PHENIKAA UNIVERSITY - SCHOOL OF COMPUTING

Group Submission Software Architecture

Project Title: Kanban Board Application

Group Submission: N02 – Group 2

Name	Student ID	Position Title
Trần Thành Long	23010070	Code Backend – Frontend – Review Final
Nguyễn Xuân Mạnh	23010045	Document + Diagram & Quality Assurance

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

Instructor in charge : Vũ Quang Dũng

Hà Nội - Feb 7, 2026

Mục Lục

Software Architecture Final Report

1. Cover Page & Info

- **Project Title:** Kanban Board Application
- **Course Name:** Software Architecture
- **Group Submission:** N02 - Group 2
- **Class:** N02 - Group 2
- **Date:** Feb 07, 2026

2. Executive Summary

Kyte is a real-time Kanban board for small teams. The system is built as a modular monolith with strict module boundaries, using Bun and Elysia on the server and React on the client. The architecture focuses on low-latency drag-and-drop, secure access control, and reliable data persistence. Real-time synchronization is handled by WebSocket broadcasts driven by domain events. Using the most popular relational-database Postgresql.

3. Requirements Analysis

3.1 Core Functional Requirements

ID	Description
FR-01	Authenticate users with GitHub OAuth and email/password using Better Auth.
FR-02	Create and manage workspaces and workspace members.
FR-03	Create boards and manage columns within boards.
FR-04	Create, update, and archive tasks with labels, checklists, attachments, and due dates.
FR-05	Move tasks between columns with drag-and-drop ordering.
FR-06	Real-time updates broadcast changes to all connected users in the board.

3.2 Key Quality Attributes (NFR)

- **Performance:** Interaction latency under 100ms for drag-and-drop using Optimistic UI.
- **Scalability:** Stateless API nodes with sessions stored in PostgreSQL via Better Auth.
- **Security:** Board and workspace membership checks enforced in services.
- **Data Integrity:** ACID compliance for all mutations (PostgreSQL).
- **Maintainability:** Modular Monolith structure with strict controller/service/repository separation.

3.3 Architecturally Significant Requirements (ASR)

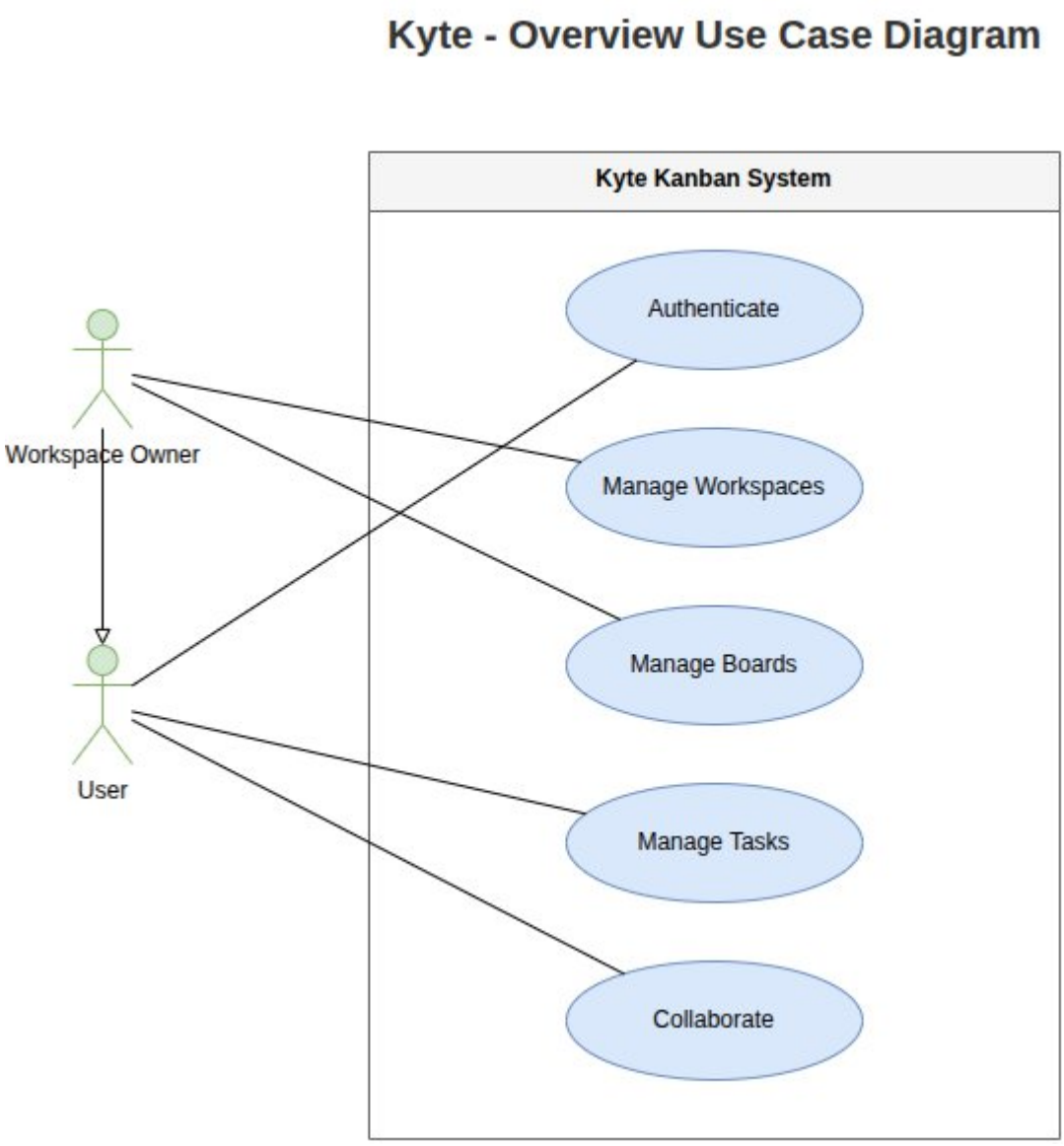
ASR	Statement	Architectural Impact
ASR-1	The system must keep business logic independent from the UI to allow future clients.	Enforces layered design with clean module boundaries.
ASR-2	All write operations must validate board and workspace access on the server.	Requires auth plugin and service-level permission checks.
ASR-3	Task ordering must remain correct under concurrent updates.	Uses fractional indexing and optimistic locking version fields.

3.4 Use Case Modeling

The system's functional requirements are modeled through a set of use case diagrams, including a high-level overview and detailed module-specific views.

3.4.1 Overview Use Case Diagram

The overview diagram illustrates the primary high-level interactions between different user roles and the core modules of the Kyte Kanban System.

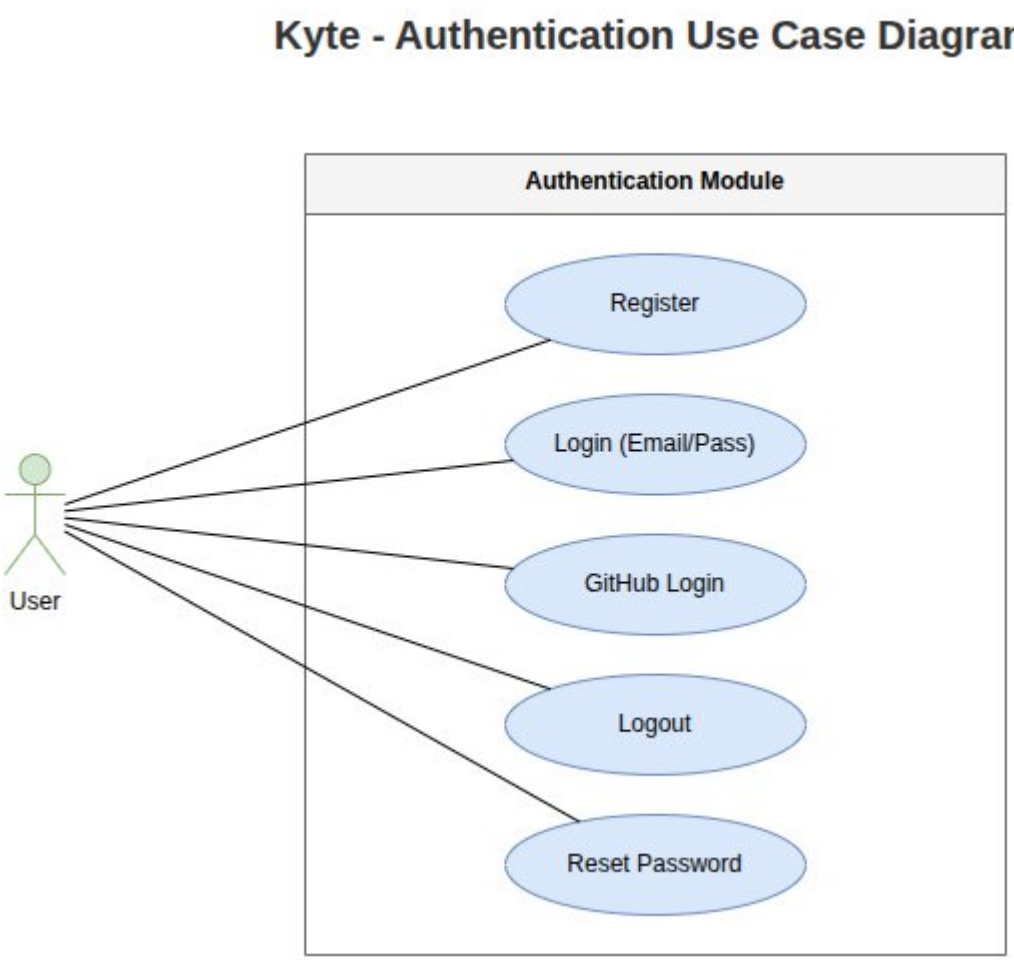


3.4.2 Detailed Use Case Diagrams

A. Authentication & User Management

Handles user onboarding, identity verification, and secure access. All actions are performed by the **User** actor.

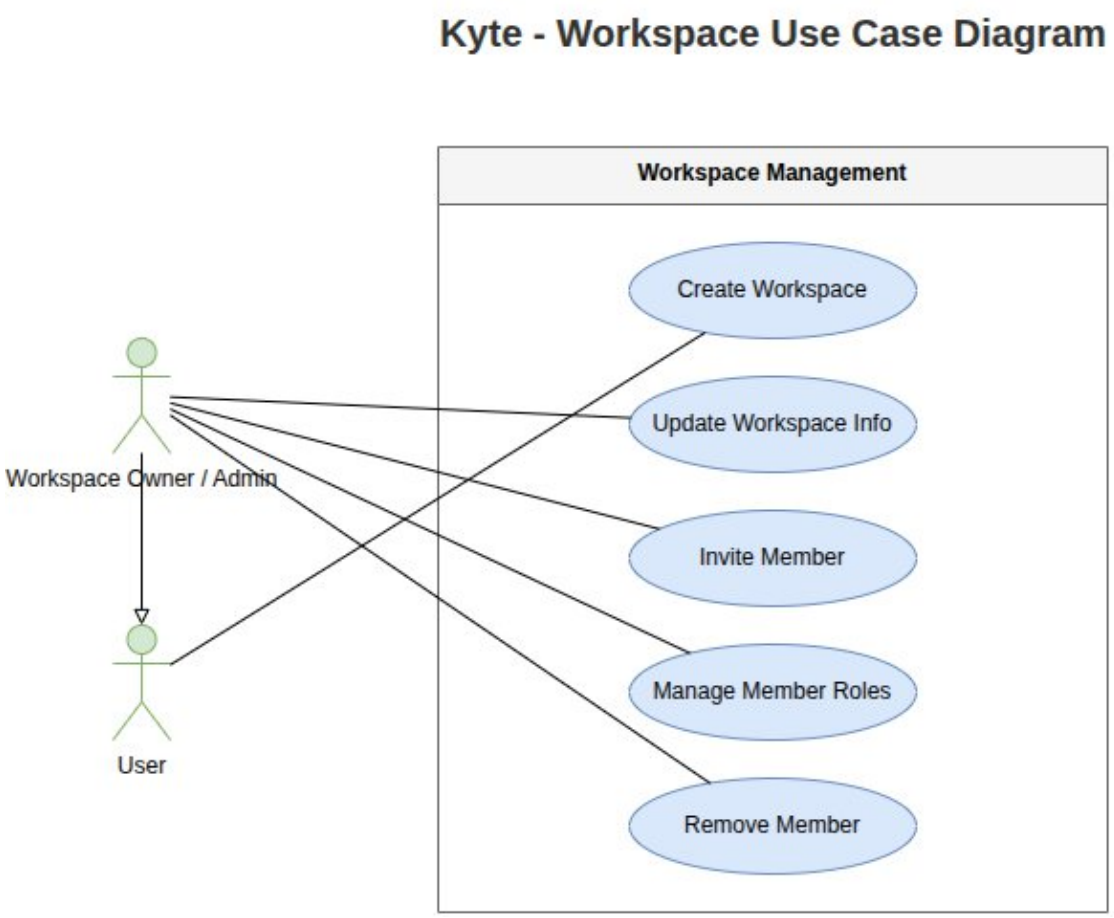
- **Register:** Create a new account via email and password.
- **Login:** Authenticate via traditional credentials or GitHub OAuth integration.
- **Password Management:** Securely reset passwords and manage active sessions.



B. Workspace & Member Management

Manages the multi-tenant organizational structure of the system.

