# Project - High Level Design

# College Management System

## Course Name: DevOps Foundation

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

| Sr no | Student Name | Enrolment Number |
|---|---|---|
| 01 | Sneha Jain | EN22CS301966 |
| 02 | Soumya Barve | EN22CS301975 |
| 03 | Sumit Pawar | EN22CS301100 |
| 04 | Vikas Devnani | EN22CS3011080 |
| 05 | Vikas Dhakad | EN22CS3011081 |

***Group Name:***

*Group 09D9*

***Project Number:*** *DO-09*

***Industry Mentor Name:***

*Mr. Vaibhav*

***University Mentor Name:***

*Prof. Ritesh Joshi*

***Academic Year:*** *2026*

# Table of Contents

# 1. Introduction

## 1.1 Scope of the Document

This High-Level Design (HLD) document provides a comprehensive overview of the architecture and deployment design of the College Management System.
The system is designed to demonstrate modern frontend development integrated with advanced DevOps practices including containerization, image registry management, CI/CD automation, cloud hosting, and Kubernetes-based orchestration.
This document describes:
• Overall system architecture
• Core application and infrastructure components
• Deployment and CI/CD workflow
• Kubernetes-based orchestration using K3s
• Service exposure using NodePort
• Rollout-based deployment strategy
• Security and performance considerations
• Non-functional requirements
This HLD serves as a high-level architectural and deployment blueprint before diving into implementation-level details.

## 1.2 Intended Audience

This document is intended for:
• Academic evaluators reviewing the DevOps implementation
• Industry mentors supervising the project
• Developers maintaining or extending the system
• DevOps engineers managing CI/CD pipelines
• System administrators managing AWS and Kubernetes infrastructure

## 1.3 System Overview

The College Management System is a web-based application developed using React (Vite). It provides a user interface for managing college-related academic information such as student records, faculty information, courses, attendance, and results.
From a DevOps perspective, the project demonstrates:
• Containerization using Docker with multi-stage builds
• Image storage using Docker Hub
• Automated CI/CD using GitHub Actions
• Cloud hosting on AWS EC2 (t3.micro)
• Lightweight Kubernetes orchestration using K3s
• External service exposure using NodePort
• Deployment updates using rollout restart
• Secure SSH-based deployment to EC2
The system follows a container-based deployment architecture:

1. Presentation Layer – React (Vite) frontend application
2. Container Layer – Docker image built via multi-stage build
3. Orchestration Layer – K3s Kubernetes cluster
4. Cloud Infrastructure – AWS EC2 instance

The application is built as a production-ready Docker image, pushed to Docker Hub, deployed on

# 2. System Design

## 2.1 Application Design

The system is designed with clear separation between development, containerization, deployment, and orchestration layers.

**Frontend Layer (React with Vite)**

The frontend is built using React with Vite for fast development and optimized production builds.

Responsibilities:

- Rendering dashboards and user interfaces
- Managing routing
- Handling form inputs
- Communicating with backend APIs (if integrated)
- Managing client-side state

The production build is generated using Vite and served as a static optimized bundle inside a Docker container.

**Containerization Layer (Docker – Multi-Stage Build)**

Docker is used to create optimized production images.

Multi-stage build ensures:

- Development dependencies are removed in final image
- Reduced image size
- Improved security
- Faster deployment

The final production image is pushed to Docker Hub.

**CI/CD Layer (GitHub Actions)**

GitHub Actions automates the CI/CD process.

Workflow:

1. Code pushed to GitHub repository
2. GitHub Actions workflow triggered
3. Docker image built
4. Image pushed to Docker Hub
5. SSH connection established to EC2
6. Kubernetes deployment updated

This eliminates manual deployment steps.

**Cloud & Orchestration Layer**

The application is deployed on:

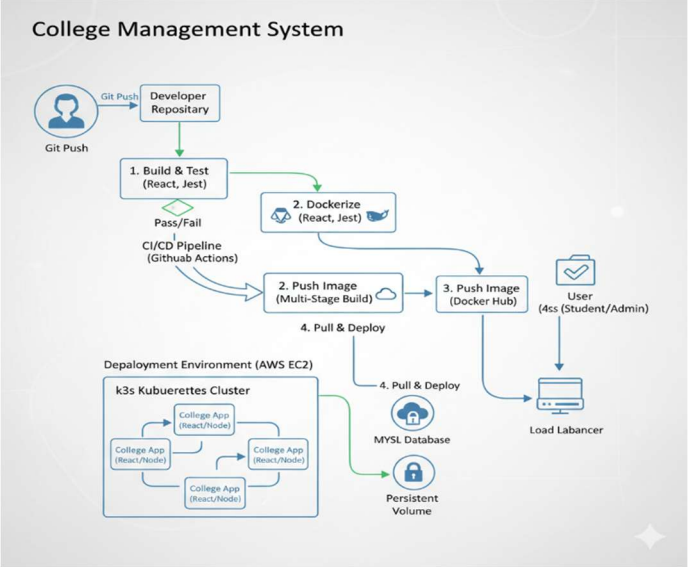
- AWS EC2 (t3.micro instance)
- K3s (Lightweight Kubernetes distribution)

