Project - High Level Design

on

CI/CD for Node.js retail app

Course Name: DevOps

Institution Name: Medi caps University – Datagami Skill Based Course

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

Sr no	Student Name	Enrolment Number
1.	Aditya Agrawal	EN22IT301006
2.	Atharv Shukla	EN22IT301025
3.	Malya Singh	EN24CA5030092
4.	Mohit Gupta	EN21CA301036
5.	Nikhil Kumar Verma	EN23IT3L1002

Group Name: Group 03D12

Project Number: DO-41

Industry Mentor Name: Mr. Vaibhav

University Mentor Name: Prof. Akshay Saxena

Academic Year: 2026

Table of Contents

1. Introduction

This document presents the High-Level Design (HLD) for the **CI/CD-enabled Node.js Retail Application**. It describes the system architecture, major components, workflows, and technical decisions involved in implementing an automated software delivery pipeline using modern DevOps practices.

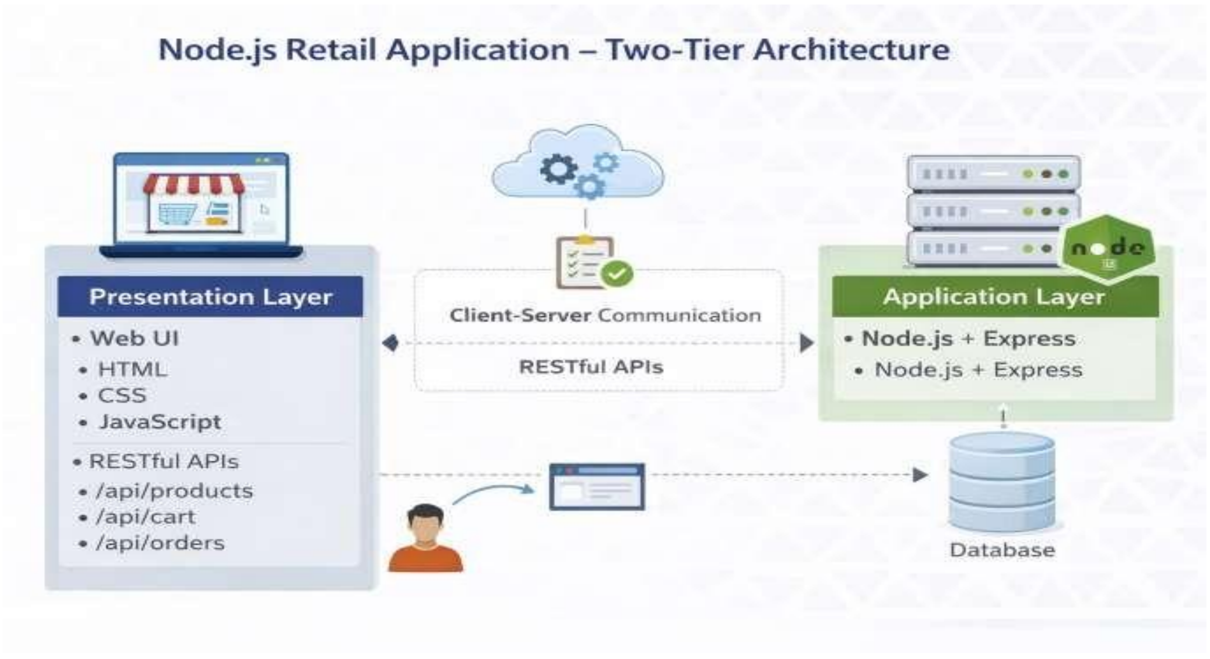
1.1 Scope of the Document

- Design of the Node.js Retail Application
- CI/CD pipeline implementation
- Integration with GitHub Actions/Jenkins
- Containerization using Docker
- Deployment architecture
- Security and performance considerations

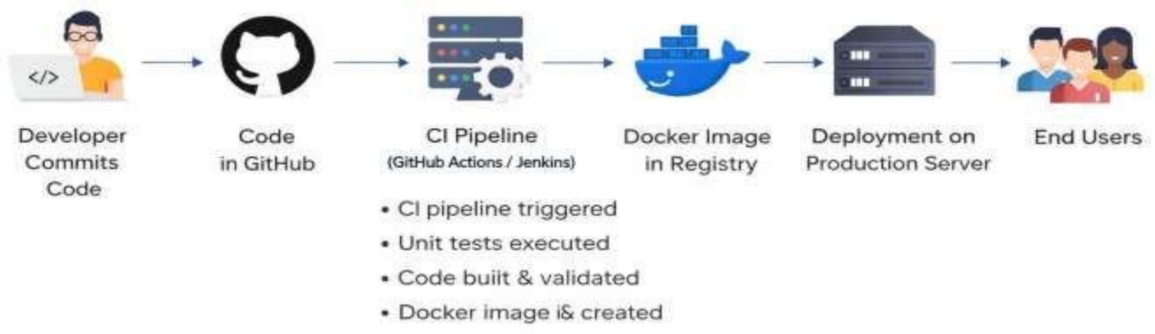
1.2 Intended Audience

- Software Developers
- DevOps Engineers
- System Architects
- Project Evaluators
- Faculty and Review Committees