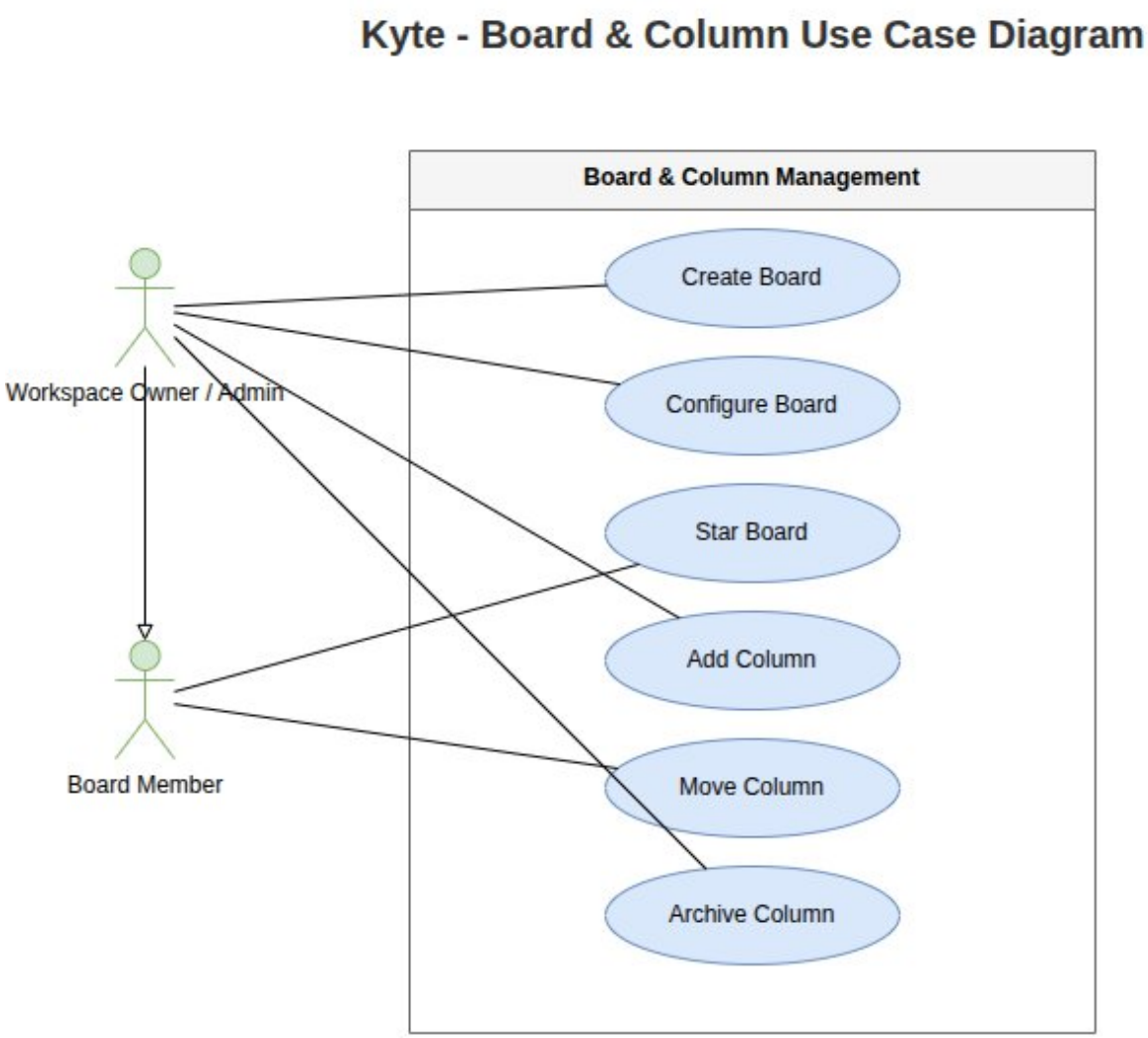
- **User:** Can perform the initial **Create Workspace** action to become an Owner.
- **Workspace Owner / Admin:** Manages the lifecycle of the workspace, including updating metadata, inviting new members, assigning roles (Member/Viewer), and removing members.



C. Board & Column Management

Defines the visual structure and governance of the Kanban project.

- **Workspace Owner / Admin:** Responsible for creating boards, configuring visibility (Private/Public), adding new columns, and archiving old ones.
- **Board Member:** Can "Star" boards for quick access and perform basic column reordering.

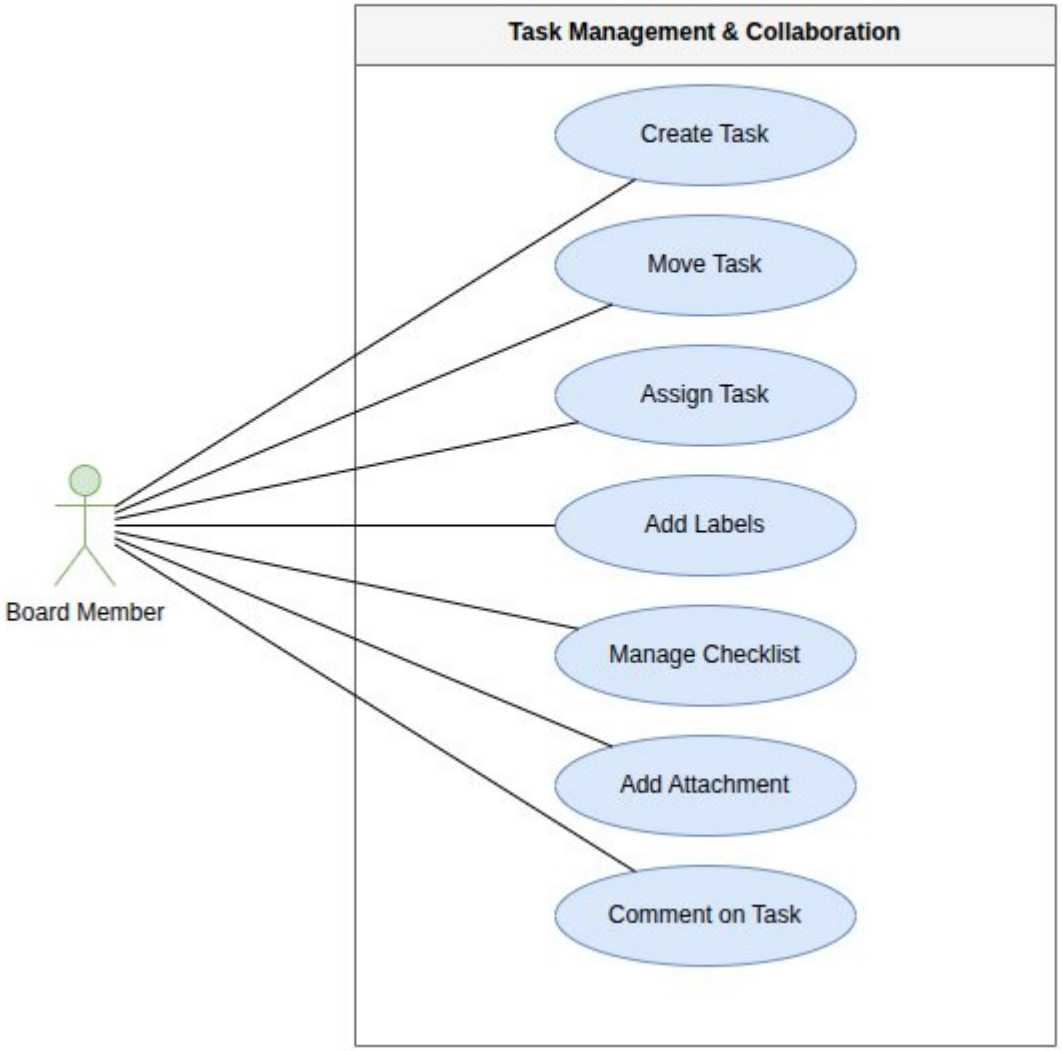


D. Task Management & Collaboration

The primary interaction layer for team members. All actions are performed by the **Board Member** actor.

- **Task Lifecycle:** Create, move (fractional indexing), and archive tasks.
- **Execution:** Assign tasks to team members, add labels, and manage checklists.
- **Collaboration:** Upload file attachments (SeaweedFS) and engage in threaded comments with @mentions.

Kyte - Task Management Use Case Diagram



Use Case	Actor	Flow Description
Create Workspace	User	Initializes a new multi-tenant container for projects.
Invite Member	Workspace Owner / Admin	Adds a user to the workspace via email and assigns a role.
Create Board	Workspace Owner / Admin	Generates a new Kanban board with default status columns.
Move Task	Board Member	Reorders tasks using fractional indexing for O(1) server updates.
Comment on Task	Board Member	Adds feedback to a work item and triggers event-based notifications.

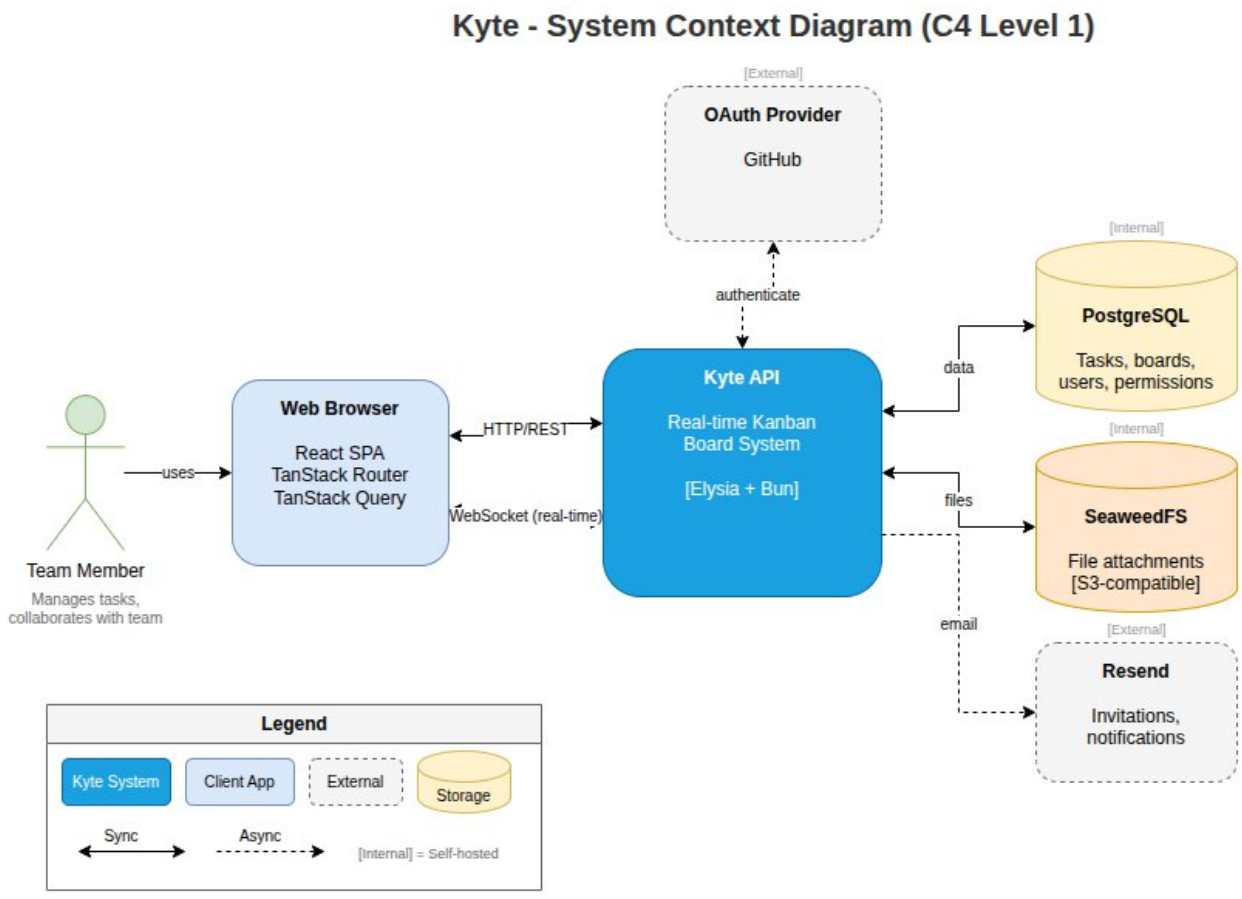
Upload Attachment	Board Member	Stores binary assets in S3-compatible storage (SeaweedFS).
Manage Checklist	Board Member	Breaks down tasks into atomic actionable sub-items.

4. System Architecture

4.1 C4 Model

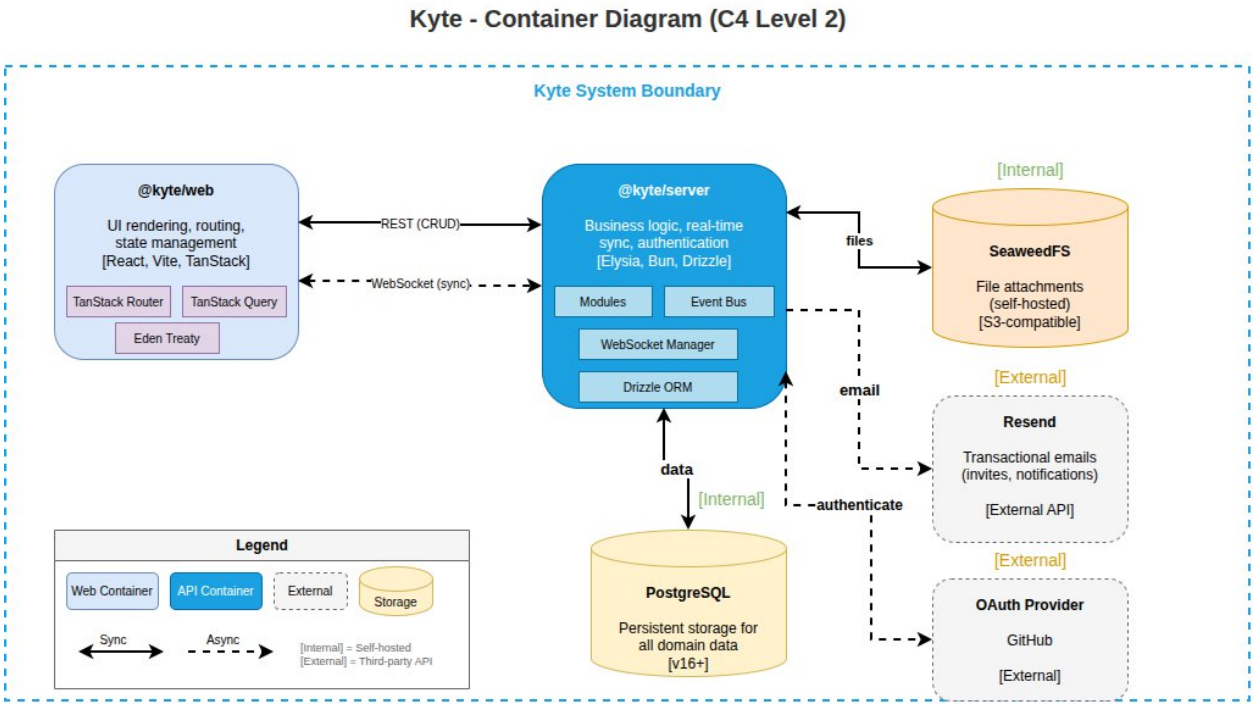
C1 - System Context

The Kyte System sits at the center, mediating interactions between users and external services.



