Project - High Level Design

on

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 Containerized Construction Management System is a cloud-native web application built using the MERN stack (MongoDB, Express, React, Node.js) and containerized with Docker. It provides centralized project tracking, workforce management, material inventory control, and analytics through a scalable three-tier architecture deployed on AWS EC2.

The system addresses critical challenges in the construction industry including manual processes, data silos, lack of real-time visibility, and inconsistent environments across development and production. By leveraging containerization and modern DevOps practices, the application ensures consistent behavior across all environments while maintaining security, performance, and ease of deployment.

1.1 Scope of the Document

This document defines the complete technical architecture of the system including:

- **Frontend (React SPA)** – Component structure, routing, state management, and UI design
- **Backend (Node.js/Express REST API)** – Route handlers, business logic, and middleware
- **MongoDB database design** – Collections, schemas, relationships, and indexing
- **Docker containerization strategy** – Multi-stage builds, networking, and orchestration
- **AWS EC2 deployment setup** – Infrastructure configuration and deployment procedures
- **API contracts and integration flow** – Endpoint specifications and data formats
- **Security and performance requirements** – Best practices and optimization strategies

It serves as the technical reference for development, deployment, testing, and maintenance activities throughout the project lifecycle.

1.2 Intended Audience

Development Team – Implements and maintains frontend, backend, and database components. Uses this document to understand architecture, API contracts, and data models.

DevOps Engineers – Deploy and manage Docker containers on AWS EC2. References deployment procedures, health checks, and infrastructure configuration.

QA Engineers – Design test cases and validate system behavior. Uses API specifications and process flows to create comprehensive test scenarios.

Project Managers – Plan sprints and allocate resources. Reviews system overview and component design to understand project scope and dependencies.

System Architects & Reviewers – Evaluate design and scalability. Assesses architectural decisions, security measures, and performance considerations.

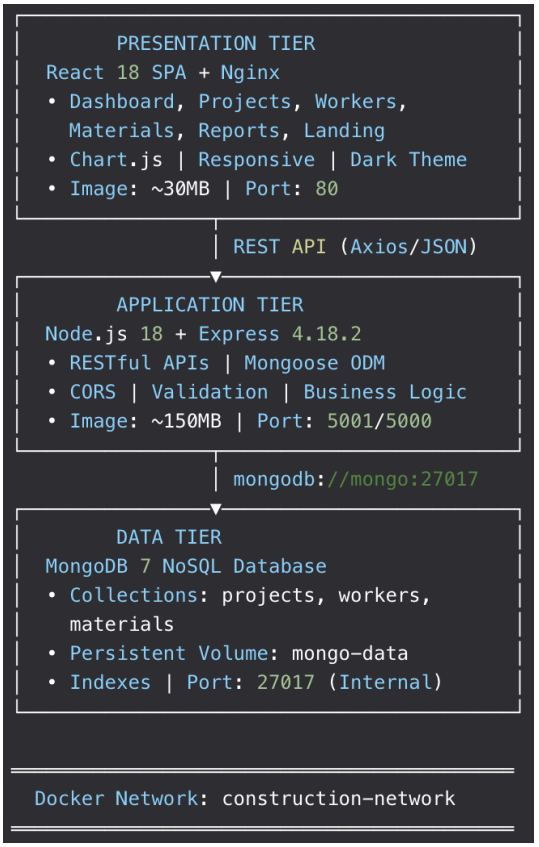
1.3 System Overview

The system follows a three-tier architecture that separates presentation, application logic, and data persistence into independent layers:

- **Presentation Tier** – React SPA served via Nginx, providing responsive UI accessible from desktop, tablet, and mobile devices.
- **Application Tier** – Node.js/Express REST API handling business logic, validation, and database operations.
- **Data Tier** – MongoDB database storing projects, workers, and materials with persistent volume storage.

All components are containerized using Docker and orchestrated via Docker Compose, ensuring consistent deployment across environments.

