# RePlan Design Document

**Developer:** Copoeru Matei **Version** 2

## **1. System Overview**

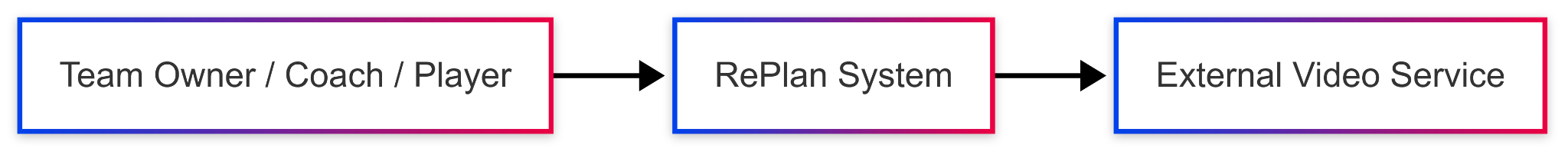
**RePlan** is a web-based platform designed to address the challenges of team organization and video review in the amateur eSports scene. The application enables team owners to create and manage multiple teams, annotate embedded videos with timestamped notes, schedule practices, and facilitate real-time collaboration among team members. RePlan aims to streamline the review process and improve team communication through an intuitive, interactive dashboard.

## **2. Architecture Overview**

RePlan is architected as a modular and scalable application using modern web technologies. Its design follows the C4 model and Clean Architecture principles to ensure maintainability, testability, and separation of concerns.

### **2.1 C1 Diagram (System Context)**

The system context diagram shows how various actors interact with RePlan:

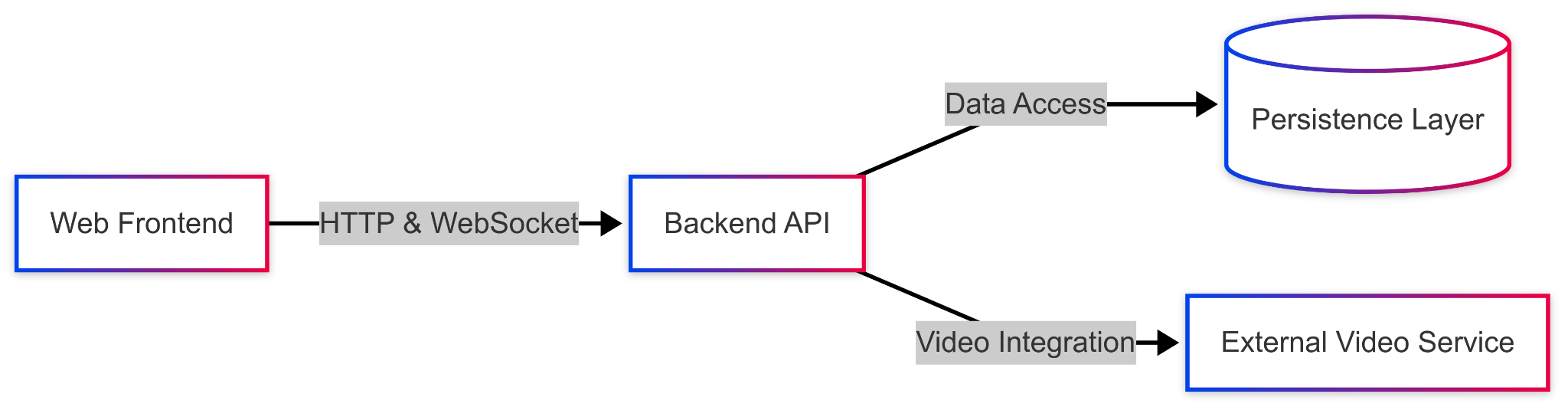


* **Actors:**
  + **Team Owner:** Creates and manages teams.
  + **Team Member (Coach, Player, Other Owner):** Participates in team activities.
  + **Viewer:** Reviews annotated videos.
* **External Systems:**
  + **YouTube:** Provides video embedding capabilities.
  + **Optional Notification Services:** For real-time alerts.

*Explanation:* RePlan sits between its users and external services. It receives events (e.g., video annotations, team actions) and communicates with YouTube to embed videos, ensuring a seamless user experience.