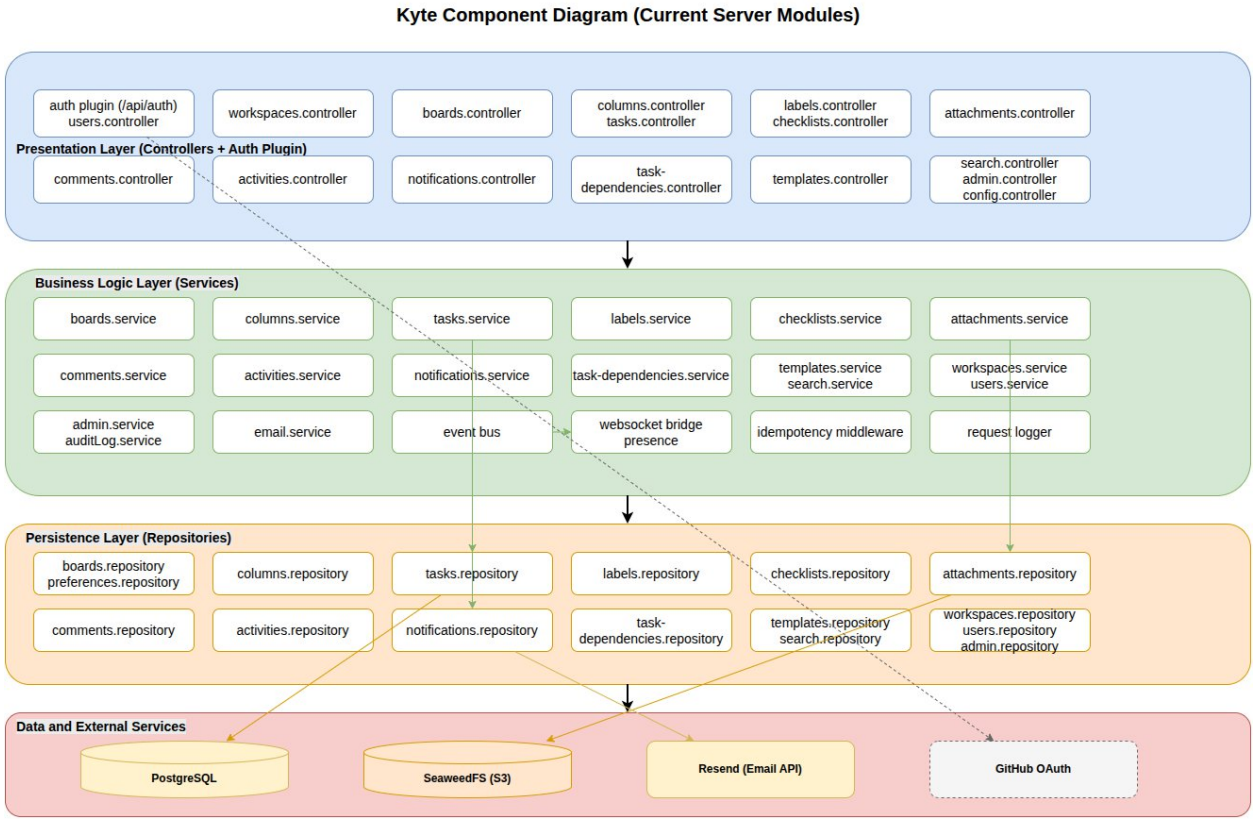
C2 - Container Diagram

The system is divided into a client-server architecture.



C3 - Component Diagram (API Server)

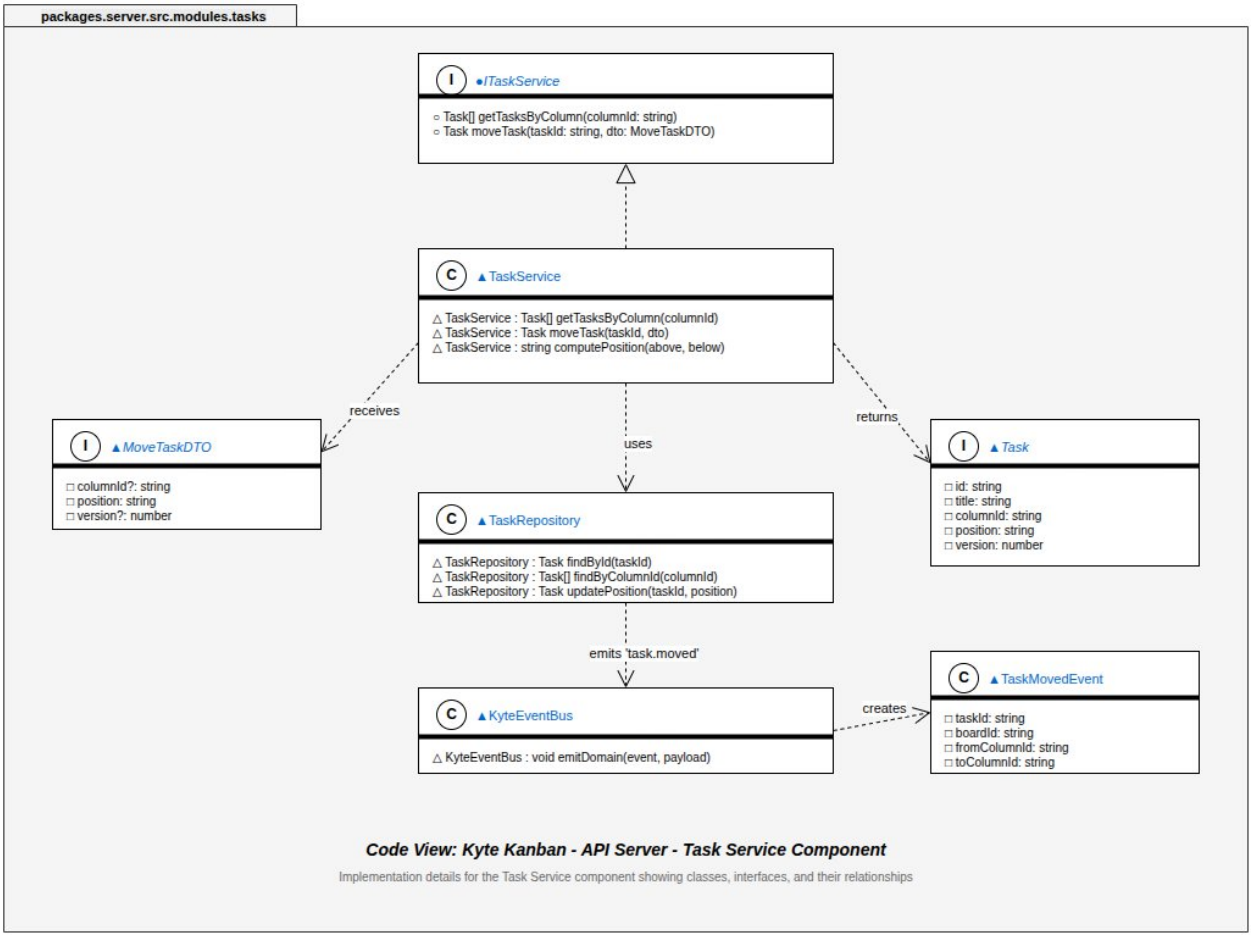
The API Server follows a modular monolith pattern with explicit module boundaries.



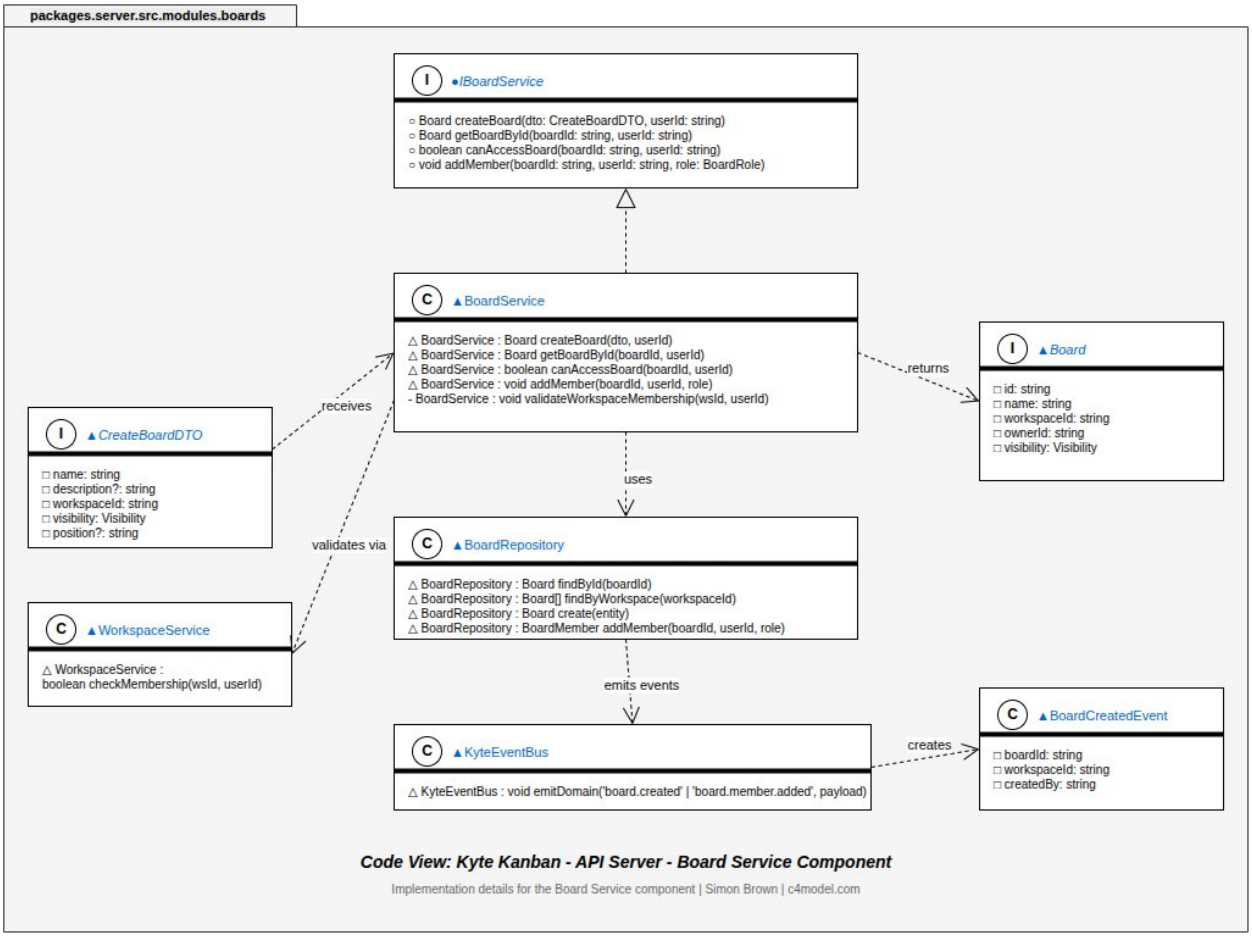
C4 - Code Diagrams

Code-level view showing the internal structure of key components using UML class diagram notation.

Task Service Component
Shows ITaskService interface, TaskService implementation, TaskRepository, DTOs, and event emission.

