OUTLINE

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviations	Meaning
API	Application Programming Interface
CSS	Cascading Style Sheet
HTML	HyperText Markup Language
IP	Internet Protocol
REST	Representational Stat Transfer
PK	Primary Key
FK	Foreign Key
NN	Not Null
U	Unique
DB	Database
CRUD	Create (adding new data), read (viewing data), update (modifying existing data), and delete (removing data)
GCS	Google Cloud Storage

INTRODUCTION

In the era of digital transformation, the academic environment has evolved significantly, shifting from traditional methods to technology-driven learning models. University students are increasingly required to manage complex coursework, collaborate on group projects, and maintain self-study schedules simultaneously. To cope with these demands, students often rely on a multitude of specialized software applications: project management tools like Jira or ClickUp for tasks, cloud storage solutions like Google Drive for documents, social platforms like Facebook or Messenger for communication, scheduling apps like Google Calendar for private planning, and communities like StackOverflow for knowledge exchange.

However, the reliance on this disparately connected ecosystem creates a critical issue known as **"tool fragmentation."** Students are forced to constantly switch contexts between applications that do not communicate with one another, leading to information silos, missed deadlines, and inefficient collaboration. Furthermore, existing industrial tools are often overly complex for academic needs and fail to account for the personal well-being of the student, such as balancing academic rigor with necessary rest and health constraints.

In this context, the development of a unified platform that consolidates these essential functions into a single ecosystem is imperative. This thesis proposes the design and implementation of a collaborative platform built upon a Microservices architecture. By leveraging robust technologies, the system focuses on seamlessly integrating personal scheduling, project collaboration, and community knowledge sharing.

This report aims to demonstrate how a unified software solution can streamline the academic lifecycle, fostering a more efficient and balanced learning environment for students.

CHAPTER 1: INTRODUCTION TO THE SYSTEM AND TECHNOLOGY STACK

1.1. System Introduction

1.1.1. Thesis Objective

The primary objective of this thesis is to design and develop a unified knowledge-sharing and learning support system entitled “Building a collaborative system to support learning using Java and NextJS. The project aims to create a centralized web-based platform that integrates essential tools for students, including task management, scheduling, and document collaboration. Rather than competing with enterprise-level project management software, this system is specifically tailored to the academic environment. It focuses on providing a "streamlined" experience, offering sufficient functionality to manage university projects effectively without the unnecessary complexity of industrial tools.

1.1.2. Thesis Significance

In the current academic landscape, students often face the challenge of "tool fragmentation." To complete a single group project, teams frequently rely on a disparate ecosystem: Google Drive for storage, separate applications for task assignment and risk management, distinct forums for communication, and unconnected personal calendars. This thesis addresses these inefficiencies by consolidating these functions into a single ecosystem.

- **The significance of the system lies in its ability to:**
    - **Reduce Context Switching:** By integrating a private calendar with a team calendar and task board, students can manage their workload without constantly switching between applications.
    - **Enhance Collaboration:** The system provides a dedicated space for assigning tasks, arranging workflows, and sharing knowledge, directly supporting students in developing their teamwork skills and career trajectory.
    - **Optimize Learning:** By centering the application on the user's learning goals and career development, the system supports a more structured and effective educational process.

1.1.3. Thesis Scope

- **Platform:** The project is developed exclusively as a Web Application.

- **Target Audience:** The system is designed for university students across various disciplines who need to manage personal studies and collaborate on group projects.
- **Functional Boundaries:** The application focuses on the core needs of student life:
    - **Task & Project Management:** Creating, assigning, and tracking the status of academic tasks.
    - **Resource Management:** A simplified document storage and sharing system similar to a dedicated drive.
    - **Scheduling:** Integrated personal and team calendars to manage deadlines and meetings.
    - **Community:** A community space for Q&A and knowledge exchange.

1.1.4. Problems to be Solved

The core problem identified is the critical **fragmentation of the digital academic ecosystem**. Currently, there is a distinct lack of a unified solution that adequately covers the full lifecycle of a university student's academic and personal progression. Students are forced to manually bridge the gaps between disparate platforms, leading to specific operational inefficiencies:

- **Resource and Context Isolation:** Storage solutions (e.g., Google Drive) are often disconnected from task management workflows (e.g., Jira or Trello). This separation means that the deliverables and documentation required to complete a task are not immediately accessible within the task's context, leading to information silos and time wasted searching for assets.
- **Temporal Conflicts and Well-being Neglect:** Standard planning tools fail to integrate **personal constraints** with **collaborative obligations**. Industrial tools do not account for a student's personal well-being, such as necessary sleep hours or daily workload limits. This results in unrealistic planning where group deadlines are set without visibility into a member's actual availability or health requirements.
- **Fragmentation of Knowledge:** Academic knowledge exchange often occurs on external platforms (e.g., StackOverflow, Facebook groups) that are isolated from the actual project workspace. Valuable solutions and discussions regarding specific project challenges are lost in external chat logs or forum threads, rather than being preserved as a knowledge base linked to the project itself.

- **Operational Complexity:** Existing industrial-grade project management tools are often overly complex for the academic environment. They offer excessive features that clutter the user experience, while lacking integration with a private planning calendar to create a comprehensive scheduling function.

1.1.5. Proposed Solutions

To address the user needs described above, the thesis proposes a user-centric solution built upon a Microservices Architecture. This architectural choice ensures that the system is modular, maintainable, and capable of handling diverse functionalities—such as scheduling, content management, and forums—independently.

The proposed technical solution includes:

- **Architecture:** A Microservices approach allows the application to be scalable and flexible. It enables the independent development and deployment of distinct modules, ensuring that a failure in one service (e.g., Document Service) does not compromise the availability of others (e.g., Scheduling Service).
- **Backend:** Developed using **Java (Spring Boot)** to ensure robust logic processing, type safety, and seamless integration with the Spring ecosystem.
- **Frontend:** Built with **Next.js** to provide a fast, server-side rendered user interface that optimizes performance and search engine visibility.
- **Database:** A polyglot persistence strategy is employed, using **PostgreSQL** for structured relational data (users, project structure) and **MongoDB** for unstructured data (documents, forum content), alongside **Redis** for high-performance caching.
- **Message Broker: Apache Kafka** is integrated to manage asynchronous inter-service communication. It specifically serves as the backbone for the **Notification Service**, effectively handling the high throughput of system events—such as project invitations, task assignments, and forum replies—generated by services. This ensures that data is propagated reliably to the user without tightly coupling the services.

1.2. Technology Stack

1.2.1. Front-end

- **HTML (HyperText Markup Language)**

- **Structural Foundation:** HTML serves as the skeleton of any web application, defining the fundamental structure and organization of content on the interface.
- **Element Definition:** It utilizes a system of "tags" (e.g., <header>, , <table>) to categorize content types, distinguishing between headings, paragraphs, lists, and data containers.
- **Semantic Integrity:** Modern HTML emphasizes semantic markup, ensuring that elements convey meaning not just for display, but for accessibility tools (screen readers) and search engine optimization (SEO).

- **CSS (Cascading Style Sheets)**
  - **Presentation Layer:** CSS is responsible for the visual aesthetics of the application, separating the design (colors, fonts, spacing) from the structural content defined by HTML.
  - **Layout Management:** It provides powerful layout systems (such as Flexbox and Grid) that control the positioning of elements, enabling complex designs like the project's Kanban boards or Calendar grids.
  - **Responsiveness:** CSS facilitates "Responsive Web Design," ensuring that the application adapts fluidly to different screen sizes and devices, from desktop monitors to mobile phones.
- **Tailwind CSS (Utility-First CSS Framework)**
  - **Definition**: Tailwind CSS is a utility-first CSS framework that provides low-level utility classes that can be composed to build any design, directly in the HTML markup.
  - **Rapid Prototyping**: It allows for faster development by using predefined classes (e.g., flex, pt-4, text-center) instead of writing custom CSS rules from scratch.
  - **Design Consistency**: By utilizing a predefined design system (spacing scales, color palettes), Tailwind ensures visual consistency across the entire application interface.
  - **Application in System**: Tailwind is used to style the project's complex UI components, such as the document editor, task cards, and navigation menus, while maintaining a small CSS bundle size through the removal of unused styles during production.
- **JavaScript**

- **Interactivity and Behavior:** JavaScript is the programming language that breathes life into the web page, transforming static content into a dynamic and interactive experience.
- **Logic Execution:** It handles client-side logic, such as validating user inputs (e.g., checking if a password meets criteria) before sending data to the backend server.
- **DOM Manipulation:** In the context of modern frameworks like Next.js (used in your project), JavaScript manages the Document Object Model (DOM), allowing for real-time updates—such as dragging a task card or receiving a notification—without requiring a full page reload.

- **Next.js (React Framework)**
  - **Definition:** Next.js is a robust web development framework built on top of the React JavaScript library. It extends React's capabilities by providing production-grade features such as hybrid static and server rendering.
  - **Key Features:**
    - **Server-Side Rendering (SSR):** Unlike standard Client-Side Rendering, Next.js can pre-render pages on the server. This significantly improves the initial load performance and search engine optimization (SEO), which is crucial for a knowledge-sharing platform.
    - **Component-Based Architecture:** Inherited from React, this allows the interface to be broken down into reusable, independent components (e.g., navigation bars, task cards), enhancing code maintainability.
  - **Application in System:** Next.js serves as the primary view layer. It is responsible for "purely display" logic: receiving data from the backend APIs and rendering the user interface for features such as the calendar view and document editor, without processing complex business rules.

- **RxJS (Reactive Extensions for JavaScript)**
  - **Definition:** RxJS is a library for reactive programming using Observables, creating an event-based architecture for managing asynchronous data calls.
  - **Key Features:**

    - **Observables:** Unlike standard Promises, Observables treat data as a stream that can be observed over time. This allows the application to react instantly when API data arrives.
    - **Operators:** RxJS uses operators to manipulate data streams before they reach the component. For example, the .pipe(map(...)) function transforms raw API responses into a usable format cleanly and consistently.
- **Application in System:** RxJS is utilized within the Repository Pattern. It acts as the bridge between the frontend components and the backend API, managing HTTP requests (GET, POST, PUT, DELETE) and ensuring the UI remains responsive while waiting for server operations to complete.

1.2.2. Back-end

- **Java**
    - **Object-Oriented Paradigm:** Java is strictly built upon the Object-Oriented Programming (OOP) model, organizing software design around data, or objects, rather than functions and logic. This promotes modularity, code reusability, and easier maintenance of complex systems.
    - **Platform Independence:** Operating on the "Write Once, Run Anywhere" (WORA) principle, Java code is compiled into bytecode that runs on the Java Virtual Machine (JVM). This ensures that the backend application is portable across different operating systems and hardware environments without modification.
    - **Strong Typing and Robustness:** Java enforces strict static typing and utilizes automatic memory management (Garbage Collection). These features mitigate common programming errors, such as memory leaks and type mismatches, ensuring high stability and reliability for critical backend services.
- **Spring Boot**
    - **Definition:** Spring Boot is a robust backend development framework built on top of the Java Spring ecosystem. It extends the standard Spring capabilities by providing production-grade features such as convention-over-configuration, automatic dependency management, and embedded server execution.
    - **Key Features:**

    - **Dependency Injection (DI):** Promotes loose coupling between classes, making the application easier to test and manage.
    - **Spring Security:** Provides a powerful and highly customizable authentication and access-control framework, ensuring secure login and protection against common vulnerabilities.
    - **Spring Data JPA:** Simplifies the data access layer by abstracting the interaction with the database.
    - **Spring Data MongoDB:** Streamlines interaction with document-oriented databases by providing a repository abstraction similar to JPA. It maps Java objects directly to JSON-like documents, enabling flexible data storage for unstructured content like forum posts and rich-text files.
- **Application in System:** Java is used to build the core Microservices. It handles all business logic, including task scheduling algorithms and permission validation. The system acts as a set of independent services exposed via RESTful APIs to communicate with the frontend.

1.2.3. Database

The system uses different database technologies to handle specific data types efficiently.

- **PostgreSQL**
    - **Definition:** An advanced, open-source object-relational database management system (ORDBMS).
    - **Application in System:** Used for storing highly structured and relational data that requires strict consistency (ACID compliance). This includes User Identities, Task assignments, Calendar schedules,... where data integrity and complex relationships are paramount.
- **MongoDB**
    - **Definition:** A NoSQL database that stores data in flexible, JSON-like documents.
    - **Application in System:** Utilized for managing unstructured or semi-structured data, which may vary in format. Specifically, it

stores Documents and Forum posts, allowing for rapid iteration of content schemas without affecting the core relational logic.

- **Redis**
    - **Definition:** An open-source, in-memory data structure store.
    - **Application in System:** Acts as a high-performance caching layer and session store. By keeping frequently accessed data (such as active user sessions or real-time notification states) in memory, Redis significantly reduces latency and improves the overall responsiveness of the application.

1.2.4. External Services & Integrations

To enhance system scalability and user convenience, the application integrates with robust third-party services provided by the Google Cloud ecosystem.

- **Google OAuth 2.0 (Identity Provider)**
    - **Definition:** The system leverages Google OAuth 2.0 as an external identity provider to facilitate secure and streamlined user access.
    - **Application in System:**
        - **Federated Identity:** Instead of only handling the storage of sensitive credentials (such as passwords) for every user, the system also delegates the initial authentication process to Google. This allows students to log in using their existing institutional or personal Google accounts.
        - **User Experience:** This integration removes the friction of creating new account credentials, thereby reducing entry barriers and enhancing the overall user experience (UX) for the users.
- **Google Cloud Storage (Object Storage)**
    - **Definition:** Google Cloud Storage is an enterprise-grade service for storing and accessing unstructured data.
    - **Application in System:**
        - **Asset Management:** The system utilizes Google Cloud Storage buckets to handle the persistence of static assets.
        - **Scalability:** By offloading files—such as user avatars, shared project source—to a dedicated cloud storage provider, the main application server remains lightweight and performant. This separation of concerns ensures that the application can

handle increased data loads without degrading the performance of the core backend services.

1.2.5. Real-time Collaboration Server with Hocuspocus

a) Overview

Hocuspocus is an extensible, opinionated WebSocket backend designed specifically for real-time collaboration. Built by the Tiptap team, it acts as a plug-and-play server for Yjs, a high-performance CRDT (Conflict-free Replicated Data Type) framework. It enables multiple users to edit the same document simultaneously while ensuring data consistency across all clients without the need for complex manual conflict resolution.

b) Main features

- **Conflict-Free Data Sync (CRDTs):** Leverages Yjs to merge changes from different users automatically. This ensures that even if users edit the same line or work offline and reconnect later, the document state remains consistent across all devices without data loss.
- **Extensible Lifecycle Hooks:** Provides a robust system of "Hooks" (such as onConnect, onAuthenticate, and onStoreDocument) that allow developers to intercept server events. This makes it easy to integrate custom logic for authentication, authorization, and logging into an existing infrastructure.
- **Pluggable Persistence:** Offers flexible storage options through official extensions. While it keeps the active document in memory for speed, it can asynchronously persist data to various databases like PostgreSQL, MongoDB, SQLite, or even S3 buckets, ensuring that collaborative sessions are saved for future use.
- **Awareness & Presence:** Facilitates "Awareness" features that track user presence in real-time. This allows developers to implement UI elements like shared cursors, live typing indicators, and "who is online" lists, significantly enhancing the collaborative user experience.
- **Framework Agnostic**: Although developed by the Tiptap team, Hocuspocus is compatible with any editor or application that uses Yjs, including ProseMirror, Slate, Quill, and Monaco, making it a versatile choice for diverse full-stack projects.

c) Advantages and Technical Limitations

- **System Advantages:**

- **Development Efficiency:** Building a custom real-time sync engine from scratch is an immense undertaking; leveraging Hocuspocus significantly reduces development time and infrastructure costs.
- **Conflict Resolution:** By utilizing CRDTs, the system naturally resolves "edit conflicts" (e.g., two users typing in the same paragraph) without the need for complex server-side locking mechanisms.

- **Technical Limitations and Challenges:**
  - **Limited Library Support:** Integrating real-time collaborative indicators—such as shared cursors and presence markers—within existing interface frameworks remains challenging as high-level library support is still evolving.
  - **Data Structure Integrity:** Maintaining consistency during the transformation of complex document structures between the client-side editor and the collaboration server is critical. Any discrepancy in schema interpretation during real-time data exchange can lead to structural malformation or the loss of document information.

1.2.6. System Architecture

a) Overview

The system is constructed using a microservices architectural style, which structures the application as a suite of loosely coupled, independently deployable services. Unlike traditional architectures, each service in this model is organized around a specific business capability—such as User Management, Scheduling, or Project Collaboration—and operates in its own dedicated process. This paradigm facilitates continuous delivery and deployment, allowing individual modules to be developed, tested, and scaled without impacting the integrity of the entire ecosystem.

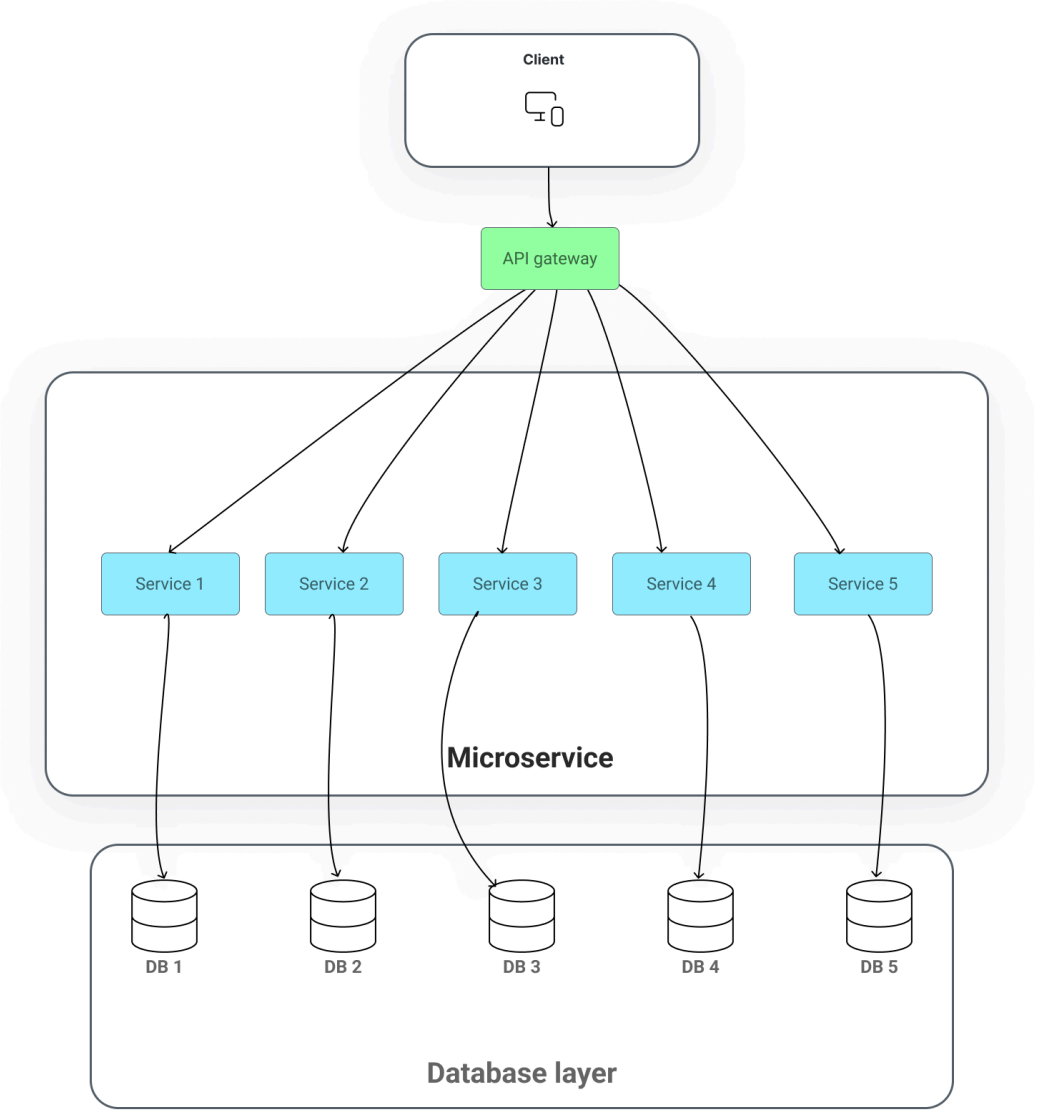

Image 1.1. Microservice architecture

b) Comparative Analysis: Monolithic vs. Microservices

The following table synthesizes the fundamental differences between these two architectural paradigms, highlighting why a decentralized approach is superior for the "Collaborative System to Support Learning."

Table 1.1. Monolithic and Microservice architecture comparison

Feature	**Monolithic Architecture**	**Microservices Architecture (The System)**

Structure & Deployment	All modules (User, Project, Forum, etc.) are bundled into a single codebase and a single executable unit.	The application is decomposed into independent services (Scheduling, Notification, etc.) running in distinct processes.
Coupling & Agility	**Tight Coupling:** A minor update to the Forum's comment logic requires a full rebuild and redeployment of the entire platform.	**Loose Coupling:** The **Scheduling Service** can be updated with new health-balance algorithms without affecting the **User Service**.
Fault Tolerance	**Single Point of Failure:** A memory leak or fatal error in the Notification module can crash the entire application process.	**Failure Isolation:** If the Forum Service fails, the core Project Service and Scheduling Service remain fully operational for students.
Scalability	**Uniform Scaling:** The entire application must be replicated to handle increased load, wasting resources on low-traffic modules.	**Granular Scaling:** The Notification Service can be scaled horizontally to handle high-throughput Kafka traffic without scaling the User Service.
Technology Stack	**Homogeneous:** Usually restricted to a single programming language and framework for the entire system.	**Heterogeneous:** Allows the use of different programming languages for different modules.

Data Management	**Centralized Database:** All modules share a single schema, often leading to complex, rigid relational structures.	**Polyglot Persistence:** Each service manages its own data; Project Service uses PostgreSQL for integrity, while Document Service uses MongoDB for flexibility.
Development Speed	Slows down as the codebase grows; teams often interfere with each other's code, leading to merge conflicts.	Accelerated development; small teams can own and iterate on specific services (e.g., the Forum service) independently.

- **Analysis of Selection:**

  As demonstrated in the comparison, the Monolithic model is insufficient for a modern collaborative platform. The primary driver for selecting **Microservices** for this project is the need for **system resilience** and **independent scalability**.