System Architecture Diagram



Key Capabilities

- **Project lifecycle tracking** – Manage projects from planning through completion with status monitoring
- **Workforce management** – Role-based classification across seven worker types with project assignment
- **Inventory tracking** – Automated low-stock alerts with configurable thresholds per material
- **Dashboard analytics** – Real-time visualizations using Chart.js for project status distribution

- **Reports and financial insights** – Comprehensive analytics for workforce, inventory, and budgets

Deployment target: AWS EC2 (t3.micro instance, Amazon Linux 2023, Public IP: 15.206.173.12)

---

2. System Design

2.1 Application Design

Presentation Tier (Frontend)

Built using React 18 with a component-based architecture. React Router manages navigation, Axios handles API calls, and Chart.js provides visualizations.

Responsive UI (desktop, tablet, mobile) with dark theme.

Key Technologies: React, React Router DOM, Axios, Chart.js, Bootstrap.

Multi-stage Docker build reduces final image size to ~30MB using Nginx to serve static files.

Application Tier (Backend)

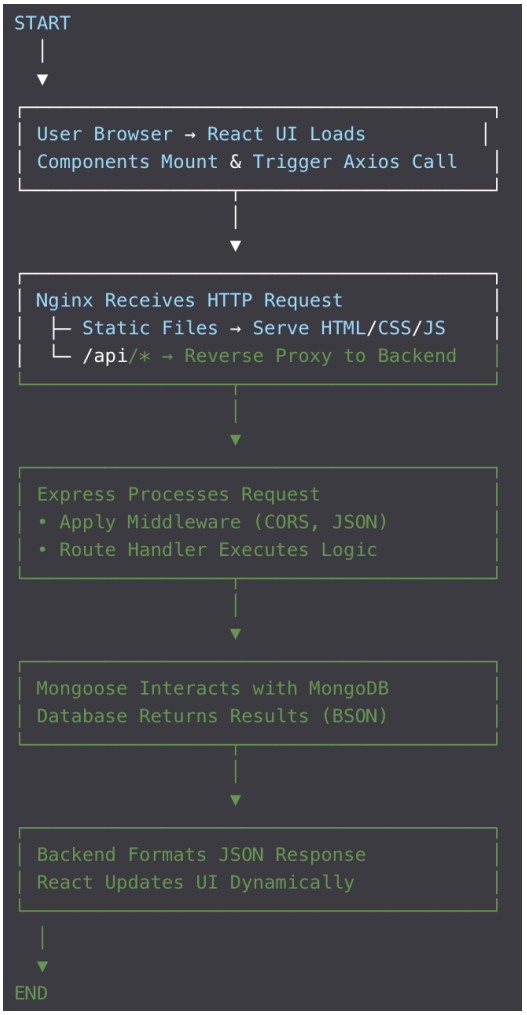
Built using Node.js 18 and Express. Provides RESTful APIs for CRUD operations. Mongoose manages schema validation and database interaction.

Key Technologies: Node.js, Express, Mongoose, CORS, Dotenv.
Optimized Docker image (~150MB) using production-only dependencies.