### **2.2 C2 Diagram (Container Diagram)**

RePlan is divided into three primary containers:

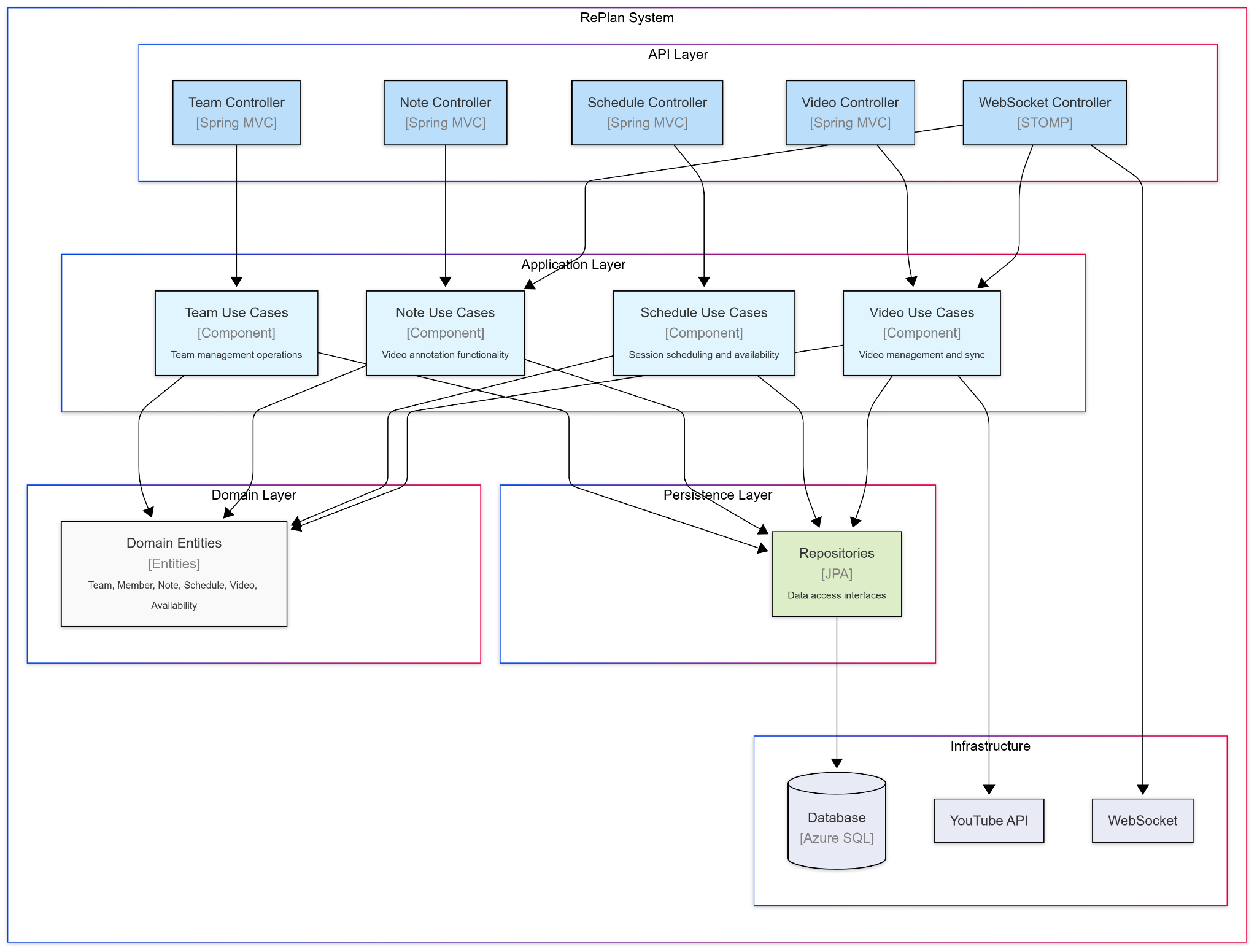


* **Frontend (React, Shadcn, Tailwind CSS):**
  + Responsible for rendering the user interface, handling user interactions, and consuming backend APIs.
* **Backend (Spring Boot with Clean Architecture):**
  + Exposes REST and WebSocket endpoints to manage team creation, member management, event processing, and scheduling.
  + Organized into layers (Domain, Use Cases, Adapters, and Frameworks/Drivers) to enforce SOLID principles.
* **Database:**
  + Initially implemented as an in-memory store (with plans for JPA/MySQL integration) for persisting teams, members, and event data.
* **External Services:**
  + **YouTube:** Embedded videos.

