Containerized Construction Management System: A Cloud-Native DevOps Implementation

Course Name: DevOps

Institution Name: Medicaps University – Datagami Skill Based Course

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

Sr no	Student Name	Enrolment Number
1.	Vivek Sharma	EN22ME304116
2.	Jayesh Chouhan	EN23MEEL4001
3.	Sakina Modi	EN22EL301044
4.	Vinay Birla	EN22CS3011083
5.	Dev Singh Ghuraiya	EN22ME304022

Group Name:05D10

Project Number:DO-18

Industry Mentor Name:

University Mentor Name: Avnesh Joshi

Academic Year:2025-26

Table of Contents

1. Introduction

The Construction Management System is a full-stack, containerized web application designed to modernize construction operations through centralized project tracking, workforce management, material inventory control, and real-time analytics. Built using the MERN stack and deployed using Docker on AWS EC2, the system ensures consistent performance across development and production environments.

The project integrates DevOps practices, automated deployment workflows, RESTful API architecture, and responsive UI design to deliver a scalable and production-ready solution. It demonstrates how containerization and cloud infrastructure can transform traditional construction workflows into a structured, data-driven management system.

---

2. Problem Statement & Objectives

2.1 Problem Statement

The construction industry faces inefficiencies due to fragmented systems, manual processes, and inconsistent application environments. Deployment complexity, lack of scalability, and absence of centralized dashboards reduce operational visibility and increase project risk.

Key Challenges:

- Environment inconsistency across systems
- Manual and error-prone deployment
- Limited scalability of traditional systems
- No automated alerts for delays or low inventory
- Poor real-time reporting and analytics

2.2 Project Objectives

The objective is to design and deploy a scalable, secure, and containerized Construction Management System using modern DevOps principles.

Technical Objectives:

- Full-stack MERN implementation
- Docker multi-stage builds for optimized images
- 15+ RESTful API endpoints
- Health checks ensuring 99.9% uptime
- API response time < 200ms
- Deployment on AWS EC2

Functional Objectives:

- Project lifecycle management
- Worker tracking and assignment
- Inventory management with automated low-stock alerts and
- Real-time dashboards and reports

1.3 Scope of the Project

The project includes frontend development (React SPA), backend API development (Node.js & Express), database modeling (MongoDB), containerization (Docker), orchestration (Docker Compose), and cloud deployment (AWS EC2).

Core Modules Covered:

- Projects management
- Workers management
- Materials inventory
- Dashboard analytics
- Financial and operational reporting

3. Proposed Solution

3.1 Key Features

- Containerized deployment using Docker
- Three-tier architecture (Frontend, Backend, Database)
- Multi-stage Docker builds for optimization
- RESTful API with structured endpoints
- Automated health checks
- Low-stock alert system
- Interactive dashboard with charts
- Responsive UI design
- Cloud deployment on AWS

3.2 Overall Architecture / Workflow

The system follows a structured three-tier containerized architecture where user requests flow sequentially from the browser to the frontend, backend, and database layers through a secure Docker network.

Step 1: User Interaction Layer

Users access the application through a web browser (Desktop, Tablet, or Mobile). They perform actions such as:

- Viewing dashboard analytics
- Creating, editing, or deleting projects
- Managing workers and assignments
- Tracking material inventory
- Generating analytical reports

All user actions generate HTTP requests (GET, POST, PUT, DELETE) over **Port 80**.

Step 2: Frontend Container (construction-frontend)

The frontend runs inside a Docker container using **Nginx Alpine + React SPA**.

Nginx Web Server Responsibilities:

- Serves static files (HTML, CSS, JS, images)
- Enables Gzip compression
- Reverse proxies `/api/*` requests to `http://backend:5000`
- Performs health checks using `wget localhost:80`
- Runs under a non-root user (`nginx-user`)

React Application Components:

- Landing.js – Feature overview
- Dashboard.js – Real-time analytics
- Projects.js – Project management
- Workers.js – Workforce tracking
- Materials.js – Inventory management
- Reports.js – Business intelligence
- Navigation.js – Routing and navigation

Frontend Optimization:

- Image size: ~30MB (90% optimized via multi-stage build)
- Base image: `nginx:alpine`
- Health check interval: Every 30 seconds (3 retries)

When users perform actions, the React app sends API calls using **Axios** with JSON request/response format to `/api/*`.

