Project - High Level Design

on

Dockerized Sports Rust System

Course Name: Devops

Institution Name: Medicaps University – Datagami Skill Based Course

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

Sr no	Student Name	Enrolment Number
1	ABHAY SINGH THAKUR	EN22CS304003
2	SARTHAK GOKHALE	EN22CS304057
3	HIMANSHI PORWAL	EN22CS306021
4	AKASH MUVEL	EN23EE3L1001
5	PRATHAM JOSHI	EN22CS306037

Group Name: ***Group 04D11***

*Project Number:****DO-30***

Industry Mentor Name:

University Mentor Name:

*Academic Year:****2022-2026***

1.Introduction.

The project **"Containerize a Sports Service (built with MERN Stack) using Docker best practices"** focuses on designing and deploying a containerized sports analytics service using modern DevOps and containerization technologies. The system provides sports data through REST APIs and a web-based frontend interface while ensuring portability, scalability, and consistent runtime environments.

The application consists of a **Node.js (Express) backend sports service**, a React (Vite) frontend interface, and a MongoDB database. All components are packaged as Docker containers and orchestrated using Docker Compose to enable seamless multi-service deployment. The use of multi-stage Docker builds reduces image size and improves performance, making the application suitable for both development and production environments.

This High-Level Design (HLD) document describes the overall architecture, major components, workflows, and deployment structure of the containerized sports service system at a conceptual level without implementation-specific details.

1.1 Scope of the Document

This document presents the high-level architectural design of the project **"Containerize a Sports Service (built with MERN Stack) using Docker best practices."** It describes how the Node.js backend service, React frontend, and MongoDB database are structured, integrated, and deployed using Docker and Docker Compose.

The scope includes system architecture, component interactions, containerization strategy, API exposure, data flow, and deployment workflow. It explains how Docker ensures consistent environments across systems and how the services communicate within a containerized network.

Detailed coding logic, internal Node.js implementation, and configuration-level details are outside the scope of this document and are covered in the Low-Level Design (LLD).

1.2 Intended Audience

This High-Level Design document is intended for:

- Academic evaluators and project reviewers
- University and industry mentors
- Developers and DevOps engineers working on the project

• Team members responsible for deployment and maintenance
• Stakeholders requiring architectural understanding of the system

1.3 System Overview

The system developed in this project is a containerized sports service platform that delivers sports information through a web interface and REST APIs. The architecture follows a three-tier structure consisting of a frontend layer, backend service layer, and data storage layer.

Users interact with the application through a React (Vite) frontend interface. The frontend sends HTTP API requests to the **Node.js (Express) backend sports service**. The backend processes the requests, retrieves sports data from the MongoDB database, and returns structured JSON responses to the frontend for display.

All components run inside Docker containers and are orchestrated using Docker Compose. Docker Compose manages service networking, environment variables, container startup sequence, and inter-service communication. This ensures that the application runs consistently across different development and production environments.

The system supports static sports data, database-driven sports data, and a health-check API endpoint. Automatic database seeding is performed during container initialization to ensure data availability. The containerized architecture makes the system portable, reproducible, and ready for future deployment on cloud platforms or container orchestration environments such as Kubernetes.

2. System Design

The system design of the project **"Containerize a Sports Service (built with MERN Stack) using Docker best practices"** follows a containerized three-tier architecture consisting of a presentation layer, application layer, and data layer.

Each layer is implemented as an independent Docker container and orchestrated using Docker Compose to ensure seamless communication and deployment.

• Presentation Layer → React (Vite) frontend container
• Application Layer → Node.js (Express) backend container
• Data Layer → MongoDB database container

Docker Compose manages service configuration, container networking, environment variables, and startup dependencies. This architecture ensures modularity,

portability, scalability, and consistent execution across development and production environments.

2.1 Application Design

The application is designed as a client–server system where the frontend acts as the client and the Node.js backend service acts as the server. The frontend provides user interaction and sends HTTP requests to backend API endpoints. The backend validates requests, executes business logic, and retrieves or stores data in MongoDB.