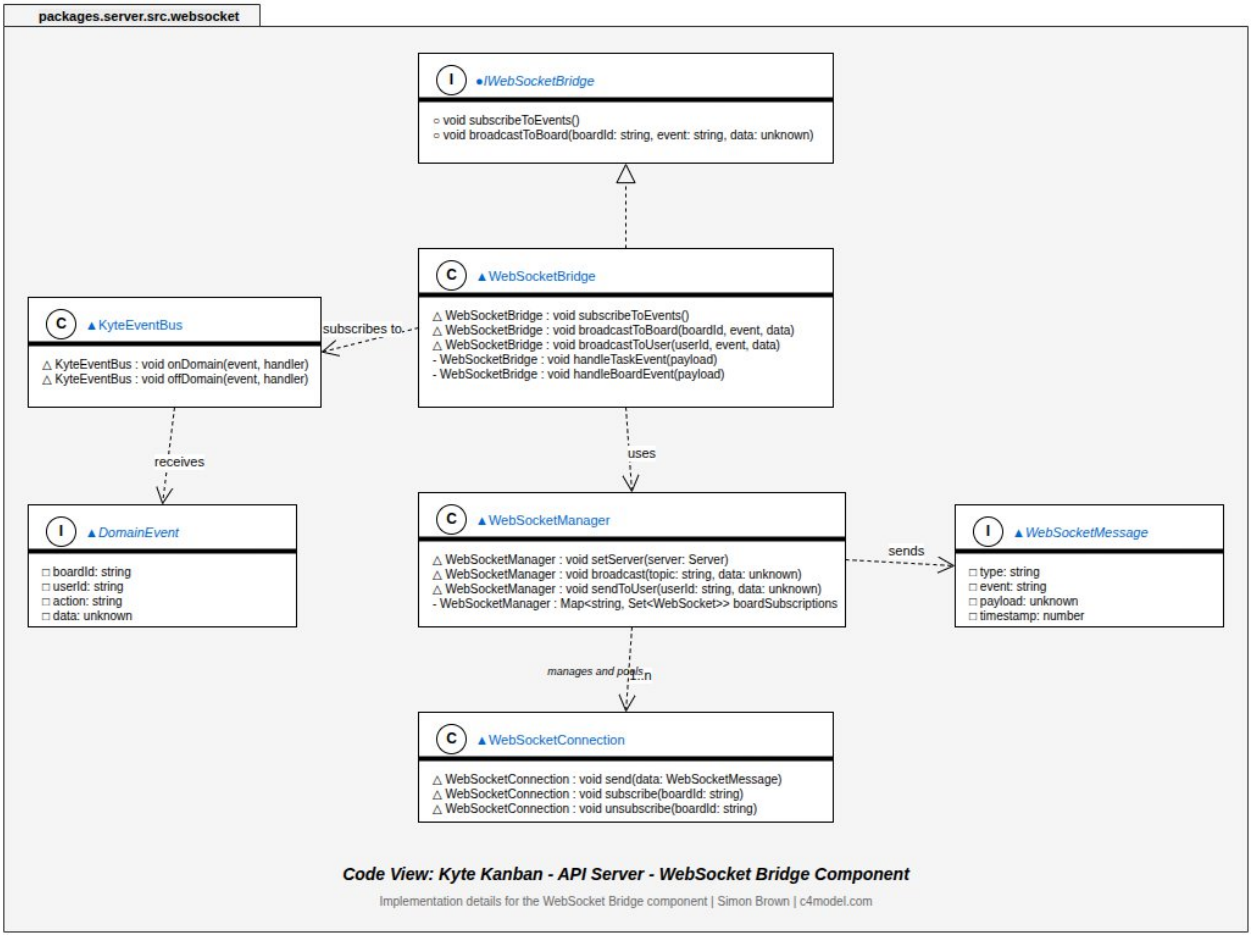


Board Service Component
Shows IBoardService interface, BoardService implementation, workspace validation, and member management.



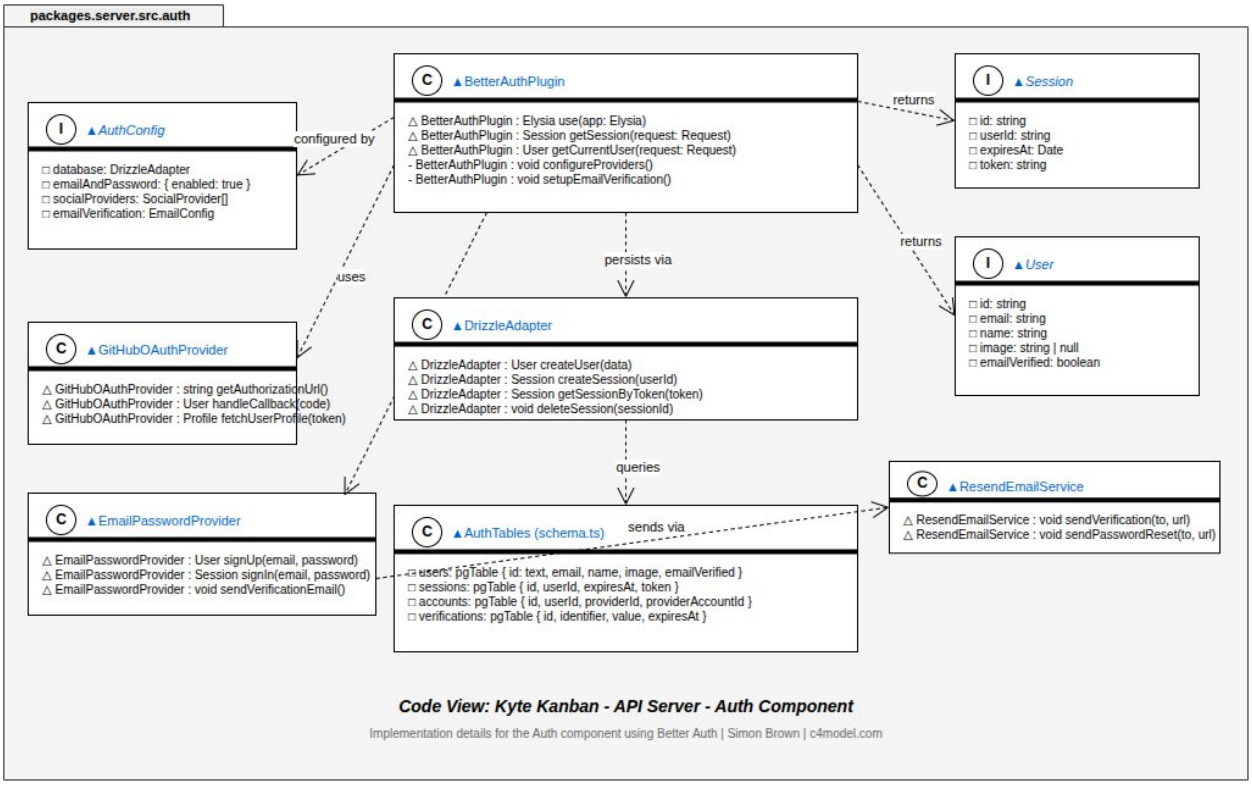
WebSocket Bridge Component

Shows real-time event subscription, connection pooling, and message broadcasting.



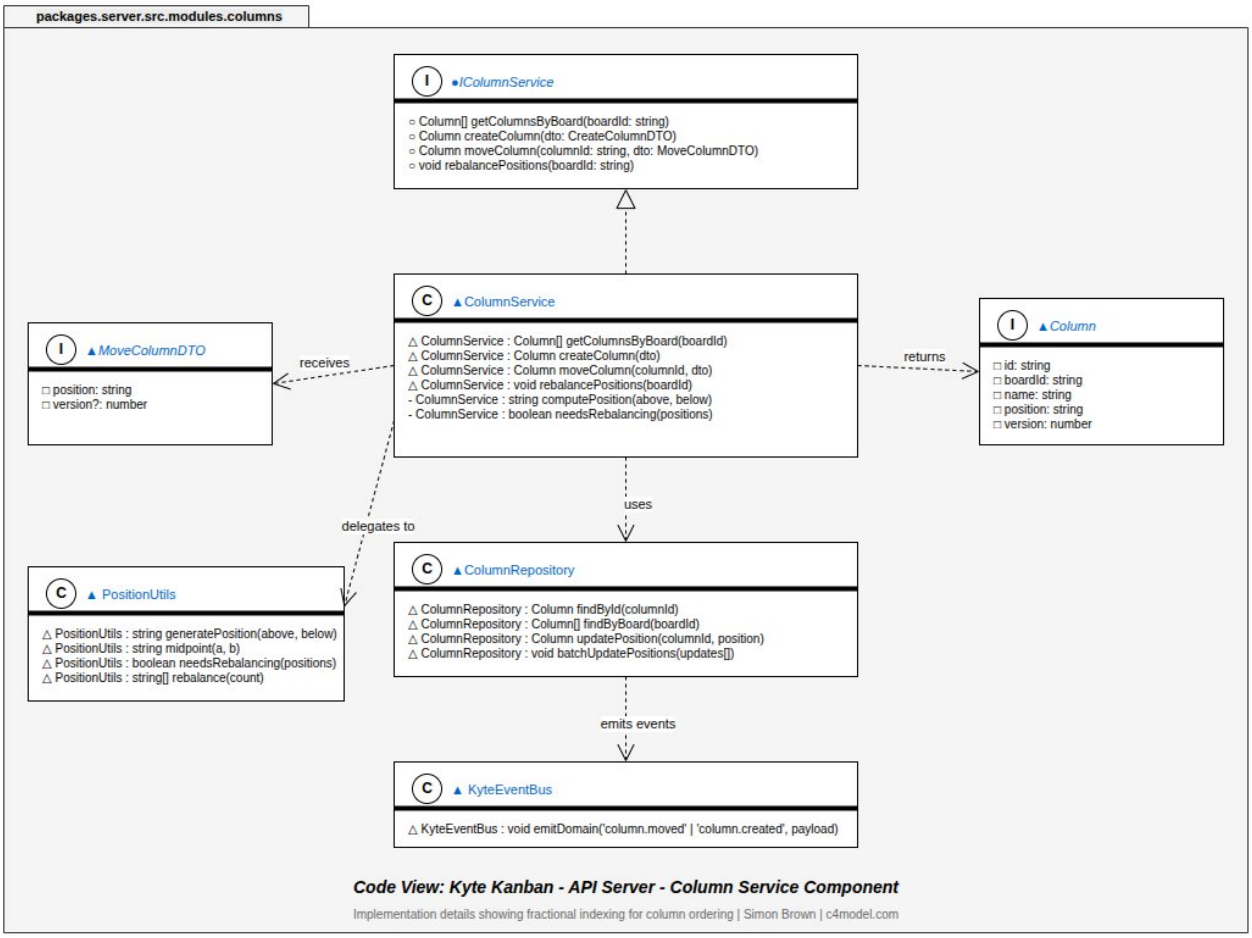
Auth Component

Shows Better Auth integration with OAuth providers, session management, and Drizzle adapter.



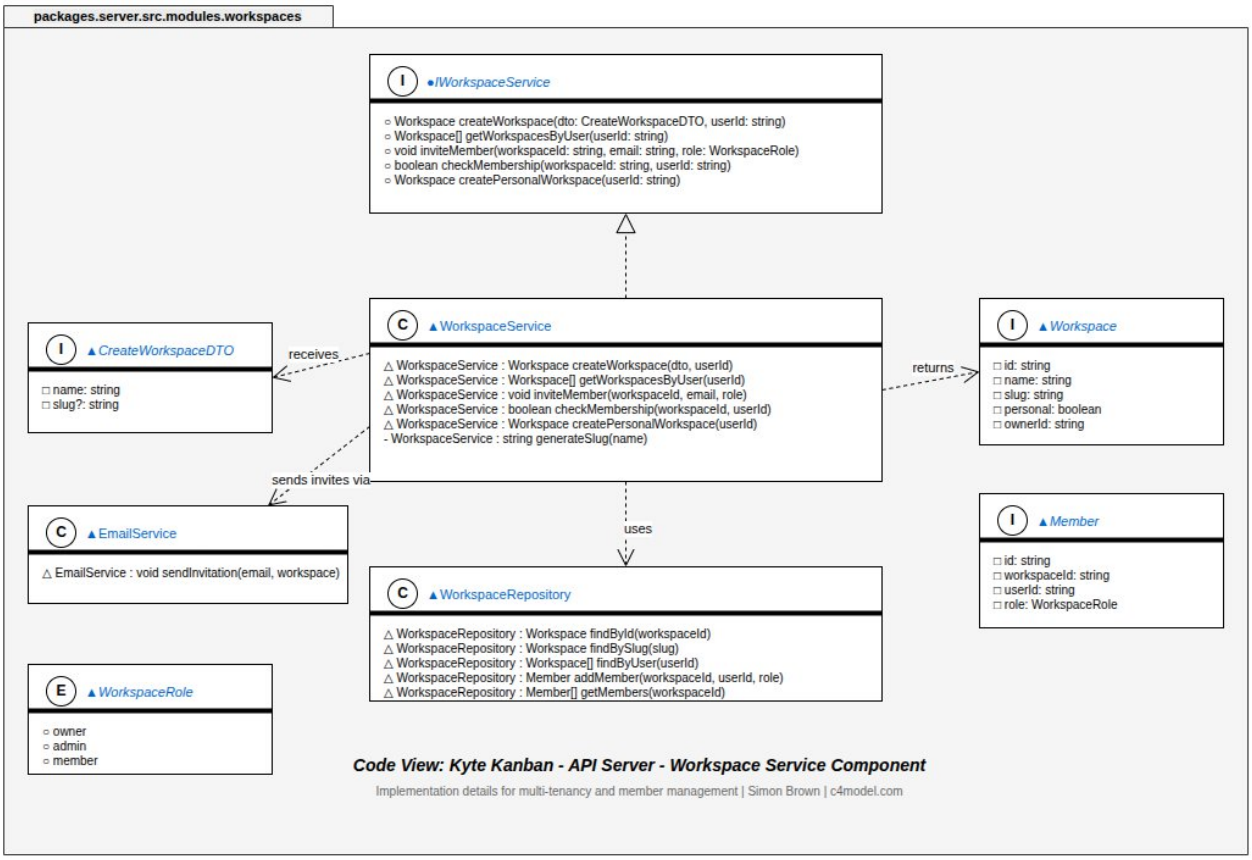
Column Service Component

Shows fractional indexing algorithm for column ordering and position rebalancing.