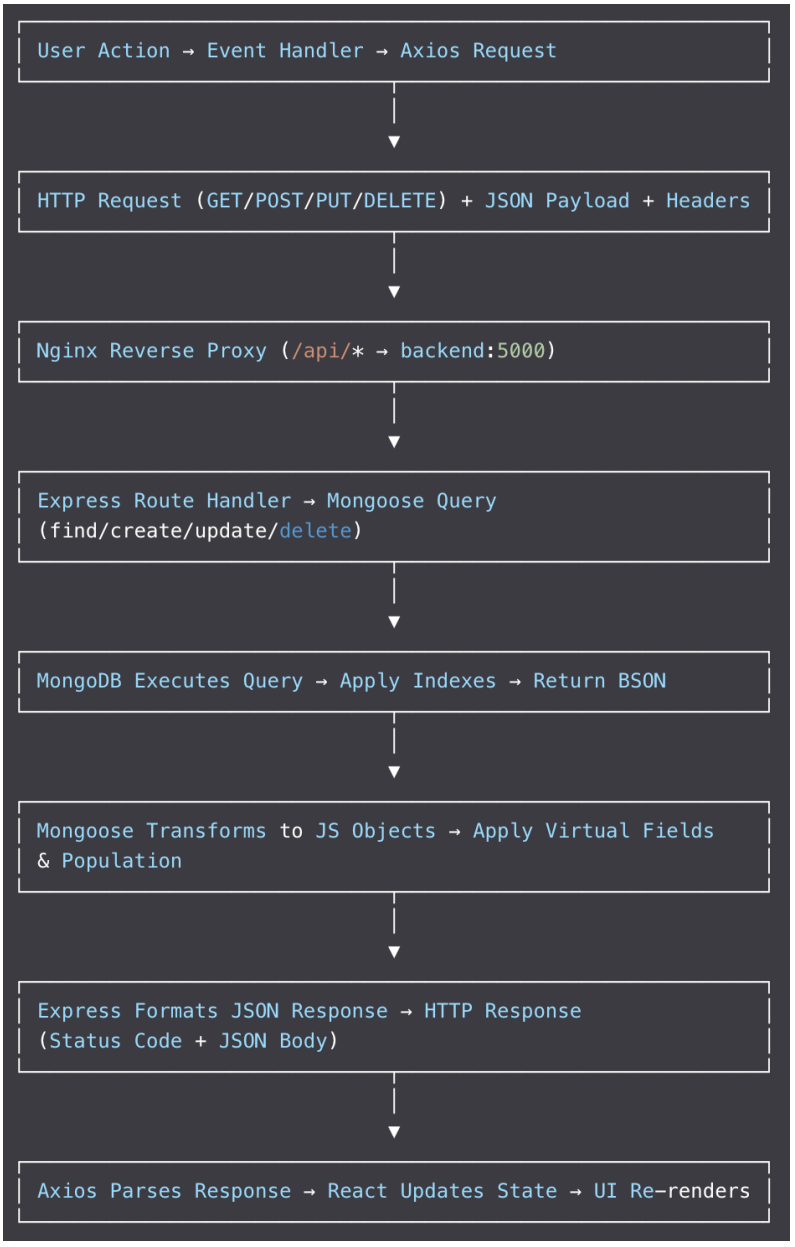
Data Tier (Database)
MongoDB 7 stores projects, workers, and materials.

- Persistent Docker volume ensures durability.
- Indexed fields improve query performance.

2.2 Process Flow



2.3 Information Flow



2.4 Components Design

```
├── 🐳 docker-compose.yml                # Local development
├── 🐳 docker-compose.prod.yml           # Production deployment
│
├── 📁 backend/
│   ├── 📁 models/
│   │   ├── Project.js                   # Project schema
│   │   ├── Worker.js                    # Worker schema
│   │   └── Material.js                  # Material schema
│   ├── 📁 routes/
│   │   ├── projects.js                  # Project API routes
│   │   ├── workers.js                   # Worker API routes
│   │   └── materials.js                 # Material API routes
│   ├── server.js                        # Express server
│   ├── package.json                     # Dependencies
│   ├── 🐳 Dockerfile                    # Multi-stage build
│   ├── 📄 DOCKERFILE-EXPLAINED.md       # Docker documentation
│   ├── .dockerignore                    # Docker ignore
│   └── .env.example                     # Environment template
│
├── 📁 frontend/
│   ├── 📁 public/
│   │   └── index.html                   # HTML template
│   ├── 📁 src/
│   │   ├── 📁 components/
│   │   │   └── Navigation.js            # Navigation bar
│   │   ├── 📁 pages/
│   │   │   ├── Dashboard.js             # Dashboard with charts
│   │   │   ├── Projects.js              # Project management
│   │   │   ├── Workers.js               # Worker management
│   │   │   ├── Materials.js             # Material tracking
│   │   │   └── Reports.js               # Analytics
│   │   ├── 📁 api/
│   │   │   └── axios.js                 # API configuration
│   │   ├── App.js                       # Main component
│   │   ├── index.js                     # Entry point
│   │   └── index.css                    # Global styles
│   ├── package.json                     # Dependencies
│   ├── 🐳 Dockerfile                    # Multi-stage build
│   ├── 📄 DOCKERFILE-EXPLAINED.md       # Docker documentation
│   ├── nginx.conf                       # Nginx configuration
│   ├── .dockerignore                    # Docker ignore
│   └── .env.example                     # Environment template
│
└── 📁 .github/
    └── 📁 workflows/
        └── deploy.yml                   # GitHub Actions CI/CD
```