  Specifically, because "The System" integrates high-frequency real-time events (via the Notification Service and Kafka) alongside critical academic document management, the architecture must ensure that a surge in community interaction on the Forum does not impede a student's ability to edit their graduation project files. Furthermore, the microservices model accommodates the diverse technical requirements of the platform, enabling the seamless integration of both relational data and unstructured collaborative content.

c) Advantages

- **Scalability:** Microservices allow for precise scaling. Services experiencing high demand can be scaled horizontally without the need to replicate the entire system. This granular approach optimizes resource allocation and ensures cost-efficiency.
- **Fault Tolerance:** The architecture provides strong isolation. If a specific service encounters a failure, it does not necessarily cause a total system outage, thereby preserving the availability of core functions and maintaining user experience.

- **Flexibility:** Services can be developed using different programming languages or technologies best suited for their specific tasks. This decoupling allows for the integration of new technologies without refactoring the entire application.
- **Organizational Efficiency:** The architecture supports a decentralized governance model. Small, cross-functional teams can own specific services, reducing dependency conflicts and accelerating the development lifecycle.

d) Disadvantages

- **Operational Complexity:** Decomposing a monolithic system into services increases complexity. It requires sophisticated mechanisms for inter-service communication (via APIs or Message Queues), global data consistency, and robust monitoring.
- **Infrastructure Costs:** Running multiple independent services often requires significant overhead in terms of infrastructure management and deployment automation compared to a single monolithic application.

1.2.7. Event Streaming with Apache Kafka

a) Overview

Apache Kafka is an open-source event streaming platform designed for high-performance data pipelines and asynchronous communication. In this project, it serves as the messaging backbone that decouples core services, allowing them to interact without direct dependencies. By treating data as a continuous stream of events, Kafka ensures that system-wide notifications and updates are handled reliably and at scale.

b) Architecture and Core Components

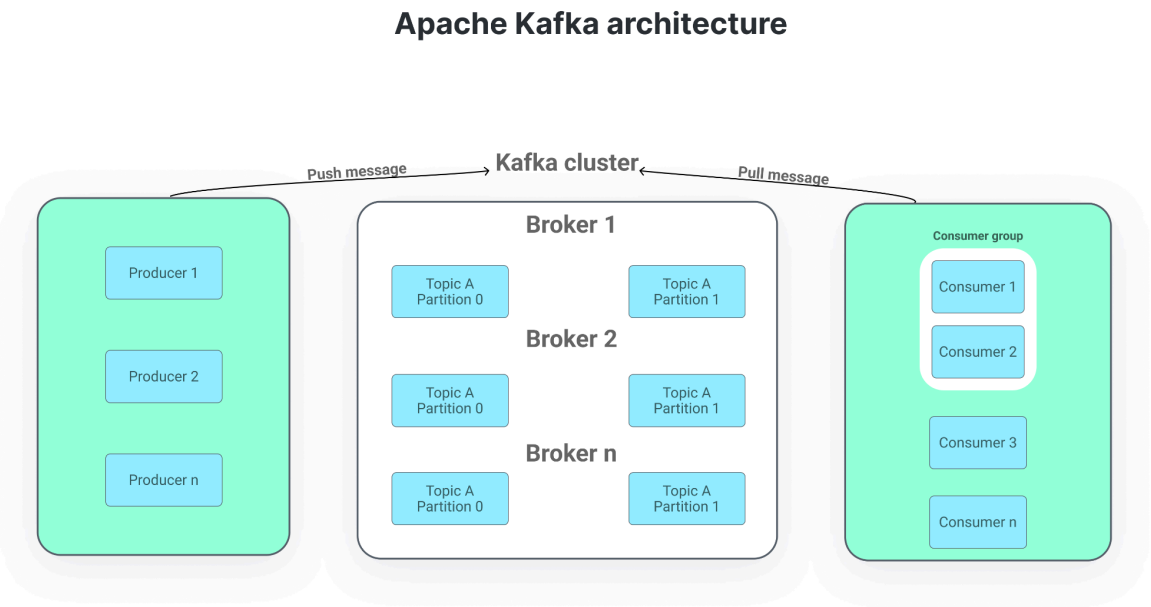

Image 1.2. Apache Kafka architecture

The Kafka infrastructure utilizes a distributed architecture to provide high-throughput, fault-tolerant data streaming through the following core components:

- **Kafka Brokers:** These server instances function as the cluster's physical layer, managing data persistence, replication for high availability, and metadata synchronization. They act as the primary gateway for client communication and ensure cluster-wide data consistency.
- **Topics and Partitions:** Data is logically organized into **Topics**, which are further divided into **Partitions** to enable parallelism. This structure allows for horizontal scaling across brokers. **Key-based partitioning** is employed to guarantee strict message ordering for specific entities, such as User or Project IDs.
- **Producers:** Independent microservices that publish event data to specific topics. They define the partitioning logic to ensure data is routed correctly within the cluster.
- **Consumers and Consumer Groups:** Subscriber services that retrieve data from topics. By utilizing **Consumer Groups**, the system achieves efficient load balancing, ensuring that each message is processed by only one consumer within a group while allowing for simultaneous processing across multiple partitions.

c, Advantages and Limitations

- **System Advantages:**

- **Scalability:** Kafka's partitioned architecture allows the system to handle millions of messages, supporting future user growth.
- **Fault Tolerance:** Persistent storage and data replication ensure that events are not lost during service interruptions.

- **System Limitations:**
  - **Operational Complexity:** Maintaining a distributed message broker increases infrastructure overhead and deployment difficulty.
  - **Idempotency Challenges:** Kafka's "at-least-once" delivery model can lead to duplicate message processing. To mitigate this, the system implements a **processedMessage** tracking mechanism, where unique message IDs are stored and verified to ensure each notification is delivered exactly once.

d) System Implementation and Justification

The implementation of Kafka in this project is driven by three primary strategic needs:

- **Decoupling of Services:** By using Kafka, the **Notification Service** remains independent from the services generating events. This ensures that if the Notification Service is temporarily down, messages are queued in Kafka and processed once the service resumes.
- **Future Scalability:** Compared to simpler brokers like RabbitMQ, Kafka is better suited for a high-growth environment where the Notification Service must aggregate massive amounts of data from four different core services.
- **Asynchronous Communication:** Kafka handles the high throughput of cross-service events—such as project task assigned notifications from the **Project Service** or replies from the **Forum Service**—without blocking the main application flow.
- **Resilience via Dead Letter Queues:** Implementing a Dead Letter Queue (DLQ) in the Notification Service prevents transient errors from causing cascading failures. This mechanism acts as a buffer for problematic data—like invalid project IDs or malformed payloads—ensuring high availability while isolating errors for later debugging.

1.3. Chapter 1 Conclusion

Chapter 1 provides a comprehensive overview of the Knowledge Sharing and Learning Support System, which is designed using a robust technology stack comprising Java (Spring Boot) and Next.js. The proposed system not only fulfills the essential needs of individual students—such as personal scheduling, task management, and document storage—but also facilitates effective teamwork through collaboration and shared resource management.

By adopting a Microservices Architecture and integrating modern solutions like Google OAuth 2.0 and a polyglot database strategy (PostgreSQL, MongoDB, Redis), the system addresses the critical issue of tool fragmentation in the academic environment. These technologies ensure that the platform is scalable, secure, and user-centric. The fundamental concepts and architectural decisions presented in this chapter serve as the solid foundation for the detailed system analysis and design that will be developed in the subsequent chapters.

CHAPTER 2: SYSTEM ANALYSIS AND DESIGN

2.1. Actor Description, Terminology, and Requirement Identification

2.1.1. Identify and describe actors

Table 2.1. Actor identity

No.	Actor Name	Description	Interaction System
1	**Student**	The primary user of the system. Students can register accounts, manage personal profiles, and utilize core features such as the personal calendar (creating tasks, routines, events), month planning, and resource management.	Web Application
2	**Project Owner**	A specific role assumed by a Student when they create a new project. The Project Owner has full control over the project features.	Project Management Module
3	**Project Member**	A Student who is invited to join a project. They have "Read-only" access to the project structure (Deliverables, Phases, Timeline) but can actively collaborate by updating task statuses (e.g., "To Do" to "Done"), sharing and editing files, and creating risks.	Project Management Module
4	**Post creator**	A student who posts a question. They can edit, comment, answer, or delete (with conditions) with their posts	Forum Module

5	**Post respondent**	A student who can interact with posts: like posting an answer, commenting on a post, editing, deleting their comments, or answers.	Forum Module

2.1.2. Glossary of Terms

Purpose: To clarify the specific terminology used within domains of the system, ensuring consistent understanding throughout the software development process.

- **Selected terms used in the system:**

Table 2.2. Terms used in the system

No.	Term	Explanation
1	**Student**	Represents the main user of the system who interacts with calendar features, collaborative projects, and the forum community.
2	**Routine**	A recurring activity (e.g., "Morning Exercise") that repeats on specific days of the week. Unlike tasks, routines do not have a "completed" status but serve to block out time in the calendar.
3	**Calendar Task**	An actionable item with a specific deadline and status (Incomplete/Complete). It can contain subtasks.
4	**Event**	A specific occurrence scheduled for a distinct date and time range (e.g., "Team Meeting") that does not necessarily recur or require completion tracking.
5	**Calendar Item**	A term to talk about routine, task, or event generally.
6	**Month Plan**	A high-level planning entity where students define their "Big Tasks", events, and routines (only name) for the upcoming months.

7	**Big Task**	A major goal set during Month Planning (e.g., "Complete Thesis"). It acts as a parent container for smaller, actionable tasks scheduled in the weekly view.
8	**Project**	A collaborative workspace where a group of students works together. It contains members, milestones, a structured hierarchy of work, risk management, file collaboration, and a Kanban board.
9	**Deliverable**	The highest level of work hierarchy in a project. It represents a major output (e.g., "Frontend Interface") and contains multiple Phases.
10	**Phase**	A subset of a Deliverable (e.g., "Design Phase"). Phases are used to group specific Project Tasks.
11	**Task**	The smallest unit of work in a project containing several fields of information, such as priority, file attachments,...
12	**Risk**	A potential issue identified in a project (e.g., "Server Crash"). It is tracked by probability and impact levels to allow for mitigation planning.
13	**Unscheduled List**	A sidebar buffer containing tasks or routines that have been defined (e.g., in the Month Plan and Projects) but have not yet been assigned a specific time slot on the calendar.
14	**Kanban Board**	A visual interface in the Project module that allows members to manage task progress by dragging cards between status columns (To Do, In Progress, Done).
15	**Memorable Event**	Special annual events (e.g., Birthdays, Anniversaries) that are configured in settings and auto-generated into the calendar for future years.
16	**File**	Members upload static files from the local machine to the shared source of the project.

17	**Document**	A document type is created by members supporting real-time collaborative editing.
18	**Folder**	A component in the shared source of a project created to contain other folders, files, and documents.
19	**Post**	A post or question that is created by a student contains rich documents such as files, images,... to ask other students to help.
20	**Answer**	A solution in the post is created by students to help the student who created the post solve the issue.
21	**Comment**	A text can be created inside both the post and the answer by students who want to talk about the post and the answer.

2.1.3. Identify requirements

❖ **Functional requirements**

The system should handle the following functions:

a) User Management Module

- Register, Login (via Email or Google OAuth), Logout.
- Reset Password, Change Password, Verify Email.
- Manage Personal Profile: Update name, date of birth, and avatar image.
- Manage Calendar Settings: Configure Sleep Hours, Daily Work Limits, and Timezone.
- Manage Memorable Events: Set up recurring annual events (e.g., birthdays).

b) Personal Scheduling Module

- **Calendar View:**
  - View calendar by Day, Week, Month, Year.
  - Create, Update, and Delete Calendar Items (Tasks, Routines, Events).

- Categorize items by type with specific attributes (e.g., Tasks have subtasks, Routines have patterns).

- **Month Planning :**
  - Create a Month Plan: Define "Big Tasks", events for the month.
  - Schedule from Plan: Drag "Unscheduled" items from the sidebar to the calendar grid.
- **Sidebar Buffer:**
  - View unscheduled tasks and routines grouped by month.

c) Project Management Module

- **Project Workspace:**
  - Create, Edit, and Delete Projects.
  - Manage Members: Invite users via emails, notifications, assign roles, and remove members.
- **Scope & Timeline:**
  - Manage Deliverables and Phases.
  - Manage Project Tasks: Create tasks, assign members, set priorities, and change status.
  - View Timeline (Gantt Chart): Visualize project progress and dependencies.
- **Execution & Monitoring:**
  - Kanban Board: Drag and drop tasks to update status (To Do -> Done).
  - Risk Management: Identify risks, assess impact/probability, and define mitigation plans.
  - Dashboard Summary: View task statistics, active risks, and teammate workload.
- **Shared source**:
  - Organizing source: Upload files, create a folder, and create a document.
  - Can real-time edit a document with a simple function simulating a Notion page.

d) Forum Module

- **Forum Interaction:**
  - **Post Management:** Create, edit, and delete questions (with conditions) and tag categorization.

- **Response System:** Post answers to questions, with the ability for the post owner to mark a specific answer as the "Accepted Solution".
- **Comment System:** Add text-based comments or threaded replies to both posts and answers to facilitate discussion.
- **Feedback & Scoring:** Upvote or downvote posts and answers to determine helpfulness and visibility within the community feed.

- **Discovery & Personalization:**
  - **Feed & Search:** Browse a global feed with support for full-text search, tag filtering, and sorting by newest or most helpful posts.
  - **Resource Saving:** Save posts privately for personal reference or link them directly to a specific **Project Workspace** for team knowledge sharing.
  - **Tagging System:** Use an autocomplete search to find existing tags or create new technology tags explicitly.
- **Integration & Utilities:**
  - **User Interaction:** View user "hover cards" to see profiles and quickly invite other students to join your owned projects.

e) Supporting Modules (Notification service)

- **Automated Business Triggers:** Notifications are automatically generated based on specific business events across the platform, such as a project invitation from the Project Management Module.
- **User-Centric Notification Management**
  - **Smart Inbox:** Provides a paginated view where users can toggle between an "Inbox" (unread) and "All" history to manage their daily alerts.
  - **Interaction Logic:** Users can mark individual items as read, "Mark all as read" to clear their dashboard, or toggle a "Mark unread" status for items they wish to revisit later.
  - **History Cleanup:** Allows users to hard-delete single alerts or perform a bulk "Clear read notifications" to maintain an organized workspace.
  - **Support workflow:** Depending on each type of notification, it not only provides information, but it can also let users interact with the system instantly (e.g., clicking a "Task

Assigned" notification takes the user directly to that specific task on the List of tasks page).

❖ **Non-functional requirements**

Table 2.3. Non-functional requirements

Non-functional Requirement	Description	Example
Performance	System response time and data loading speed.	- "Lazy loading" is used for project tasks to ensure fast initial page load.
Constraint Enforcement	The system must enforce logic rules to prevent user error.	- The system prevents scheduling tasks during configured "Sleep Hours". - The system validates that daily work limits are not exceeded.
Security	Access control based on roles.	- Only "Project Owners" can implement full functions; "Members" have certain functions. - Authentication required for all API access. (Except for certain APIs such as: Oauth2, forget password,...)

2.2. Use Case Identification and Description

2.2.1. General system use case diagram

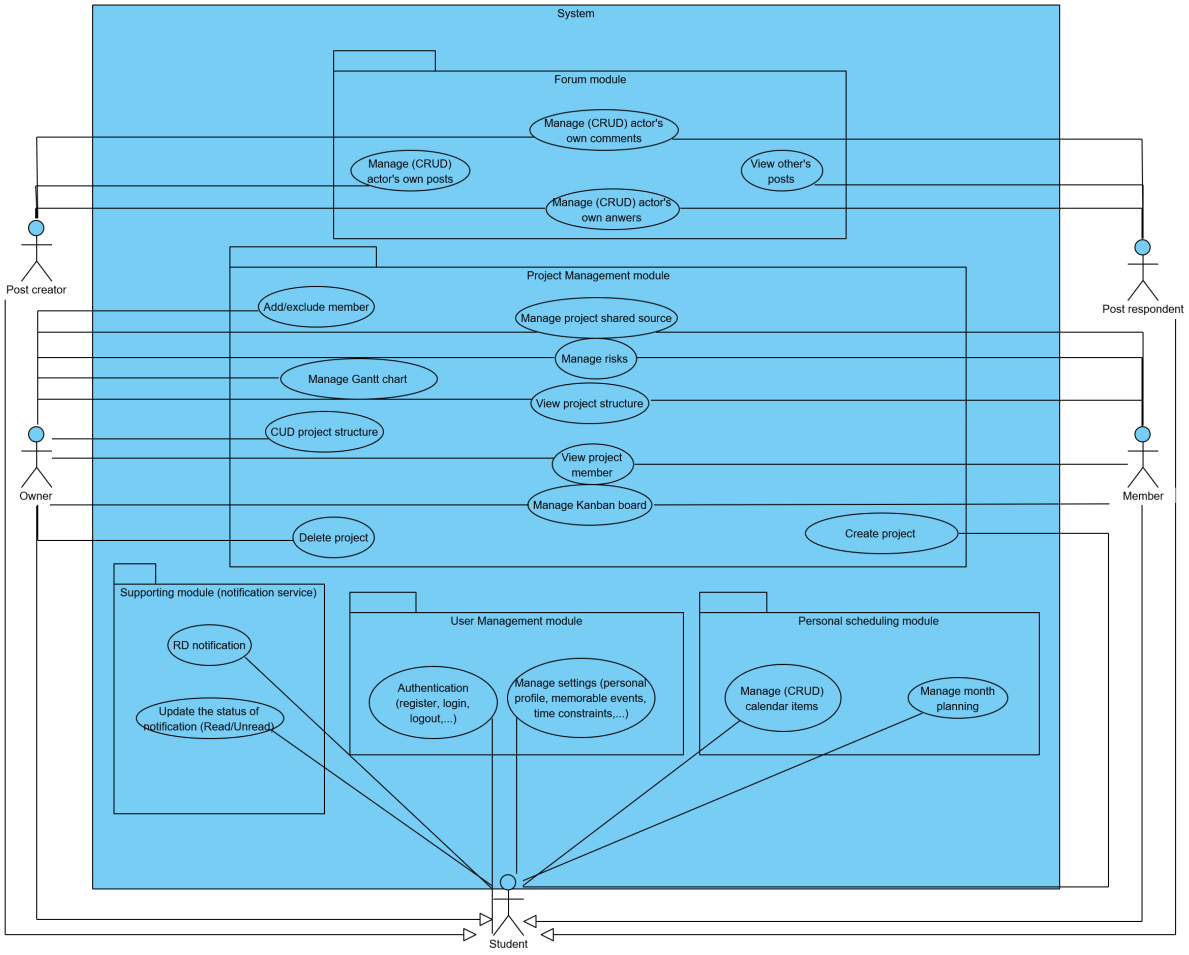

Image 2.1. General system use case diagram

- **NOTE:**
  - 'Manage project shared source' UC mentioned here is high-level. This UC depends on each type of file and who uploaded it to decide that a member of a project will have specific permissions. The details UC will be illustrated in the next part of the report.
  - Post creator can be a Post respondent.

2.2.2. Detailed use cases

Due to the large number of UCs and the report's length constraints, basic CRUD operations have been consolidated into unified 'Management' UCs. This report also only demonstrates the UCs that are the most complex to better illustrate the system's core logic.

- **Month plan management (Personal Scheduling Module):**

- **Description:** This UC allows students to define high-level tasks (big tasks), events, and routines (only names) for the upcoming months.

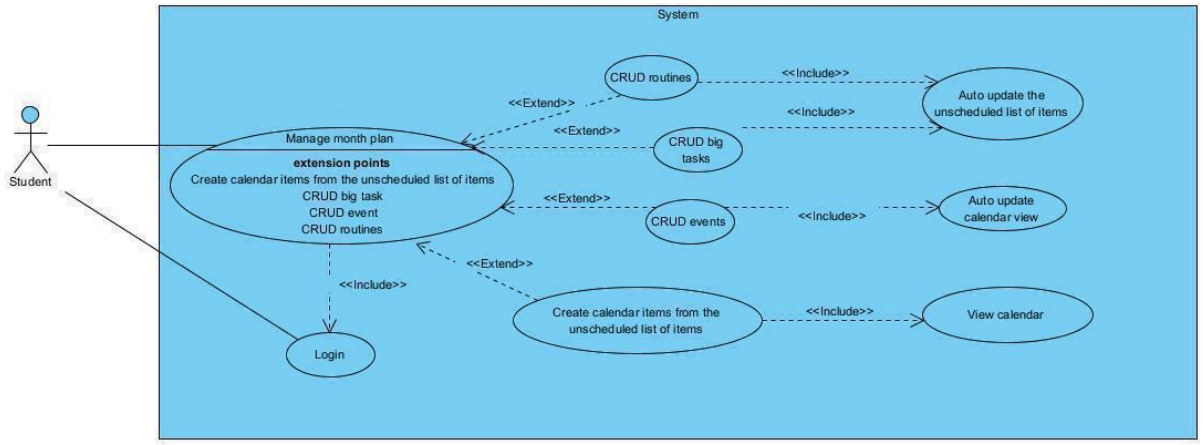

Image 2.2. Detailed UC of Month plan management

- **Team member management (Project Management Module):**
  - **Description:** This is implemented by the project owner who created the project, supporting the owner and managing their members in the project.

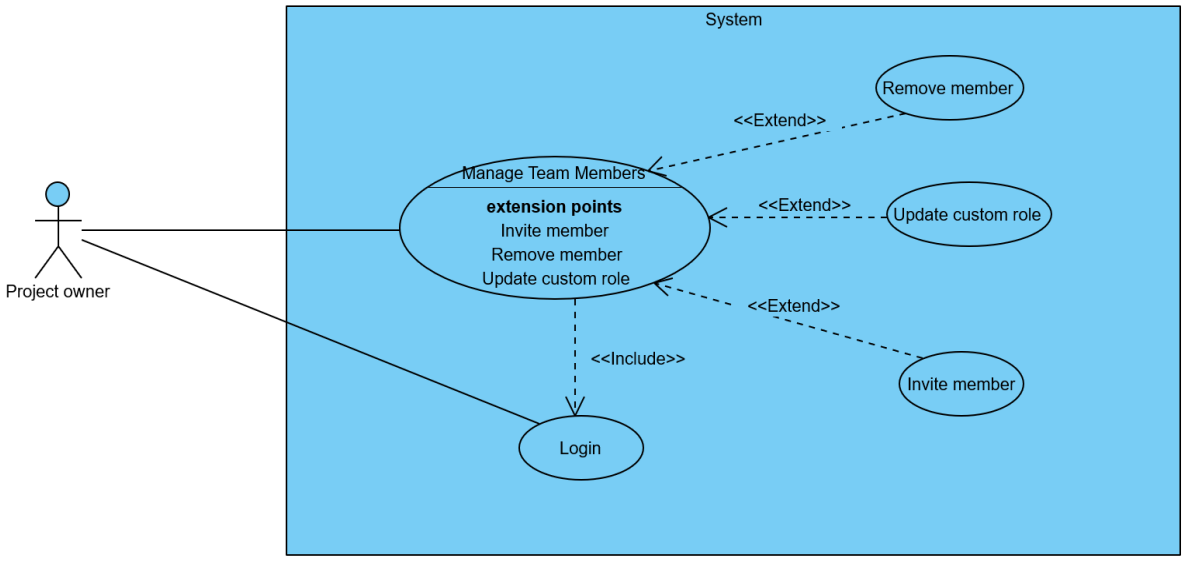

Image 2.3. Detailed UC of Team member management

- **Project task structure management (Project Management Module):**
  - **Description:** Defining the hierarchy of work. Only the Owner can create Deliverables and Phases.