The backend exposes RESTful endpoints such as health checks and sports data retrieval APIs. The system supports both static sports data embedded in the service and dynamic sports data stored in MongoDB. Automatic database seeding ensures that initial sports data is available when the containers start.

The frontend displays retrieved sports data in a structured format, allowing users to view information fetched from both static sources and the database.

2.4 Components Design

The system consists of the following major components:

Frontend Component (React + Vite)
Provides the graphical user interface and interacts with backend APIs. It runs in a Docker container and communicates with the backend through HTTP requests.

Backend Component (Node.js + Express Sports Service)
Implements REST APIs, handles business logic, processes requests, and communicates with MongoDB using Mongoose ODM. It is containerized using a Docker build for optimized deployment.

Database Component (MongoDB)
Stores sports-related data such as teams, matches, and statistics in document collections. It runs as a separate Docker container and is accessible only within the Docker network.

Containerization Component (Docker)
Packages each service into isolated containers, ensuring consistent runtime environments and portability across systems.

Orchestration Component (Docker Compose)
Manages multi-container deployment, service networking, environment variables, and startup dependencies between frontend, backend, and database services.

DevOps Component (GitHub & Docker Hub)

Source code is maintained in GitHub, and container images are stored and distributed via Docker Hub for deployment.

2.6 API Catalogue

The Node.js (Express) backend exposes the following REST APIs:

GET /health
Returns the health status of the backend service. Used to verify that the service is running correctly inside the container.

GET /sports/static
Returns predefined static sports data stored within the backend service. Useful for testing API functionality without database dependency.

GET /sports/db
Retrieves sports data stored in MongoDB collections using Mongoose. Demonstrates backend–database integration and dynamic data retrieval.

3.2 Data Access Mechanism

The Node.js backend service accesses MongoDB using **Mongoose ODM**, configured through environment variables defined in Docker Compose. The backend establishes a secure connection to the MongoDB container over the internal Docker network.

All data operations are performed by the backend service, including retrieval of sports records and reading seeded data. The backend converts database documents into structured JSON responses before sending them to the frontend through REST APIs.

Direct database access from the frontend is not permitted, ensuring proper separation of concerns and improved security.

4.3 Database Interface

The database interface enables communication between the **Node.js backend service** and the MongoDB database container. The backend connects to MongoDB using Mongoose and connection parameters defined through Docker environment variables.

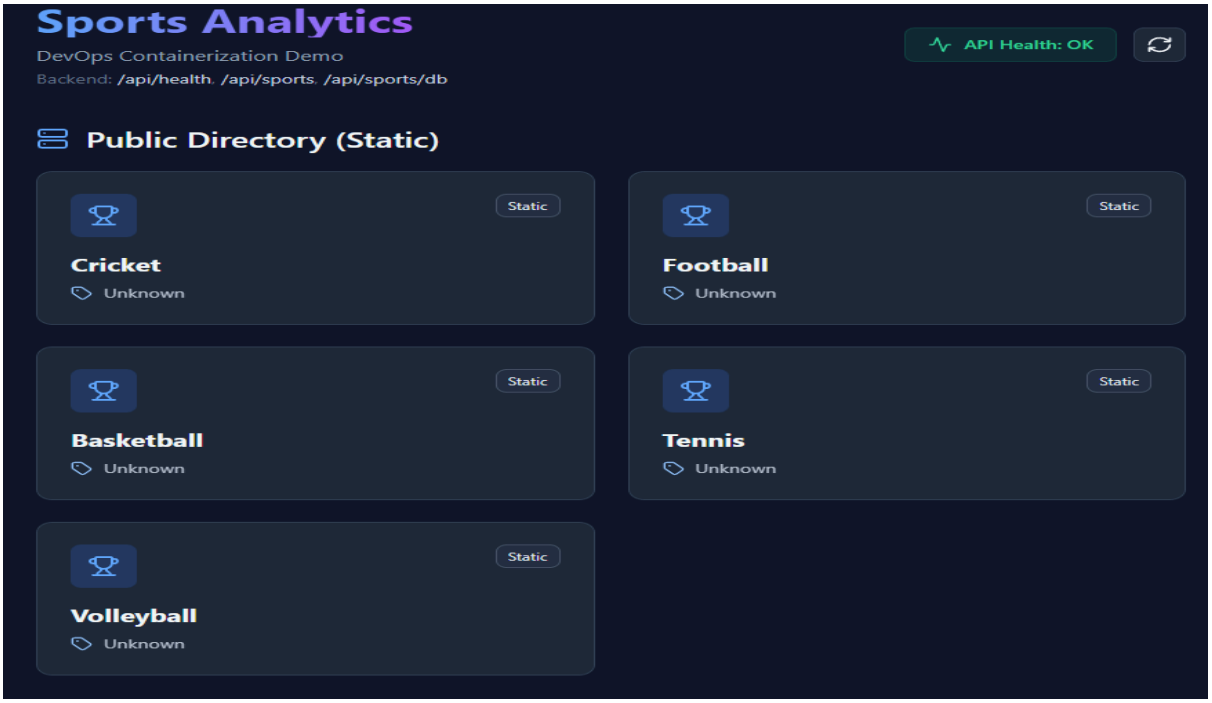
All database operations, including reading seeded sports data, are performed by the backend service. The database is not directly exposed to the frontend or external clients, ensuring secure and controlled access.