K3s manages:

- Deployment creation
- Pod lifecycle management
- Container scheduling
- Service exposure

• Application accessible via EC2 Public IP + NodePort
• No external load balancer required
• Suitable for lightweight deployment environments

Deployment Strategy (Rollout Restart)
Application updates are deployed using:
kubectl rollout restart deployment
Benefits:
• Zero downtime deployment
• Automatic pod replacement
• Smooth version transition




## 2.2 Process Flow

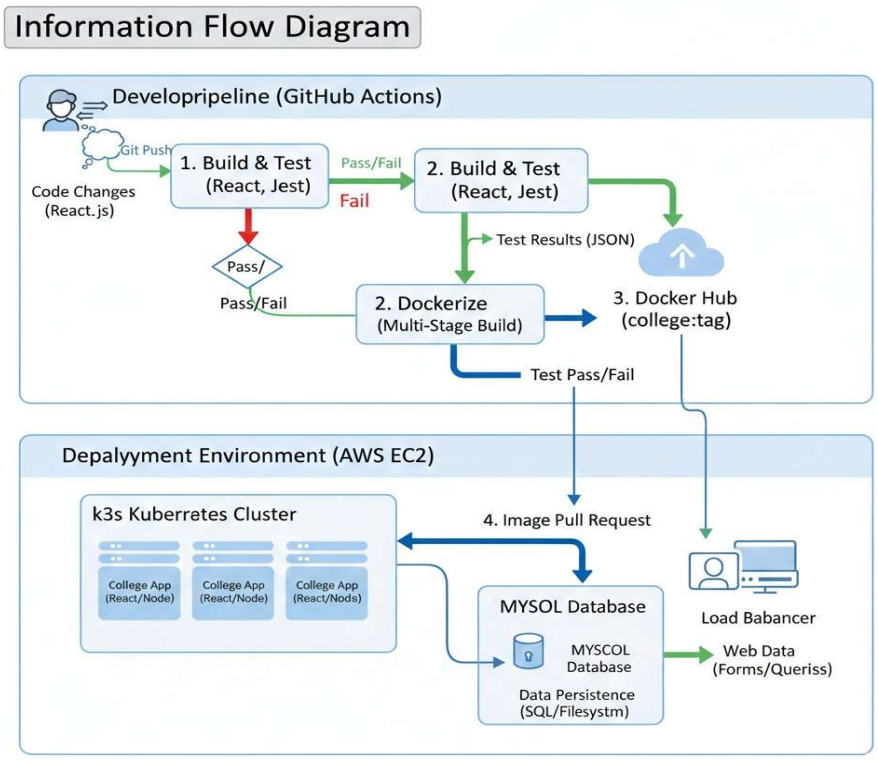
Application Deployment Flow

1. Developer pushes code to GitHub repository.
2. GitHub Actions workflow is triggered.
3. Docker multi-stage build creates optimized production image.
4. Image pushed to Docker Hub.
5. SSH-based connection established to AWS EC2.
6. K3s pulls latest image from Docker Hub.
7. Kubernetes deployment updated.
8. Rollout restart ensures updated pods are running.
9. NodePort exposes application to users.

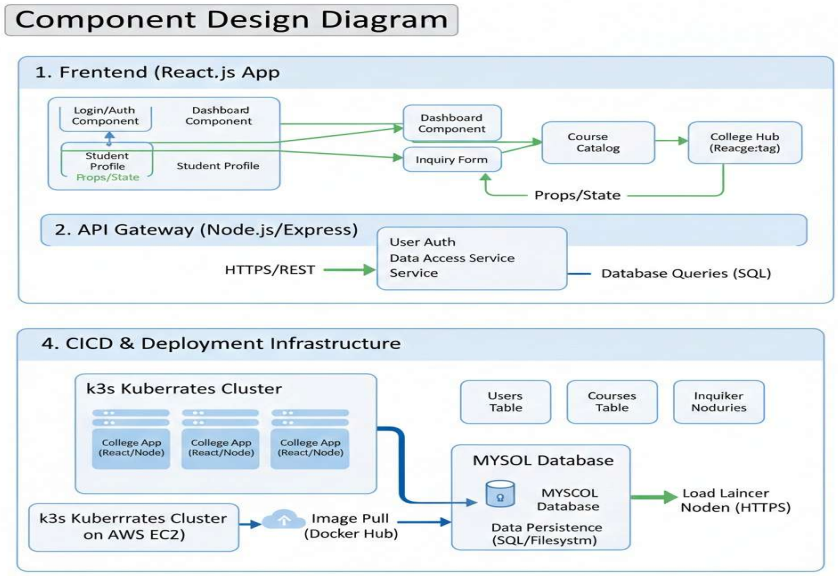
## 2.3 Information Flow

• Developer → GitHub Repository
• GitHub → GitHub Actions
• GitHub Actions → Docker Build
• Docker Image → Docker Hub

• Users → Access via Public IP
All deployment communication occurs securely via SSH and Kubernetes APIs.




