Technical Specification Document

1. Introduction

- **Project Title**: AI powered Student and Alumni support network
- **Document Version**: 3
- **Publication Date**: 03/12/2025
- **Prepared by**: Vasundhara Ravikumar
- **Topic leader**: Professor Tamás Storcz

2. Version history

#	Date	Consultant	Topic
1	2025.11.11	Professor Tamás Storcz	- Consulted about the way AI can be incorporated into my work - Was advised to research on the different kinds of AI and LLMs out there to see the models that meet the functional specification that I have mentioned in my document
2	2025.18.11	Professor Tamás Storcz	- Asked questions about each section of the technical specification document. - Clarified what is required as regards technicality in each section and how it ties up to the functional specification - Got some tips on what to think about when writing the documentation for each section of the document
3	2025.26.11	Professor Tamás Storcz	- What is the difference between the system architecture and component details? - How many users should the system support? - How can I determine the best architecture to use - Where should I speak in detail about APIs - How long should the presentation be? - Is it possible to design my system in such a way so as to make it scalable in case many users cannot be supported?
4	2025.10.12	Professor Tamás Storcz	- Consider the type of AI model to integrate regarding encoding, decoding and other such models

3. Purpose

Purpose Statement: Briefly describe the purpose of the document and what it aims to achieve.

The purpose of this technical specification document is to present a comprehensive engineering overview of the **AI-powered Student and Alumni Support Network**. It details the system's technical design, including the architecture, frameworks, APIs, database schema, and AI integration methods that will support both web and mobile platforms. The document outlines how the system's components will interact to enable intelligent matchmaking, secure data exchange, and real-time communication between users. It also establishes development standards, scalability considerations, and deployment strategies to ensure the platform operates efficiently across various devices and environments.

Scope: Define the boundaries of the technical specifications covered in this document.

In – Scope :

- Cross platform availability – The system will be developed as a web application as well as a mobile application, ensuring accessibility across desktops, tablets and mobile devices. Both systems will be built using Single Page Application (SPA) for consistent user experience across devices.
- Feature parity across platforms – Most of the core features such as notes collaboration, 'ask a doubt' threads, AI assisted topic linking and others will be available on both platforms. However, mobile-only functionalities may exist where required such as the geo – based location feature.
- Unified backend architecture – A single backend will serve both the web and mobile clients through standardized APIs, ensuring consistent data handling, user authentication and business logic across platforms.
- Design of the system architecture, including frontend, backend, database and AI components.
- Development of AI modules – This includes reccomender systems for mentor matching, hybrid ranking models for a personalized campus feed, text classification models for question routing, embedding models for content linking, and clustering models for study group formation. AI will be integrated through dedicated backend services and APIs, allowing the SPA frontend to request predictions, suggestions or reccomendations in real-time.
- Creation of a centralized database to store user profiles, notes, discussion threads, academic queries, mentor mapping data and activity logs.
- Integration of real time communication mechanisms e.g for chat, collaborative notes and notifications.

- Establishment of security measures, including authentication flows, role based access control, secure data transmission and privacy protections.
- Integration with external services such as email/notification systems and cloud storage.
- Deployment and hosting of the system on a cloud infrastructure with scalability considerations.

Out-of-Scope :

- Development of offline capabilities – The platform requires an active internet connection to function for both web and mobile applications
- Content creation, curation, or moderation – Writing FAQs, adding academic resources, validating alumni posts or managing discussion threads beyond the technical mechanisms for storage and retrieval.
- Integration with external university systems – For example the PTE Moodle or Neptun. The platform will use the same authentication method as neptun does – the 2 factor authentication system, however, besides this the system will function independently.
- Gamification features – Users will not be awarded any points, badges, streaks or rewards for using the app.
- Building new AI models from the ground up – This projekt is going to rely on existing pre – trained models and some light – fine tuning only based on the features of this app.
- Advanced analytics dashboards – Visualizations or detailed insights for admins e.g student engagement heatmaps or AI model performance metrics.
- Multi – language support – This application is only in its initial phase, hence it will support only a single language which in this case is English.
- Marketing, branding or any promotion related work – this includes social media integration beyond basic notifications.
- Hardware development – device specific sensors, or embedded systems – this application is only designed for standard web browsers and mobile devices.
- Manual data backup and recovery services – Beyond the automated backup mechanisms defined for the database environment, there will be no other forms of backup available to restore the data in case of any data losses by the system.
- Third party financial transactions – payment integrations or e – commerce functionality is not included in this application.
- Community moderation policies or legal governance protocols – all this falls under institutional control not technical implementation.