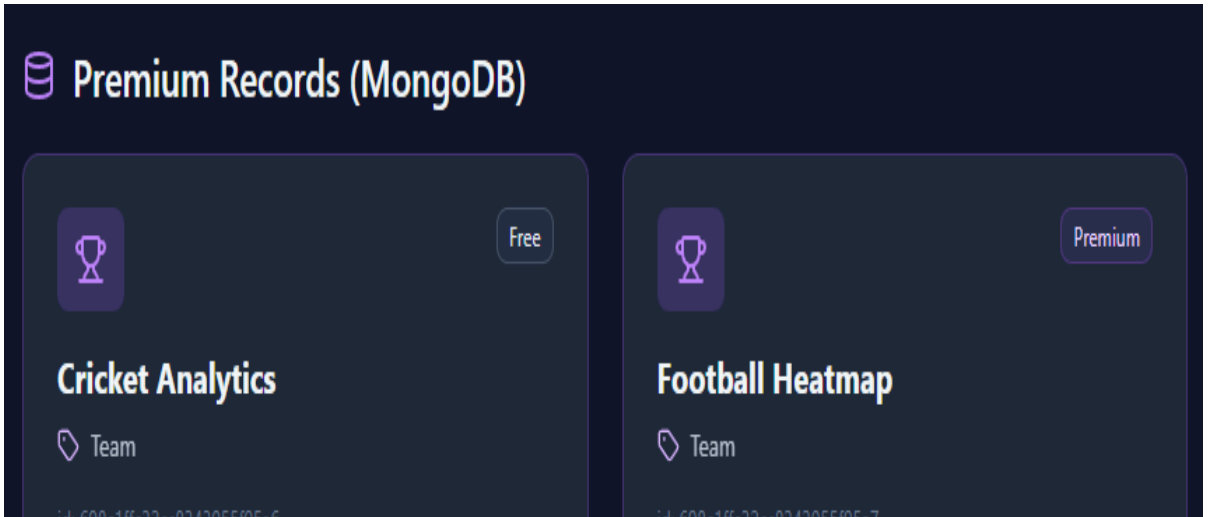
5.1 Backend State Management

The Node.js backend service is designed to be stateless. It does not store user session data or request context in memory across multiple requests. Every API call contains all the information required for processing, and the backend retrieves necessary sports data from

MongoDB as needed.
This stateless design allows multiple backend containers to run concurrently in future scalable deployments without requiring session synchronization.

1. *Reports / dashboards / models*





5.2 Application Data State

Persistent application state, such as sports data (teams, matches, statistics), is stored in the MongoDB database container. MongoDB acts as the single source of truth for dynamic sports data used by the Node.js backend service.