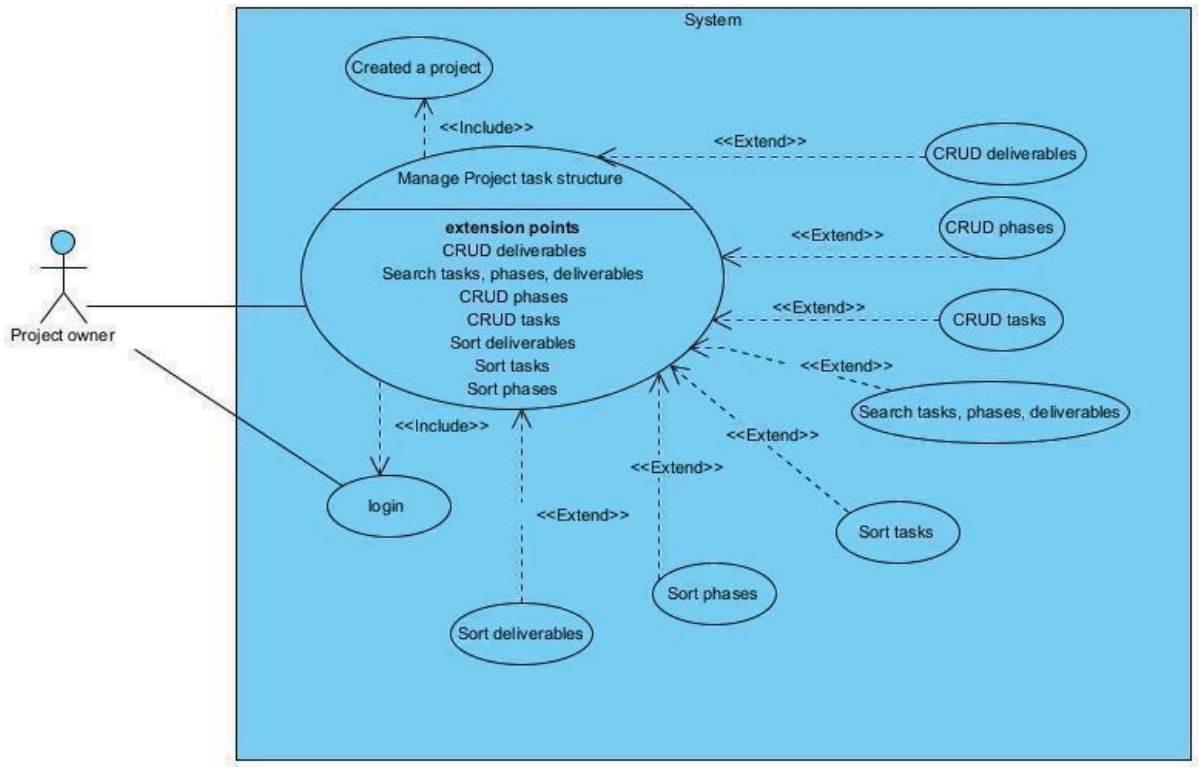

Image 2.4. Detailed UC of Project task structure management

- **Note:** When the project owner creates a task, it must belong to an existing phase. Similarly, the phase must belong to an existing deliverable.
- **Document management (Project Management Module):**
    - **Description:** Showing the functionalities related to documents such as creating, reverting the document to the previous versions,…

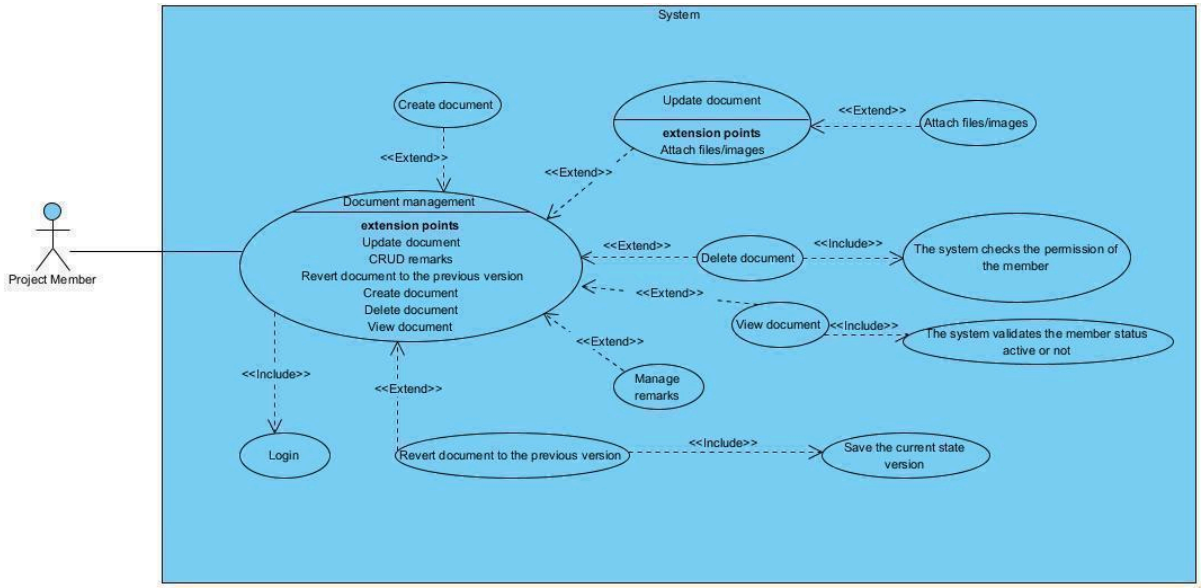

Image 2.5. Detailed UC of Document management

- **Note:** The system automatically saves the versions of documents every 30 minutes or when the last users leave the document.

- **Post management (Forum Module):**
  - **Description:** Facilitating academic exchange through a structured Q&A and discussion platform.

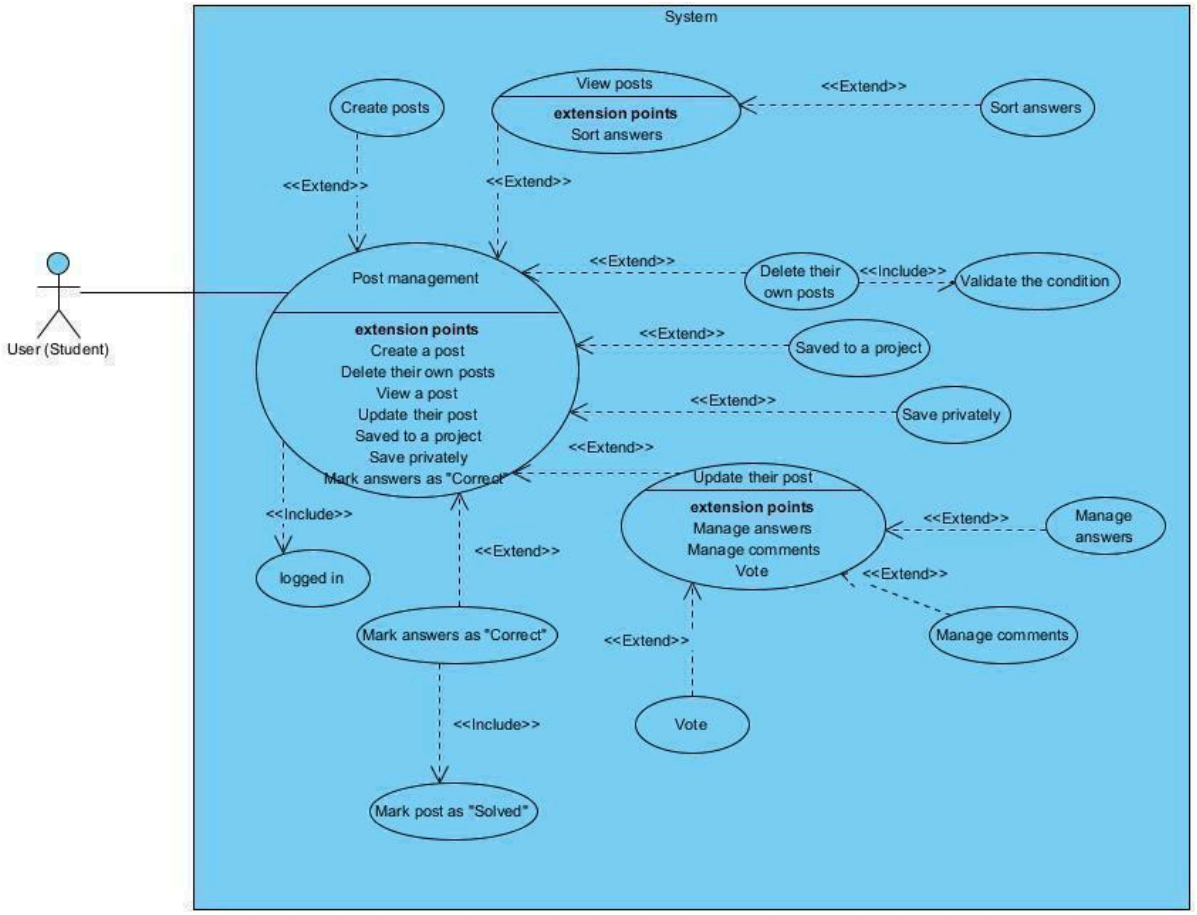

Image 2.6. Detailed UC of Post management

- **Note:** The 'condition' will be demonstrated explicitly in the next part.

2.2.3. Main scenarios

- **Month plan management**

Table 2.4. Scenario for the Month plan management UC

Actor	Student
Precondition	The Student has successfully logged into the system and navigated to the Month Planning interface.
Post-condition	The month plan is updated and saved to the database.

Main Scenario	1. The Student selects a target month for planning (ranging from the current month up to 6 months in the future). 2. The Student initiates the creation of a new item by selecting one of three categories: **Big Task**, **Event**, or **Routine**. 3. The Student provides the following details based on the item type: ● **Big Tasks:** Title, start/end date range, and optional notes. ● **Events:** Title, specific date, start/end hours, and optional notes. ● **Routines:** Routine name. 4. The Student clicks **Save**. The system validates the input, persists the data, and updates the "Unscheduled Items" list (if it is a routine or a big task item), or calendar views (if it is an event item). 5. The Student navigates to the Calendar View (selectable by Day, Week, Month, or Year) and focuses on the target month. 6. The student opens the unscheduled item list and drags items onto the calendar grid to assign specific dates and times. 7. The system runs background constraint checks (e.g., time conflicts) and saves the scheduled state successfully 8. ● To edit, the student selects an item to view its details, modifies the data, and saves changes. ● To remove, the student selects the delete option; the system removes the record and reflects the change in the calendar view.
Exceptions	4.1. If the Big Task, Event dates fall outside the selected month, the system prompts the user with a warning to correct the dates. 4.2. The system triggers a warning if the "Start Time" is set later than the "End Time." 7.1. The system blocks scheduling if a sub-task's date falls outside the pre-defined date range of its parent Big Task.

	7.2. The system identifies and warns the student of "Hard Conflicts": ● Overlapping with pre-defined "Sleep Time". ● Exceeding the maximum time limit allocated for a specific category. ● Overlapping with established items.

- **Team member management**

Table 2.5. Scenario for the Team member management UC

Actor	Project Owner
Precondition	The Student logged in and created a project.
Post-condition	The team member view displays an updated list of all members currently associated with the owner-managed project.
Main Scenario	1. The project owner accesses the "Invite Member" form, enters the target user's email address in the search bar, and clicks Add. 2. The system retrieves the metadata for both the sender and the recipient. It then concurrently dispatches an invitation via Email and a real-time In-app Notification. 3. The recipient is added to the project database with a temporary status of "INVITED". 4. After the recipient accepts the invitation through either the link provided in the email or the in-app notification, the system validates the invitation token/response and updates the recipient's status from "INVITED" to "MEMBER". 5. The team member's view is updated to display the new member with his/her “Member” status. 6. From the management form, the owner can perform the following administrative actions: ● **Edit Role:** The owner clicks on a member's custom role name to modify. ● **Remove Member:** The owner selects the delete option corresponding to a specific user.

	7. The system validates the administrative request, persists the changes in the database, and updates the view for all relevant members.
Exceptions	1.1. If the email entered by the owner does not correspond to an existing account in the system, the system triggers a "User Not Found" error and prompts the owner to check the email address. 4.1. If the receiver chooses to decline the invitation or fails to respond within the expiration period: a. The system validates the rejection response. b. The system removes the temporary "INVITED" record from the project database. c. A notification is sent to the project owner regarding the recipient's decision. d. The "Invited" entry is removed from the team member's view.

- **Project task structure management:**

Table 2.6. Scenario for the Project task structure management UC

Actor	Project Owner
Precondition	The Project Owner logged in and navigated to the List of Tasks page.
Post-condition	The project hierarchy and data (Deliverables, Phases, Tasks) are modified or reorganized.
Main scenario	1. Create & Edit: Owner adds, renames, or deletes Deliverables, Phases, and Tasks; system validates and updates the dashboard. 2. Manage Details: Owner updates task priority, progress, and schedules (start/end times) on the Timeline. 3. Assign & Attach: Owner assigns members and links project files to tasks. 4. Collaborate: Members discuss via real-time threaded comments (post, reply, edit, delete). 5. Reorder: Owner reorganizes components via drag-and-drop.

Exceptions	7. a. The system sends the task to the unscheduled list of tasks of the project in the calendar view. b. Students drag and drop the task into the calendar view to create items. c. The systems validate the request successfully and display items in the calendar view. 10. a. The system sends a notification to the task-assigned members.

- **Document management**

Table 2.7. Scenario for the Document management UC

Actor	Project Member
Precondition	The Project Member has logged in and navigated to the 'Shared Source' page.
Post-condition	The page displays a document containing the edited data.
Main Scenario	1. A user initiates the creation of a new document. The system automatically generates the file with default metadata, including the document name, creator identity, and timestamp. 2. Users can modify the document title and body. The editor supports rich-text input and custom commands to format various data types. 3. Media Integration: ● **Images:** Users can insert images via clipboard pasting, local file uploads, or external URLs. ● **External Storage:** Upon validation, the system offloads images to a third-party cloud storage service and renders the media within the document. ● **File Attachments:** Users can link existing project files to the document. The system retrieves a list of available assets from the 'Shared Source' page for selection. 4. Other members click into the file to open it. 5. The system validates the permissions of users before letting them see the document.

	6. The document header displays avatars of all active members currently viewing or editing the file. 7. Members can highlight specific text to add remarks. The system supports a discussion thread for including replying, editing, and resolving remarks. 8. The remark creators delete, edit, or resolve their own remarks, while others reply and edit them. 9. Users can access a "Full Comment" view to manage active and resolved discussions, including the ability to reopen resolved threads. 10. The system automatically persists a new version of the document every 30 minutes. 11. Users can access the "Version History" panel to browse all saved snapshots. 12. Users can revert the document to a previous state. Before restoring an old version, the system captures a snapshot of the current state to prevent accidental data loss. 13. The member navigates to the 'Shared Source' to view and then delete the document. 14. The system validates permissions, removes the record from the database, and refreshes the file list.
Exceptions	3.1. The system enforces a 10MB limit on image uploads. If a file exceeds this threshold, the system rejects the upload and issues a warning notification. 14.1. The system checks the permission of the member: the member is not the creator or project owner. It rejects the request and triggers a response.

- **Post management**

Table 2.8. Scenario for the Post management UC

Actor	User (Student)
Precondition	The user has logged in and navigated to the forum page.
Post-condition	The post is created and displays the modified information.
Main Scenario	1. The user clicks the "Ask a Question" button. The system presents a form requiring a Title, Body content, and Tags.

	1. As the user types in the tag field, the system performs real-time filtering of existing topics. Pressing "Enter" validates the selection and appends the tag to the post's metadata. 2. Upon clicking "Post," the system validates that all fields are populated and redirects the user to the Post Detail page. 3. The user clicks edit the post. The form of a post with a title, body, and tag for the user to update. The user clicks update, then the system loads and displays the new content post. 4. **Collaborative Contribution:** ● **Answers:** Other users (or the author) can submit answers using a structured input form. ● **Comments:** Users can post comments (text type) on either the original question or specific answers to request clarification or provide feedback. 5. Authors of answers or comments can edit, delete, and comment on their own contributions. **Note:** Deleting an answer automatically triggers a cascade deletion of all nested comments associated with it. 6. Users can downvote posts or answers deemed unhelpful for everyone. The system recalculates the aggregate score and refreshes the display immediately. 7. **Knowledge Integration:** ● **Project Linkage:** Users can select "Save Post to Project." The system retrieves a list of the user's active projects. Upon selection, the post is linked to the project's Shared Source page. ● **File Preservation:** If a post contains attachments, users can save these files directly to their project's storage. ● **Private Bookmarking:** Users can save posts privately to their personal "Saved Posts" collection.

	8. The post author can mark a specific answer as "Correct." This action updates the post status to "Solved" and visually highlights the answer as the verified solution. 9. Users can sort the answer lists by clicking on sorting by time or by score. 10. Users can click on avatars to view profiles. From another user's profile, a project owner can send a project invitation (reusing the "Team Member Management" logic). 11. The user clicks delete the post. The system loads, rejects the request, and triggers a warning (the system only lets the user delete the post when it does not contain any answers).
Extensions (Sub-flows)	7.a. ● The system hides the post if the total votes are over 10 and the number of downvotes is bigger than upvotes, and there is no positive-score answer. ● The system hides the answer if the total votes are over 10 and the number of downvotes is greater than the upvotes.
Exceptions	3.1. The system blocks publication and issues a warning because the Title, Body, or Tag fields are empty during post. 5.1. The system rejects the request and triggers a warning that the Title, Body, or Tag fields are empty. 7.1. The system triggers a warning when a field of the answer is empty.

2.3. System Analysis

2.3.1. Extract system entity classes

The system is designed to manage entities revolving around two main domains: Personal Academic Management (Calendar/Planning) and Collaborative Project Management.

- **Main Entities:**
    - User: User
    - User Constraints: UserConstraints

- Calendar: Calendar
- Calendar Item: CalendarItem
- Routine: Routine
- Personal Task: Task
- Event: Event
- Memorable Event: MemorableEvents
- Month Plan: MonthPlan
- Week Plan: WeekPlan
- Big Task: BigTask
- Project: PM_Project
- Project Member: PM_ProjectMember
- Project Deliverable: PM_Deliverable
- Project Phase: PM_Phase
- Project Task: PM_Task
- Project Risk: PM_Risk
- Project Milestone: PM_Milestone
- Project File: PM_File
- Forum Post: ForumPost
- Forum Answer: ForumAnswer
- Forum Comment: ForumComment
- Forum Tag: ForumTag
- Document Content: DocContent
- Document Snapshot: DocSnapshot
- Notification: Notification

2.3.2. Identify relationships between classes

- Each User possesses specific User Constraints (e.g., sleep hours) and owns a Calendar containing multiple Calendar Items (Routines, Tasks, Events).
- A User creates a Month Plan, which consists of multiple Week Plans and high-level Big Tasks, Routines (only name), and Events.
- A User can receive multiple Notifications.
- A Month Plan contains multiple Events, Big Tasks, and Routines.
- A Big Task can be broken down into smaller Tasks.
- A User can participate in multiple Projects. The relationship is managed through the Project Member entity, which defines the specific role (Owner, Member, Invited) of a user within a specific project.

- A Project is composed of multiple Deliverables. Each Deliverable contains multiple Phases, and each Phase contains multiple Project Tasks.
- A Project Task can be assigned to one or multiple Project Members.
- A Project contains multiple Files.
- A Project contains multiple Risks. Each Risk can be assigned to specific members.
- A Project contains multiple Milestones.
- Project Tasks, Phases, and Deliverables can have predecessor/successor relationships to establish a timeline dependency chain.
- Each User can create multiple Posts, Answers, and Comments.
- Each Post contains multiple Answers, Comments, and Tags.
- Each Answer contains multiple Comments.
- Each Document can have multiple DocSnapshot.
- Each Notification is owned by a User.

2.4. Design Phase Class Diagram

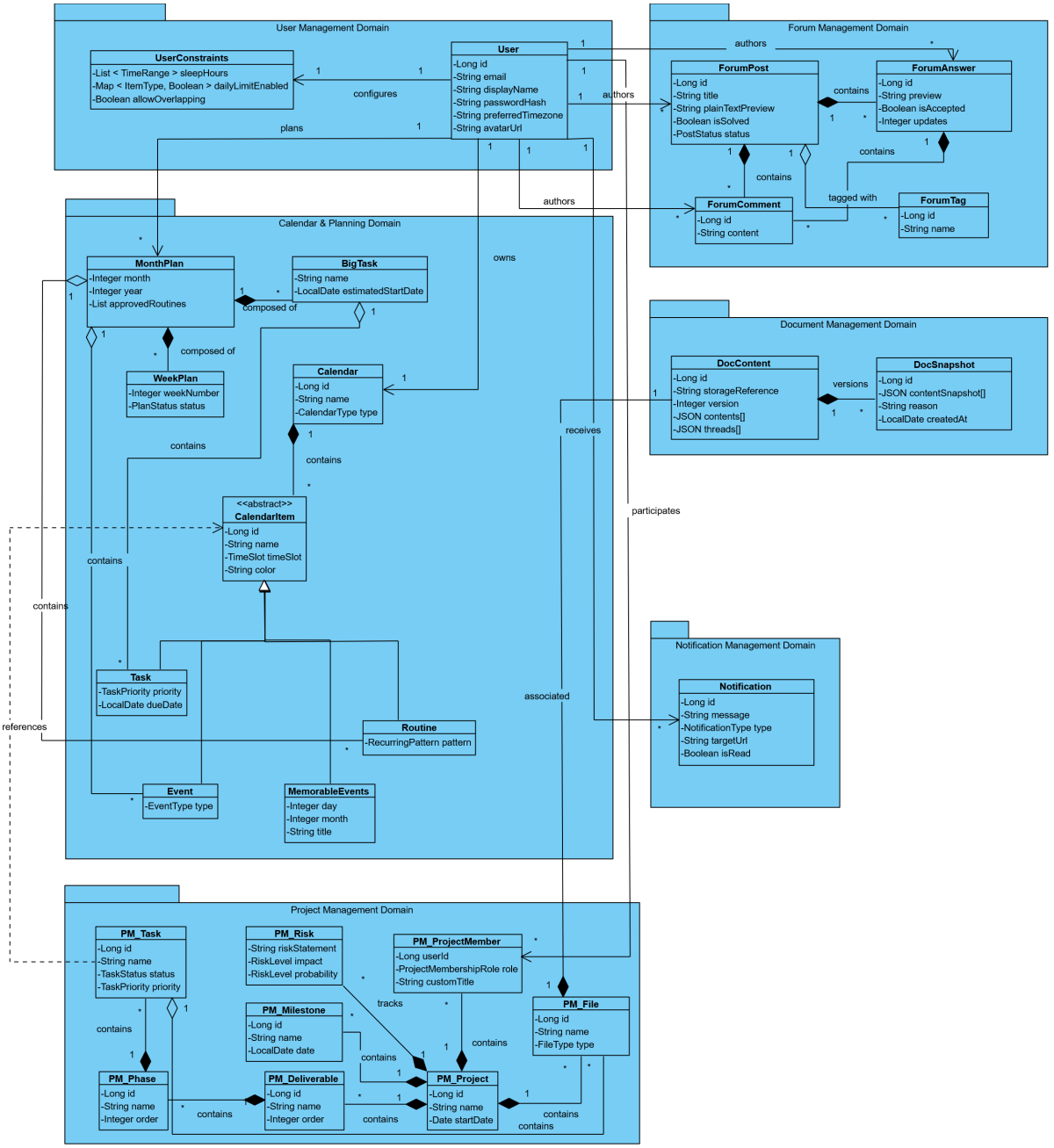

Image 2.7. Design Phase Class Diagram

2.5. System Sequence Diagrams

As mentioned above, due to the length of the report, this section only presents a partial view of the full scenario to ensure clarity in reading. These chosen parts are the core contents of my project.