4. System Overview

System Description: High-level description of the system and its objectives.

The system is a **cross-platform Single Page Application (SPA)** designed to run seamlessly on both web browsers and mobile devices. Both clients interact with a **unified backend** that centralizes business logic, data processing, AI requests, and real-time communication. The application provides a streamlined user experience by maintaining state on the client side and updating only necessary UI components without full page reloads. The backend is responsible for authentication, user management, storage of academic content, recommendation logic, AI – assisted routing and real time communication.

Technical Objectives of the system :

- Deliver a responsive cross platform, user experience. The application should provide unified functionality across web SPA and mobile app. It should also maintain a consistent user state and smooth navigation.
- Enable real time chat and interaction. Messages should be delivered instantly across both platforms and users should be able to experience live typing updates, note editing and discussion update.
- Personalized feed generated from embeddings, semantic search and activity analysis.
- Scalability and modularity. The system should allow the future expansion of AI components, new academic features and additional communication tools.

Architecture Overview: A summary of the system architecture, including major challenges, designed components and their interactions.

The system follows a Clean Architecture design, complemented by dedicated real-time and AI service components. This approach has been chosen specifically because the core business logic will remain stable and independent during development while infrastructure components (databases, AI models, notification systems and real-time services) can evolve or even be replaced without the core of the application being affected.

The clean architecture will be used primarily for the backend, where the system is divided into layered components.

Designed Layers and Interactions :

- **Presentation layer (Web + Mobile SPA Clients)**
  - Provides a unified interface for users across devices
  - Handles user interactions like creating notes, posting questions, sending messages, viewing notifications, and browsing personalized feeds.

- Maintains the client – side state to ensure smooth updates without reloading the page.
- Sends user requests to the backend and displays results received from the system.

➔ Example in application :

- A student opens a discussion thread → views comments in real time
- A user scrolls through the feed → receives a personalized list of posts and threads based on what they usually show interest in.

◆ **Controllers (Interface Adaptors) – API Layers**

- Acts as the intermediary between the user – facing layer and the application logic.
- Translates incoming user requests into application level actions
- Validates the inputs and enforces access rules or security checks.
- Formats output from the application logic to be delivered back to users

➔ Example in application :

- Receives a request to generate a personalized feed → invokes appropriate use case
- Receives a new chat message → passes it to the message-handling use case and triggers real-time notification events.

◆ **Application (Use Case Layer)**

- Implements the workflows and business rules for the system's features
- Coordinates data between the domain entites and external services (AI modules, real-time events, storage)
- Makes decisions about what happens when a user performs an action, without knowing how external systems work

➔ Example in application :

- Personalized feed generation: selects relevant posts, invokes AI to rank content, combines results with business rules, and returns the final feed.
- Question routing: decides which mentors or peers should receive notifications for a query.

◆ **Domain Layer (Entities/Bussines rules)**

- Contains the core concepts and rules of the application

- Maintains the integrity of data and enforces business constraints
- Independent of how the application is delivered, stored or integrated with AI

➔ Example in application :

- `User` entity: defines roles, permissions, and user profile consistency.
- `Note` entity: defines ownership, version control, and valid edits.
- `MentorMatch` entity: ensures recommendations always include a confidence score and timestamp.

◆ **Infrastructure Layer**

- Implements all the external dependencies required by the system
- Provides mechanisms for storing and retrieving data, performing the AI computations and returning results, sending notifications and real time events and managing the shared resources or external services.

➔ Example in application :

- Saves notes, threads, and user activity in the database.
- Processes AI-based recommendation queries and vector searches.
- Broadcasts live messages or notifications to connected clients.
- Maintains backup and consistency of stored data.

How the layers interact (functionally) :

1) **User performs an action** → SPA (Presentation Layer).
2) **Controller** validates request → sends it to the Use Case Layer.
3) **Use Case Layer** applies application rules → interacts with Entities and Infrastructure.
4) **Entities** enforce core business rules → ensure valid results.
5) **Infrastructure** executes AI, persistence, or notifications → returns data to Use Case.
6) **Controller** formats results.
7) **Presentation** Layer displays output to user.