Step 3: Backend Container (construction-backend)

The backend runs on **Node.js 18 Alpine + Express 4.18.2**.

Express Server Configuration:

- Internal Port: 5000
- External Port: 5001
- CORS middleware enabled
- JSON body parsing
- Health endpoint: `GET /health`
- Root endpoint for API info
- Runs as non-root user (`nodejs`)

API Routes:

- `GET /api/projects` – List projects
- `POST /api/projects` – Create project
- `GET /api/projects/:id` – Retrieve project
- `PUT /api/projects/:id` – Update project
- `DELETE /api/projects/:id` – Delete project
- `GET /api/projects/stats/summary` – Project statistics
- Similar CRUD endpoints for workers and materials

Mongoose Models:

- Project.js – 7 fields + timestamps
- Worker.js – Includes project reference
- Material.js – Includes virtual field for low-stock detection

Backend Optimization:

- Image size: ~150MB
- Base image: `node:18-alpine`
- Health check: Every 30 seconds (3 retries)
- Service dependency: Starts only after MongoDB is healthy

The backend connects to the database using:

`mongodb://mongo:27017/construction`

Step 4: Database Container (construction-mongo)

The database layer uses the official **MongoDB 7** Docker image.

MongoDB Configuration:

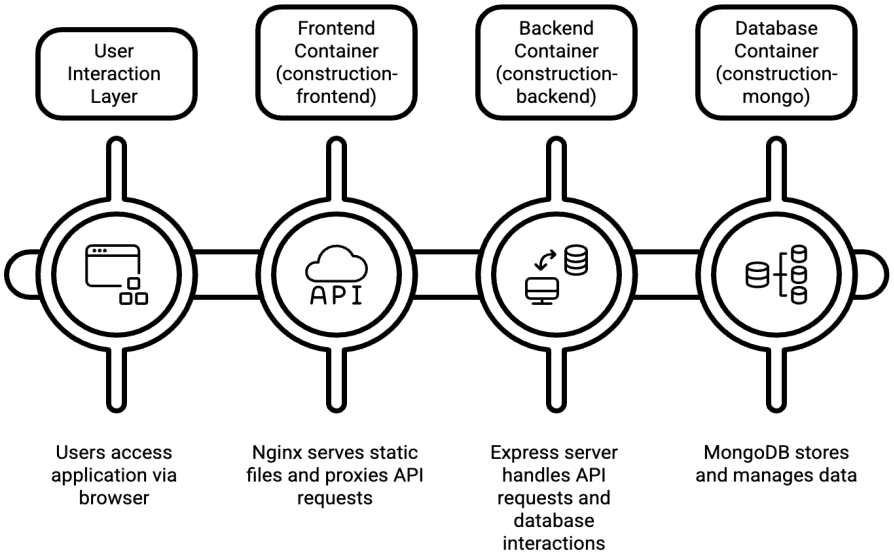
- Internal Port: 27017 (not exposed externally)
- Database: `construction`
- Collections: projects, workers, materials
- Indexed fields for performance (status, role, category)
- `MONGO_INITDB_DATABASE=construction`

Data Persistence:

- Named Docker volume: `mongo-data`
- Mounted at `/data/db`
- Survives container restarts and recreation
- Managed by Docker local driver