2.5 Key Design Considerations

- **Multi-stage Docker builds** reduce image size and improve deployment efficiency.
- **Three-tier architecture** enables independent scaling and fault isolation.
- **MongoDB** offers flexible schema and horizontal scalability.
- **RESTful APIs** ensure consistent, predictable communication.
- **Health checks** allow automatic container recovery.
- **Non-root users** enhance container security.
- **Indexing** improves database query performance.

2.6. API Catalogue

Projects

- `GET /api/projects` – List
- `GET /api/projects/:id` – Get by ID
- `POST` – Create
- `PUT` – Update
- `DELETE` – Remove
- `GET /stats/summary` – Project statistics

Workers

- `GET` – List
- `POST` – Create
- `PUT` – Update
- `DELETE` – Remove

Materials

- `GET` – List
- `POST` – Create
- `PUT` – Update
- `DELETE` – Remove

Health Check

- `GET /health` – System status

Flow: Client → Nginx → Express → MongoDB → JSON

Status Codes: 200, 201, 400, 404, 500

3. Data Design

3.1 Data Model

Three collections store application data with Mongoose schemas enforcing structure and validation:

Projects Collection

- _id (ObjectId, auto-generated)
- name (String, required, 3-100 chars)
- location (String, required)
- status (Enum: Planning, Ongoing, Completed, Delayed)
- startDate (Date, required)
- endDate (Date, optional)
- budget (Number, required, min 1000)
- description (String, optional)
- createdAt, updatedAt (auto-generated timestamps)

Workers Collection

- _id (ObjectId, auto-generated)
- name (String, required, 2-50 chars)
- role (Enum: Engineer, Supervisor, Labor, Electrician, Plumber, Carpenter, Mason)
- phone (String, required, validated format)
- email (String, optional, validated format)
- assignedProject (ObjectId, ref 'Project')
- salary (Number, required, min 1000)
- status (Enum: Active, Inactive, On Leave)
- createdAt, updatedAt (auto-generated)

Materials Collection

- _id (ObjectId, auto-generated)
- name (String, required, 2-50 chars)
- category (Enum: Cement, Steel, Bricks, Sand, Wood, Paint, Electrical, Plumbing, Other)
- quantity (Number, required, min 0)
- unit (Enum: kg, ton, pieces, bags, liters, meters, sqft)
- pricePerUnit (Number, required, min 0.01)
- supplier (String, optional)
- lowStockThreshold (Number, default 10)
- isLowStock (Virtual field: quantity <= threshold)
- createdAt, updatedAt (auto-generated)

3.2 Data Access Mechanism

Mongoose ODM provides abstraction between application code and MongoDB:

- Schema validation – Enforces required fields, data types, enum constraints
- Type casting – Automatically converts compatible values to correct types
- Query building – Chainable methods for complex queries
- Population – Resolves ObjectId references by fetching related documents
- Virtual fields – Computed properties not stored in database

Query Patterns:

- find() – Retrieve multiple documents with optional filters
- findById() – Retrieve single document by ObjectId
- create() – Insert new document after validation
- findByIdAndUpdate() – Atomically update document
- findByIdAndDelete() – Remove document

Example: Worker.find().populate('assignedProject') fetches workers with complete project details.

3.3 Data Retention Policies

Proposed retention policies:

- Completed projects – Retain 7 years for compliance and historical analysis
- Inactive workers – Retain 3 years after termination for reference checks
- Inventory records – Retain 3 fiscal years for financial audits

Implementation: Use scheduled cleanup jobs or MongoDB TTL indexes for automatic deletion after specified time periods.