Major Challenges :

◆ Maintaining strict layer separation – Clean architecture relies on strict boundaries : inner layers such as entities and use cases cannot depend on outer layers (Controller, Infrastructure). During development, the infrastructure logic

and AI service calls could get leaked into use cases and this could make it difficult to test, maintain and extend the application.

- Integrating the AI modules – This application relies on external AI services for mentor matching, semantic search, question routing, etc. Each call introduces latency and potential unreliability(network errors, rate limits, service downtime). This makes it essential to implement asynchronous handling, caching, retries and possibly even fallback logic. Furthermore it will be tricky to balance performance with accurate AI results for a real – time system.
- Real time features – SPA requires instant updates for chat, notifications, and collaborative note editing. Integrating WebSockets or similar services while keeping business logic seperated is non – trivial. This means that there is a risk of race conditions, duplicated notifications, or inconsistent state. Testing real-time flows is more difficult than traditional request-response flows.
- Synchronizing relational and vector databases – For this application, relational databases will be used for structured data and vector databases for embeddings and semantic search. Keeping these two systems in sync while performing any updates will be challenging.
- Cross Platform SPA Support – The unified backend must handle requests from both the web and mobile SPAs. Different devices have varying screen sizes, performance and connectivity. The project may require the implementation of adaptive data handling, efficient caching and offline fallback.
- Security and Role based access control – This application has specific roles assigned to each user – students, professors, alumni and possibly more depending on the type of stakeholder you may be so implementing the system will require strict access rules. In addition protecting sensitive data such as the notes that students generate, chats, anything AI – generated will be critical.

Design options: High level technical questions with possible solution options and explanation of selections.

Questions :

- Which architecture provides a good balance of structure, scalability and ease of implementation for an application that uses AI integration and enables real – time communication between users?
- Should the application be implemented as an SPA or MPA for consistent cross – platform experience?
- Should AI be run via external APIs or custom – trained models?
- How can real – time messaging and notifications be delivered?

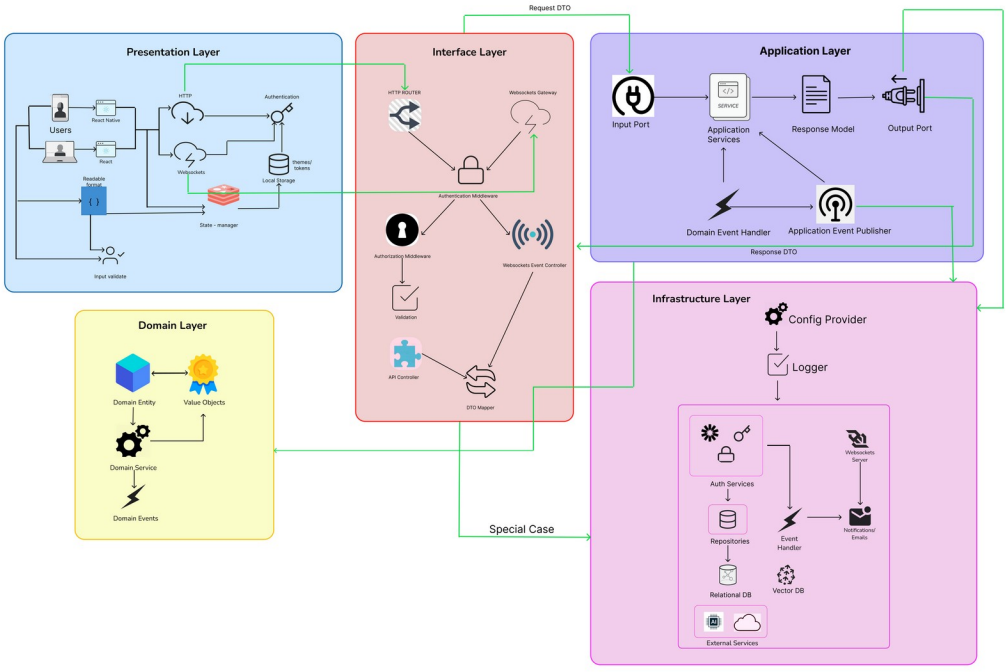
- What kind of storage models support structured user data and AI driven semantic search?