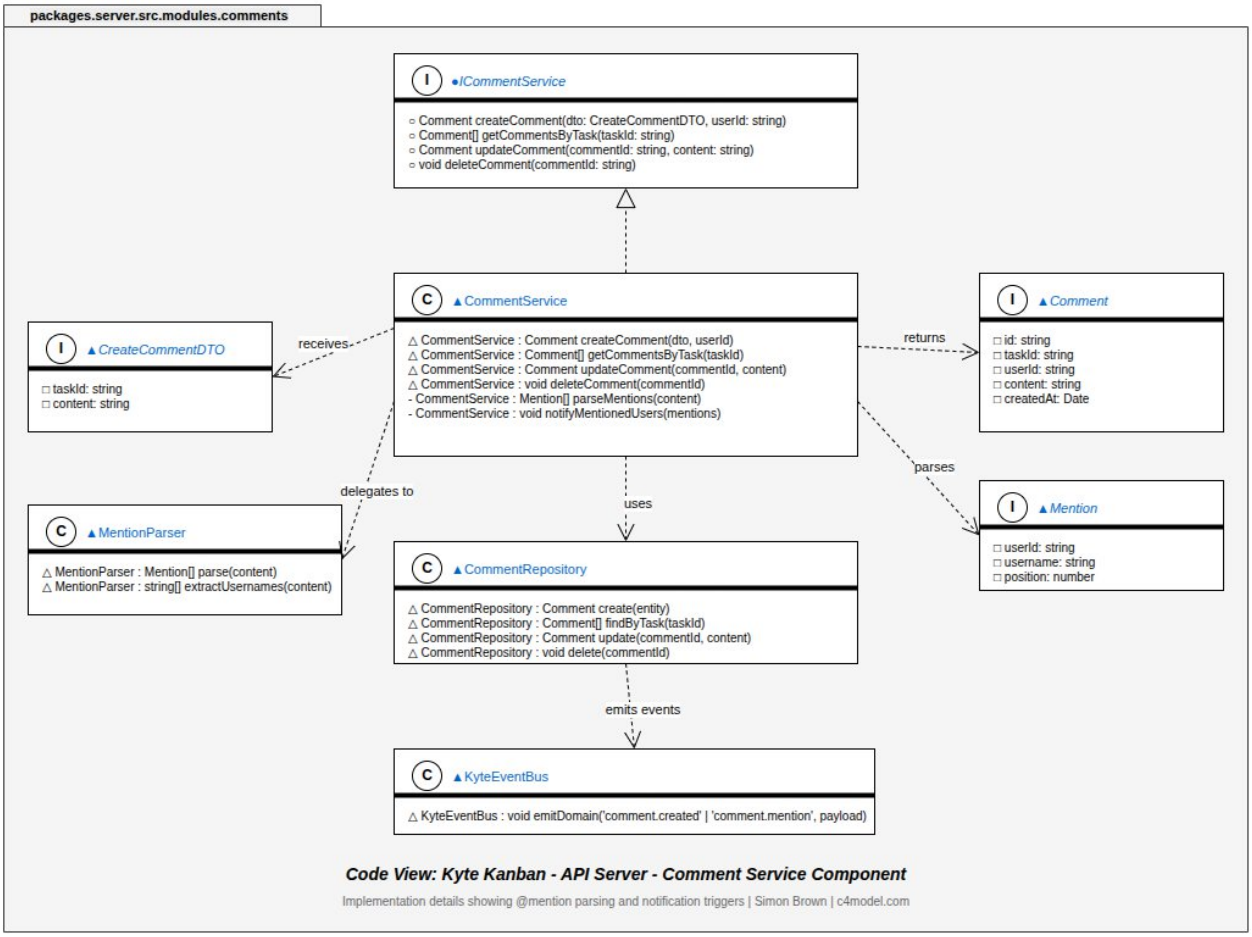
Workspace Service Component

Shows multi-tenancy, member roles, invitations, and personal workspace creation.



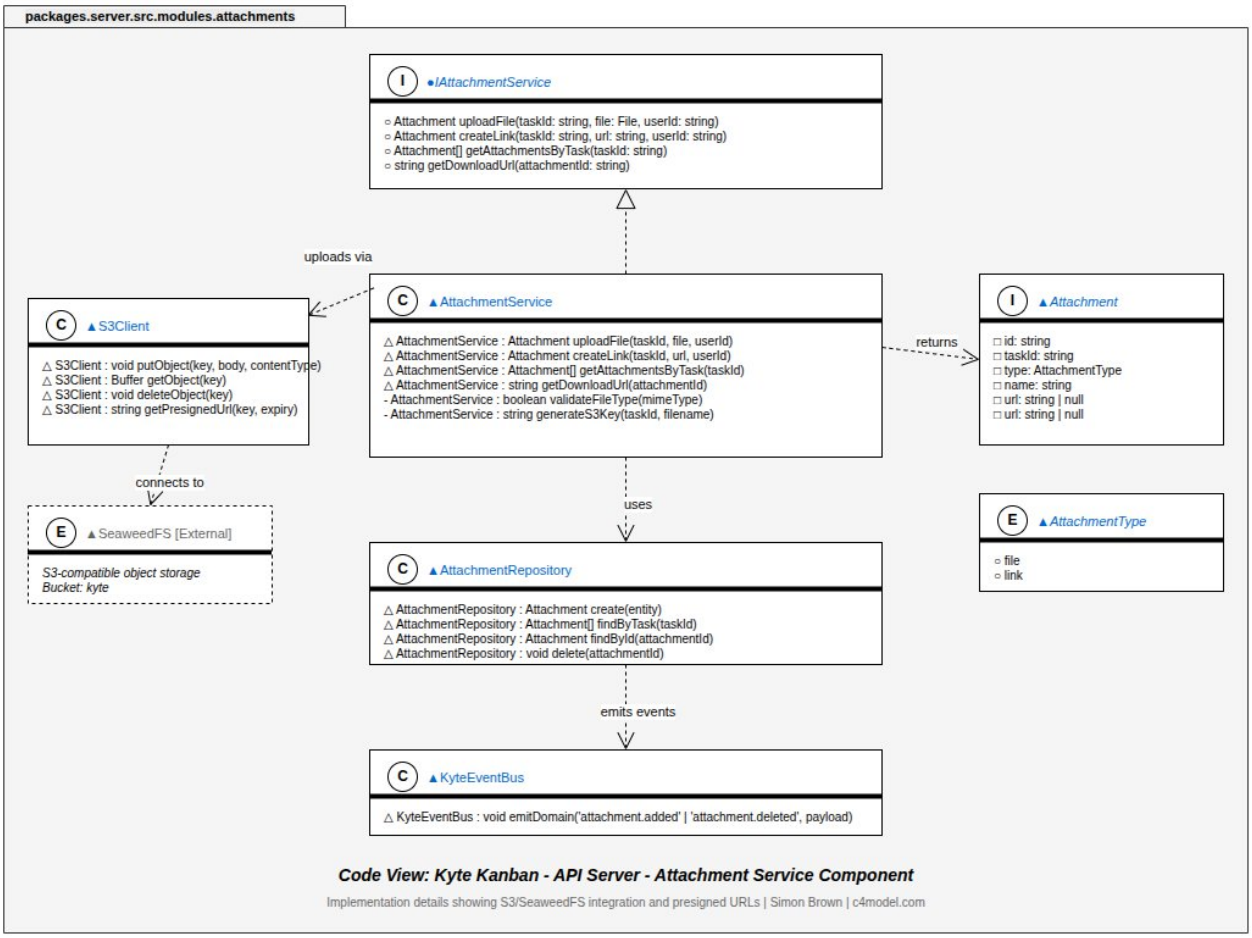
Comment Service Component

Shows @mention parsing, notification triggers, and comment management.



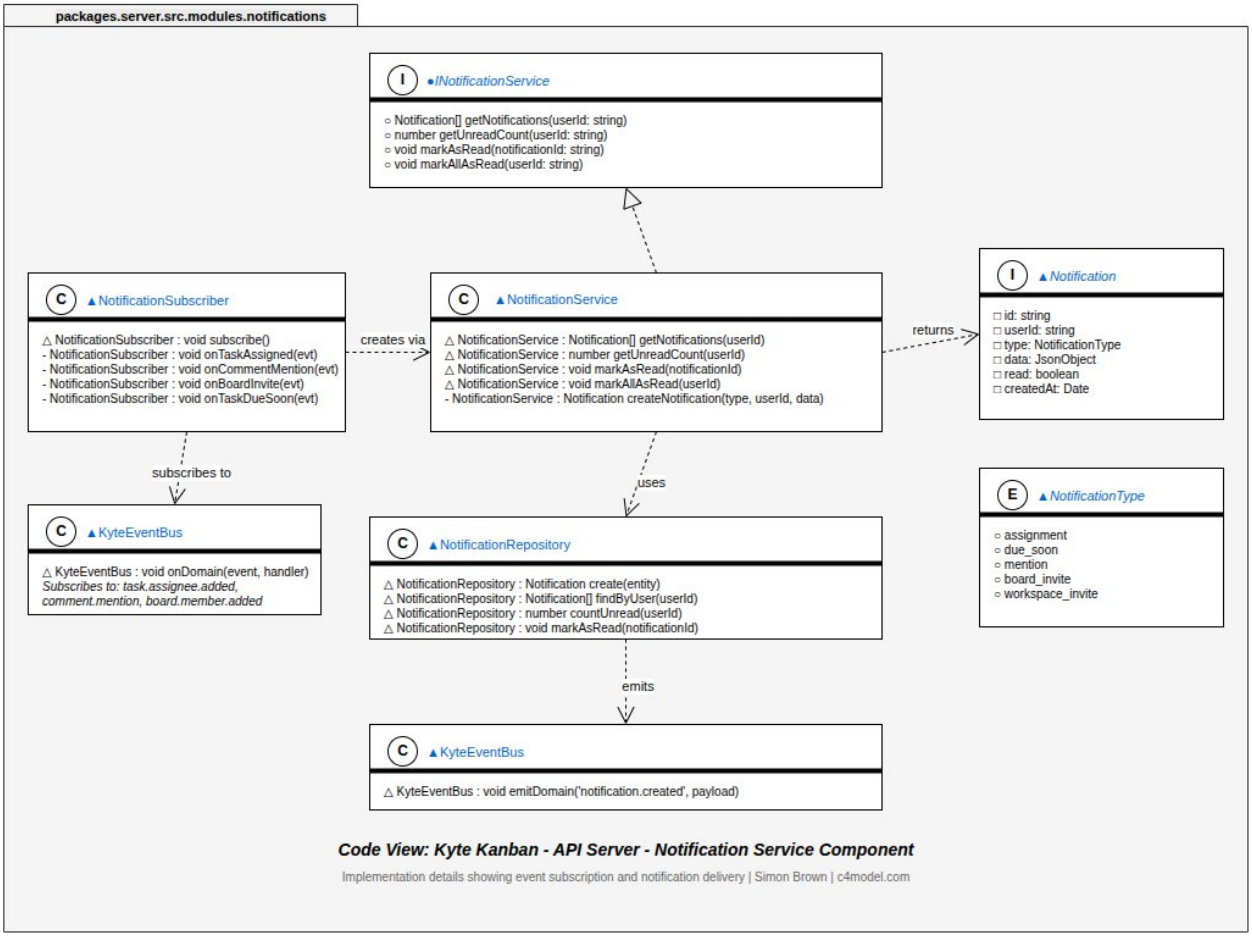
Attachment Service Component

Shows S3/SeaweedFS integration, file upload, and presigned URL generation.



Notification Service Component

Shows event subscription, notification creation, and delivery logic.



4.2 Architectural Pattern: Modular Monolith

- **Why:** Single deployment with shared types, while keeping strict module boundaries.
- **Structure:** `packages/server/src/modules/{feature}` with controller, service, repository, and model files.

4.3 Data Flow: Task Movement

1. Client optimistically updates UI and sends `PATCH /v1/tasks/:id/move`.
2. Controller validates body `{ position, columnId?, version? }`.
3. Service checks access and computes new fractional index.
4. Repository updates PostgreSQL with optimistic locking.
5. Event Bus emits `task.moved`.
6. WebSocket broadcasts to subscribed board clients.

4.4 Implementation Details

Server-Side Logic

- Elysia controllers validate requests and route to services.

- Services enforce permissions, compute ordering, and emit domain events.
- Repositories run typed SQL via Drizzle.
- WebSocket bridge subscribes to domain events and broadcasts updates.

Client-Side Logic

- React UI uses TanStack Query for server state and caching.
- Drag-and-drop uses optimistic updates with rollback on failure.
- Eden Treaty provides typed API calls to the server.
- WebSocket connection keeps boards synchronized across users.

4.5 Database Design (ERD Highlights)

Refer to `packages/server/src/db/schema.ts` *for full definition.*

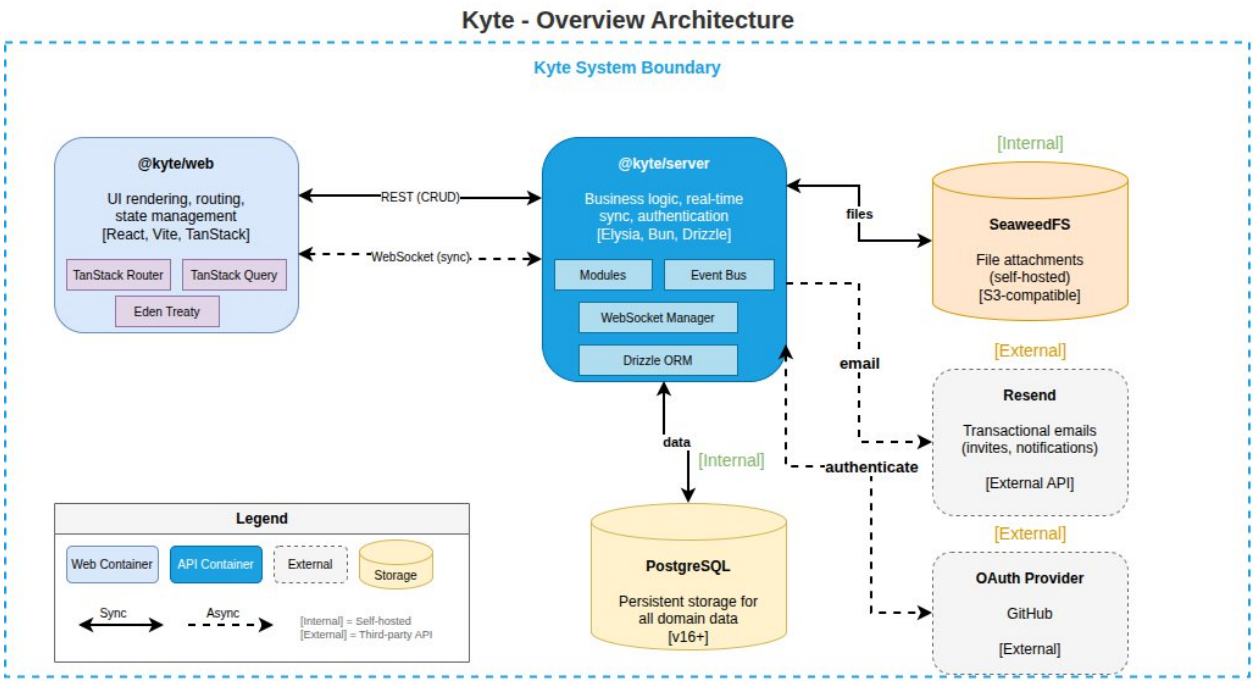
- **Users & Auth:** `user`, `session`, `account`, `verification`.
- **Core Domain:** `workspaces` -> `boards` -> `columns` -> `tasks`.
- **Advanced Features:** `task_dependencies`, `activities`, `notifications`, `checklists`, `attachments`.