When the system starts, MongoDB is automatically seeded with predefined sports data to ensure that the application has valid data available. Since the data resides in the database rather than the backend container, application state remains consistent even if the backend container is rebuilt or restarted.

5.3 Frontend State Management

The React frontend manages temporary UI state within the browser session. This includes rendered sports data, loading states, and API response handling. The frontend does not store persistent user data or sessions and relies on backend APIs for retrieving sports information.

Frontend state exists only during the user's browser interaction and is refreshed whenever new API requests are made to the backend.

5.4 Session Handling

The current system does not implement user authentication or login functionality; therefore, no user session management is required. All users access the same publicly available sports data through stateless API requests handled by the Node.js backend service.

This simplified session model aligns with the project scope, which focuses on containerization and service integration rather than user identity management. Future enhancements may introduce authentication mechanisms such as JWT tokens and role-based access control if user-specific features are added.

5.5 Scalability Considerations

Because the Node.js backend is stateless and persistent data is stored in MongoDB, the system is inherently scalable. Multiple backend containers can be deployed behind a load balancer in future cloud or Kubernetes environments without session conflicts.

This approach supports horizontal scaling and distributed deployment while maintaining consistent application behavior.

6. Caching

The current implementation of the system does not include a dedicated caching layer. The Node.js backend service retrieves sports data directly from MongoDB or from static data embedded within the service. Since the project primarily focuses on containerization, service orchestration, and API integration, caching optimization is not required at this stage.

However, the system architecture allows the future integration of caching mechanisms to improve performance and reduce database load. A caching layer such as Redis or in-memory caching within the backend service can be introduced to store frequently accessed sports data, such as team lists or match summaries. This would reduce repeated database queries and improve response time.

Because the backend service is stateless and containerized, any future caching solution should use an external cache store (e.g., Redis container) rather than in-container memory to ensure consistency across multiple backend instances in scalable deployments.

7. Non-Functional Requirements

The system **"Containerize a Sports Service (built with MERN Stack) using Docker best practices"** is designed to satisfy key non-functional requirements related to portability, scalability, reliability, maintainability, and usability. These requirements ensure that the containerized sports service operates consistently across environments and can be extended for future deployment scenarios.

Portability
Docker containers ensure the application can run consistently across different operating systems and machines without dependency conflicts.

Scalability
The stateless Node.js backend and containerized architecture allow horizontal scaling in future cloud or Kubernetes deployments.

Reliability
Docker Compose manages service dependencies and networking, ensuring that frontend, backend, and database services start and communicate correctly.

Maintainability
The separation of frontend, backend, and database into independent containers simplifies updates, debugging, and component replacement.

Usability
The React-based web interface provides a simple and accessible way for users to view

Deployability
Docker builds and Docker Compose configuration enable quick and repeatable deployment across development and production environments.

7.1 Security Aspects

Security in the system is primarily achieved through container isolation and controlled service communication.

Each component (frontend, backend, MongoDB) runs in a separate Docker container, preventing direct interference between services. MongoDB is not exposed publicly and is

accessible only through the internal Docker network by the Node.js backend service. This prevents unauthorized direct database access from external clients.

Sensitive configuration parameters such as database connection strings and ports are managed using environment variables defined in Docker Compose rather than hardcoded values. This improves security and deployment flexibility.

Since the current system does not include user authentication, no user credential storage or session security is required. Future enhancements may introduce authentication mechanisms such as JWT tokens and role-based access control for secure user-specific access.

7.2 Performance Aspects

The system performance is optimized through containerization and efficient backend–database interaction.

Docker containerization ensures consistent runtime environments and predictable performance across machines. Lightweight Node.js backend containers enable fast startup and low resource consumption. Containerized services also eliminate environment inconsistencies that can degrade performance.

The backend retrieves sports data using MongoDB queries via Mongoose, which are efficient for document-based retrieval. Static sports data endpoints further reduce database dependency for simple requests.