Solution options and explanation of selections

Technical Question	Considered Solutions	Chosen Solution	Reasoning
Architecture Type	Clean, Hybrid, Event driven, Microservices	Clean Architecture	Best balance of modularity, scalability, simplicity for prototype
App Type	SPA vs MPA	SPA	Required for real-time and mobile compatibility
AI Integration	External API calls vs AI built from the ground up	External APIs	Faster implementation, easier maintenance
Real – Time features	WebSockets vs Polling vs SSE	WebSockets	Needed for chat + real-time collaboration
Databases	SQL vs NoSQL vs Vector	SQL + Vector DB	Supports structure + semantic AI features

5. System Architecture

High-Level Architecture Diagram: Visual representation of the system architecture.



Component Descriptions: Detailed descriptions of each major component.

Domain/Entities Layer :

This is the innermost, most protected layer within the architecture.

Component 1: Aggregates

An aggregate is a cluster of domain objects (entities + value objects) that must remain **consistent** as a single unit.
Example in your app:

- A **ConversationAggregate** contains messages, participants, timestamps, permissions.
- A **UserProfileAggregate** contains profile info, academic info, interests, verification status.

The responsibility of an aggregate include grouping related domain data and rules into a transactional boundary, ensuring that the rules always remain true and enforcing business logic consistency across the data. For example, a conversation aggregate will ensure that only students, alumni or professors can post or respond to messages in the discussion panel, the thread status (open/closed) is respected and that messages append correctly.

The aggregates only interact with the application/use case layer. They do not interact with any other layers of the architecture because they must remain pure and independent. They should know nothing about APIs, databases, frameworks or any other infrastructure.

Component 2 : Aggregate Root

The aggregate root is the entry point to an aggregate. All external operations must go through this root. For example, Conversation is the root for messages, participants, typing indicators, read recipients.

Main responsibilities of the root include protecting invariant rules, control access to children entities, expose only safe operations.

The aggregate roots interact with the use case layer when they are called by them only. This keeps the domain consistent and prevents the application layer from modifying the „child entities” by mistake.

Component 3 : Entities

Entities are domain objects **with** identity that persist over time. Examples of entities include User, Message, Note, Conversation, Notification, etc.

Entities are responsible for storing domain state that may change over time and they contain business logic related to themselves.

Only the application layer manipulates entities. The infrastructure layer loads/saves entities, however, the entities themselves do not depend on the infrastructure layer.

Component 4 : Value Objects

Value objects are small, **immutable** domain objects **without identity**, defines only by their value. Examples of such objects include Email, Username, Message Content, Timestamp, etc.

Their responsibilities include validating and enforcing the correctness of data for example an email has to have a valid format, they cannot be modified and they need to help prevent invalid states from entering the domain.

They interact only within the domain layer, specifically with entities or aggregates.

Component 5 : Domain Events

Domain events represent *something significant that happend within the domain.* Examples include when a message is sent, when a user starts to follow a thread, if a profile has been updated, etc.

The responsibility of domain events include notifying the system of importnat domain changes and allowing decoupled reactions such as notifying users, triggering async tasks, etc.

Domain events are created inside the domain layer. They are published by the Application layer and are handled usually by application layer event handlers or infrastructure subscribers.

Component 6 : Domain Services

A domain service is domain logic that does not belong to a single entity, involves multiple aggregates or external rules and is strictly business logic. For example, matching a student to an alumni mentor(s), determining conversation visibility rules, AI query prepocessing, etc.

This component's responsibilities include holding pure business logic independent of infrastructure and implementing rules involving multiple entities.

This component is also called by the application layer in order to help perform computations such as what kind of mentor to match with a student based on their common interests, profile information, etc.

Component 7 : Domain Interfaces – Repository and Service interfaces

These are abstractions that define how the domain expects to load/save aggregates.

They are responsible for defining abstract operations including finding, saving or updating. They also decouple the domain from databases, APIs or frameworks. The actual implementation of this component lives in the infrastructure layer.

They are used by the application layer and implemented in the infrastructure layer.

Component integration: Specify the interfaces and communication protocols used for integration of components.

The components of the domain layer communicate only through method calls via interface abstractions and domain events emitted to the application layer. There are no protocols such as HTTP or WebSockets used here. The domain layer is framework – agnostic, transport – agnostic and database – agnostic making it the purest layer within the system architecture.