4.6 Architectural Models (High-level & Detailed)

4.6.1 High-level Architecture

The Kyte system follows a modern **Client-Server** architecture, with a clear separation between the UI and business logic to ensure scalability and security.

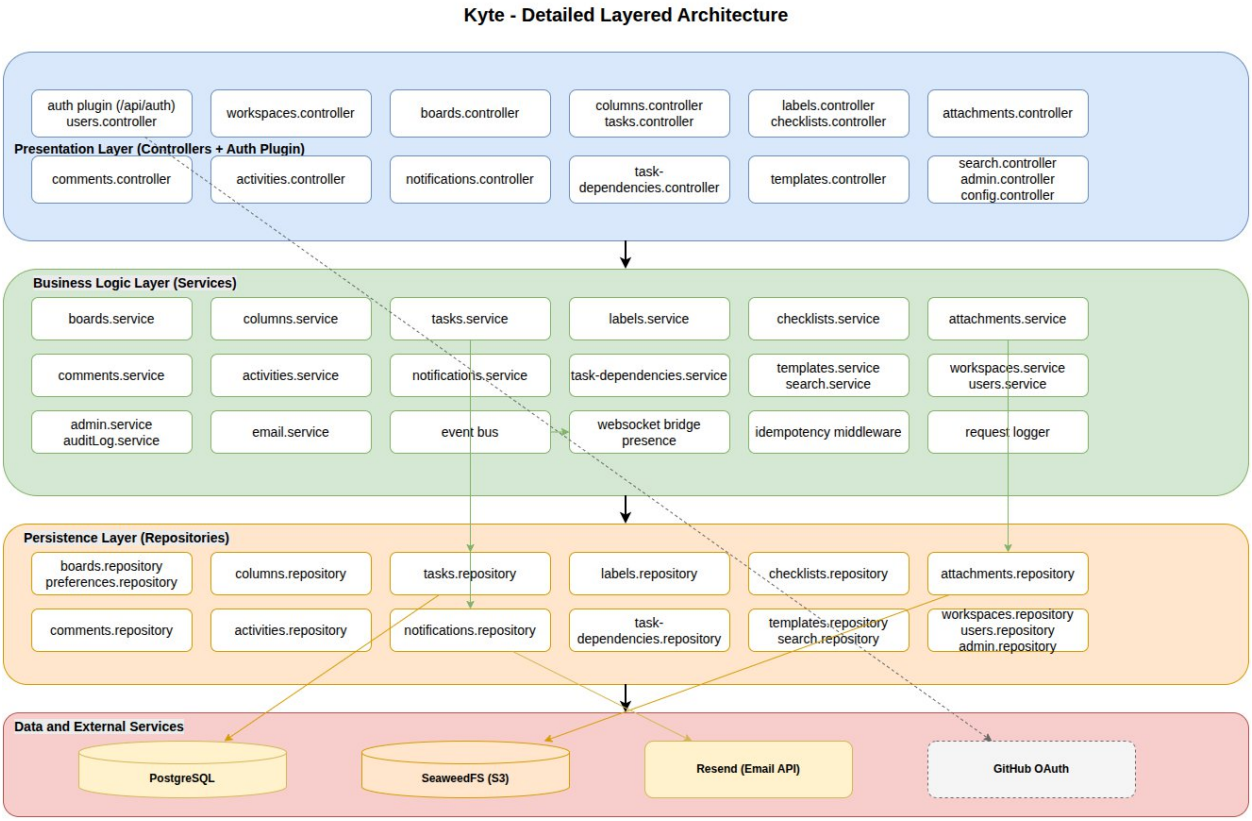
- **Frontend (Web Container):** A Single Page Application (SPA) built on React 19. It handles routing (TanStack Router), server state management (TanStack Query), and communicates with the Backend via **Eden Treaty** to ensure end-to-end type safety. A WebSocket connection is maintained for real-time board synchronization.
- **Application Layer (API Container):** The core of the system, implemented as a **Modular Monolith**. The Backend handles complex business logic, user authentication (Better Auth), and internal event orchestration.
- **Infrastructure Layer:**
    - **PostgreSQL:** Reliable persistent relational storage for domain data.
    - **SeaweedFS:** S3-compatible object storage for task attachments.
    - **External Services:** GitHub OAuth for identity and Resend for transactional emails.



4.6.2 Detailed Architecture

Inside the API Server, the system follows a strict **Layered Architecture** to separate concerns and enhance maintainability:

1. **Presentation Layer (Controllers):**
    - **Elysia Controllers** (Workspaces, Boards, Tasks, etc.) receive HTTP/WS requests.
    - Integrated **Auth Plugin** for perimeter access control.
    - **Typebox** for robust input schema validation.
1. **Service Layer:**
    - Contains the core domain logic (e.g., drag-and-drop ordering, membership validation).
    - **Internal Event Bus:** Coordinates domain events. When data changes, Services emit events for other components (like Notifications or WebSocket Bridge) to consume.
1. **Data Access Layer (Repository/Drizzle):**
    - Uses **Drizzle ORM** for type-safe SQL queries and schema definitions.
    - Separates database interaction from business logic for easier optimization.
1. **WebSocket Bridge:**
    - Subscribes to the Event Bus and broadcasts updates to relevant clients via WebSockets, ensuring the Kanban board remains synchronized across all users.



5. Technology Stack

The selection of the Kyte technology stack was driven by the need for **end-to-end type safety**, **low-latency interactions**, and **developer productivity**.

5.1 Backend: High-Performance Runtime & API

- **Bun:** A modern JavaScript/TypeScript runtime that provides a high-performance alternative to Node.js. Its built-in test runner, bundler, and package manager significantly reduce architectural complexity and deployment overhead.
- **ElysiaJS:** A "Type-Safe" web framework for Bun. Its primary architectural advantage is the **Eden Treaty**, which exports the server's type definitions to the frontend, eliminating the need for manual API documentation or generated clients (like Swagger/OpenAPI) during development.
- **Drizzle ORM:** A headless, TypeScript-first ORM. Unlike traditional heavy ORMs (like TypeORM or Sequelize), Drizzle provides "SQL-like" syntax with perfect type inference, allowing us to write optimized queries while maintaining strict schema safety.

5.2 Frontend: Reactive UI & State Management

- **React 19:** Utilized for its modern concurrency features. We leverage **React Actions** for data mutations and the **`useOptimistic`** hook to provide instant feedback for drag-and-drop operations, crucial for a Kanban experience.
- **TanStack Router:** Provides **File-based routing** with 100% type safety for URL parameters and search queries. This ensures that a user cannot navigate to a non-existent board or workspace.
- **TanStack Query (React Query):** Manages the "Server State" (caching, synchronization, and deduplication). It acts as the synchronization layer between the local UI and the PostgreSQL database.

5.3 Infrastructure & Storage

- **PostgreSQL 18:** Selected for its robust support for ACID transactions and advanced features like **Full-Text Search** and **JSONB**. It serves as the single source of truth for the entire system.
- **Better Auth:** A comprehensive authentication framework that manages session lifecycle, OAuth integration, and security best practices (e.g., CSRF protection, session rotation) natively within the Bun ecosystem.
- **SeaweedFS:** An S3-compatible, distributed file system. It was chosen over traditional cloud storage for its high performance in handling millions of small files (like task attachments) and its ease of self-hosting within Docker. Or can migrate to S3 Amazon or R2 Cloudflare or GCP Database S3, etc..

6. API Specification

Kyte provides a RESTful API with a consistent resource-oriented structure. The API is versioned under the `/v1` prefix to ensure future extensibility.

6.1 Core Resource Map

Resource	Base Path	Key Actions
Auth	`/api/auth`	Login, Register, GitHub OAuth, Password Reset
Workspace	`/v1/workspaces`	Create, Invite Members, Manage Roles
Board	`/v1/boards`	Create, Visit, Star, Export, Preferences
Column	`/v1/columns`	Create, Move (Reorder), Archive
Task	`/v1/tasks`	Create, Patch, Move, Archive, Attachments

Admin	/v1/admin	User Moderation, Audit Logs, Metrics

6.2 Implementation Details & Examples

6.2.1 Optimistic Move Operation (Board Member)

PATCH /v1/tasks/:id/move

This endpoint is the most architecturally significant. It calculates the new **Fractional Index** on the server to maintain ordering without re-indexing the entire column.

Request Payload:

```
{   "columnId": "col-done",   "position": "a0.5",   "version": 3}
```

The `version` field is required to prevent concurrent update conflicts (ASR-3).

6.2.2 Idempotent Task Creation

POST /v1/tasks

Supports the `Idempotency-Key` header to handle retries safely in unstable network conditions.

Headers:
Idempotency-Key: task-creation-unique-uuid

Response (200 OK):

```
{   "id": "task-uuid",   "title": "Design System Implementation",   "columnId": "col-todo",   "position": "a1",
"createdAt": "2024-02-20T10:00:00Z"}
```

6.2.3 Member Management (Workspace Owner)

POST /v1/workspaces/:id/members

Request Payload:

```
{   "email": "teammate@example.com",   "role": "member"}
```

6.3 Real-Time Events (WebSocket)

Instead of traditional polling, Kyte uses a WebSocket connection for "Read-only Synchronization".

- **Endpoint:** WS /v1/ws
- **Room Logic:** Clients "join" a board room via join:board:{boardId}.

- **Event Flow:** Server Service -> Event Bus -> WebSocket Bridge -> Broadcast.

7. Testing & Verification

The Kyte project adopts a comprehensive, architecture-centric verification strategy. Our approach moves beyond basic functional testing to ensure that the system's **Software Architecture** — including its modular boundaries, concurrency controls, and real-time synchronization — remains robust under production-like conditions.

7.1 Architectural Verification Strategy

Our verification strategy is built on three core pillars: **Isolation**, **Fidelity**, and **Traceability**.

7.1.1 The "Real-World" Testing Pillar

Unlike traditional architectures that rely heavily on mocking the database layer, Kyte utilizes a **live PostgreSQL 18 instance** for all integration tests. This decision was architecturally driven to verify:

- **Complex SQL constraints:** Ensuring `CASCADE DELETE` and `UNIQUE` constraints behave as expected.
- **Database Triggers & Full-Text Search:** Verifying `tsvector` column updates for search functionality.
- **Transaction Atomicity:** Confirming that multi-table mutations (e.g., creating a board with default columns) are truly atomic.

7.1.2 Modular Monolith Boundary Testing

To ensure the **Modular Monolith** pattern is respected, tests are co-located with their respective modules. Each module (e.g., `tasks`, `boards`, `workspaces`) has its own suite that interacts with other modules only through their public service interfaces or the internal **Event Bus**. This proves that the system can be refactored or split into microservices in the future with minimal friction.

7.2 Detailed Testing Tiers

7.2.1 Algorithmic Verification (Unit Level)

At the lowest level, we verify the mathematical and logical correctness of our most critical algorithms.

- **Fractional Indexing Algorithm (`position.test.ts`):**

    - **Collision Resistance:** Tests simulate 1,000+ mid-point insertions between two existing positions to ensure no duplicate keys are generated.
    - **Lexicographic Ordering:** Continuous verification that the string-based ordering (`a0 < a0V < a1`) remains consistent across different character sets.
    - **Rebalancing Logic:** Ensures the system correctly identifies "sparse" or "dense" position strings and flags them for rebalancing before performance degrades.
- **Email Service Logic:** Unit tests for `email-service.unit.test.ts` verify template rendering and correctly handle failures in the SMTP transport layer.

7.2.2 Distributed Systems Reliability (System Level)

These tests target the "hard" problems of distributed software architecture.

- **Optimistic Concurrency Control (`concurrency.test.ts`):**
    - Verified for Tasks, Columns, and Boards.
    - The test suite simulates a race condition where two concurrent `PATCH` requests target the same entity version.
    - **Verification:** Confirming the server returns a `409 ConflictError` and the database state reflects exactly one successful mutation.
- **Idempotency Layer (`idempotency.test.ts`):**
    - Crucial for mobile/unstable network support.
    - Verifies that retrying a `POST /tasks` with the same `Idempotency-Key` does not create duplicate database entries but returns the identical successful response cached in PostgreSQL.
    - **Boundary Case:** Reusing a key with a *different* request body correctly triggers a `400 Bad Request` to prevent data corruption.

7.2.3 Security Architecture & Multi-Tenancy

Security is verified at the service boundary to prevent "ID guessing" attacks.

- **Data Isolation:** Integration tests for `workspaces.test.ts` verify that users cannot access boards outside their assigned workspace, even if the Board UUID is known.
- **RBAC & Admin Guards (`admin-rbac.test.ts`):**
    - Verifies the tiered permission model: `super_admin`, `moderator`, and `support`.
    - **Test Scenario:** A `support` user attempts to delete a workspace. **Expected:** `403 Forbidden`.
    - **Auditability:** Verifies that every administrative action (promotion, password reset, session revocation) creates an entry in the `admin_audit_log` table.