Database Settings: - Health check

Structured Three-Tier Containerized Architecture



Workflow Diagram

```
User Browser (Port 80)
Desktop / Tablet / Mobile Access
        |
        | HTTP Requests
        v
Frontend Container (construction-frontend)
Nginx Alpine + React SPA

• Nginx serves static files (HTML/CSS/JS)
• React pages: Dashboard, Projects, Workers,
  Materials, Reports, Landing
• Reverse proxy: /api/* → backend:5000
• Image: ~30MB | Health: 30s intervals
        |
        | API Calls (/api/*)
        v
Backend Container (construction-backend)
Node.js 18 Alpine + Express 4.18.2

• RESTful API: GET/POST/PUT/DELETE
• Routes: /api/projects, /api/workers,
  /api/materials, /health
• Mongoose Models: Project, Worker, Material
• Port: 5000 (internal), 5001 (external)
• Image: ~150MB | Health: 30s intervals
        |
        | MongoDB Connection
        | mongodb://mongo:27017
        v
Database Container (construction-mongo)
MongoDB 7 Official

• Database: construction
• Collections: projects, workers, materials
• Persistent Volume: mongo-data (/data/db)
• Port: 27017 (internal only)
• Health: 10s intervals, 5 retries

═══════════════════════════════════════
Docker Network: construction-network (Bridge)
• Service discovery via DNS
• Network isolation & security
═══════════════════════════════════════
```

2.3 Tools & Technologies Used

Component	Technologies
Frontend	React, React Router DOM, Axios, Chart.js, Css
Backend	Node.js, Express, Mongoose, CORS, Dotenv
Database	MongoDB
DevOps & Infrastructure	Docker, Docker Compose, AWS EC2, Amazon Linux 2023, GitHub Actions, Git & GitHub

4. Results & Output

4.1 Screenshots / Outputs

1. **Docker Compose Deployment Execution**

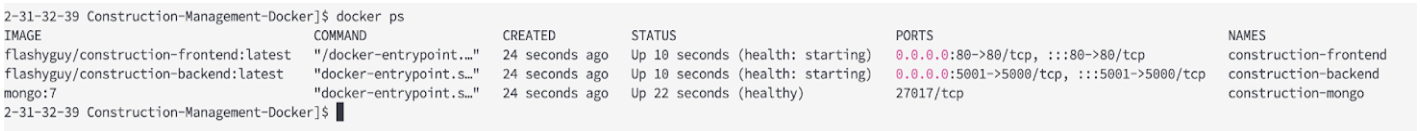
Shows successful execution of **`docker-compose up -d`**.
The network and all three containers (frontend, backend, mongo) are created, and MongoDB is marked as **healthy**, confirming proper service initialization.

2. **Running Containers Status (`docker ps`)**

Displays active containers with correct port mappings:

- Frontend → Port 80
- Backend → Port 5001
- MongoDB → Internal port 27017

Confirms all services are running successfully.



