation
- Efficient CI/CD automation

When traffic increases, Kubernetes automatically increases the number of pod replicas. When traffic decreases, replicas are reduced to save resources.

Rolling deployments ensure no downtime during application updates. Containerization ensures consistent runtime environments and optimized resource utilization.

Performance goals include:

- Minimal response time
- Zero downtime deployments
- Efficient CPU and memory usage
- Fast build and deployment cycles

---

## 8. References

The project design and implementation are based on official documentation and industry best practices from:

- Spring Boot Documentation
- Docker Official Documentation
- Kubernetes Documentation
- Terraform Documentation
- GitHub Actions Documentation
- AWS EC2 Documentation