Project - High Level Design

on

CI/CD Pipeline For Retail Application

Course Name: Devops

Institution Name: Medicaps University – Datagami Skill Based Course

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

Sr no	Student Name	Enrolment Number
1	SUJAY KUMAR MISHRA	EN22CS301996
2	SUKHENDRA SINGH	EN22CS301997
3	TANYA SOMWANSHI	EN22CS3011032
4	VAIBHAVI MEHTA	EN22CS3011052
5	VEDANT BHATORE	EN22CS3011070

Group Name: 01D9

Project Number:01

Industry Mentor Name: Mr. Vaibhav

University Mentor Name: Dr. Ritesh Joshi

Academic Year:2025-2026

Table of Contents

INTRODUCTION

This project focuses on the design and implementation of a professional DevOps-based Continuous Integration and Continuous Delivery (CI/CD) pipeline for a Retail Backend Application developed using Node.js and Express.

In modern software development, manual testing and deployment processes often lead to delays, configuration errors, and inconsistent environments. To overcome these challenges, this project integrates DevOps practices to automate the complete software lifecycle — from code integration to deployment.

The primary objectives of this project are:

- To develop a Retail Backend system for managing products, customers, and orders.
- To implement automated testing using Continuous Integration.
- To create containerized application artifacts using Docker.
- To automate deployment using Continuous Delivery workflows.
- To ensure reliable, scalable, and version-controlled releases.

By integrating backend development with DevOps automation, the system ensures faster releases, improved reliability, and reduced manual intervention.

1.1 Scope of the Document

This document defines the High-Level Design (HLD) of the DevOps-based Retail Backend Application. It outlines the architectural structure, workflow automation, and deployment strategy used in the system.

The scope of this document includes:

- Overall system architecture
- Application design and modules
- CI/CD workflow implementation
- Docker-based containerization
- Artifact versioning and registry management
- Deployment process
- Security and performance considerations

This document serves as a technical reference for understanding how DevOps practices are integrated with backend application development.

It focuses specifically on automation of:

- Code validation
- Artifact generation
- Deployment to production environment

1.2 Intended Audience

This document is intended for:

- Faculty members evaluating the project
- Project mentors and academic reviewers
- DevOps engineers
- Backend developers
- System architects
- Technical stakeholders

It provides architectural clarity and explains how backend development and DevOps automation work together in a structured and scalable way.
The document is written to provide both technical depth and conceptual clarity, making it suitable for academic as well as industry-level evaluation.

1.3 System Overview

The system consists of two major integrated components:

1) Retail Backend Application (Application Layer)
The backend is developed using Node.js and Express and provides RESTful APIs for managing retail operations.
It supports:

- Product management
- Customer management
- Order processing (validate stock, calculate total, update inventory)

The Orders module contains business logic that ensures product availability before confirming orders.
Currently, the system uses in-memory data storage for demonstration purposes. In production, it can be extended with a persistent database such as MongoDB or MySQL.

2)DevOps Automation Layer
The DevOps layer automates the application lifecycle using:

- GitHub Actions for CI/CD
- Docker for containerization

- Container Registry for artifact storage
- Kubernetes K8s for deployment

Whenever a developer pushes code:

1. Continuous Integration (CI) pipeline runs automated tests.

2. If tests pass, Continuous Delivery (CD) pipeline builds a Docker image.

3. The image is tagged using commit SHA for version control.
4. The artifact is pushed to a container registry.
5. The deployment environment pulls the latest image.
6. The updated application runs automatically.
   This eliminates manual deployment and ensures consistent environments across development, testing, and production.

2. System Design

The system design defines the overall technical architecture of the Retail Backend Application integrated with DevOps automation. It explains how different layers interact to ensure automated testing, containerized artifact creation, and reliable deployment.

The system follows a **layered architectural model** consisting of:

1. Application Layer
2. CI/CD Automation Layer
3. Containerization Layer
4. Artifact Management Layer
5. Deployment Layer

   Each layer is logically separated to maintain modularity, scalability, and maintainability.

2.1 Application Design

The application is developed using **Node.js and Express** following RESTful architecture principles.

The backend is structured into modular components to separate concerns and improve maintainability.

Core Modules

1) Products Module

This module is responsible for product management.

Functions include:

- Creating new product
- Updating product details
- Deleting products
- Viewing available products

Each product contains the following attributes:

- id
- name
- price

- stock quantity

This module ensures proper inventory management.

2)Customers Module

This module handles customer data.
Functions include:

- Registering customers
- Viewing customer details

Each customer contains:

- id
- name
- email

This module ensures customer identification during order processing.

3) Orders Module (Core Business Logic)

This is the most critical module in the application.
When an order request is received:

1. The system validates whether the product exists.
2. It verifies available stock.
3. If stock is sufficient, it reduces stock quantity.
4. It calculates total order value.
5. It generates an order record with status "PLACED".