- **The Month plan management** diagram below represents the flow of creating a Big Task and creating a Task from it.

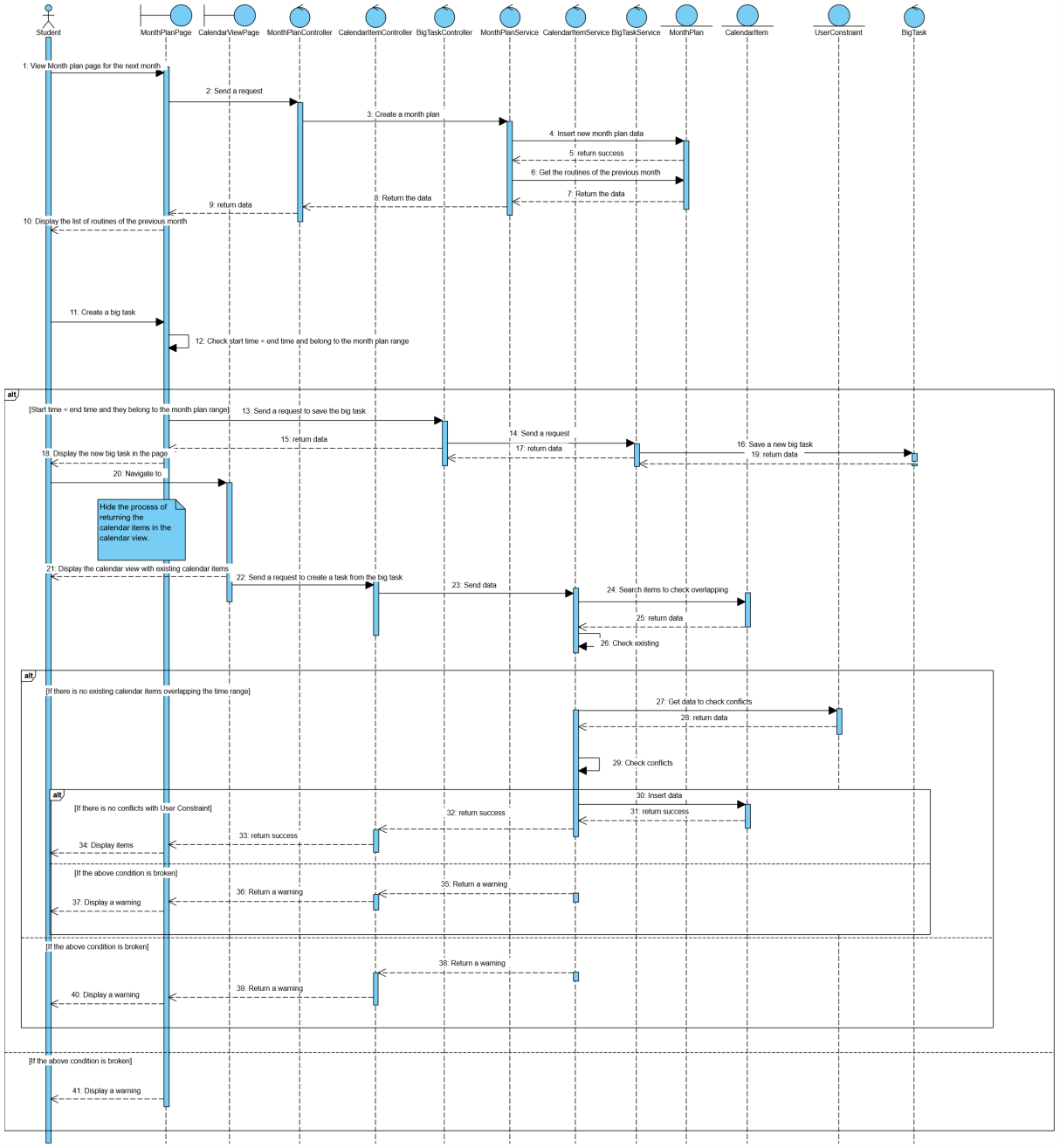

Image 2.8. Sequence diagram for the Month management UC

- **The 'Team member management'** diagram below focuses on demonstrating the flow of member invitation.

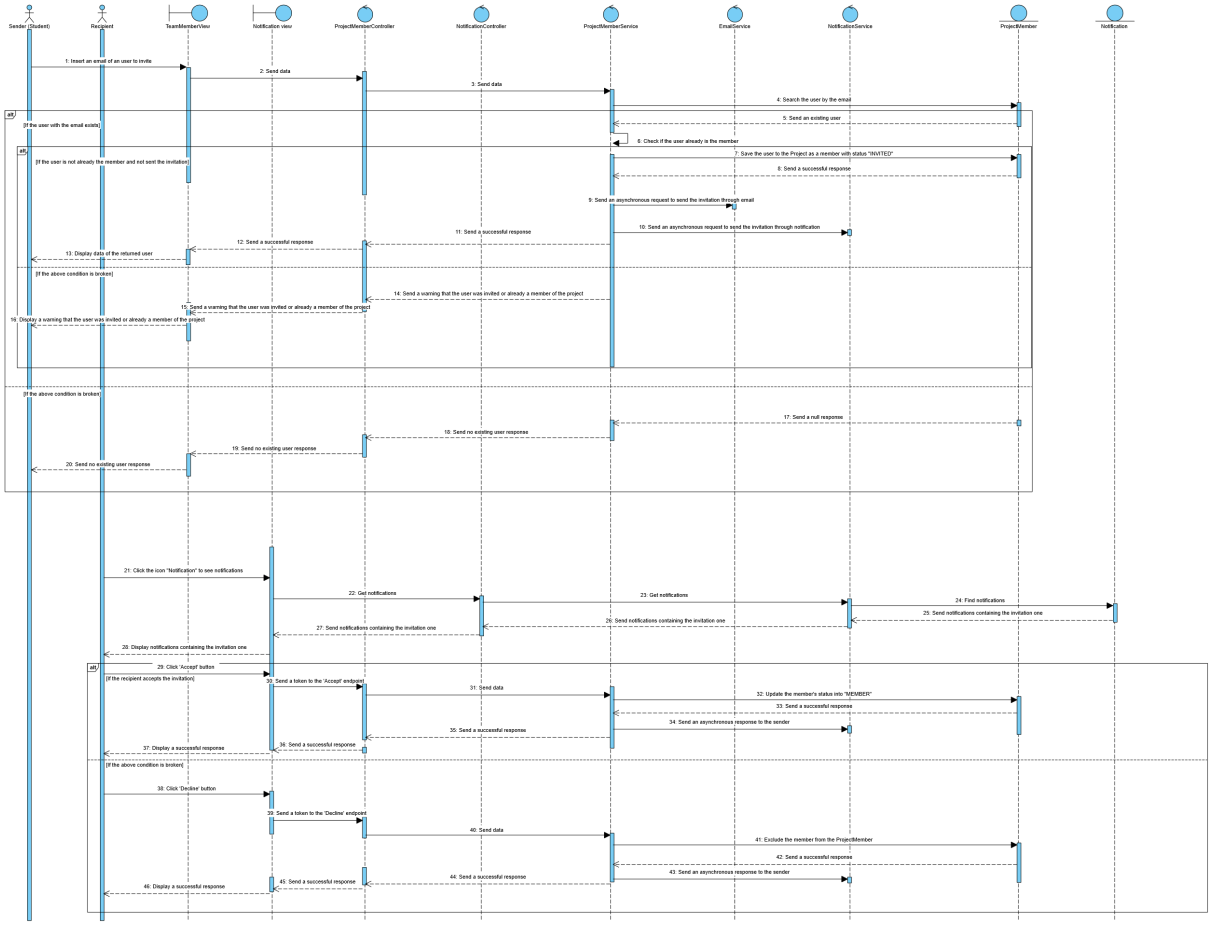

Image 2.9. Sequence diagram for Team member management UC

- **Note:** This diagram demonstrates only the flow of reciever respond the invitation through the notification.

- **Project task structure management:** The diagram below concentrates on demonstrating the flow of creating and modifying a task.

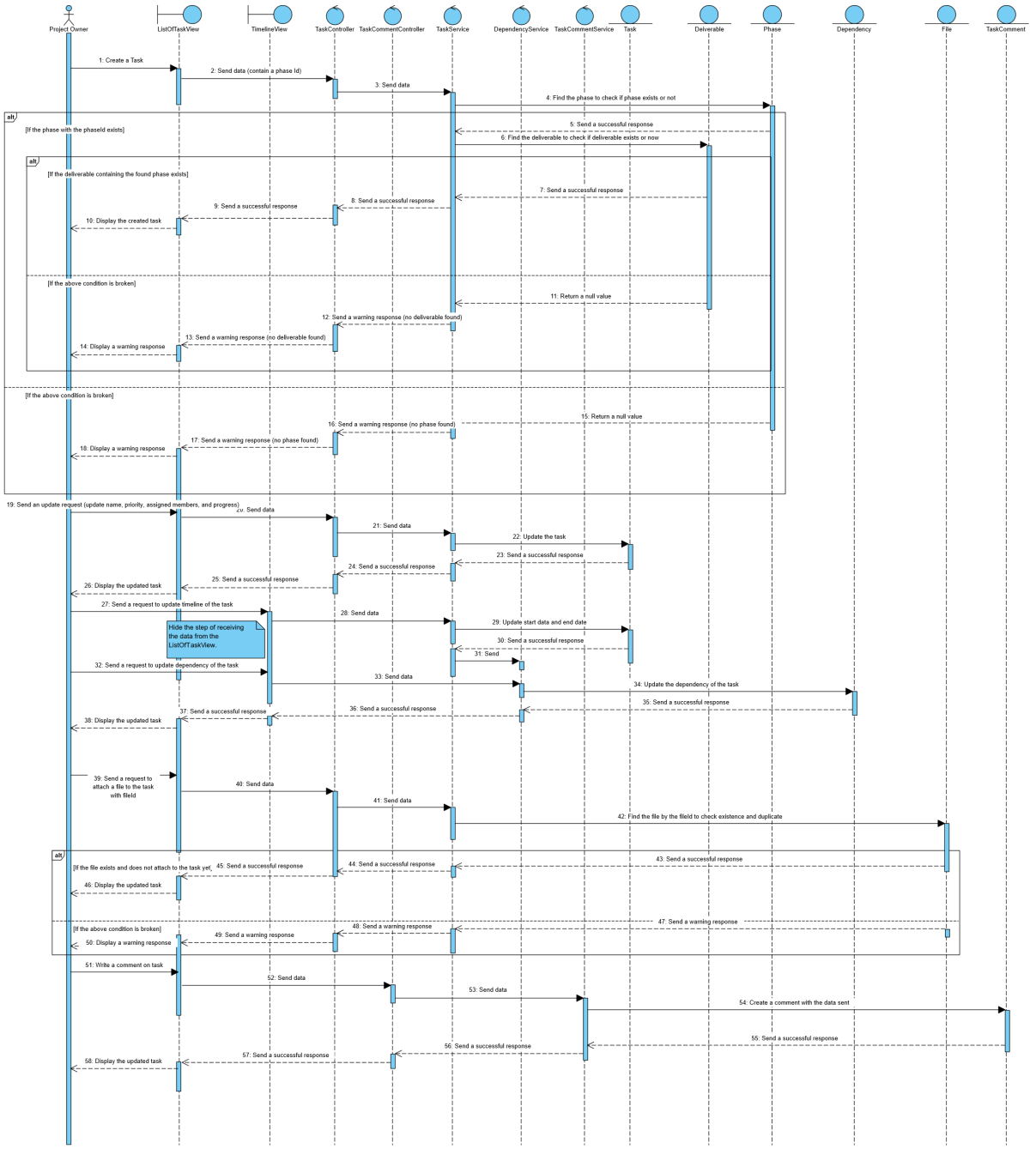

Image 2.10. Sequence diagram for the Project task structure management UC

- **Note:** The ‘Timeline’ and ‘Comment on tasks’ are demonstrated at a high level.
- **Document management:** The diagram below demonstrates the flow of creating and updating the document.

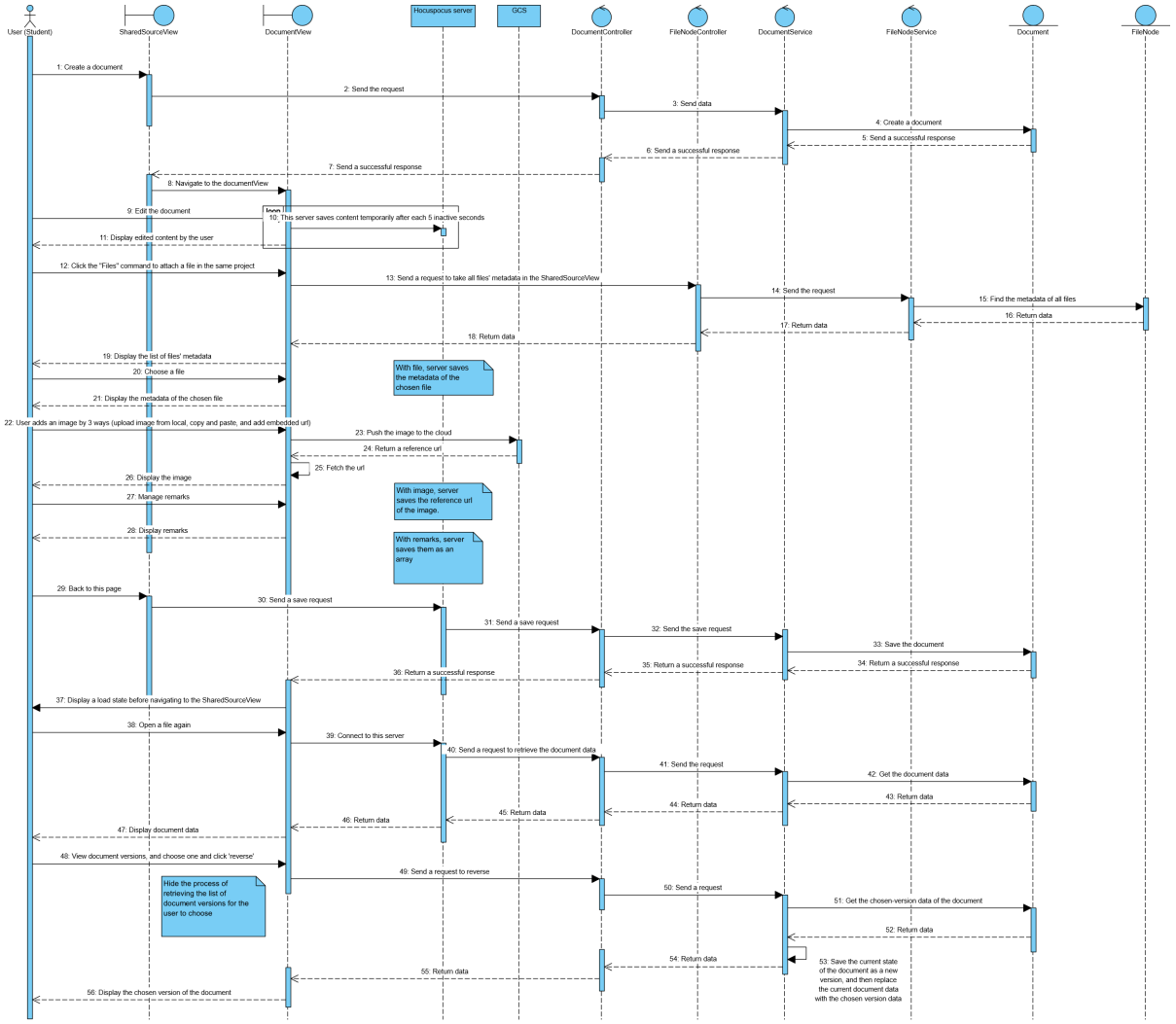

Image 2.11. Sequence diagram for the Document management UC

- **Note:** Before sending the creation request to the documentService, the system creates a file node to mark the document (a document is a type of file) in the 'FileService'. The diagram hides this step to ensure clarity.
- **Post management:** The diagram below illustrates the flow of creating, answering, commenting, and marking posts as being solved.

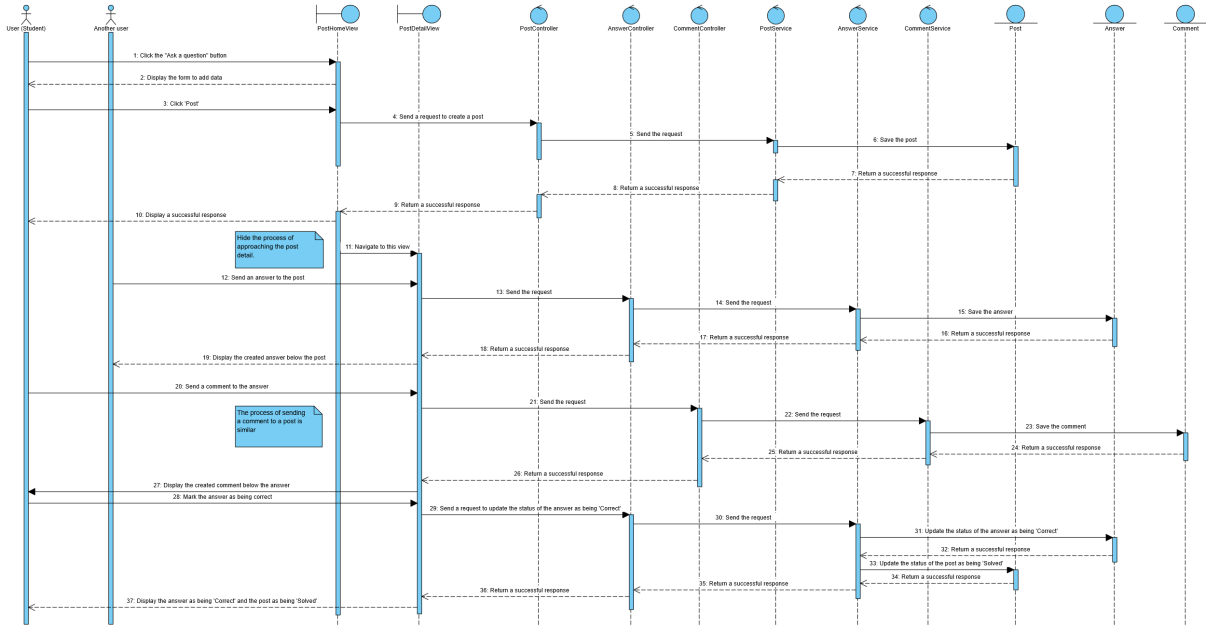

Image 2.12. Sequence diagram for the Post management UC

2.6. Database Design

2.6.1. Information about tables in the database

Due to report length constraints, this section provides descriptions for only the special tables. For a complete view of all tables, please refer to the database schema diagrams provided.

- User Management Module

Table 2.9. Information about tables in the User Management Module DB

No.	Table Name	Function of the table
1	users	Stores core user information such as email, password hash, display name, and profile settings.
2	user_sessions	Manages the active state of logged-in users.
3	oauth_providers	Handles external identity provider (Google) linked to a local User account.

- Personal Scheduling Module

Table 2.10. Information about tables in the Personal Scheduling Module DB

No.	Table Name	Function of the table
1	calendar_items	The central, polymorphic table that stores *all* time-bound entities (Tasks, Routines, Events) in a unified structure
2	month_plans	Represents the high-level planning unit for a specific month, explicitly linking "big tasks" (monthly goals) to the daily schedule
3	week_plans	Organizes goals into an actionable 7-day window.
4	user_constraints	Defines the "rules of engagement" for the scheduling engine customized for each user.

- Project Management Module

Table 2.11. Information about tables in the Project Management Module DB

No.	Table Name	Function of the table
1	pm_projects	Stores general metadata for a project, including name, start date, and counters for ID generation.
2	pm_project_members	Manages the many-to-many relationship between users and projects, including their specific roles (e.g., OWNER, MEMBER).
3	pm_deliverables	Stores high-level deliverables that group project work, linked to a specific project.
4	pm_phases	Represents distinct phases within a deliverable.

5	pm_tasks	Stores individual project tasks with details like status, priority, and timeline, linked to a phase.

- Forum Module

Table 2.12. Information about tables in the Forum Module DB

No.	Table Name	Function of the table
1	forum_posts	The central entity of the forum. It acts as the "root" for all discussions.
2	forum_answers	Represents the solutions or major responses to a post.
3	forum_comments	Facilitates the "Discussion" layer.
4	project_saved_posts	The integration link between the Forum Domain and the Project Management Domain.