3. Application Accessibility Test (cURL Output)

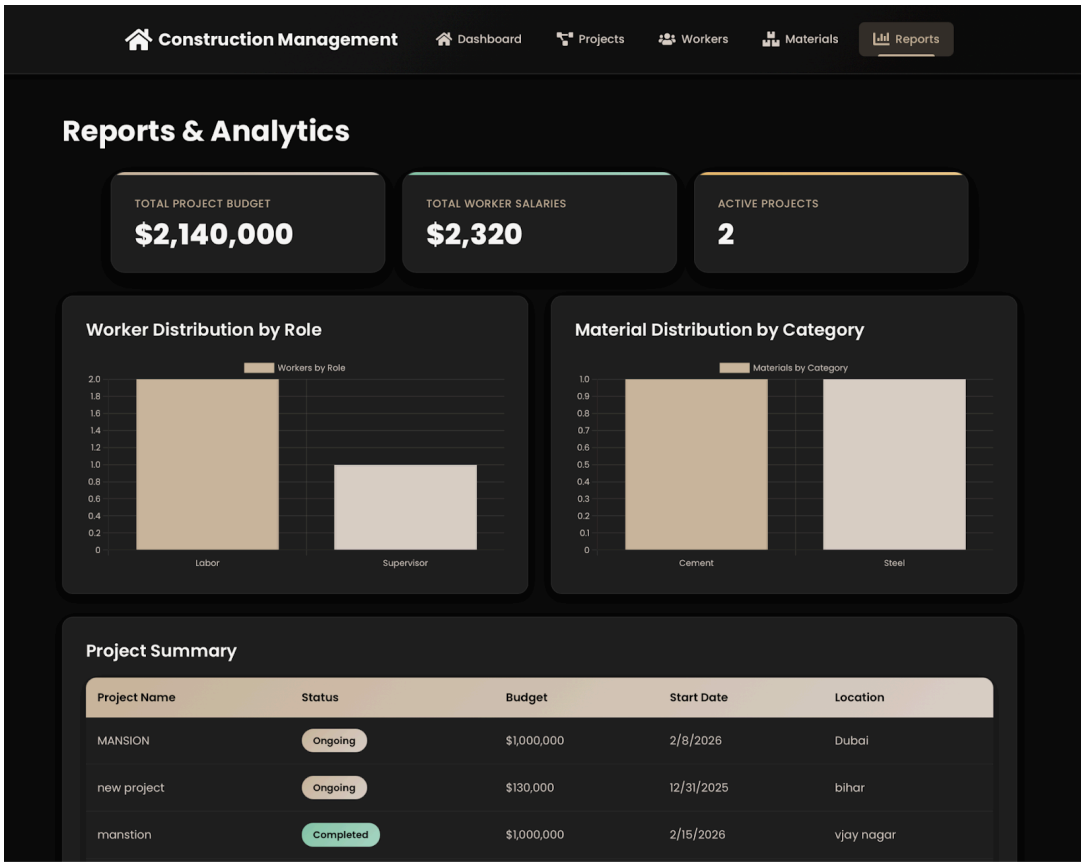
Shows `HTTP/1.1 200 OK` response from the public IP.
Confirms Nginx is serving the React application correctly and the deployment on AWS EC2 is working as expected.

```
[ec2-user@ip-172-31-32-39 Construction-Management-Docker]$ curl -v http://13.233.238.214
*   Trying 13.233.238.214:80...
* Connected to 13.233.238.214 (13.233.238.214) port 80
* using HTTP/1.x
> GET / HTTP/1.1
> Host: 13.233.238.214
> User-Agent: curl/8.15.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx/1.29.5
< Date: Thu, 19 Feb 2026 17:41:44 GMT
< Content-Type: text/html
< Content-Length: 776
< Last-Modified: Sun, 15 Feb 2026 15:39:57 GMT
< Connection: keep-alive
< ETag: "6991e8cd-308"
< X-Frame-Options: SAMEORIGIN
< X-Content-Type-Options: nosniff
< X-XSS-Protection: 1; mode=block
< Accept-Ranges: bytes
<
* Connection #0 to host 13.233.238.214 left intact
```

4. Application User Interface

Dashboard

Reports



4.2 Reports

Project Management Reports

The Project Portfolio Summary report displays active and completed projects with status, budget, and timeline details using color-coded badges (blue for planning, yellow for ongoing, green for completed, red for delayed).

Key Metrics:

- Total projects across four status categories
- Budget variance (planned vs actual)
- Timeline performance (on-time vs delayed)

Data Model:

```
Project Schema:
- name (String, required)
- description (String)
- budget (Number)
- startDate (Date)
- endDate (Date)
- status (Enum: Planning, Ongoing, Completed, Delayed)
- createdAt (Timestamp)
- updatedAt (Timestamp)
```

Worker Analytics Reports

The Worker Distribution by Role report visualizes workforce composition across seven roles.The Project-wise Worker Allocation report identifies understaffed and overstaffed projects.The Salary Analysis Report calculates payroll obligations and average salaries by role.

Key Insights:

- Workforce utilization percentage
- Skill gap identification
- Payroll cost analysis by project

Data Model:

```
Worker Schema:
- name (String, required)
- role (Enum, required)
- contactNumber (String)
- salary (Number)
- status (Enum: Active, Inactive, On Leave)
- projectId (Reference to Project)
- joinDate (Date)
- createdAt (Timestamp)
```

Inventory Analytics Reports

The Material Distribution by Category report visualizes inventory levels across nine material categories.The Low-Stock Items Report identifies materials below threshold with procurement details.The Inventory Valuation Report calculates total inventory investment.