Because the backend is stateless, multiple instances can be deployed in parallel in future scalable environments, improving throughput and availability. The architecture also supports future performance enhancements such as caching layers or load balancing.

8. References

The design and implementation of the system are based on the following technologies and documentation sources:

- Docker official documentation
- Docker Compose documentation
- Node.js documentation
- Express.js documentation
- MongoDB documentation
- Mongoose ODM documentation
- React and Vite documentation
- REST API design best practices
- DevOps and containerization best practices

9. Detailed Architecture Description

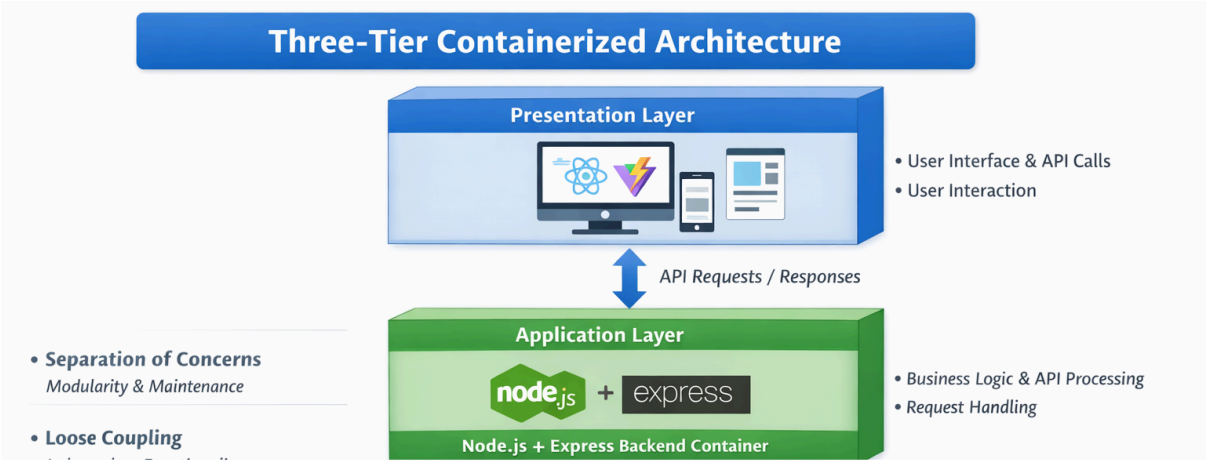
9.1 Three-Tier Architecture Deep Dive

The system follows a three-tier containerized architecture:

- Presentation Layer – Built using React with Vite
- Application Layer – Built using Node.js with Express
- Data Layer – Implemented using MongoDB

Explain:

- Why three-tier architecture?
- Benefits of separation of concerns
- Loose coupling between layers
- Independent scaling of layers



10. Deployment Architecture

The deployment architecture of the system is based on containerization principles using Docker and multi-container orchestration through Docker Compose. The application components — frontend, backend, and database — are packaged into independent Docker images and deployed as isolated containers within a shared virtual network.

This approach ensures consistency across development, testing, and production environments while maintaining modularity and scalability.

The deployment architecture includes:

- Docker Images
- Docker Containers
- Docker Networking
- Volume Mapping
- Port Mapping
- Multi-Stage Docker Builds

Each of these elements plays a critical role in ensuring reliable and efficient deployment.

10.1 Container Architecture

The container architecture defines how the application services are packaged, networked, and executed within Docker.

10.1.1 Docker Images

A Docker image is a lightweight, standalone, and executable package that includes everything needed to run an application: source code, runtime, system libraries, dependencies, and configuration files.

In this project:

-
- The frontend image contains the React application built using Vite.
- The backend image contains the Node.js and Express application.

- The MongoDB image is based on the official MongoDB image from Docker Hub.

Each image is built using a Dockerfile, which defines the steps required to create the runtime environment.

Key characteristics of Docker images:

- Immutable once built
- Version-controlled
- Reproducible across systems
- Portable across environments

Images are stored locally or pushed to container registries such as Docker Hub for distribution.