This module ensures business rules are enforced before confirming transactions.

2.2 Process Flow

The DevOps-enabled process flow ensures automation from development to deployment.
The step-by-step workflow is:

1. Developer writes code locally.
2. Code is pushed to GitHub repository.
3. Continuous Integration (CI) pipeline is triggered automatically.
4. Dependencies are installed.
5. Automated unit tests are executed.
6. If tests pass, Continuous Delivery (CD) pipeline is triggered.
7. Docker image is built.
8. The image is tagged using commit SHA (unique version).
9. The image is pushed to a container registry.
10. Production environment pulls the latest image.
11. Application runs inside a container.

This workflow eliminates manual testing and deployment errors.

2.3 Information Flow

Information flow explains how data moves inside the system.

Application-Level Flow

Client Request
→ Express Route
→ Business Logic
→ Data Store
→ API Response
For example:
Client sends POST /orders
→ Order validation logic executes
→ Stock updated
→ Order response returned

DevOps-Level Flow

Source Code
→ CI Pipeline
→ Test Execution
→ CD Pipeline
→ Docker Image Creation
→ Registry Storage
→ Deployment Server
This ensures that only tested and validated code reaches production.

2.4 Components Design

The system consists of the following components:

Component	Description	Responsibility
Express Server	Node.js application	Handles HTTP requests
Business Logic Layer	Order processing logic	Validates stock and calculates totals
In-Memory Data Store	Temporary data storage	Stores products, customers, orders
Docker Engine	Container platform	Builds application image

Component	Description	Responsibility
GitHub Actions	CI/CD automation tool	Runs tests and builds artifacts
Container Registry	Artifact repository	Stores versioned Docker images
Deployment Server (K8s)	Runtime environment	Runs containerized application

Component Interaction

- Express server processes user requests.
- CI validates code quality.
- CD generates containerized artifacts.
- Deployment layer ensures scalable runtime execution.

2.5 Key Design Considerations

During system design, the following factors were prioritized:

1) Automation

Manual testing and deployment are eliminated through CI/CD.

2) Reliability

Only tested code is deployed to production.

3) Version Control of Artifacts

Docker images are tagged using commit SHA to enable rollback and traceability.

4) Scalability

Container-based deployment allows horizontal scaling.

5) Portability

Docker ensures the application runs consistently across environments.

6) Separation of CI and CD

- CI validates code.

- CD handles artifact creation and deployment.
  This separation ensures clean DevOps workflow.

2.6 API Catalogue

The application exposes RESTful APIs as follows:

HTTP Method	Endpoint	Description
GET	/	Health check
GET	/products	Fetches the list of all retail products
GET	/products/:id	Fetches details of a specific product
POST	/products	Adds a new product to inventory
GET	/customers	Fetches registered customer profile
GET	/orders	List all transactions and order history
POST	/orders	Creates a new order

Each API follows REST standards:

- JSON request and response
- Proper HTTP status codes
- Clear endpoint structure

3. Data Design

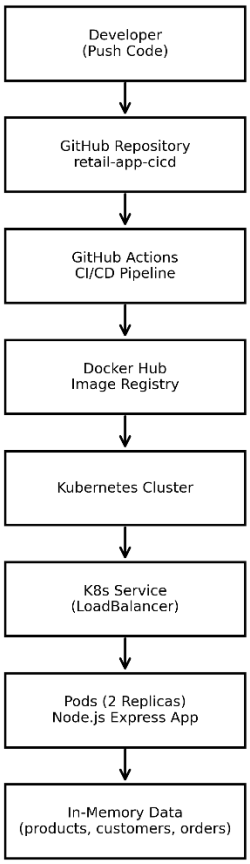
The Data Design section defines how data is structured, stored, accessed, and maintained within the Retail Backend Application.
In this project, the application uses in-memory data storage for demonstration purposes. However, the design follows a structured data modeling approach that can easily be extended to persistent databases such as MongoDB or MySQL in real-world scenarios.
The system manages three core entities:

- Product
- Customer
- Order

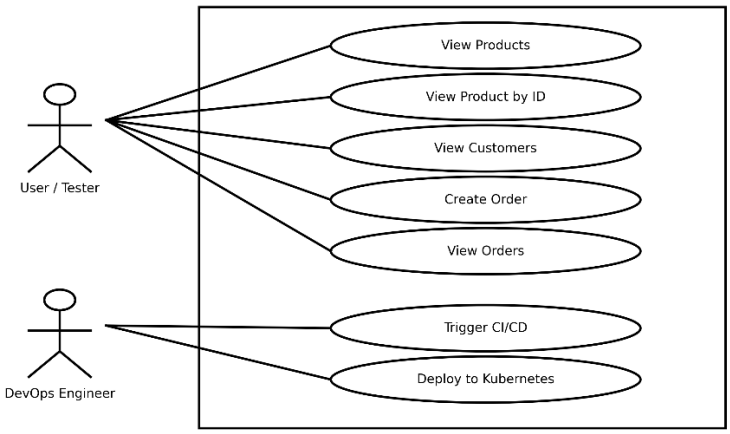
The data design ensures data consistency, integrity, and clear relationships between entities.