*Explanation:* The separation into distinct containers allows us to isolate concerns, making it easier to scale and modify each part independently. The front end communicates with the backend over HTTP/REST and WebSockets for real-time updates.

### **2.3 C3 Diagram (Component Diagram)**

The component diagram breaks down the internal structure of the RePlan backend, showing how the system is organized following Clean Architecture principles:



The backend is organized into four distinct layers:

* **API Layer:**
  + **Team Controller:** Handles team creation, joining, and management requests
  + **Note Controller:** Processes video annotation operations
  + **Schedule Controller:** Manages practice scheduling functionality
  + **Video Controller:** Handles video embedding and metadata
  + **WebSocket Controller:** Enables real-time collaboration and updates
* **Application Layer:**
  + **Team Use Cases:** Implements business logic for team management operations
  + **Note Use Cases:** Manages video annotation functionality and timestamps
  + **Schedule Use Cases:** Coordinates scheduling and availability tracking
  + **Video Use Cases:** Handles video management and synchronisation
* **Domain Layer:**
  + Contains the core entities that represent the system's business concepts (Team, Member, Note, Schedule, Video, Availability)
  + Defines the business rules independent of any external concerns
* **Persistence Layer:**
  + Provides data access interfaces through repositories
  + Separates business logic from data storage implementation details

*Explanation:* This architecture ensures a clean separation of concerns, with dependencies pointing inward toward the domain layer. Controllers depend on use cases, which in turn depend on domain entities. The system connects to external infrastructure like the database, YouTube API, and WebSocket services without compromising the core business logic. This approach makes the system more maintainable, testable, and adaptable to changing requirements.

The component diagram visually demonstrates how the various parts of RePlan work together to support the key features: video annotation, multi-user collaboration, team management, and practice scheduling.

### **2.4 SOLID**

#### S- Single Responsibility Principle

Each class has one responsibility:

* Domain Classes (Team, User) represent data structures
* Use case implementations (CreateTeamImpl, LoginUserImpl) handle specific operations
* Controllers (TeamController, UserController) manage API endpoints
* Repositories handle data access for specific entities

#### O - Open/Closed Principle

The code is open for extension without modification through:

* Business logic accessed via interfaces (CreateTeamUseCase, LoginUserUseCase)
* Repository interfaces that can have multiple implementations
* Controllers that depend on interfaces, not concrete implementations

#### L - Liskov Substitution Principle

Implementations can be substituted for their interfaces:

* Mock repositories in tests (TeamControllerTest, CreateTeamImplTest)
* Different authentication strategies can replace JwtAuthenticationFilter
* Test configurations show this in practice (TeamTestConfig, UserTestConfig)

#### I - Interface Segregation Principle

Interfaces are focused and client-specific:

* Each use case has its own interface with a single method (e.g., CreateTeamUseCase)
* Repositories have specific methods for their domain (TeamRepository.findByOwnerId)
* Request/response objects are operation-specific (CreateTeamRequest/Response)

#### D - Dependency Inversion Principle

High-level modules depend on abstractions:

* Controllers depend on use case interfaces, not implementations
* Use cases depend on repository interfaces, not database implementations
* Dependency injection used throughout (CreateTeamImpl constructor)
* External concerns (JWT, database) are isolated behind interfaces

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

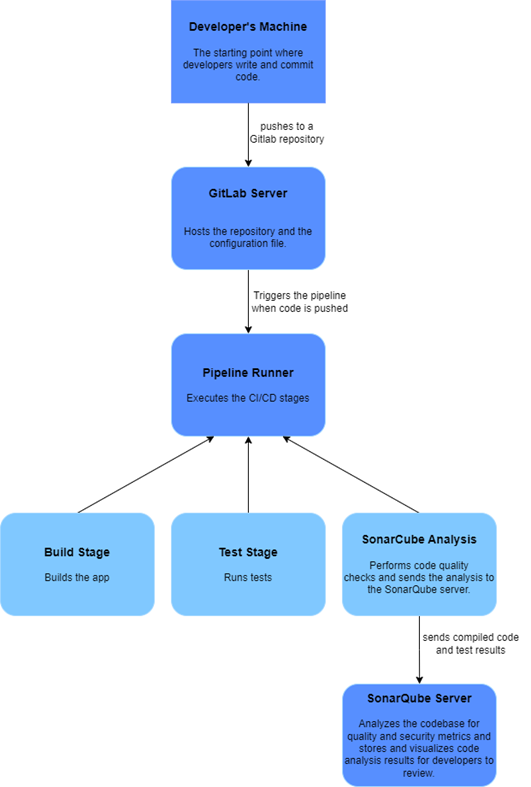
### 

### 

### 

### **2.5 CI Pipeline Diagram**

The CI pipeline automates build, test, and deployment processes:



* **Source Code Management:** Git repository for both frontend and backend.
* **Build & Test:** Gradle (for backend) and npm/yarn (for frontend) run automated tests.
* **Containerization:** Dockerfiles are used to package the backend for deployment.
* **Deployment:** Continuous Delivery pipelines (e.g., via GitHub Actions or Jenkins) deploy to staging/production environments.

*Explanation:* An automated CI/CD pipeline ensures that changes are tested rigorously and deployed reliably, supporting continuous integration and delivery practices.

## **3. Design Decisions**

### **3.1 Frontend: React with Shadcn, Tailwind CSS, and Axios**

* **Why React?** React's component-based architecture and efficient state management are ideal for creating dynamic and interactive UIs.
* **Why Tailwind & Shadcn?** These tools offer a rapid, customizable styling approach that helps maintain consistency and responsiveness.
* **Why Axios?** Axios simplifies HTTP communication, enabling easy integration with our backend services.

### **3.2 Backend: Spring Boot with Clean Architecture**

* **Why Spring Boot?** Spring Boot accelerates development with auto-configuration and robust ecosystem support. It is ideal for building secure, scalable backend services.
* **Clean Architecture:** Using Clean Architecture, the business logic is isolated from the framework and infrastructure. This ensures maintainability, testability, and adherence to SOLID principles.
* **Real-Time Capabilities:** Integration with WebSocket (or STOMP) supports real-time updates, crucial for event processing and note synchronisation.

### 

### 

### **3.3 Database: JPA/MySQL**

* **Why MySQL?:** The architecture is designed to allow a seamless switch to a persistent store like MySQL with JPA when needed.

### **3.4 Containerization and CI/CD**

* **Why Docker?** Docker ensures a consistent environment across development, testing, and production.
* **CI/CD Pipeline:** Automating builds and deployments reduces manual errors and accelerates the delivery cycle.

## **4. Architecture Constraints**

### **4.1 Performance Constraints**

* **Real-Time Updates:** The system must support low-latency real-time data propagation (e.g., for annotations and notifications).
* **Transaction Throughput:** The system should handle up to 100 transactions per second and support a growing user base.

### **4.2 Scalability Constraints**

* **Horizontal Scaling:** The architecture must support horizontal scaling to accommodate increased user load.
* **Modularity:** New features should be integrable without significant rework, thanks to the modular structure.

### **4.3 Security Constraints**

* **Data Protection:** Implement robust authentication and authorization (using Spring Security, JWT, etc.), and ensure compliance with data protection regulations.
* **CORS Configuration:** Proper CORS settings are in place to secure cross-origin communications from the frontend.

### **4.4 Technology Constraints**

* **Open-Source:** The project leverages open-source technologies to remain cost-effective.
* **Technology Compatibility:** Chosen technologies must integrate smoothly (e.g., React with Spring Boot) to ensure a cohesive solution.

## **5. Component Design**

### **5.1 Team Management Module**

* **Team Creation & Member Management:** Implements use cases like CreateTeamUseCase and AddTeamMemberUseCase to handle team creation and membership.
* **UI Components:** Frontend components such as CreateTeamForm and AddTeamMemberForm interact with the backend via dedicated API services.

### **5.2 Video Annotation & Real-Time Notes**

* **Annotation Service:** Processes and stores timestamped notes against video content, ensuring synchronization across multiple users.
* **WebSocket Integration:** Supports real-time updates to deliver immediate feedback to users.

### **5.3 Scheduling Module**

* **Practice Scheduling:** Enables team owners and coaches to schedule practices, with conflict detection and availability tracking.

### **5.4 Continuous Delivery & DevOps**

* **Docker Integration:** Dockerfiles for backend (and eventually frontend) support continuous deployment.
* **Automated Testing:** Comprehensive unit, integration, and end-to-end tests ensure high quality and reliability.

## **6. Future Work and Applied Research**

* **Applied Research Document:** Ongoing research on the most optimal ways to schedule practice sessions
* **Further Integration:**Develop the VoD watching feature and the schedule feature.