Critical Metrics:

- Total inventory value
- Number of low-stock items
- Inventory turnover trends

Data Model:

```
Material Schema:
- name (String, required)
- category (Enum, required)
- quantity (Number, required)
- unit (Enum, required)
- pricePerUnit (Number)
- supplier (String)
- threshold (Number, default: 10)
- isLowStock (Virtual field: quantity < threshold)
- createdAt (Timestamp)
```

Financial Analytics Reports

The Total Budget Overview aggregates financial commitments across projects.The Cost Analysis Report tracks budget vs actual spending to detect variances.The Material Cost and Labor Cost Reports provide cost breakdowns by project and role.

Financial Highlights:

- Budget utilization percentage
- Cost variance identification
- Material and labor cost breakdown

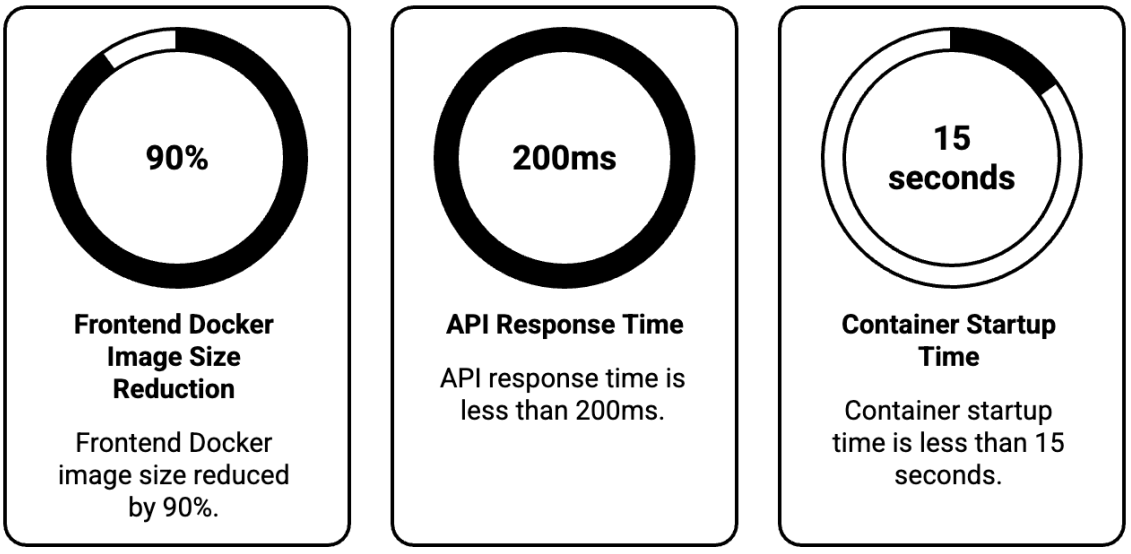
4.3 Key Outcomes

Technical Achievements:

- 90% reduction in frontend Docker image size
- Backend optimized to ~150MB
- API response time < 200ms
- Container startup < 15 seconds
- Automated health monitoring
- Successful AWS cloud deployment

Operational Benefits:

- Real-time visibility into projects and inventory
- Automated alerts preventing delays
- Improved workforce allocation
- Centralized reporting and analytics



5. Conclusion

The Construction Management System successfully demonstrates the integration of containerization, cloud deployment, and full-stack development to address real-world construction challenges. By implementing Docker multi-stage builds, secure networking, automated health checks, and AWS deployment, the system achieves scalability, reliability, and performance efficiency.

The project strengthened skills in MERN development, Docker optimization, AWS infrastructure management, RESTful API design, database modeling, and DevOps automation. It delivers both technical excellence and measurable business value.

---

6. Future Scope & Enhancements

- JWT authentication and role-based access control
- WebSocket-based real-time updates
- Kubernetes orchestration
- CI/CD pipeline automation
- Advanced monitoring (Prometheus & Grafana)
- Redis caching layer