3.1 Data Model

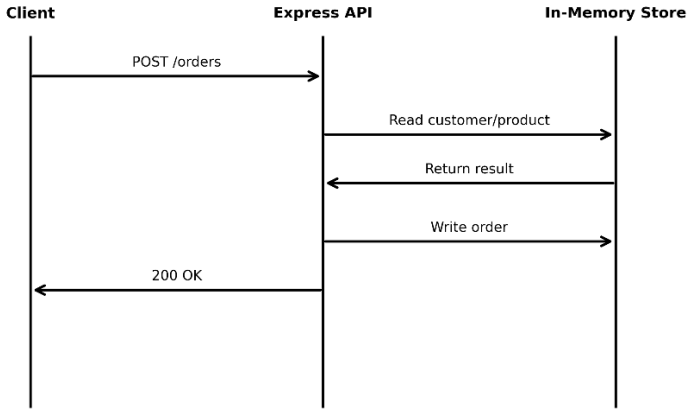
ARCHITECTURE DIAGRAM



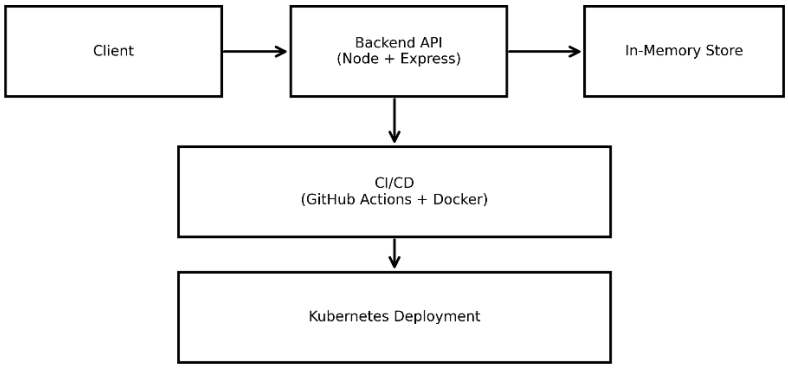
USE CASE DIAGRAM



SEQUENCE DIAGRAM



COMPONENT DIAGRAM



The data model defines the structure of the entities used in the system.

Product Entity

The Product entity represents items available for sale.

Attributes:

- id (Unique identifier)
- name (Product name)
- price (Product price)
- stock (Available quantity)

Purpose:

The Product model ensures proper inventory management and stock validation before processing orders.

Customer Entity

The Customer entity stores user information required for placing orders.

Attributes:

id (Unique identifier)

- name (Customer name)
- email (Customer email address)

Purpose:

This model helps associate orders with specific customers.

Order Entity

The Order entity represents purchase transactions.

Attributes:

- id (Unique identifier)
- customerId (Reference to customer)
- items (List of ordered product IDs)
- total (Total calculated price)
- status (Order status, e.g., "PLACED")

Purpose:

The Order model ensures that:

- Stock is validated
- Total amount is calculated
- Transaction details are recorded

3.2 Data Access Mechanism

The application uses Express route handlers to access and manipulate data.

Currently, data is stored in JavaScript arrays (in-memory storage).

Data Operations Include:

- Create (Add product/customer/order)
- Read (Retrieve records)
- Update (Modify product details)
- Delete (Remove product)

These operations are implemented using JavaScript array methods such as:

- push()
- find()
- filter()
- map()

In Real-World Production:

The data access mechanism can be extended to:

MongoDB (using Mongoose ORM)

- MySQL/PostgreSQL (using Sequelize or Prisma)

This ensures persistent and scalable data management.

3.3 Data Retention Policies

Data retention defines how long data is stored and maintained.

In the current implementation:

- Data is stored in memory.
- Data is cleared when the server restarts.
- No permanent storage mechanism is implemented.

This approach is suitable for:

- Demonstration
- Testing
- CI/CD validation

Future Implementation:

In a production environment, data retention policies would include:

- Persistent database storage
- Regular backups
- Transaction logging

- Data archiving strategies

3.4 Data Migration

Data migration refers to transferring data between storage systems or versions.

Currently:

- No migration is required due to in-memory storage.