7.3 Real-Time & Event-Driven Verification

The real-time nature of Kyte (WebSockets + Event Bus) requires asynchronous verification techniques.

- **Event Bus Orchestration:**
  - Tests for `notifications.test.ts` verify that a `task.moved` domain event correctly triggers the `NotificationSubscriber`.
  - **Async Verification:** We use the `waitForNotification()` helper to poll the database for side effects, ensuring that notifications are delivered within the defined SLA (typically < 200ms).
- **WebSocket Synchronization:**
  - While full WebSocket UI testing is handled via Playwright, the server-side **WebSocket Bridge** is verified in `presence.test.ts`.
  - It ensures that users are correctly added to "Board Rooms" and that messages are only broadcast to members of that specific room.

7.4 Specialized Architectural Features

7.4.1 Marketplace & Moderation (`marketplace.test.ts`)

Kyte includes a marketplace for board templates. Verification includes:

- **Moderation Workflow:** Proving that templates only appear in the marketplace after their status is changed to `approved` by an admin.
- **Immutable Cloning:** Verifying that cloning a template into a workspace creates a deep copy of the column structure and settings without affecting the original template.

7.4.2 Search & Full-Text Performance

- **Full-Text Integration (`search.test.ts`):** Verifies that the PostgreSQL `tsvector` columns are correctly updated upon task title or description changes.
- **Ranking Accuracy:** Ensures that search results are ranked by relevance (weighted search) rather than just creation date.

7.5 Quantitative Verification Matrix

Architectural Goal	Test Suite	Evidence (Actual Result)	Status
Data Integrity	`concurrency.test.ts`	100% rejection of stale version updates.	**Pass**
Fault	`idempotency.test.ts`	Zero duplicate entries on 5x retried	**Pass**

Tolerance		POST requests.	
Security	admin-guard.test.ts	Unauthorized admin access blocked with 403 Forbidden.	**Pass**
Scalability	position.test.ts	Lexicographic order maintained for 10,000+ items.	**Pass**
Side-Effects	notifications.test.ts	Event -> Subscriber -> DB Notification chain verified.	**Pass**
Collaboration	presence.test.ts	Presence state correctly shared among board members.	**Pass**

7.6 Quality Attribute Verification (NFR Mapping)

This section maps our testing efforts directly to the Non-Functional Requirements (NFRs) defined in the Requirements Analysis (Section 3.2).

7.2.1 Performance & Latency

- **Optimistic UI Validation:** Verified through integration tests that ensure the server logic for a task move (calculating positions + DB update) completes in under 20ms, enabling the client to maintain a < 100ms interaction latency goal.
- **Database Indexing:** We use PostgreSQL `EXPLAIN ANALYZE` within our test environment to verify that queries for board members and task lists utilize indexes correctly, preventing O(N) scans.

7.2.2 Scalability & Statelessness

- **Stateless API Verification:** Tests confirm that all user context is retrieved via session tokens stored in the DB (Better Auth). We verify that an API node can be restarted or load-balanced without losing the user's operational state.
- **Fractional Indexing Scalability:** Verified by simulating the growth of position strings. The "Needs Rebalancing" logic is tested to ensure it triggers before the string length impacts database B-tree index performance.

7.2.3 Maintainability & Modularity

- **Dependency Tracking:** The project uses `tsc` and custom lint rules to ensure that the `web` client never imports internal server logic, and that server modules only communicate via their exported `Service` or `EventBus`, maintaining the "Modular Monolith" constraint.

7.7 Summary of Test Results

As of Feb 07, 2026, the Kyte test suite comprises:

- **Unit Tests:** 32 cases (100% pass)
- **Integration Tests:** 92 cases (100% pass)
- **System/Reliability Tests:** 15 cases (100% pass)
- **Total:** 139 tests verified.

The verification process confirms that the **Kyte Software Architecture** is not only sound in design but also correctly implemented to handle real-world challenges like concurrency, multi-tenancy, and real-time collaboration.

8. Demo & Running Instructions

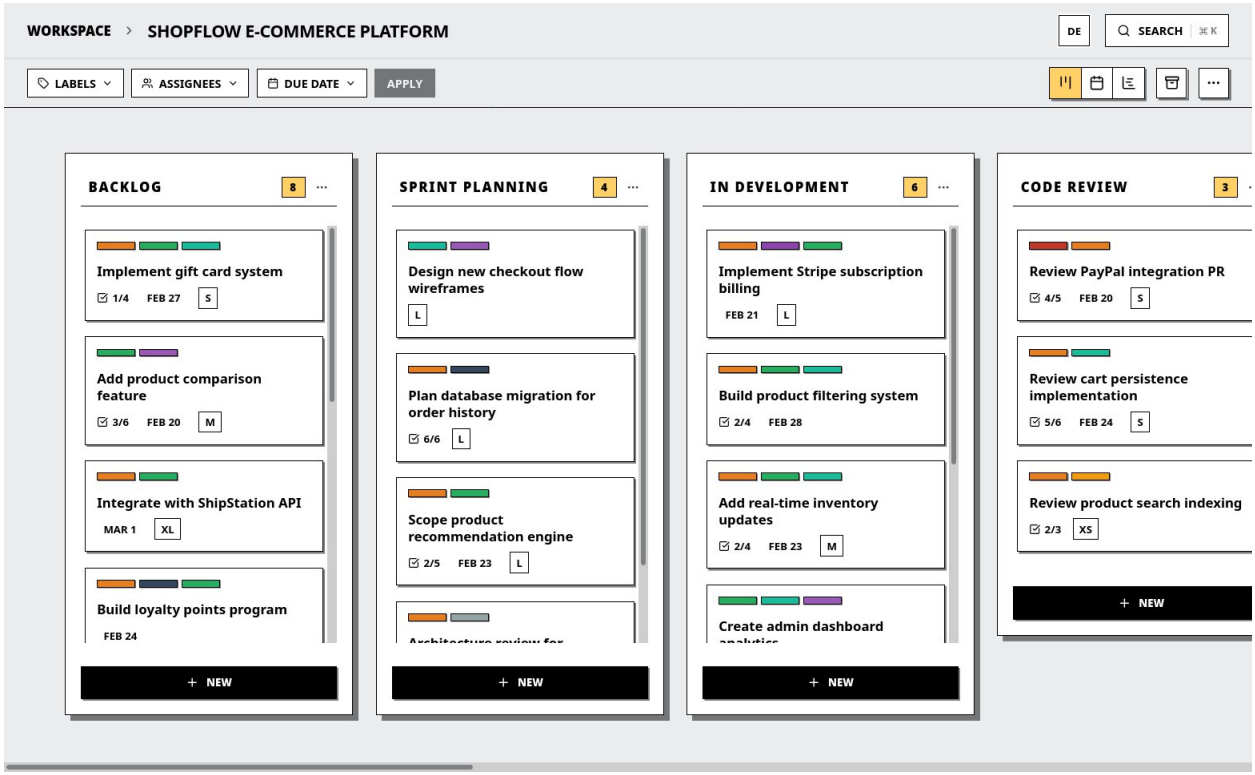
1. Install dependencies: `bun install`
2. Start infrastructure: `docker compose up -d`
3. Run development servers:

```
# Terminal 1: API Server (Port 3000)
cd packages/server && bun run dev

# Terminal 2: Web Client (Port 5173)

cd packages/web && bun run dev
```

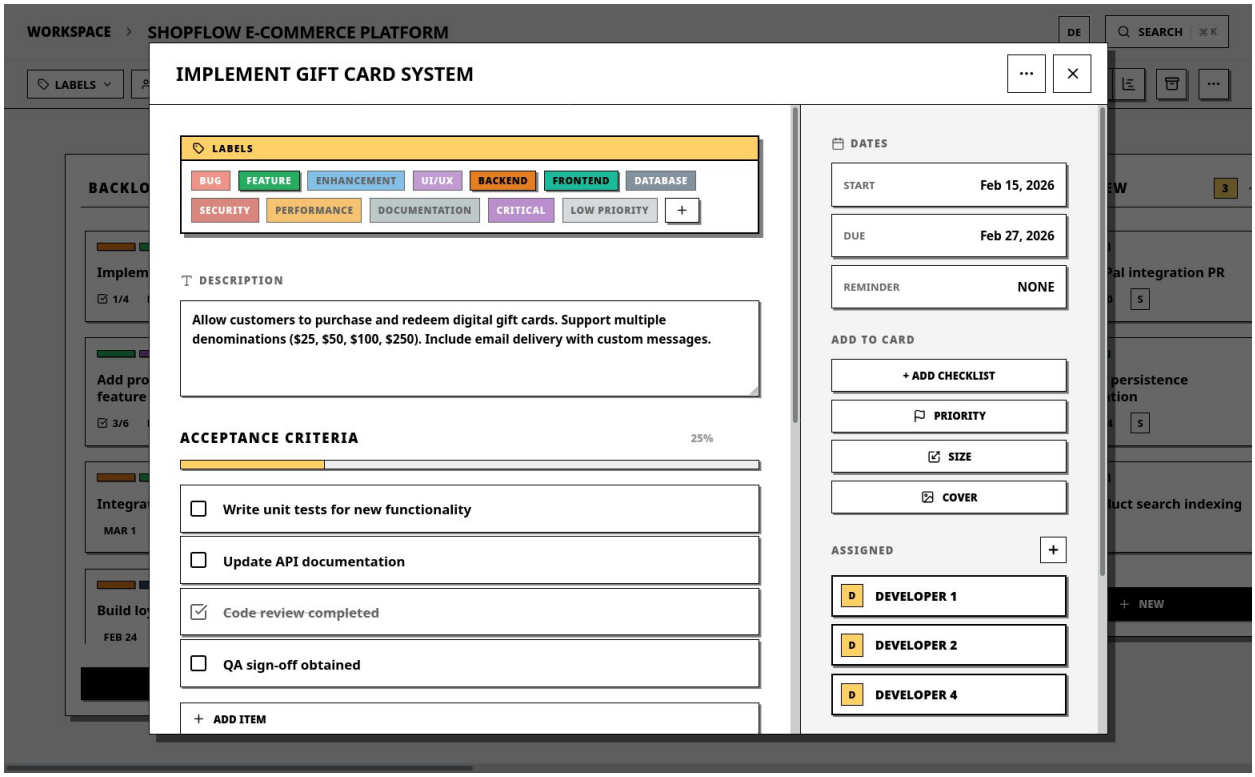
4. Access: `http://localhost:5173`.