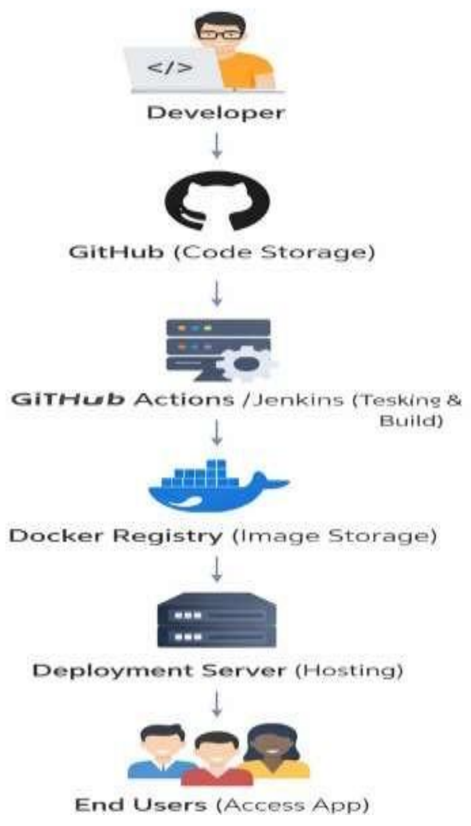
2.1 Application Design



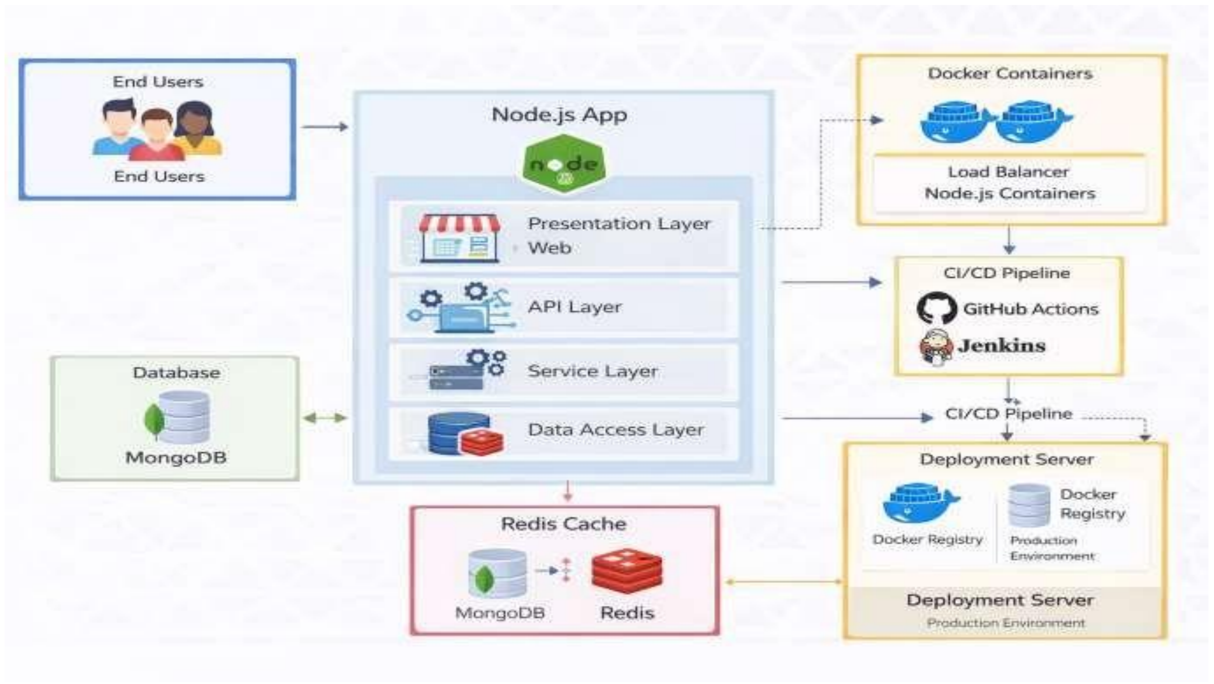
2.2 Process Flow



2.3 Information Flow


Developer
GitHub (Code Storage)
GiTHub Actions /Jenkins (Tesking & Build)
Docker Registry (Image Storage)
Deployment Server (Hosting)
End Users (Access App)

2.4 Components Design



2.5 Key Design Considerations

1. Branching Strategy
   - Use feature → develop → production flow
   - Protect production branch with PR approvals
   - Mandatory CI checks before merge

2. Automated Testing
   - Run unit tests on every commit/PR
   - Fail pipeline immediately on test failure
   - Maintain test coverage threshold

3. CI Pipeline Design
   - Fully automated and repeatable
   - Steps: checkout → install → test → build → push
   - Keep pipeline execution fast

4. Docker Best Practices
   - Use lightweight base image
   - Multi-stage builds
   - Proper image tagging (commit-id, latest)
   - No hardcoded secrets