2.6.2. System Database Schema

To ensure visual clarity and maintain a concise documentation structure across the comprehensive set of class diagrams, all diagrams are presented using a simplified notation focusing exclusively on **key fields and associated ones.**

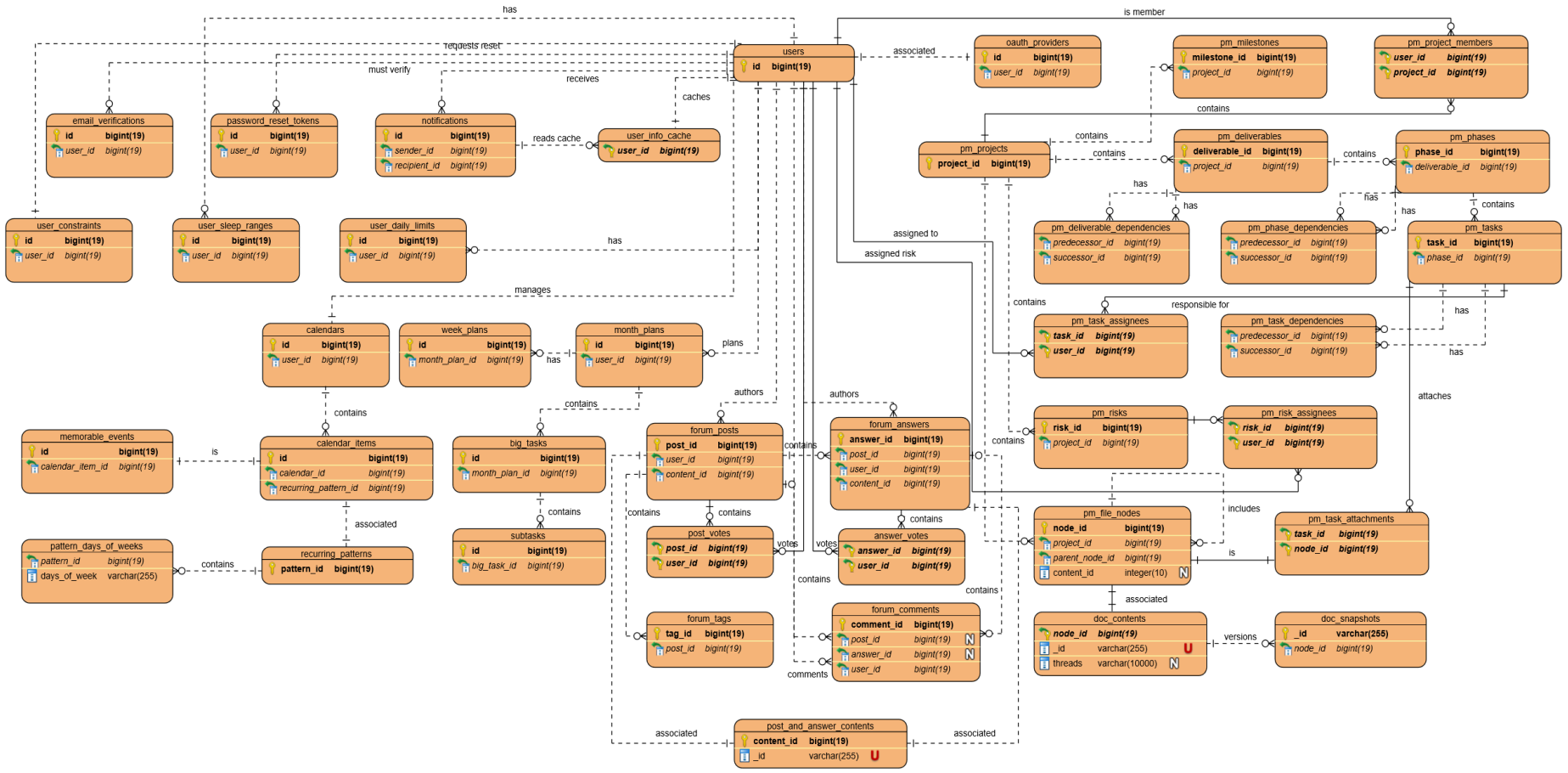

Image 2.13. System Database Schema (shortened)

2.7. Architectural Design

2.7.1. Overview of microservices architecture

The architecture consists of seven core components:

1. API Gateway Service
2. User Service
3. Scheduling Service
4. Project Service
5. Document Service (for handling live-editing documents)
6. Forum Service
7. Notification Service

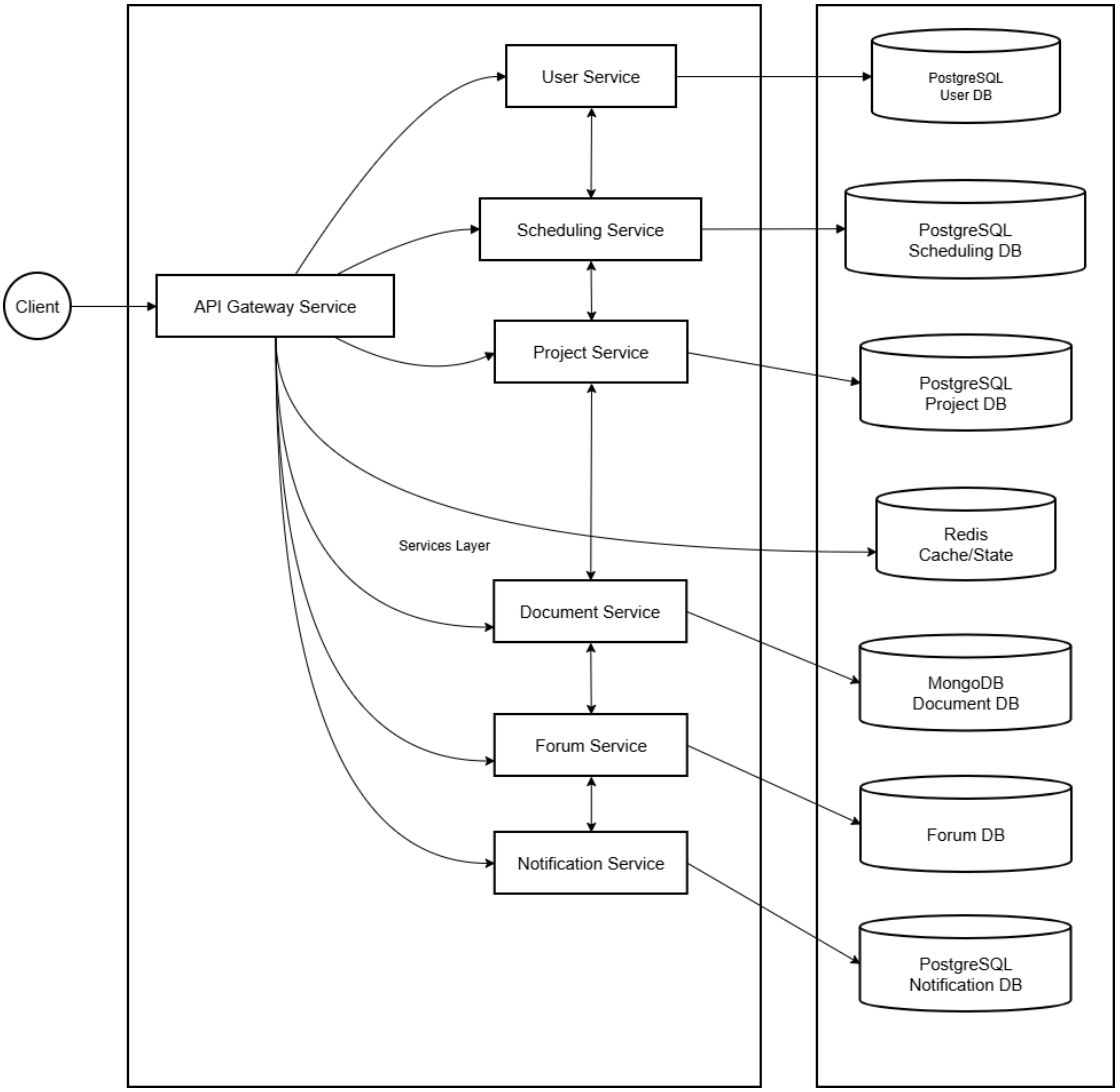

Image 2.14. Microservice Architecture Diagram

- **Note:**

- ForumDB is composed of 2 databases in 2 types: Postgres and MongoDB. However, this still ensures the Database-per-service because no other services connect to these 2 databases.
- The bi-directional arrows here are used to only illustrate that services can internally interact with each other, without mentioning the specific relationships among services, such as User Service interacts with Scheduling Service.

2.7.2. Main components

Table 2.13. System Service Elements

No.	Service Name	Description
1	**API Gateway Service**	Built on **Spring Cloud Gateway**, this service acts as the reverse proxy. It validates JWT tokens, enforcing rate limits, logs request metrics, and routes requests to the appropriate downstream services.
2	**User Service**	Responsible for user registration, authentication logic, and profile management.
3	**Scheduling Service**	Managing personal calendar items, sleep hour validation, and month planning. It owns the calendar-related data.
4	**Project Service**	Handling the collaborative features of the system, including project creation, team member management, Kanban board state, Gantt chart,...
5	**Document Service**	Supporting live-editing document functionality.
6	**Forum Service**	Providing a Q&A space for students to exchange knowledge.
7	**Notification Service**	A supporting service designed to manage and dispatch notifications. It processes triggers from across the other services, notifying users of

		relevant events like new project assigned tasks or forum interactions.

2.7.3. Communication between services

The architecture employs a hybrid communication strategy to balance immediate consistency with system scalability and decoupling.

a) Synchronous Communication

- **Internal Communication:** Services interact directly through HTTP REST APIs. The RestTemplate client is utilized within the Java Spring Boot environment to dispatch requests and manage responses between microservices.
- **Data Format:** To ensure interoperability and ease of debugging, all inter-service data exchange is standardized using JSON (JavaScript Object Notation).

b) Asynchronous Communication

- **Integration with Notification Service:** This method is specifically employed when sending data from various microservices (such as the Project or Forum Services) to the **Notification Service**. Instead of waiting for the notification to be processed, the source service simply emits an event and continues its execution.
- **Rationale and Implementation:** The decision to use an asynchronous approach is driven by the need for high throughput and service decoupling. A detailed technical justification and the architectural benefits of this choice—specifically regarding the use of Apache Kafka—are discussed in Chapter 1, Section 1.2.7 (Event Streaming with Apache Kafka).

2.7.4. API Gateway implementation

The API Gateway is the most critical component for system security and stability. It is implemented using **Spring Cloud Gateway** and includes several custom filters:

a) Security and Authentication The system implements a **Global Filter** for security (JwtAuthenticationFilter).

- **Token Validation:** For every request to a protected endpoint, the gateway extracts the JWT from the Authorization header.

- **Blacklist Check:** Before validation, the gateway checks **Redis** to ensure the token has not been logged out or revoked. If the token exists in the jwt:blacklist: key, the request is rejected immediately.
- **Context Propagation:** Upon successful validation, the User ID and Email are extracted from the token and added to the request headers (X-User-Id, X-User-Email) before being forwarded to internal services.

b) Rate Limiting To prevent abuse and ensure fair resource usage, a **Rate Limit Filter** is applied.

- **Mechanism:** The system uses a sliding window algorithm. It tracks the number of requests per client IP address within a configurable time window (e.g., 60 seconds).
- **Configuration:** Different limits can be applied to different routes. For example, sensitive routes may have stricter limits compared to read-only data routes.
- **Response:** If a limit is exceeded, the Gateway immediately returns a 429 Too Many Requests status.

c) Logging and Monitoring A high-precedence LoggingGlobalFilter is implemented to capture traffic metrics.

- It logs the incoming request method, path, and remote address.
- It calculates and logs the execution duration of the request after the downstream service responds.
- Sensitive headers (like Authorization tokens) are automatically redacted from the logs to ensure security compliance.

d) Global Error Handling The GlobalErrorHandler intercepts exceptions occurring at the Gateway level. It transforms various errors (Timeouts, 404s, Internal Errors) into a standardized JSON format containing the timestamp, status code, and a user-friendly error message, ensuring a consistent API experience for the frontend.

2.7.5 Real-time Collaboration Server with Hocuspocus Architecture Design

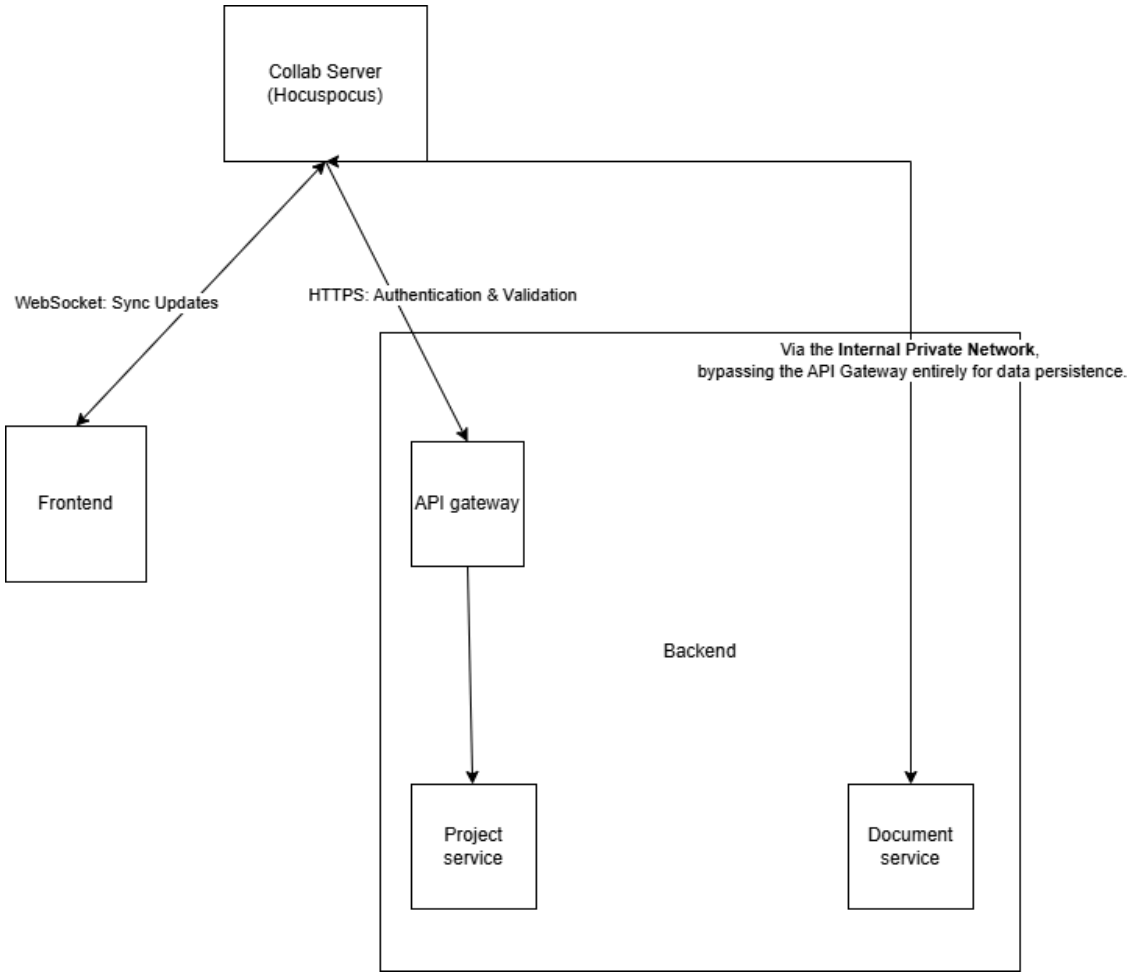

Image 2.15. High-level system architecture with Real-time Collaboration Server

a) Main Components of the Architecture

The collaborative environment is composed of three primary layers that work in synchronization:

- **The WebSocket Client (Next.js):** Utilizes a specialized synchronization module to establish a persistent connection with the Hocuspocus server via a unique resource identifier.
- **The Collaboration Server (Node.js):** Runs the Hocuspocus server instance, which manages in-memory document states and handles authentication hooks to verify user access before allowing synchronization.
- **The Backend Integration (Java):** Not only does the server validate the user access, but it also uses internal APIs to fetch the initial document state from MongoDB and periodically "flushes" (saves) the synchronized content back to the database after a period of user inactivity.

b) Operational Mechanism (The Triangle Flow)

The collaboration logic follows a triangular data flow to ensure security and persistence:

- **Connection & Authentication:** When a student opens a document, the frontend initiates a WebSocket connection. The collaboration server immediately triggers an **authentication request** to the API Gateway to verify that the user possesses the necessary "Member" or "Owner" permissions for the associated project.
- **State Loading:** Upon successful authorization, the server invokes a content retrieval service (document service) to fetch the current document structure from the database. This data is converted into a shared document state and distributed to all active participants.
- **Real-time Synchronization:** As users edit content, the editor generates binary data updates that are transmitted via the open WebSocket. The collaboration server merges these changes instantly.
- **Optimized Persistence:** To preserve system performance, the server implements a time-delayed saving strategy. Rather than recording every individual keystroke, it monitors for a period of inactivity before executing a data persistence operation to save the state.

2.8. Chapter 2 Conclusion

In this chapter, we have conducted a comprehensive analysis of the system functionalities, supported by detailed Use Case diagrams and decomposition scenarios. The chapter has presented a complete and clear Design Phase Class Diagram, defining the relationships between core entities such as Users, Calendars, and Projects.

Furthermore, Chapter 2 has finalized the system design through the development of function-specific Sequence Diagrams and a comprehensive Database Schema. The system architecture has been successfully modularized according to Microservices standards, where each service (User, Scheduling, Project) operates with its own independent database to ensure isolation and scalability. Notably, this design addresses the complex logic required for handling user time constraints and implements robust security measures at the API Gateway level.

CHAPTER 3: PROGRAM IMPLEMENTATION

3.1. Environment and Supporting Tools

3.1.1. Development environment

The system implementation was conducted using a standardized development environment to ensure compatibility and stability across the microservices architecture.

- **Java Development Kit (JDK):** Version 17 (LTS) is utilized for the backend services, chosen for its proven stability and long-term support (LTS) within the enterprise ecosystem, ensuring a reliable foundation for microservice communication.
- **Node.js:** Version 22.1.0 is used as the runtime environment to power the Next.js frontend and the Hocuspocus collaboration server.
- **Build Automation: Maven** is employed for the Java backend to manage dependencies and build the multi-module Spring Boot project.
- **Database Management System:**
    - **PostgreSQL:** Serves as the primary relational database for structured data requiring high integrity, such as user accounts, project hierarchies, and scheduling constraints.
    - **MongoDB:** Integrated specifically to handle unstructured data; it stores the JSON-based rich-text content for the "Notion-lite" document editor and Forum posts, allowing for flexible schema evolution.

3.1.2. Development Tools and Utilities

- **IntelliJ IDEA:** The primary Integrated Development Environment (IDE) used for development. Its built-in Database Tool Window was leveraged for direct connection, query execution, and management of the databases, eliminating the need for external tools.
- **Antigravity**: a state-of-the-art IDE is also used for development. It enhances development efficiency through its built-in Agent, which assists in automated task handling and intelligent code suggestions within the project environment.
- **Postman:** Used for managing and testing RESTful APIs.
- **Visual Paradigm, Draw.io, & PlantUML:** Utilized for modeling the system architecture and creating UML diagrams.
- **Figma:** The primary design tool used for prototyping the user interface (UI) and user experience (UX) flows before frontend implementation.

3.1.3. Main supporting libraries

- **Backend (Spring Ecosystem)**
  - **Spring Boot (v3.4.x):** The core framework used to bootstrap the microservices.
  - **Spring Cloud Gateway:** Acts as the entry point for the system, handling routing, security filtering, and rate limiting.
  - **Spring Data JPA:** Abstraction layer for database interactions, simplifying CRUD operations and complex queries.
  - **Spring Security & JJWT (io.jsonwebtoken):** Handles authentication and authorization via JSON Web Tokens (JWT).
  - **Spring Data Redis:** Facilitates interaction with the Redis in-memory data store for caching and managing the token blacklist.
  - **Lombok:** A utility library that reduces boilerplate code by automatically generating getters, setters, constructors, and builders via annotations.
- **Frontend (Next.js Ecosystem)**
  - **React Context:** Utilized for global state management across the application.
  - **Tailwind CSS:** A utility-first CSS framework used for rapid and responsive UI styling.
  - **ShadcnUI & Ant Design:** A hybrid approach using ShadcnUI for the core design system and Ant Design for complex interactive components (Dropdowns, OT Inputs).
  - **Tiptap:** Headless wrapper for ProseMirror used to build the rich-text "Notion-style" editor.
  - **Dnd-kit:** A lightweight, modular drag-and-drop toolkit used for the task management interfaces.
  - **React Hook Form & Zod:** Handles complex form state management and schema-based validation.
  - **Axios & RxJS:** Manages HTTP requests and asynchronous event stream handling.
  - **React Query (TanStack Query):** Manages asynchronous server state, providing efficient data fetching, caching, and synchronization to optimize application performance.