10.1.2 Docker Containers

A Docker container is a running instance of a Docker image. While images are static blueprints, containers are dynamic runtime environments.

In this system:

- One container runs the frontend service.
- One container runs the backend service.
- One container runs the MongoDB database.

Each container runs in isolation but communicates with other containers via a Docker-managed network.

Benefits of using containers:

- Process isolation
- Lightweight virtualization
- Fast startup time
- Resource efficiency
- Environment consistency

If a container crashes, it can be restarted without affecting other services.

10.1.3 Docker Network

Docker Compose automatically creates a bridge network that enables inter-container communication.

In this architecture:

- The backend container connects to the MongoDB container using the MongoDB service name as hostname.
- The frontend container connects to the backend container via internal networking.

This internal network ensures:

- Secure service communication
- No need to expose database ports publicly
- Automatic DNS-based service discovery
- Logical service separation

The database container is accessible only within the internal Docker network and is not exposed to external clients, improving security.

10.1.4 Volume Mapping

Volume mapping enables persistent data storage outside the container lifecycle.

In this system:

- MongoDB uses Docker volumes to store database files.
- Data remains intact even if the
- MongoDB container is removed or rebuilt.

Benefits of volume mapping:

- Data persistence
- Backup and restore capability
- Separation of application logic and storage
- Protection against data loss during container restarts

This ensures that sports data remains consistent across deployments.

10.1.5 Port Mapping

Port mapping connects container ports to host machine ports.

Example:

- Frontend container (Port 5173) → Host Port 5173
- Backend container (Port 5000) → Host Port 5000

This allows:

- Users to access the frontend via browser
- External API testing through tools like Postman
- Controlled exposure of services

MongoDB is typically not exposed to the host machine to prevent unauthorized direct access.

Role of Docker Compose

Docker Compose manages multi-container deployment using a YAML configuration file.

Docker Compose handles:

- Service definitions
- Environment variables
- Port bindings
- Volume configuration
- Network setup
- Service dependency order

With a single command, all services can be built and started together, ensuring consistent and repeatable deployment.

10.2 Multi-Stage Docker Build

To optimize image size and security, the project uses a multi-stage Docker build strategy.Multi-stage builds allow separating the build environment from the production runtime environment.

Why Multi-Stage Builds?

- Smaller final image size
- Reduced attack surface

- Faster container startup
- Improved performance
- Cleaner production environment

10.2.1 Builder Stage

The builder stage is responsible for:

- Installing dependencies
- Compiling source code
- Generating production-ready build files

For example:

Stage 1: Install dependencies

- Use Node.js base image
- Copy package.json
- Run npm install
- Build the React frontend

This stage may contain development tools that are not required in production.

10.2.2 Production Stage

The production stage creates a lightweight runtime image.

Stage 2: Copy production build only

- Use a minimal Node.js base image
- Copy compiled build files from builder stage
- Install only production dependencies
- Expose necessary ports
- Start the application

By copying only the final build artifacts and required runtime files, unnecessary dependencies and build tools are excluded.

Smaller Image Size

Multi-stage builds significantly reduce the size of the final Docker image.

Benefits:

- Faster image pull time
- Reduced storage usage
- Improved deployment speed
- Better performance in cloud environments

Smaller images are especially beneficial when deploying to container orchestration platforms.

Security Benefits

Multi-stage builds improve security by:

- Removing development dependencies
- Reducing unnecessary binaries
- Minimizing attack surface
- Eliminating debugging tools from production

Since only essential runtime components are included in the final image, the system becomes more secure and production-ready.

Deployment Workflow Summary

The complete deployment process follows these steps:

1. Developer writes code.
2. Docker images are built using Dockerfiles.
3. Docker Compose builds and starts containers.
4. Containers communicate via internal Docker network.
5. MongoDB stores persistent data using volumes.
6. Frontend and backend are exposed via mapped ports.
7. System becomes accessible to users.

This deployment architecture ensures portability, reproducibility, modularity, and scalability across environments