5. Secure Registry Access
   - Store credentials in Jenkins/GitHub Secrets
   - Restrict image push permissions

6. Kubernetes Deployment
   - Use rolling updates (zero downtime)
   - Define resource requests & limits
   - Configure Deployment, Service, Ingress

7. Environment Separation
   - Separate Dev / Staging / Production
   - Different configs per environment

8. Secrets Management

- Use Kubernetes Secrets
- Never store credentials in code

9. Deployment Triggering

- CI on commit/PR
- CD only on merge to production

10. Scalability & Reliability

- Minimum 2 replicas
- Enable auto-scaling
- Self-healing via Kubernetes

2.6 API Catalogue

Authentication APIs

- POST /api/auth/register → Register user
- POST /api/auth/login → Login (JWT)
- GET /api/auth/profile → Get user profile
- POST /api/auth/logout → Logout

Product APIs

- GET /api/products → List all products
- GET /api/products/:id → Get product details
- POST /api/products → Add product (Admin)
- PUT /api/products/:id → Update product (Admin)
- DELETE /api/products/:id → Delete product (Admin)

Cart APIs

- GET /api/cart → View cart
- POST /api/cart/add → Add to cart
- PUT /api/cart/update → Update quantity
- DELETE /api/cart/remove/:id → Remove item

3.1 Data Models

Main entities:

- User (UserID, Name, Email, Password)
- Product (ProductID, Name, Price, Stock)
- Order (OrderID, UserID, Date, Status) • Cart (CartID, UserID, Items)

Relationships:

- One User → Many Orders
- One Order → Many Products

3.2 Data Access Mechanism

1. ORM/ODM (Mongoose / Sequelize)

   ORM/ODM tools are used to interact with the database using JavaScript instead of complex SQL queries. They simplify database operations such as create, read, update, and delete.

2. Secure Database Connections

   All database connections are protected using authentication, encryption, and environment variables. This prevents unauthorized access and protects sensitive data.

3. Repository Pattern

   The repository pattern separates database logic from business logic. This improves code readability, maintainability, and makes future changes easier.

4. API-Based Access

Data is accessed through secure RESTful APIs instead of direct database access.This ensures controlled data flow and improves system security.

3.3 Data Retention Policies

- 1. User Data Retained as per Policy

  User information is stored securely and retained according to organizational and legal policies. Outdated or inactive user data is periodically reviewed and removed when no longer required.

- 2. Order Data Stored for Auditing

  Order and transaction records are preserved for monitoring, reporting, and audit purposes. This helps in resolving disputes and maintaining business transparency.

- 3. Automated Backups

  The database is backed up automatically at regular intervals using scheduled scripts. This ensures data recovery in case of system failure or accidental deletion.

- 4. Archival of Old Records

  Old and inactive records are moved to separate archival storage after a defined period.

4. Interfaces

- The system provides:
- Web Interface for users
- REST API Interface for services

5.State and Session Management

JWT-based authentication

- Secure cookies
- Token expiration and refresh

6. Caching

1. Browser Caching

Static files such as HTML, CSS, JavaScript, and images are stored in the user's browse. This reduces repeated downloads and improves page loading speed.

2. API Response Caching

Responses from commonly used APIs are cached for a fixed time period.This minimizes repeated processing and improves overall system efficiency.

7. Non-Functional Requirements

7.1 Security Aspects

•HTTPS encryption

•JWT authentication

•Role-based access control

•Secrets management

•Firewall and network security

•Vulnerability scanning in CI pipeline

7.2 Performance Aspects

1. Optimized CI/CD Pipeline

   Automated testing and build processes using GitHub Actions and Jenkins reduce build time. Parallel jobs and caching dependencies improve pipeline execution speed.

2. Containerized Deployment

   Docker containers ensure lightweight and consistent application environments. This reduces startup time and improves deployment efficiency.

3. Efficient API Processing

   RESTful APIs are optimized using proper request handling and indexing. This ensures faster data retrieval and reduced server response time.

4. Database Performance Optimization

   Indexes and optimized queries are used to improve database access speed. Caching mechanisms reduce frequent database calls.

5. Load Handling and Scalability

   The system supports horizontal scaling using multiple containers. This allows handling high user traffic without performance degradation.

6. Resource Monitoring and Optimization

   CPU, memory, and network usage are continuously monitored. This helps in identifying bottlenecks and improving system performance.

7. Faster Deployment Cycles

   Automated container builds and deployments reduce release time.This enables quick updates without affecting end-user experience.

8.References

GitHub Official Documentation https://docs.github.com

(Used for version control and CI with GitHub Actions)

- Jenkins User Guide

https://www.jenkins.io/doc (Used for future-ready CI/CD automation)

- Docker Documentation https://docs.docker.com (Used for containerization and deployment)
- Node.js Official Website https://nodejs.org

(Used for backend application development)

- Express.js Documentation https://expressjs.com (Used for RESTful API development)
- Git Documentation

https://git-scm.com/doc (Used for source code management)