3.2. Implementation Results

3.2.1. Implementation of the User Management Module

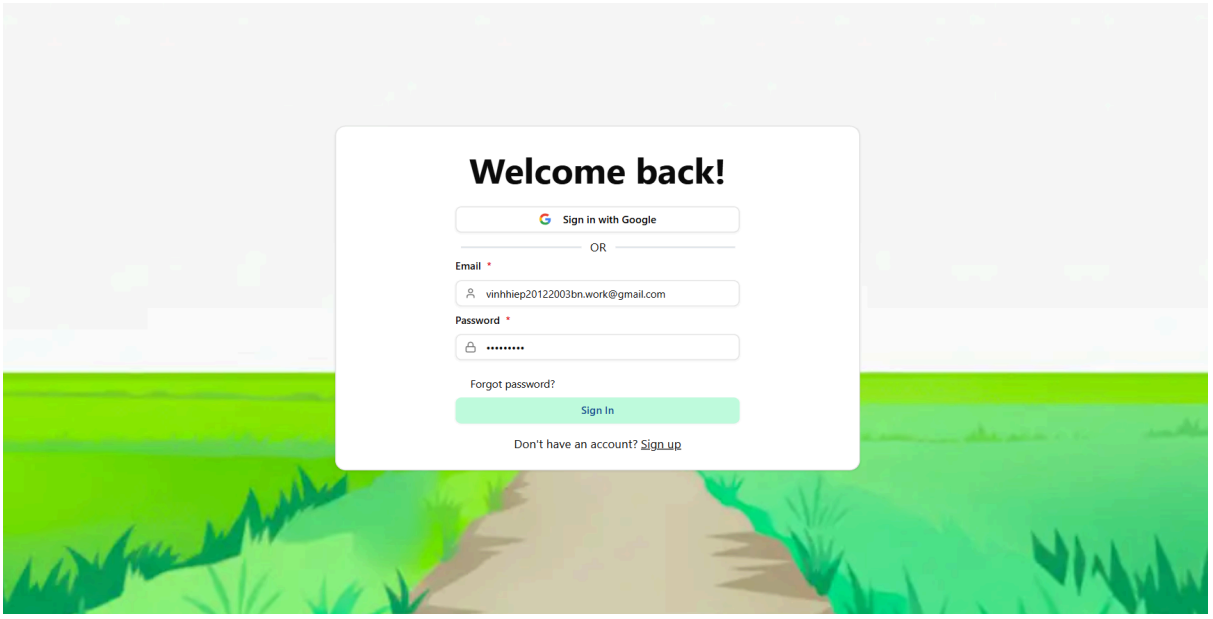

Image 3.1. Login page

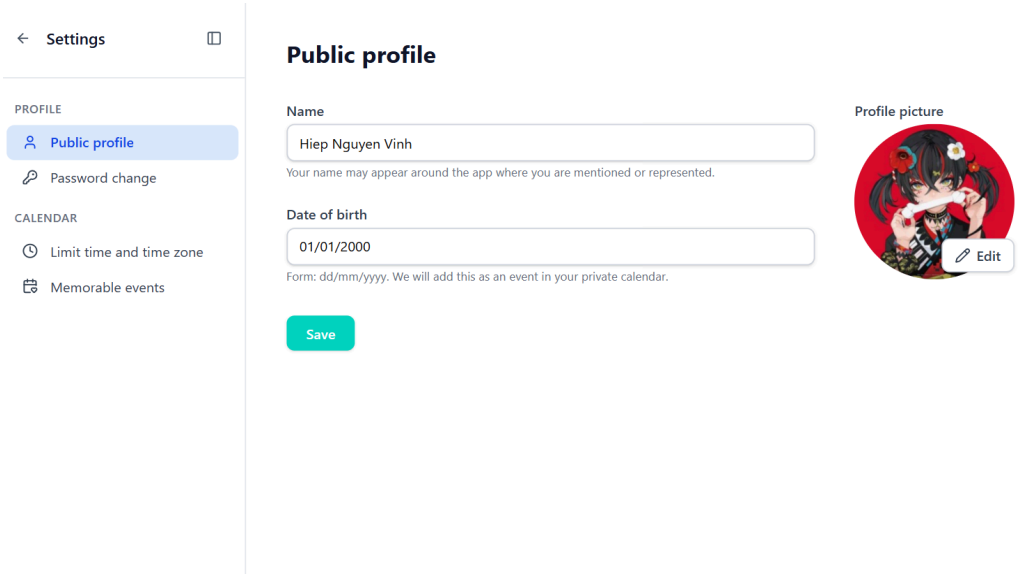

Image 3.2. Setting page (1)

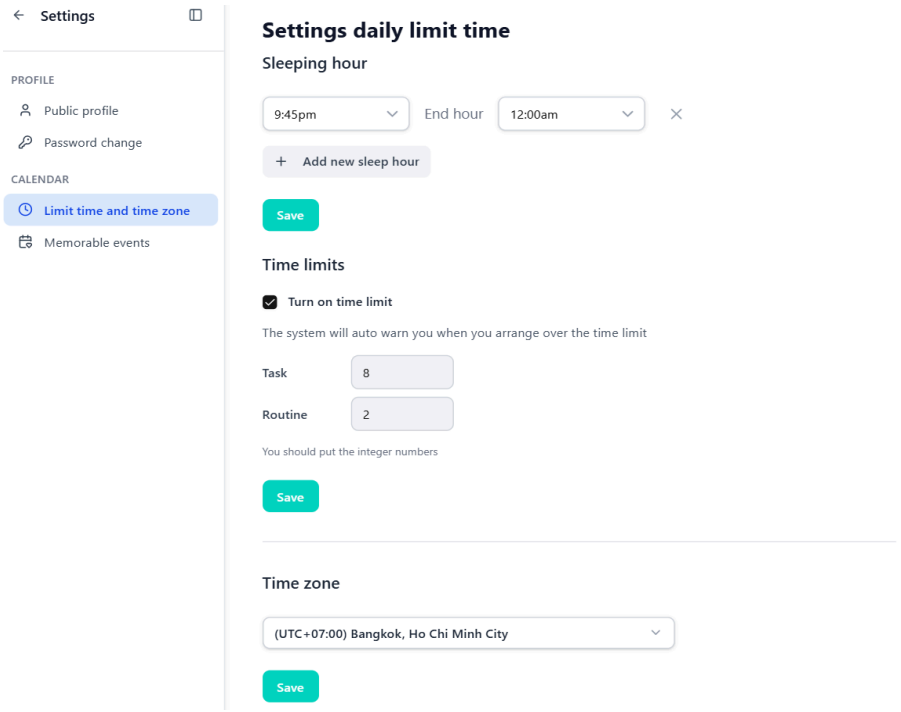

Image 3.3. Setting page (2)

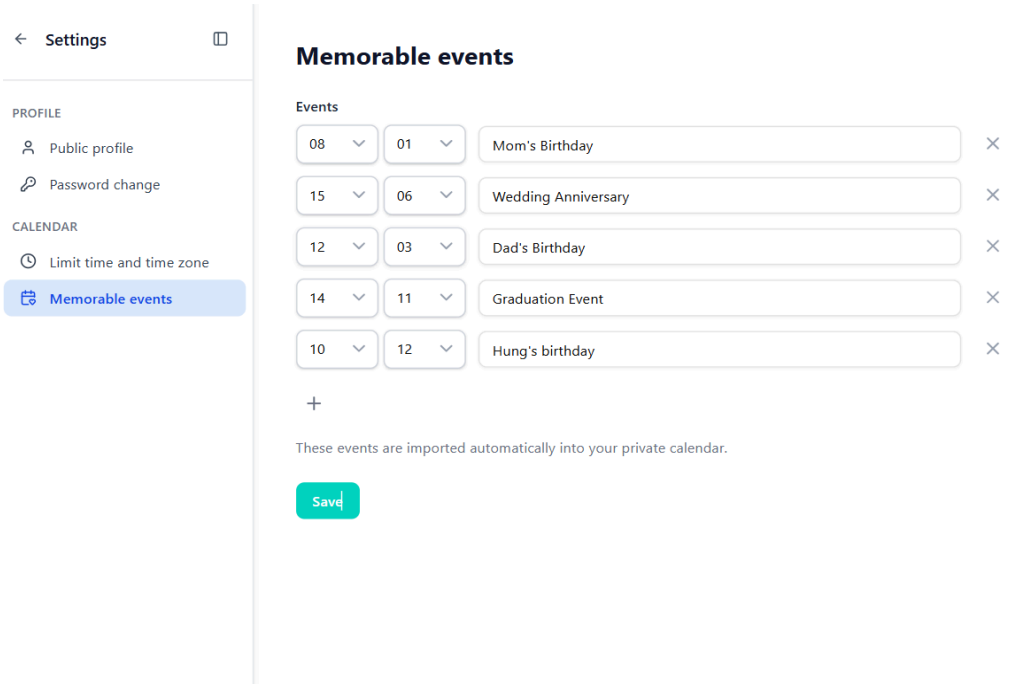

Image 3.4. Setting page (3)

3.2.2. Implementation of the Personal Scheduling Module

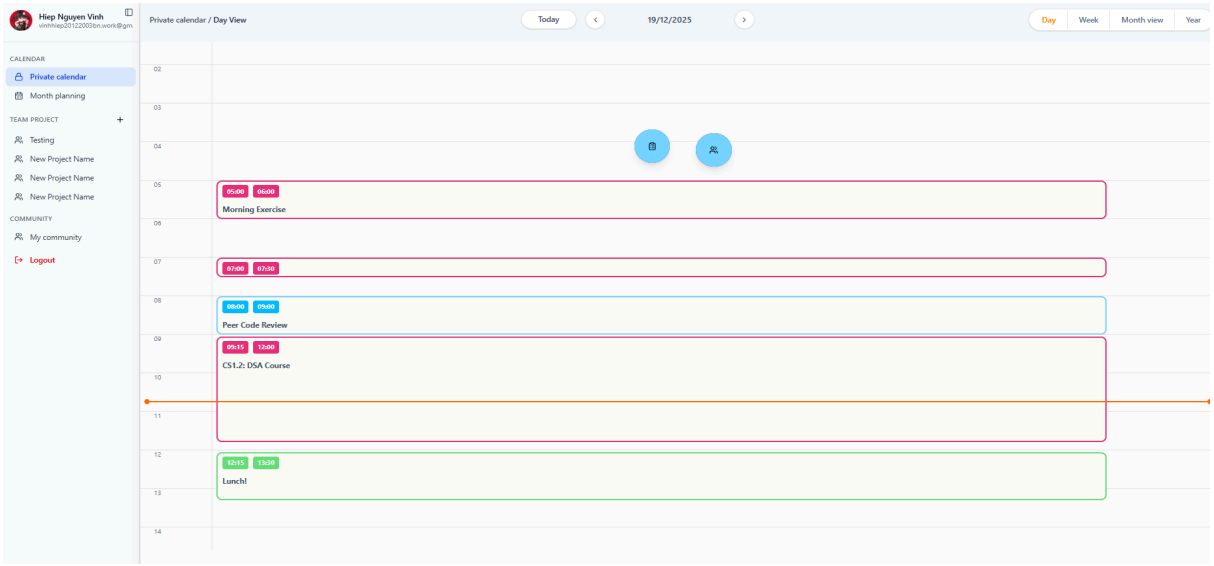

Image 3.5. Day view page

- **Description:** Day-view calendar interface utilizing color-coded categorization. Red denotes Tasks, while blue and green represent Events and Routines, respectively.

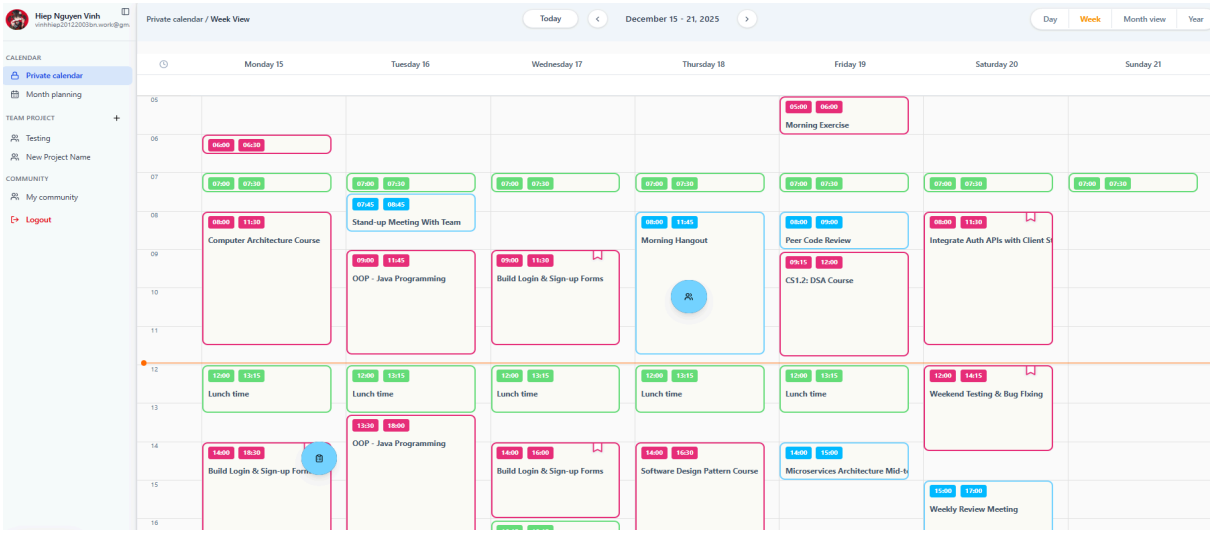

Image 3.6. Week view page (1)

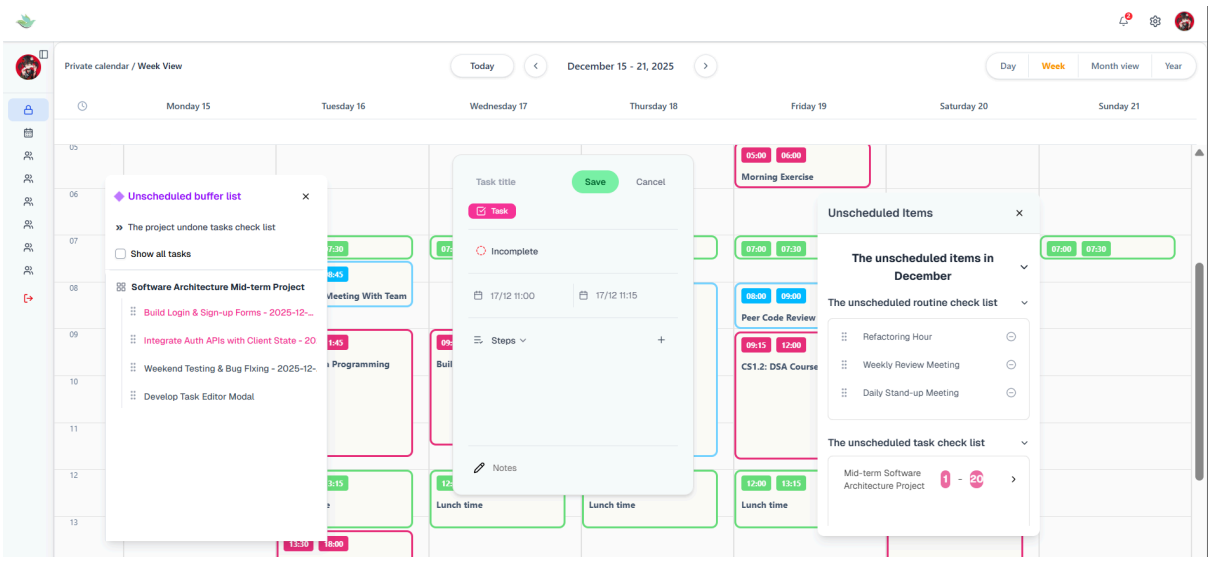

Image 3.7. Week view page (2)

- **Description:** The page contains 3 UI components featuring the Project Task Buffer, Unscheduled Items List (from the month plan page), and Item Creation Form. Red entries in the first component indicate overdue tasks. The unscheduled list exclusively displays Routines and Tasks, as Events are automatically synchronized with the calendar view.

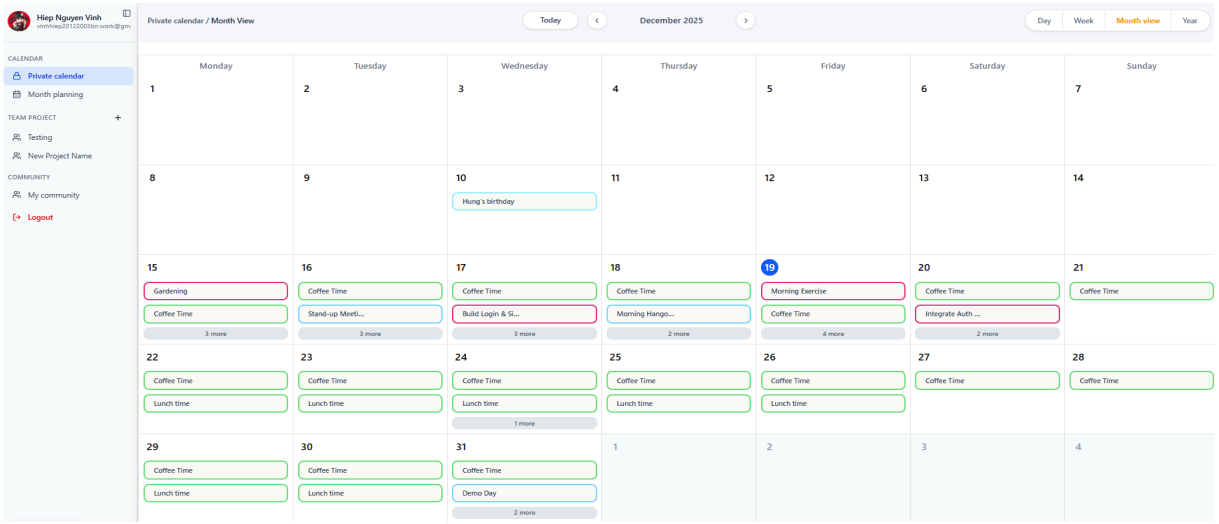

Image 3.8. Month view page

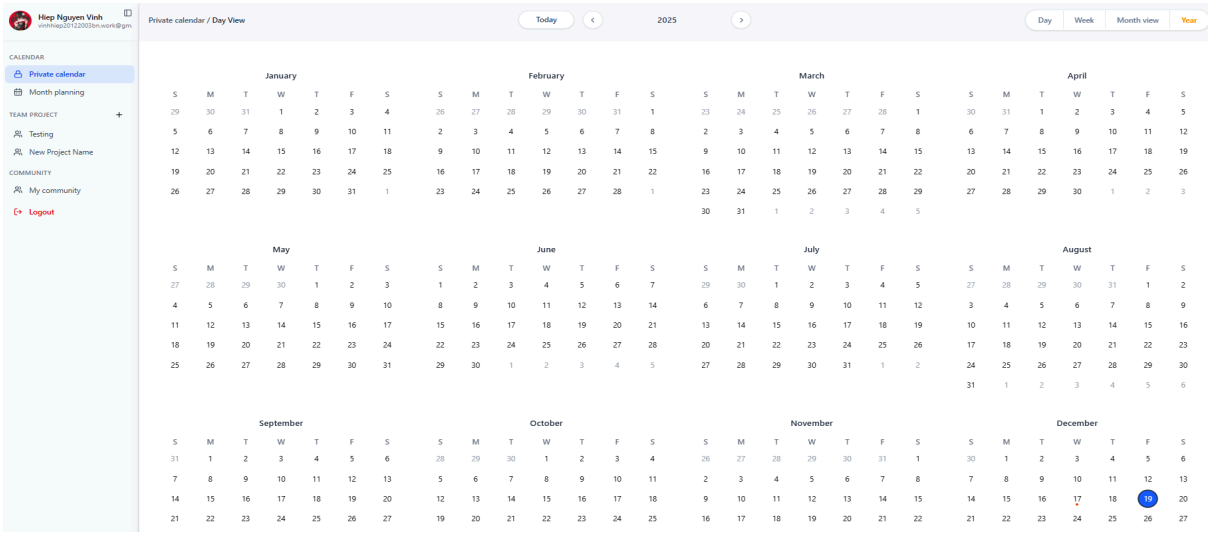

Image 3.9. Year view page (1)

