

# **PHENIKAA UNIVERSITY - SCHOOL OF COMPUTING**



**Group Submission Software Architecture**

**Project Title: Kanban Board Application**

**Group Submission: N02 – Group 2**

<b>Name</b>	<b>Student ID</b>	<b>Position Title</b>
Trần Thành Long	23010070	Code Backend – Frontend – Review Final

**Class: CSE703110-1-2-25(N02)**

**Instructor in charge : Vũ Quang Dũng**

**Hà Nội - Feb 7, 2026**

# Full-Stack Implementation & Engineering

**Project:** Kyte - Kanban Board Application

**Course:** Software Architecture

**Student:** Trần Thành Long (23010070)

**Role:** Full-stack Developer

**Class:** N02 - Group 2

**Date:** Feb 07, 2026

## 1. Executive Summary

As the Full-stack Developer for the Kyte project, I was responsible for the end-to-end implementation of the entire system. My primary focus was on transforming the architectural designs into a high-performance, real-time collaborative platform. I developed the complete backend infrastructure using Bun and Elysia, engineered the database schema with Drizzle ORM, and built the interactive React 19 frontend. A major part of my contribution was the implementation of the core domain logic including real-time event orchestration and fractional indexing algorithms and the authoring of a robust suite of over 130 automated tests to ensure system reliability and data integrity.

## 2. Technical Contribution Scope

My implementation work covered the full depth of the Kyte stack:

- **Backend Infrastructure:** Developed a modular monolith API server using ElysiaJS and Bun runtime.
- **Frontend Engineering:** Built a reactive, type-safe SPA using React 19, TanStack Router, and TanStack Query.
- **Real-Time Orchestration:** Implemented the system-wide Event Bus and WebSocket Bridge for instant synchronization.
- **Algorithmic Implementation:** Engineered the "Fractional Indexing" system and "Optimistic Concurrency Control" logic.
- **Automated Verification:** Authored the complete testing framework, including unit, integration, and reliability test suites.

## 3. Core System Implementation

### 3.1 Modular Monolith Architecture

I implemented a strict modular structure in `packages/server/src/modules/`. Each feature (Tasks, Boards, Workspaces, Members, etc.) is fully encapsulated with its own Controller, Service, and Repository layers. This design ensures that cross-module communication is explicitly handled through public interfaces, preventing "spaghetti" dependencies and simplifying future scaling.

### 3.2 Real-Time Event-Driven Infrastructure

I built the foundation for Kyte's collaborative features:

- **Internal Event Bus:** Developed a decoupled emitter system that allows Services to publish "Domain Events" without knowing who consumes them.
- **WebSocket Bridge:** Implemented a real-time bridge that subscribes to the Event Bus and broadcasts relevant changes to connected clients in specific "Board Rooms," eliminating the need for expensive HTTP polling.

### 3.3 Core Domain Algorithms

- **Fractional Indexing:** I implemented a robust ordering system using string-based keys. This allows for  $O(1)$  insertions between any two tasks without re-ranking the entire column, a critical feature for a smooth drag-and-drop experience.
- **Idempotency Layer:** I developed a specialized middleware that uses an `Idempotency-Key` header to detect and handle duplicate requests, ensuring that network retries never lead to duplicate data creation.

## 4. Frontend Engineering & Type-Safety

### 4.1 End-to-End Type Safety

I leveraged **Eden Treaty** to export the server's type definitions directly to the React frontend. This architectural decision ensures that any change in the backend API (e.g., changing a Task property) results in an immediate TypeScript error in the web client, drastically reducing integration bugs.

## 4.2 Reactive UI & Optimistic Patterns

I implemented the "Optimistic UI" pattern using React 19's `useOptimistic` hook. This provides users with sub-100ms perceived latency for card movements, as the UI updates instantly while the server synchronization happens in the background. I also engineered the complex drag-and-drop context handlers.

# 5. Testing & Verification Implementation

## 5.1 Multi-Tiered Testing Strategy

I implemented a comprehensive testing suite using **Bun Test** to verify every architectural layer:

- **Unit Tests:** Verified the mathematical correctness of the position generation and search query parsing logic.
- **Integration Tests:** Used a "No-Mocking" approach with a live PostgreSQL database to verify the full Controller -> Service -> Repository -> Database flow.
- **Reliability Tests:** Authored specific suites (e.g., `concurrency.test.ts`) to simulate race conditions and prove that our version-based optimistic locking works as intended.

## 5.2 Test Infrastructure Tooling

I developed the `test-helpers.ts` utility, which automates the lifecycle of integration tests, including atomic database resets, authenticated session generation, and polling helpers for verifying asynchronous side effects.

# 6. Lessons Learned & Reflections

- **The Power of Type Inference:** Using Drizzle and Elysia taught me how much developer velocity can be gained when the type system works for you across the entire stack.
- **Decoupling side-effects:** Implementing an Event Bus was a game-changer for maintainability; adding features like Audit Logs or Activity feeds became trivial because they were decoupled from core logic.
- **Performance Trade-offs:** I learned that while real-time features add complexity, using a high-performance runtime like Bun makes the overhead manageable even for small teams.

## 7. Future Engineering Goals

- **Performance Optimization:** Introduce Redis caching for frequently accessed board metadata.
- **Batching Support:** Enhance the API to support bulk task operations with atomic transactions.
- **Advanced PostgreSQL Integration:** Implement Row-Level Security (RLS) policies to provide an additional layer of security directly at the data tier.