**2.4 Components Design**
Application Components
• React UI Components
• Routing Components
• Dashboard Components
DevOps Components
• Dockerfile (Multi-stage build)
• Docker Image
• Docker Hub Repository
• GitHub Actions Workflow File
• AWS EC2 Instance (t3.micro)
• K3s Cluster
• Kubernetes Deployment
• NodePort Service
Each component is independently manageable and replaceable.

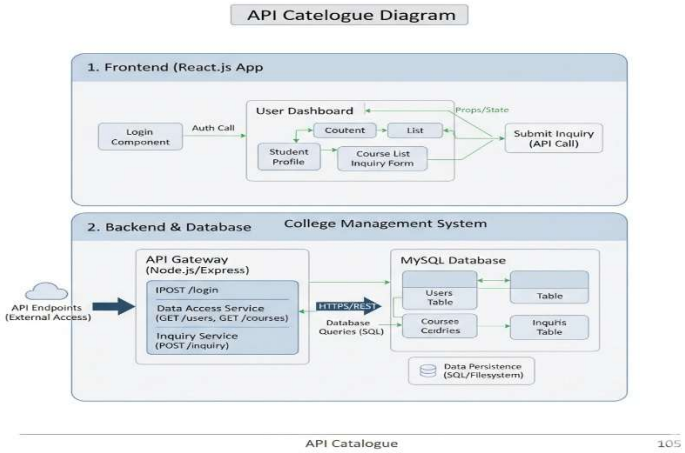



## 2.5 Key Design Considerations

• Lightweight infrastructure (t3.micro + K3s)
• Optimized Docker images
• Automated CI/CD pipeline
• Secure SSH-based deployment
• Scalable Kubernetes architecture
• Minimal manual intervention
• Cloud cost efficiency

## 2.6 API Catalogue

Since this project focuses primarily on frontend deployment and DevOps implementation, API endpoints depend on backend integration (if applicable).
If backend is integrated, APIs follow REST standards and exchange data in JSON format.



# 3. Data Design

### 3.1 Data Model

Data structure depends on backend integration. The frontend consumes structured JSON data for:

- Student records
- Faculty records
- Course data
- Attendance records
- Results

### 3.2 Data Access Mechanism

- RESTful API communication
- JSON-based data exchange
- Secure HTTP communication

### 3.3 Data Retention Policies

- Data persistence handled at backend level
- Infrastructure backups managed via AWS EC2 snapshot strategy

### 3.4 Data Migration

- Schema managed through version control
- Deployment synchronized via CI/CD pipeline

# 4. Interfaces

### 4.1 Internal Interfaces

React ↔ Container Runtime

- Static production build served inside Docker container

K3s ↔ Docker Runtime

- Kubernetes manages Docker containers
- Pods scheduled and restarted automatically

### 4.2 External Interfaces

AWS EC2

- Hosts K3s cluster
- Provides public IP
- Manages compute resources

Docker Hub

- Stores application images
- Provides image versioning

GitHub Actions

- Automates build and deployment
- Connects securely to EC2 via SSH

# 5. State and Session Management

The frontend manages client-side state using React state management.
Since deployment is container-based:
• No server-side session persistence
• Stateless deployment model
• Kubernetes manages pod lifecycle
Stateless architecture ensures scalability and reliability.

# 6. Caching

Currently no dedicated caching layer implemented.
Potential future enhancements:
• Kubernetes Horizontal Pod Autoscaling
• Reverse proxy caching
• CDN integration

# 7. Non-Functional Requirements

**7.1 Security Aspects**
• SSH-based secure deployment
• Environment variable protection
• Docker container isolation
• Kubernetes namespace isolation
• Restricted EC2 security groups

**7.2 Performance Aspects**
• Multi-stage Docker build reduces image size
• Lightweight K3s improves resource efficiency
• NodePort allows direct access without heavy load balancers
• Rollout restart ensures minimal downtime
• Optimized React production build via Vite

# 8. References

• React Documentation
• Vite Documentation
• Docker Documentation
• Docker Hub Documentation
• GitHub Actions Documentation
• Kubernetes Documentation
• K3s Documentation
• AWS EC2 Documentation