3.4 Data Migration

Flexible schema allows adding fields without migrations – new fields appear as undefined in existing documents.

Field renaming requires migration script to copy data from old field to new field, then remove old field.

Type changes require transformation script to read each document, transform field value, validate, and update.

Migrations must be:

- Idempotent (can run multiple times safely)

- Tested on development/staging before production
- Include rollback procedures

---

4. Interfaces

Web-based React SPA – Primary user interface accessible via modern browsers (Chrome, Firefox, Safari, Edge) on desktop, tablet, and mobile devices**.**

RESTful JSON API – Programmatic access for external systems, mobile apps, or custom tools. Accepts HTTP requests with JSON payloads and returns JSON responses.

Future integration with accounting software (QuickBooks, Xero), HR systems, procurement systems, and email services for automated notifications.

---

5. State and Session Management

Current Implementation

Stateless backend – Each API request contains all necessary information without server-side session storage, enabling horizontal scaling.

React local component state – Each component manages its own state using useState and useEffect hooks. No global state management (Redux/Context) currently implemented.

No authentication yet – System operates without user authentication in current version.

Future Enhancement

JWT-based authentication – After login, backend generates JWT token containing user info and expiration. Frontend stores token and includes in Authorization header of subsequent requests.

Token validation middleware – Backend validates JWT signature, checks expiration, and extracts user information on each request.

Role-based access control – Restrict operations based on user roles (Admin, Manager, Worker).

6. Caching

Current Implementation

Browser caching – Static assets (HTML, CSS, JavaScript) cached with cache-control headers. React build generates hashed filenames enabling aggressive caching.

MongoDB internal caching – WiredTiger cache stores frequently accessed documents and indexes in RAM (default 50% of available RAM minus 1GB).

Indexed queries – Database indexes serve as caching mechanism, storing frequently accessed data structures in memory for rapid query execution**.**

Future Enhancement

Redis caching – Application-level cache for frequently accessed data**:**

- Dashboard statistics cached for 5-10 minutes
- Project/worker/material lists cached with 1-2 minute TTL
- Cache invalidation on data changes

Caching strategies:

- Write-through: Update both cache and database
- Cache-aside: Load from database on cache miss
- Time-based expiration (TTL)
- Event-based invalidation

---

7. Non-Functional Requirements

11.1 Security Aspects

Non-root Docker users – All containers run as non-root users (nginx-user, nodejs) limiting potential damage from breaches.

Network isolation – Custom Docker bridge network isolates containers. Only frontend exposes port 80 publicly; backend and database remain internal.

Input validation – Mongoose schemas validate all data before database insertion, preventing invalid data and injection attacks**.**

Environment variables – Sensitive configuration (database URLs, API keys) stored in .env files excluded from version control.

7.2 Performance Aspects

API target response < 200ms – Simple CRUD operations complete in under 200ms. Complex aggregations target under 500ms.

Optimized Docker images – Multi-stage builds reduce frontend to 30MB and backend to 150MB, improving deployment speed.

Indexed database queries – Indexes on status, role, and category fields accelerate queries from full scans to targeted lookups.

Stateless backend for scaling – No session affinity required, enabling horizontal scaling with load balancers.

---

8. References

Technologies and Frameworks:

React Documentation: https://react.dev/

Node.js Documentation: https://nodejs.org/docs/

Express Framework: https://expressjs.com/

MongoDB Manual: https://www.mongodb.com/docs/

Mongoose ODM: https://mongoosejs.com/docs/

Docker Documentation: https://docs.docker.com/

Chart.js Documentation: https://www.chartjs.org/docs/

Axios Documentation: https://axios-http.com/docs/

Cloud and Infrastructure:

AWS EC2 Documentation: https://docs.aws.amazon.com/ec2/

Amazon Linux 2023: https://docs.aws.amazon.com/linux/

Nginx Documentation: https://nginx.org/en/docs/