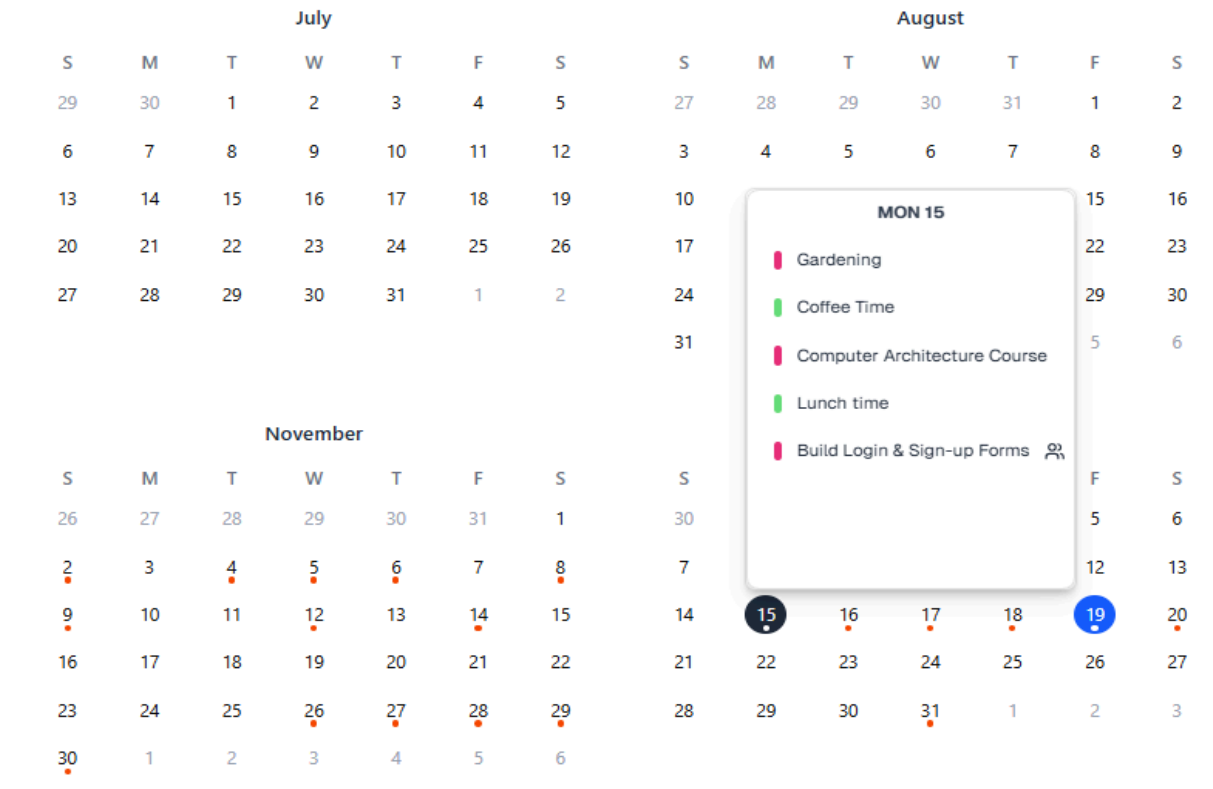

Image 3.10. Year view page(2) - Open the form to view more items in a day

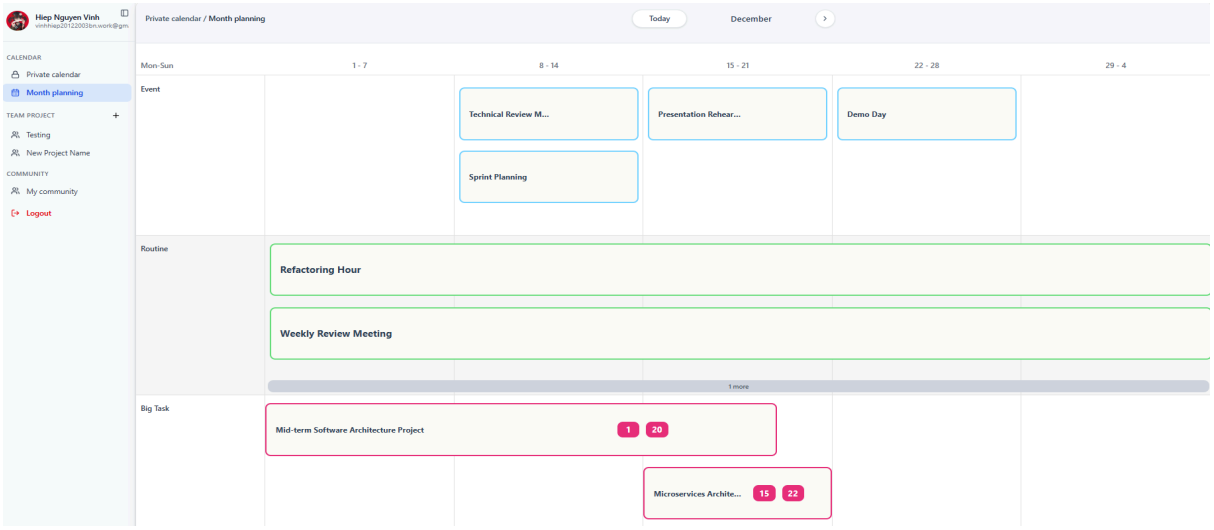

Image 3.11. Month planning page

3.2.3. Implementation of the Project Management Module

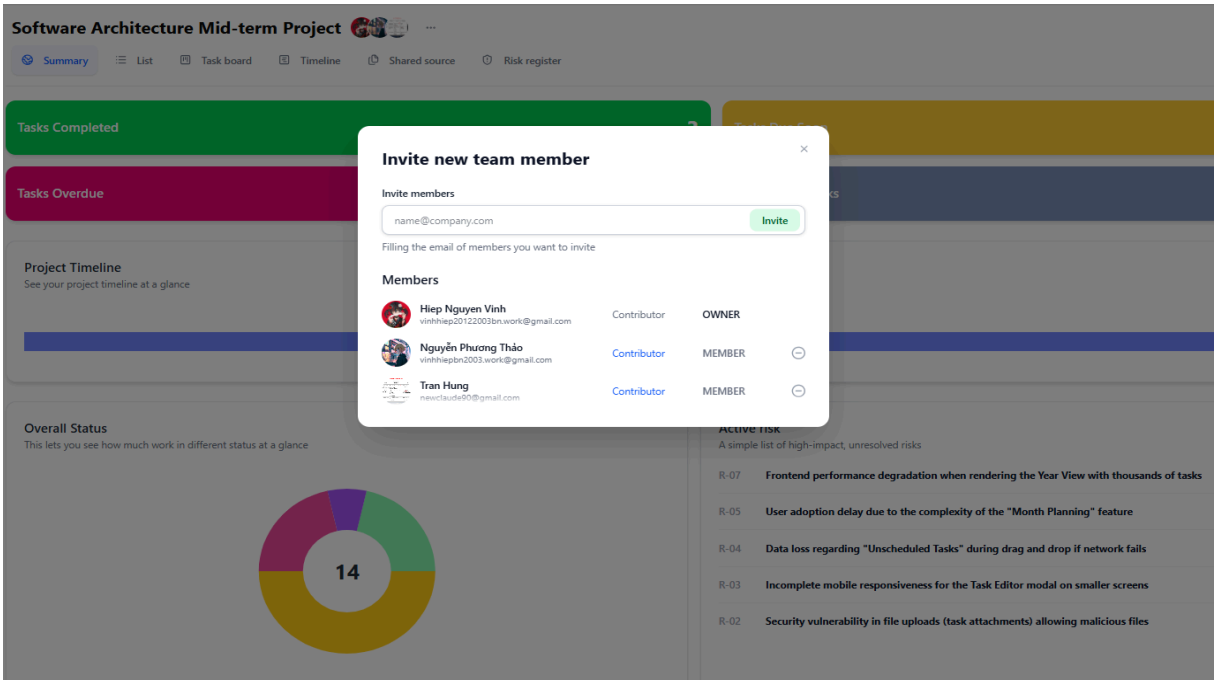

Image 3.12. Overall page (1) - with the open form of viewing members and adding members (role admin)

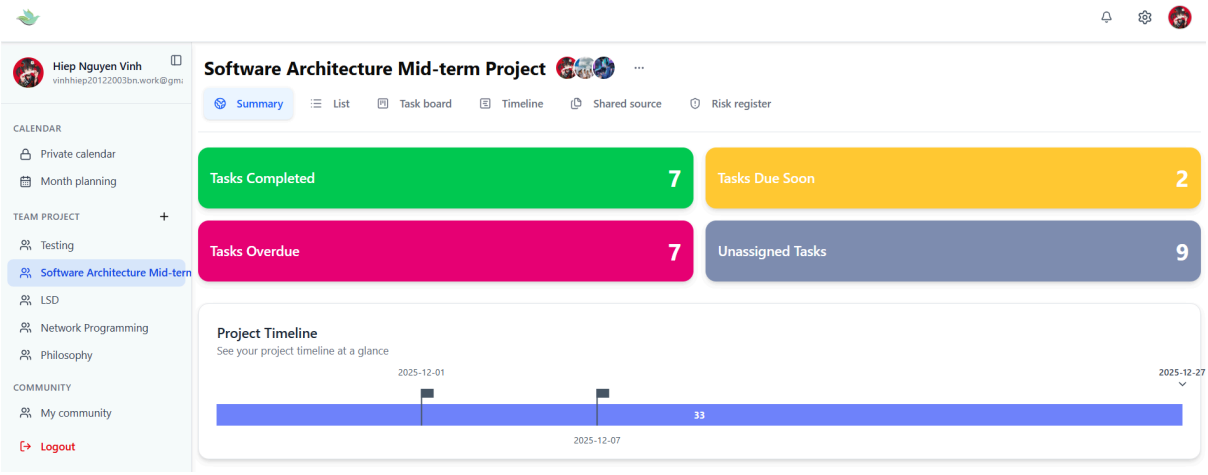

Image 3.13. Overall page (2)

- **Description:** The included diagrams offer a visualization of certain information about the project, helping members quickly grasp the important data, such as the number of tasks completed or overdue, and how many days from the start of the project.

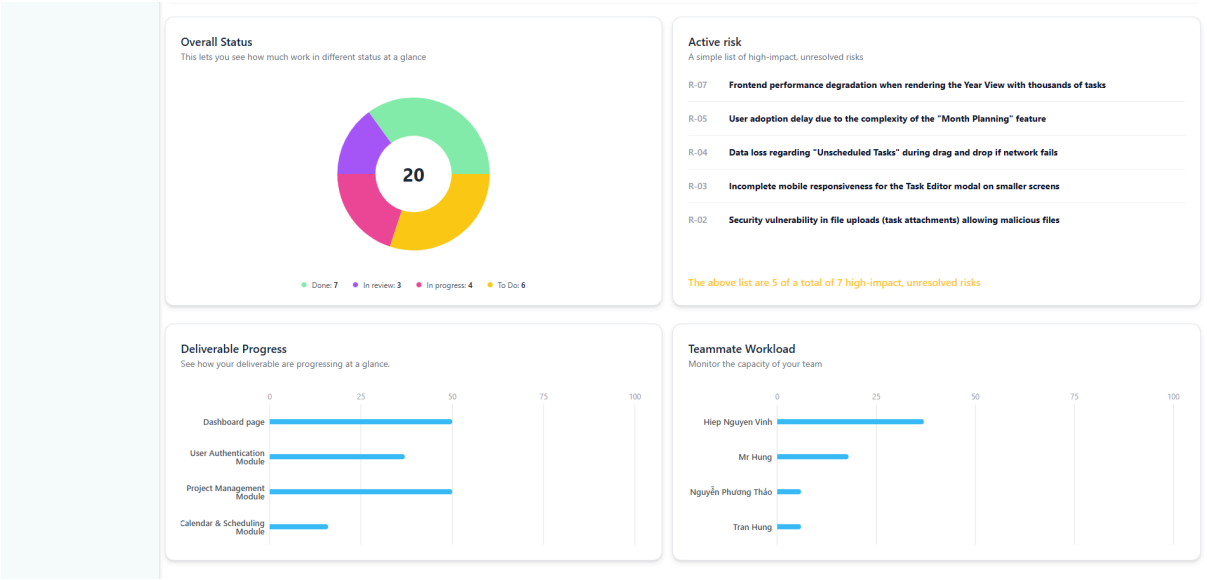

Image 3.14. Overall page (3)

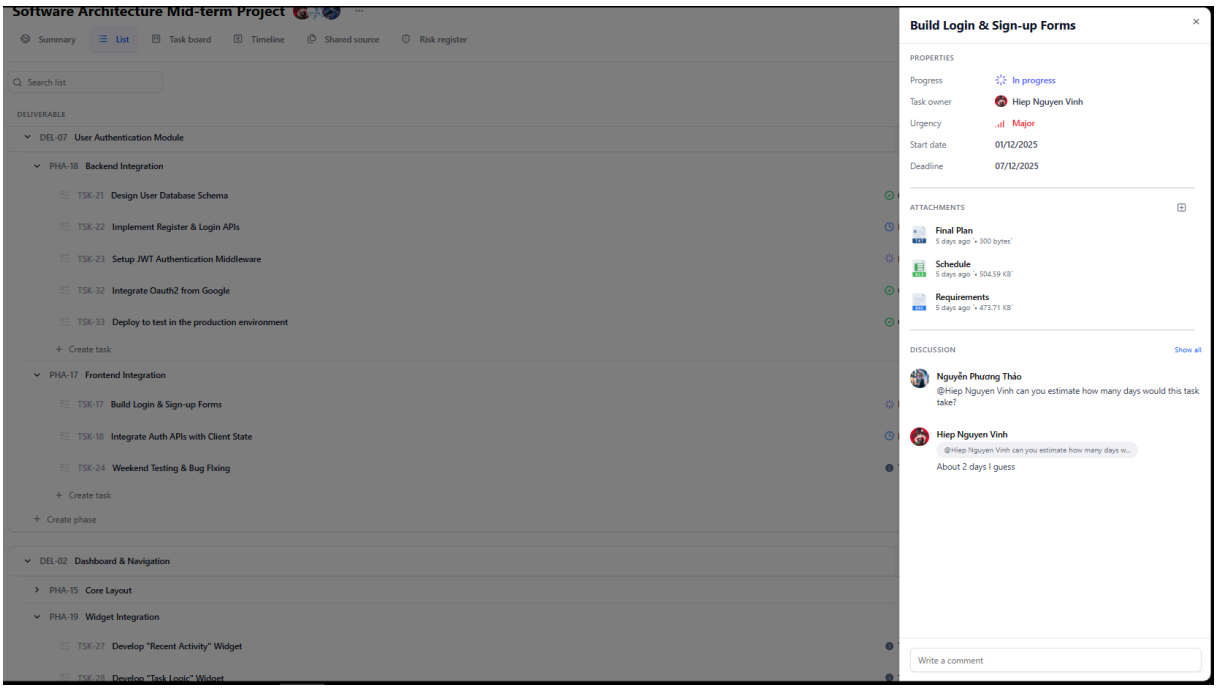

Image 3.15. List of tasks page (1) - with the task detail form

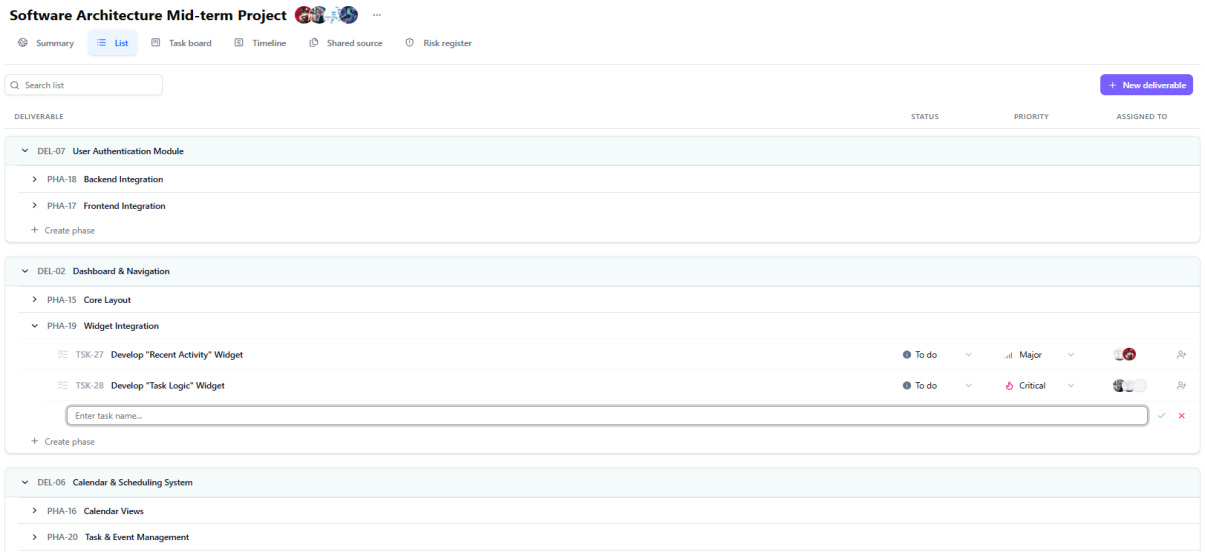

Image 3.16. List of tasks page (2) - with the form for creating a new task

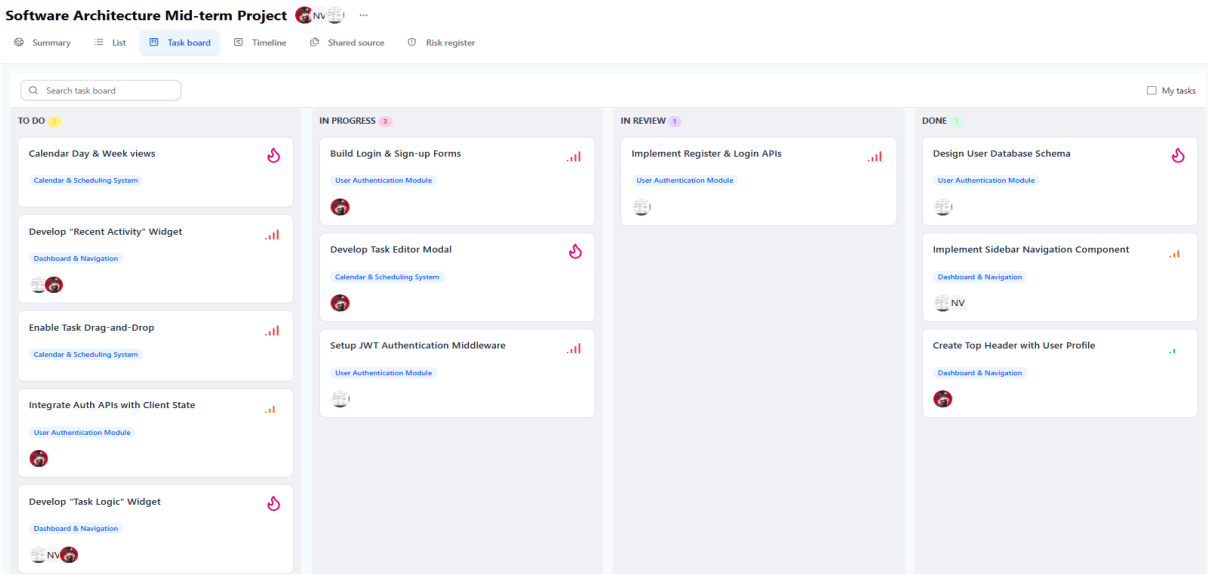

Image 3.17. Kanban board page

- **Description:** Task components display metadata—including title, associated deliverable, priority level, and assigned members—categorized by the status of tasks.

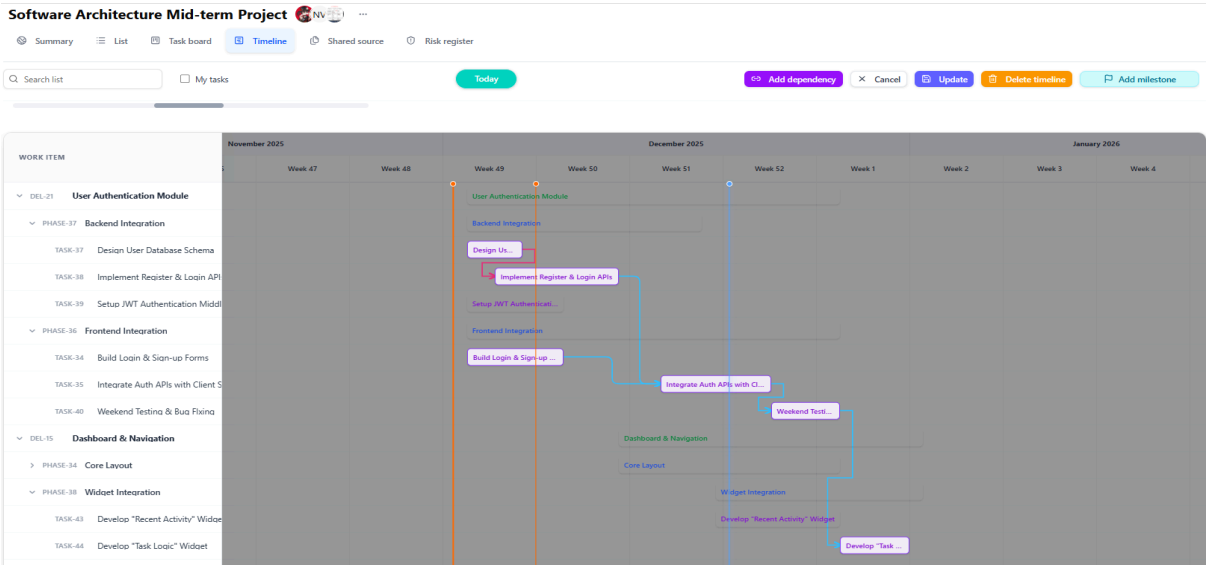

Image 3.18. Timeline page

- **Description:** The interface implements a **Gantt chart visualization** utilizing a strict **Finish-to-Start (FS)** dependency model, restricted to entities within the same hierarchical level. To enhance data clarity, the system features **interactive highlighting**: selecting an item triggers a visual emphasis on the specific timeline and its associated timelines. Temporal markers are integrated via vertical indicators, where **orange lines** signify custom milestones and a **blue line** denotes the current date.

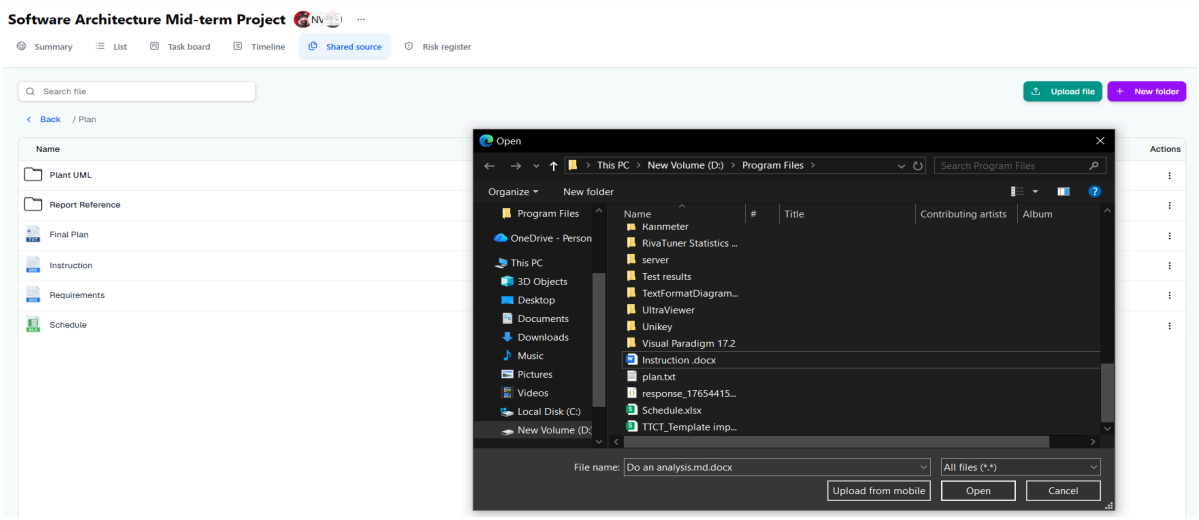

Image 3.19. Shared source page (1) - with the form for uploading a file from the local machine