However, future enhancements may include:

Database Integration

Migrating data from in-memory storage to:

- MongoDB
- MySQL
- PostgreSQL

Schema Migration

If database schema changes, migration tools such as:

- Liquibase
- Flyway
- Mongoose migration scripts

can be used to ensure version-controlled schema updates.

Backup and Restore Strategy

Data export in JSON format

Scheduled database backups

Cloud storage backup policies

4. Interfaces

Interfaces define how different components of the system communicate with each other and how external systems interact with the application.
In this DevOps-based Retail Backend system, there are two major categories of interfaces:

1. Application Interfaces
2. DevOps & Infrastructure Interfaces

4.1 Application Interfaces

The application exposes RESTful APIs over HTTP protocol.
Communication Standard:

- Protocol: HTTP
- Data Format: JSON
- Architecture Style: REST
  API Interaction Flow:
  Client (Browser/Postman/Frontend)
  → HTTP Request
  → Express Server
  → Business Logic
  → JSON Response
  Each API endpoint is designed following REST principles with proper HTTP methods such as GET and POST.
  Example:
- GET /products → Fetch product list

- POST /orders → Create new order
  These interfaces allow external applications or frontend systems to interact with the backend securely and efficiently.

4.2 DevOps Tool Interfaces

The system also interacts with external DevOps tools:
GitHub Repository

- Stores source code.
- Triggers CI/CD pipelines on push or merge.

GitHub Actions

- Runs automated testing.
- Builds Docker images.
- Executes deployment workflows.

Docker Engine

- Builds containerized artifacts.

Container Registry (DockerHub)

- Stores version-controlled Docker images.

Deployment Server (Kubernetes)

- Pulls latest Docker image.
- Runs containerized application

5. State and Session Management

The Retail Backend follows a stateless architecture, which means:

- Each request is independent.
- The server does not store session information.
- All required data must be included in every request.

Why Stateless Design?

Stateless systems provide:

- Better scalability
- Easier horizontal scaling
- Simplified load balancing
- Improved reliability

Since the application is deployed inside containers, stateless design ensures smooth scaling across multiple instances.

Current Implementation

- No session storage.
- No cookies or session IDs.

- Each API request is processed independently.

Future Enhancement

- JWT (JSON Web Token) authentication
- Role-based authorization
- Token validation per request

6. Caching

Currently, caching is not implemented in this project. However, caching is an important performance optimization strategy in scalable systems.

Purpose of Caching

Caching helps to:

- Reduce repeated computation
- Minimize database load
- Improve response time
- Enhance overall performance

Future Caching Enhancements

The system can integrate:

1) Redis (In-memory caching)

To store frequently accessed product data.

2) API-level Caching

For GET endpoints such as /products.

3)CDN Integration

For static resources in future frontend integration.

7. Non-Functional Requirements

Non-functional requirements describe how the system performs rather than what it does.

The system is designed to satisfy the following non-functional properties:

- Reliability
- Scalability
- Maintainability
- Availability
- Portability
- Automation efficiency

Reliability

- Automated CI testing prevents faulty builds.
- Only tested code is deployed.

Docker ensures consistent runtime environment.

Scalability

- Containerized architecture supports horizontal scaling.
- Stateless design supports load balancing.
- Kubernetes (optional) enables replica management.

Maintainability

- Modular code structure.
- Clear separation between CI and CD.
- Version-controlled artifacts.

Portability

- Docker ensures the application runs consistently across environments.

Availability

Automated deployment reduces downtime.

- Container restart capability ensures service continuity.

7.1 Security Aspects

Security is a critical component in DevOps systems

Current Security Measures

- Input validation at API level.
- No hardcoded credentials in code.
- Use of GitHub Secrets for Docker registry authentication.
- Controlled production branch access.

DevOps Security Practices

- CI/CD workflows protected by branch rules.
- Secure artifact versioning.
- Registry authentication using encrypted secrets.

Future Security Enhancements

- JWT authentication
- Role-Based Access Control (RBAC)
- HTTPS implementation

- Docker image vulnerability scanning
- Dependency scanning in CI pipeline

7.2 Performance Aspects

Performance depends on both application architecture and deployment strategy.

Application-Level Performance

- Node.js non-blocking asynchronous model.
- Lightweight REST APIs.
- Minimal processing overhead.

Container-Level Performance

- Fast startup time.
- Lightweight Docker image.
- Reduced environment configuration overhead.

Deployment-Level Performance

- Automated deployment reduces downtime.
- Horizontal scaling capability.
- Container restart ensures resilience.

Future Performance Enhancements

- Load balancing
- Auto-scaling using Kubernetes
- Caching integration
- Monitoring tools (Prometheus, Grafana)

8. References

The following resources were referred to during the design and implementation of this project:

1. Node.js Official Documentation
2. Docker Official Documentation
3. GitHub Actions Documentation
4. DevOps Best Practices and CI/CD Guidelines