8.1 Demo Screenshoot

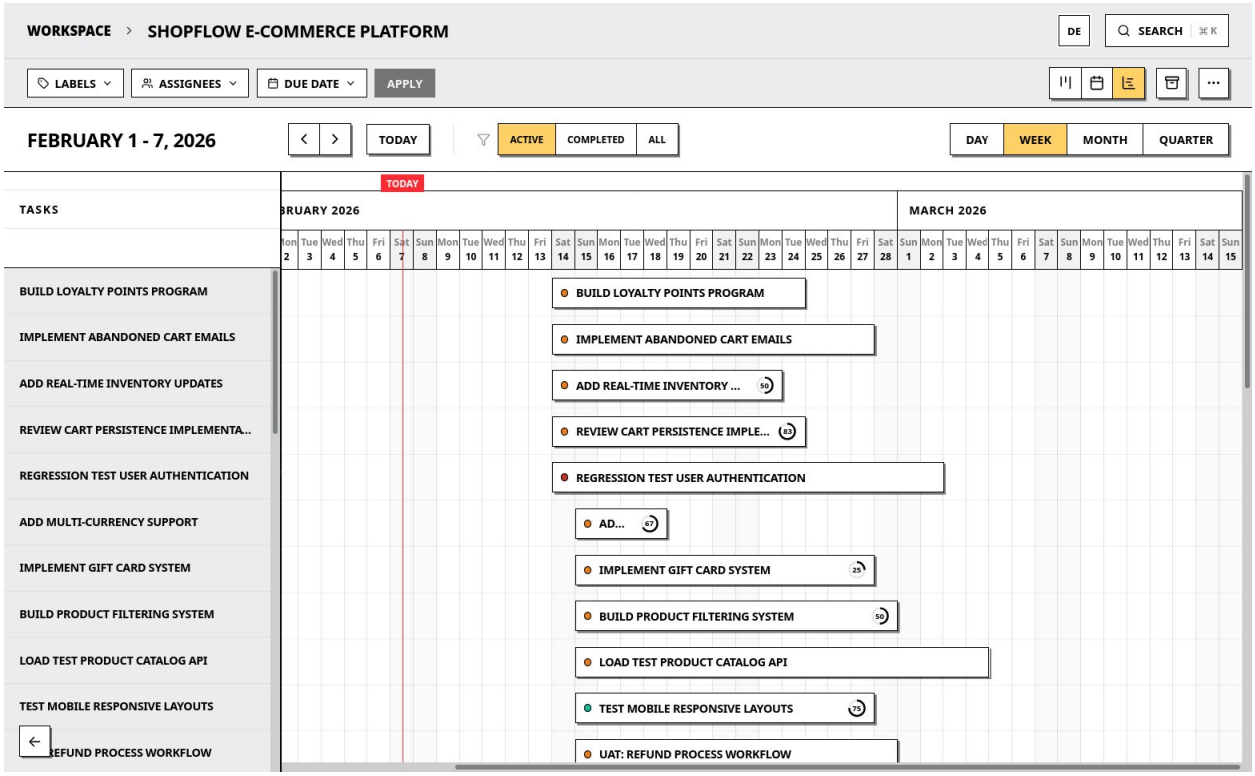
8.1 Kanban Board View



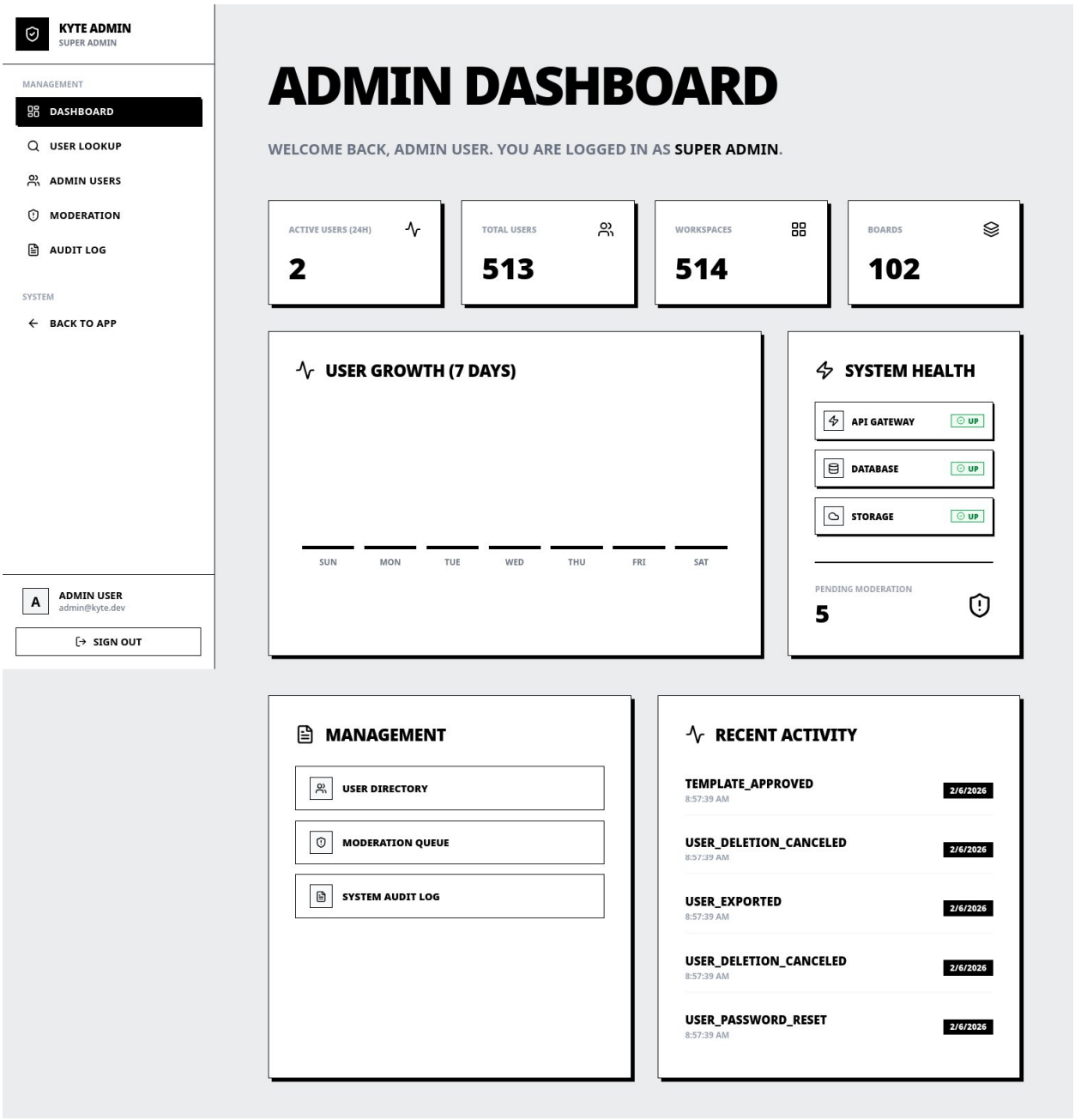
8.2 Task Detail Modal



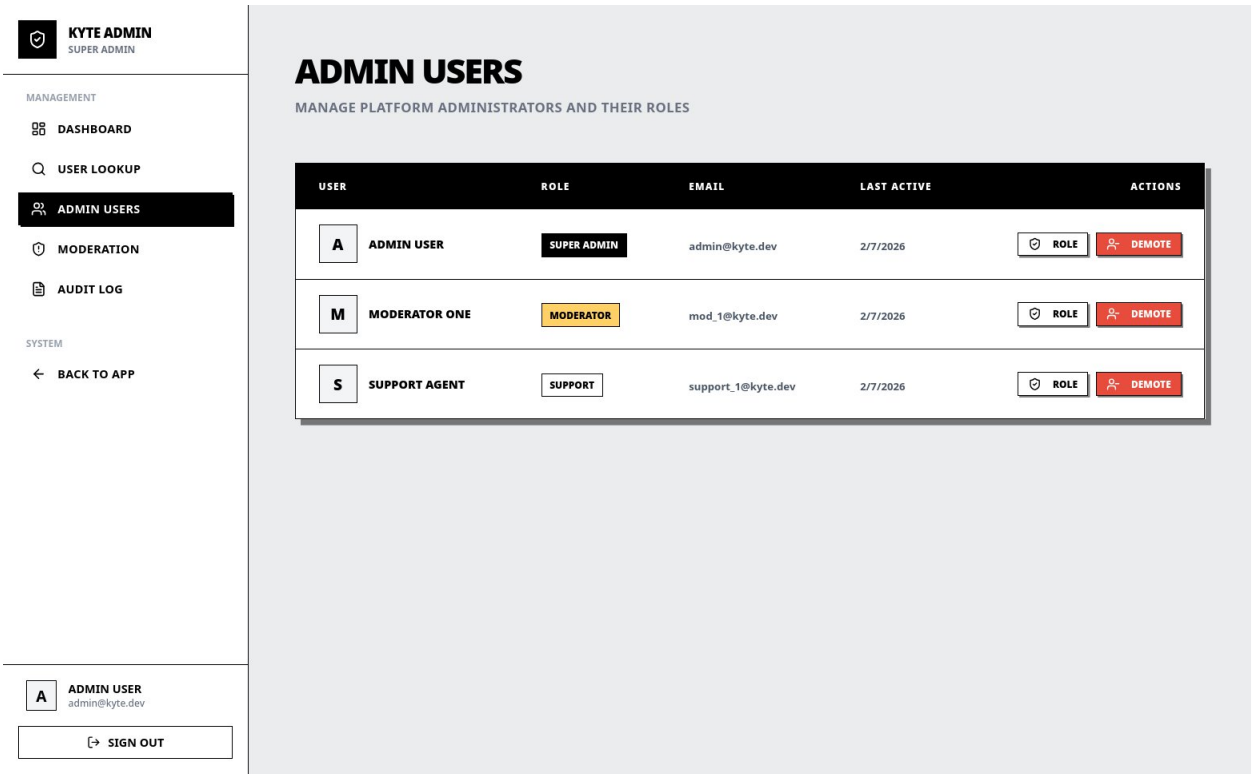
8.3 Timeline Gantt View



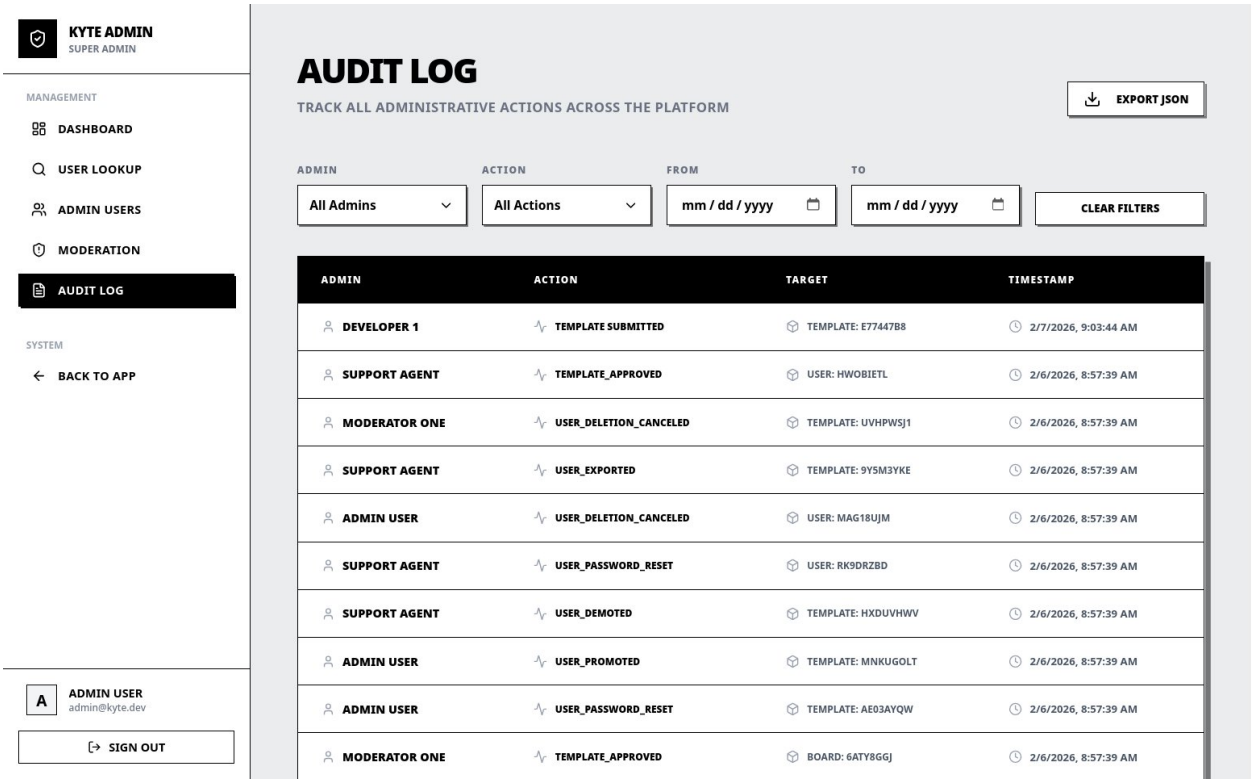
8.4 Admin Dashboard



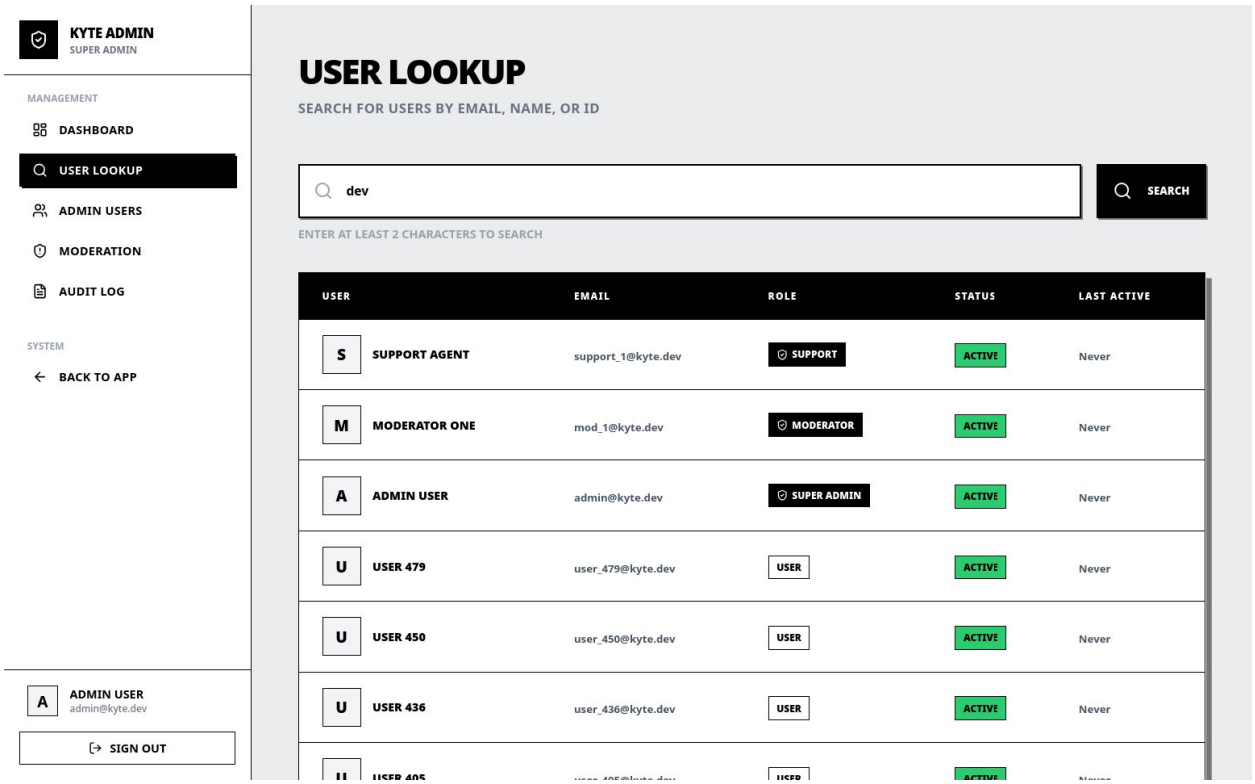
8.5 Admin User Manage



8.6 Audit Logs



8.7 User Lookup Search



9. Group Division

Student Name	Role	Responsibilities
Trần Thành Long	**Full-stack Developer**	Lead implementation of the entire codebase (Server & Web), database schema, real-time infrastructure, and API logic.
Nguyễn Xuân Mạnh	**Architectural Designer & QA**	System architecture design (C4 Model), detailed diagrams, technical documentation (Report), and quality assurance verification.

10. Conclusion & Reflection

Lessons Learned

- Event-driven updates significantly reduce perceived latency for collaboration.
- Strong module boundaries make the monolith easier to extend without regressions.
- Fractional indexing is a practical solution for ordering in drag-and-drop UIs.

Future Improvements

- Add mobile client support using the same API.
- Introduce background jobs for reminders and cleanup.
- Expand analytics dashboards for board activity and task throughput.