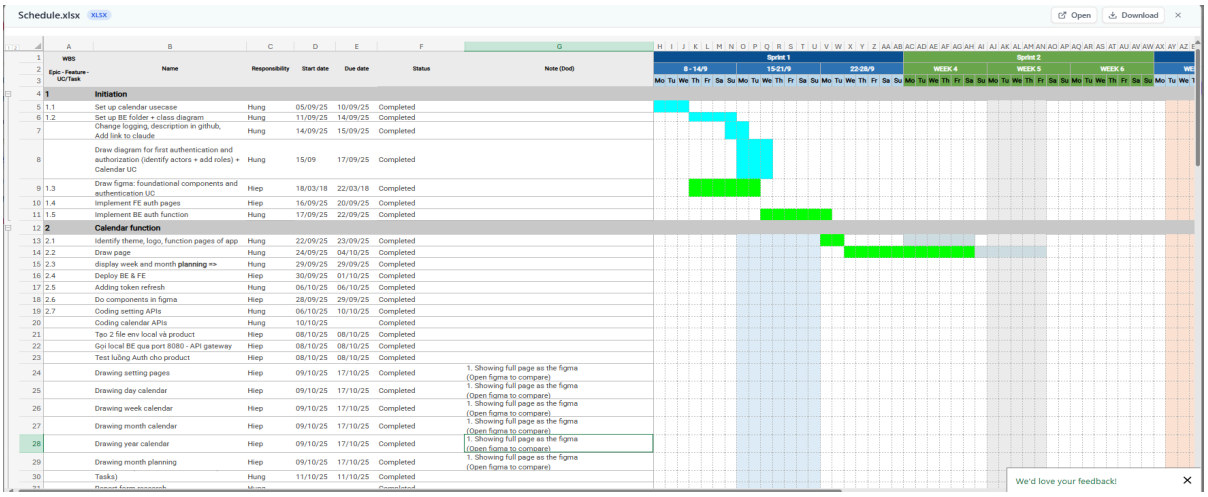

Image 3.20. Shared source page (2) - with the static file preview

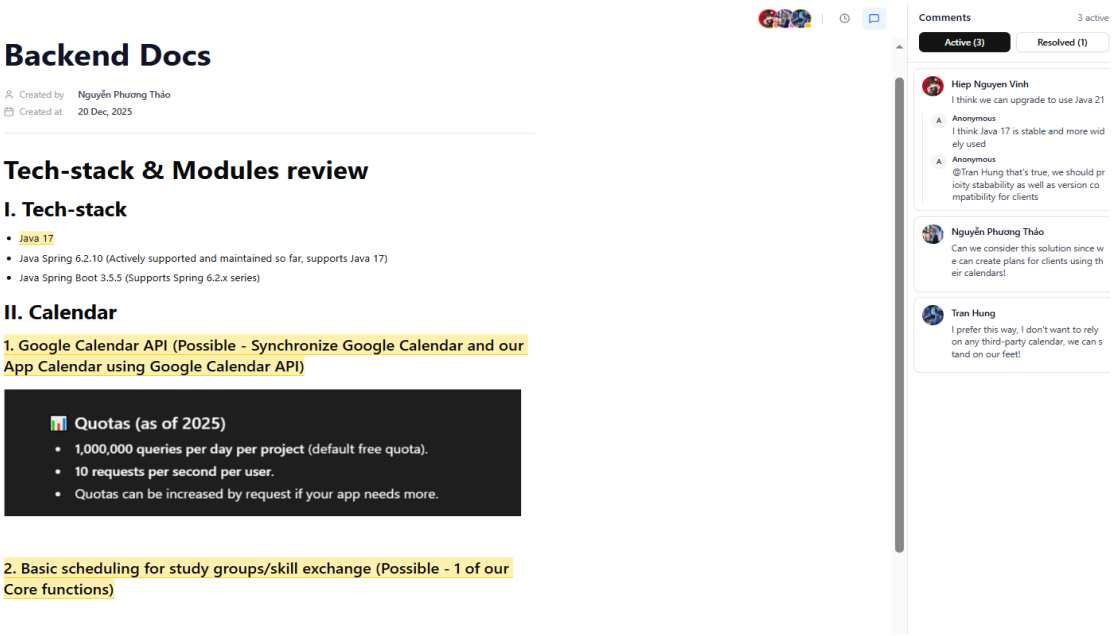

Image 3.21. Document (1) - with the 'all comments' form

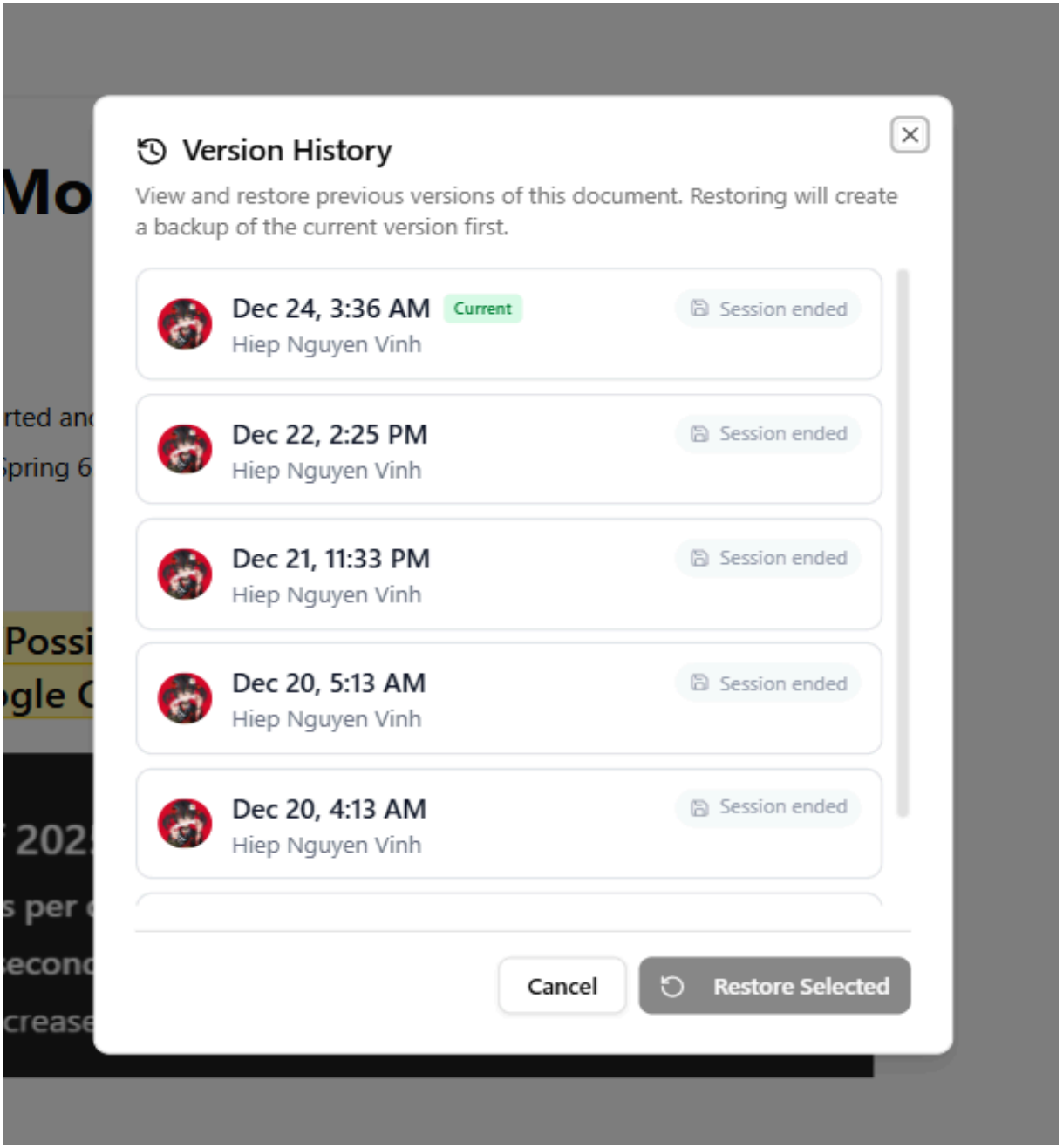

Image 3.22. Document (2) - with the version history

3.2.4. Implementation of the Forum Module

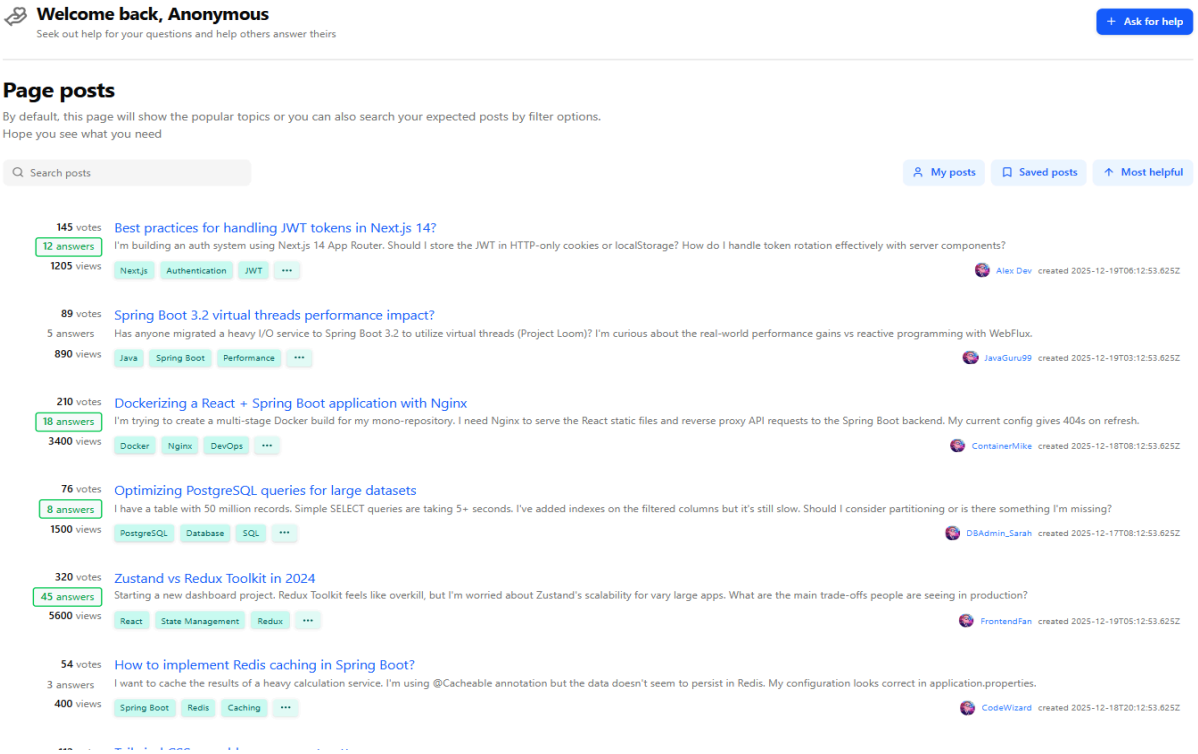

Image 3.23. Home page (1)

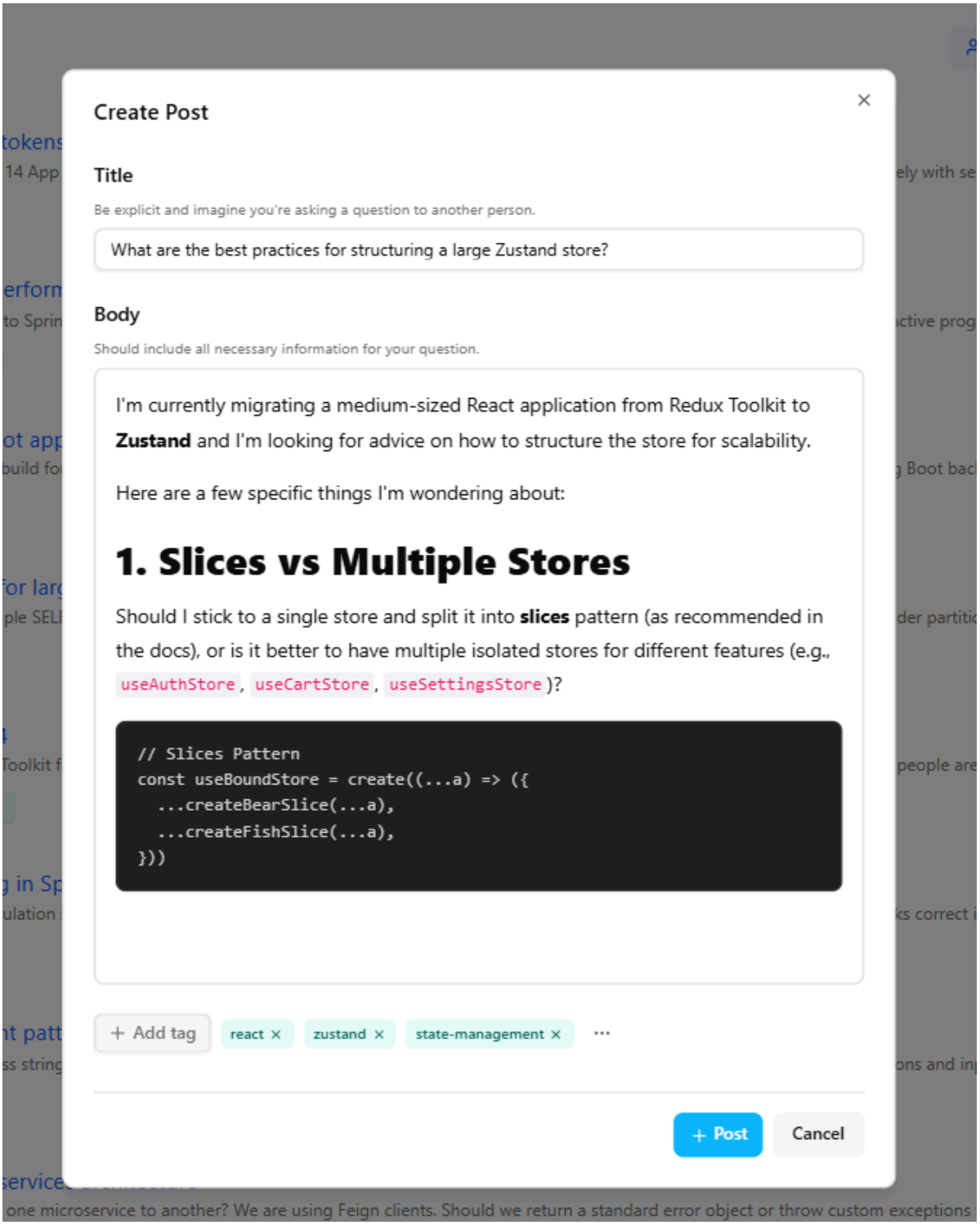

Image 3.24. Home page (2) - with the form of a post

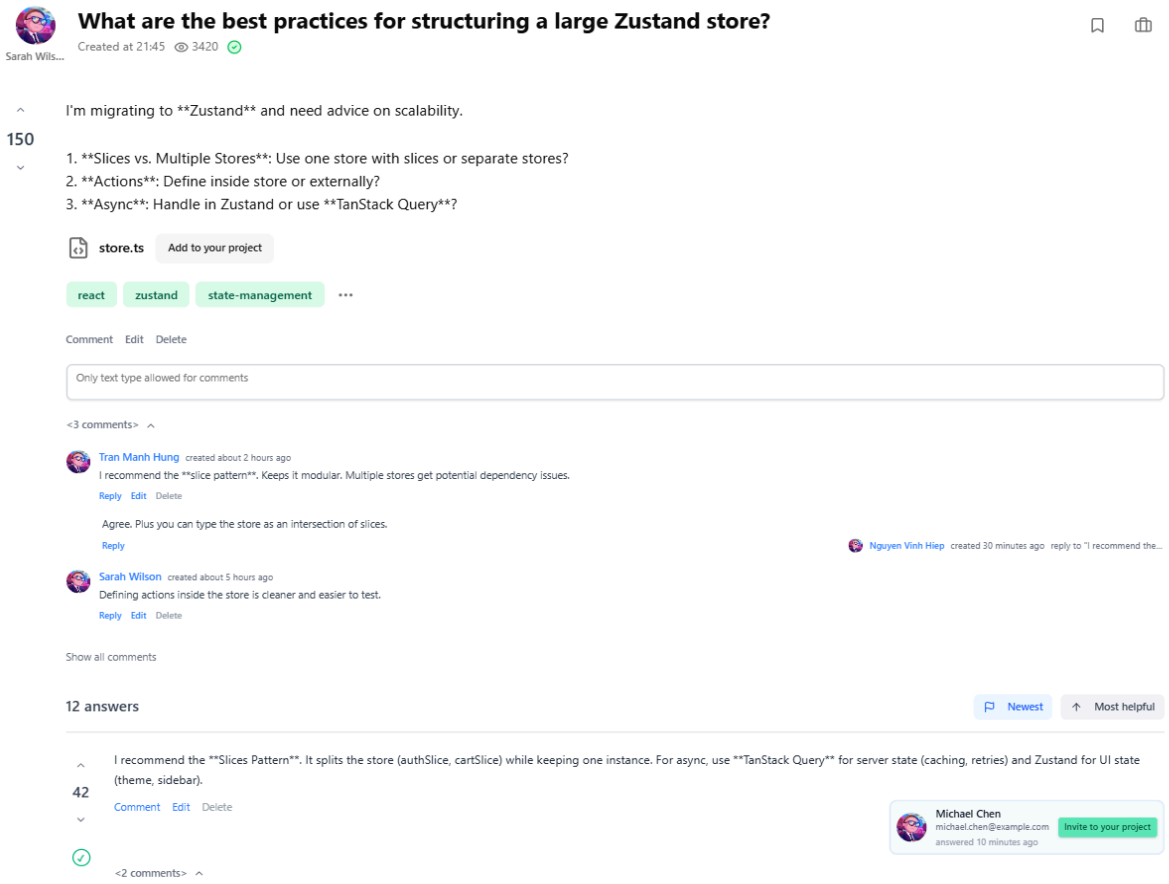

Image 3.25. Detail post

3.2.5. Implementation of the Supporting Module

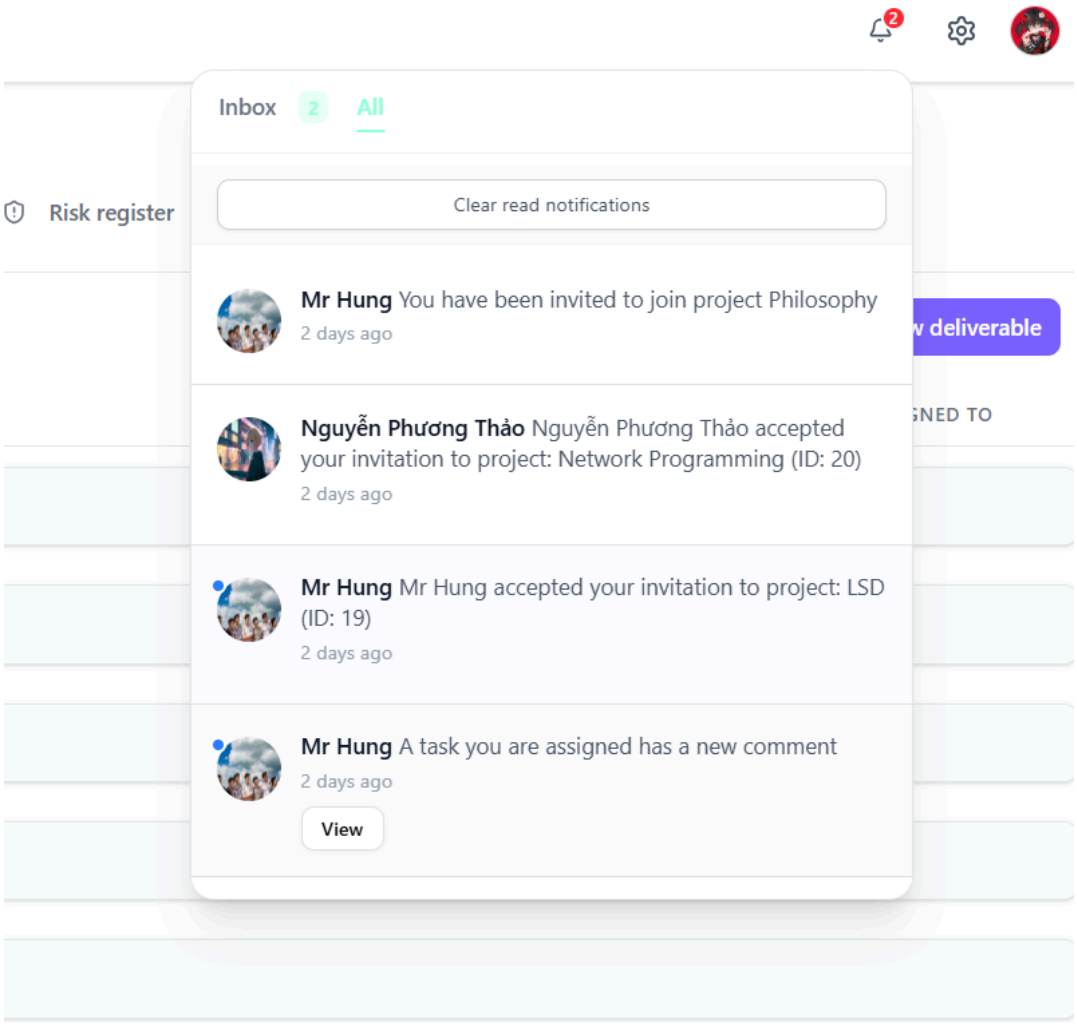

Image 3.26. Notification form

3.3. Chapter 3 Conclusion

Chapter 3 serves as the practical realization of "Building a Collaborative System to Support Learning using Java and NextJS". Through the development and implementation phases, the theoretical designs and architectural requirements established in previous chapters have been transformed into a functional, high-performance web application.

In summary, the implementation results demonstrate that the selected technology stack is robust and capable of supporting the diverse needs of university students. This chapter proves that the system is not only technically viable but also provides a streamlined, unified solution that fosters an efficient and balanced academic lifecycle.

CONCLUSION

1. Summary of completed work.

The project successfully designed and implemented a unified "Collaborative System to Support Learning" to address the issue of tool fragmentation in the academic environment. The work completed includes:

- **System Analysis and Design:** Identified five primary actors (Student, Project Owner, Member, Post Creator, and Respondent) and defined comprehensive functional requirements for five core modules.
- **Microservices Architecture:** Developed a decentralized ecosystem using Java (Spring Boot) and Next.js, where each service (User, Scheduling, Project, Document, Forum, and Notification) operates with its own independent database to ensure high availability and scalability.
- **Real-time Collaboration:** Successfully integrated Hocuspocus and Yjs (CRDTs) to support simultaneous document editing with automatic conflict resolution.
- **Advanced Scheduling Logic:** Implemented a personal scheduling engine that respects user-defined well-being constraints, such as sleep hours and daily workload limits.
- **Event-Driven Communication:** Leveraged Apache Kafka as a messaging backbone to decouple services and handle high-throughput system notifications asynchronously.
- **Data Strategy:** Realized a polyglot persistence strategy using PostgreSQL for relational data, MongoDB for unstructured document/forum content, and Redis for session management and caching.

2. Achievements and limitations of the system.

- **Achievements**
    - **Centralized Ecosystem:** The platform effectively reduces context switching by consolidating task management, resource sharing, and community knowledge exchange into a single web application.
    - **Technical Resilience:** The use of a Microservices architecture provides fault isolation, ensuring that a failure in a non-critical service (e.g., the Forum) does not impede core functions like document editing.
    - **User-Centric Design:** Unlike industrial tools, this system accounts for the student's personal progression and health by blocking scheduling during configured sleep hours.

- **Secure Authentication:** Integrated Google OAuth 2.0 and JWT with a Redis-backed blacklist to ensure secure and streamlined user access.

- **Limitations**
  - **Operational Complexity:** The transition from a monolithic to a microservices model increased the difficulty of infrastructure management and inter-service communication monitoring.
  - **Infrastructure Overhead:** Running multiple independent services requires more significant server resources and higher operational costs compared to a single-unit application.
  - **Idempotency Challenges:** The Kafka "at-least-once" delivery model presents risks of duplicate message processing, requiring additional tracking logic.
  - **UI/UX Sophistication:** Implementing advanced real-time indicators, such as shared cursors and presence markers, remains challenging due to the evolving nature of high-level library support.

3. Future development directions.

To further enhance the platform's value and technical depth, the following directions are proposed:

- **Mobile Platform Expansion:** Transition the current web-exclusive application into a cross-platform mobile experience to support students on the go.
- **AI-Driven Assistance:** Integrate AI agents to provide personalized recommended posts in the forum module and automated schedule optimization based on student habits.

REFERENCES

[1] Geeksforgeeks:
https://www.geeksforgeeks.org/apache-kafka/apache-kafka-cluster-architecture
[2] Visual paradigm:
https://www.visual-paradigm.com/learning/handbooks/software-design-handbook/sequence-diagram.jsp
[3] Jira: https://www.atlassian.com
[4] Google Calendar: https://calendar.google.com
[5] Notion: https://www.notion.com
[6] Stack Overflow: https://stackoverflow.com