Application Layer Components

The application layer is responsible for coordinating domain logic and orchestrates workflows.

Component 1 : Use Cases/Application Services

Use cases define the actions the system can perform, independent of the UI or infrastructure. For example, sending a message, starting a conversation thread, registering a user, updating a user profile, etc.

Responsibilities of this component include coordinating the business logic by calling domain entities and aggregates, use repository interfaces to load/save aggregates, applying application – level rules (workflow logic, sequence of steps) and producing output models for the controllers layer.

It accepts input only through the input ports. It calls the domain aggregates, entities and domain services such as the user repository interfaces, publishes the domain events and returns the data through the output ports back to the controllers layer.

Component 2 : Input port :

The input port is the interface that defines what each use case requires as input.

The responsibilities of the input port include providing a stable API from the controller layer to the application layer and to make the use cases easy to mock and test.

The controller layer calles the input port. The DTO mapper from the controller feeds input DTO into the input port. The use case then implements the input port interface.

Component 3 : Response Model :

The response model is also called the Output DTO. This is the structured data that is produced by the application layer for the presentation layer. It only contains presentation friendly fields – no domain intervals.

It is produced by the application service and then consumed by the controller layer which serializes to a HTTP/WS response and then sends the data to the UI.

Component 4 : Output port :

The output port is an interface used by the use cases to send results to a presenter or view model. It abstracts the formatting/presentation concern away from the use case, meaning that it keeps the use case UI - agnostic.

It is called by the application service and implemented by controller – level presenter which maps the domain objects.

Component 5 : Domain event handler/Application event handler

The application event handler reacts to domain events emitted by aggregates for example message sent, user registered, etc. It performs follow up actions like sending notifications, calling the external APIs.

This component is triggered by a domain event and is routed by the application event publisher. It calls repositories, notification adapters, event bus, websocket adapter and even external APIs via infrastructure implementation.

The handlers can be executed synchronously or dispatched to a message queue for async processing.

Component 6 : Application Event Publisher

This is the interface that the use case calls to publish domain events to the rest of the system. It decouples event production from event consumption.

It is called by the application service after the state changes. It is implemented by the infrastructure layer – event bus, message queue adapter or the synchronous dispatcher. It delivers to the domain event handlers.

Controllers Layer Components

Component 1 : HTTP Router

The HTTP router is responsible for mapping incoming HTTP requests to specific API controllers and their methods. These include GET, POST, CREATE and other such requests.

It routes incoming HTTP requests to the correct controller method. It supports route grouping for example /auth/*, /thread/*, /messages/*. It applies global middleware such as logging, rate limiting, etc. Another major role of the router is to ensure seperation between transport level routing and the application logic.

In the controller layer, the HTTP router invokes validation middleware, auth middleware, the DTO mapper and the API controllers. It defines the order the middlewares are executed in. Outside the controller layer, the HTTP router is responsible for calling the input port in the application layer through the controller and sends the final response model generated in the application layer back to the presentation layer as a HTTP response for the user to view.

Component 2 : WebSocket Gateway

The WebSocket Gateway handles real – time, bidirectional communication with the frontend. It listens for an event such as „send_message”, „join_conversation”, etc.

Its role is to establish WebSocket connections and manage client sessions. It defines event channels (rooms) such as conversation : <id>, user : <id> : notifications, apply

connection – level authentication and forwards incoming WS events to the appropriate WebSocket Event Controller.

Within the controllers layer, the gateway works with websocket authentication middleware and forwards the events to the websocket event controller. Outside the controller layer, the websocket gateway calls the application input ports for real – time use cases and then receives outputs from the application use cases to broadcast the updates to the clients.

Component 3 : Authentication Middleware

The authentication middleware is used to verify the identity of the requester. It works for both HTTP and WebSocket connections.

The main responsibilites of the authentication middleware is to validate session tokens/JWTs, extract and attach user identity into the request context and reject any unauthorized/unverified users.

Within the controller layer, the authentication middleware runs before the API controllers and WebSocket Event controllers and works alongside authorization middleware. Outside the controllers layer, the authentication middleware is responsible for providing the application layer with the user that is currently logged in, however, it does NOT apply any business logic.

Component 4 : Authorization Middleware

This component ensures that the authenticated user has the correct permissions for the requested action.

It checks the user roles (student/alumni/professor), enforces rules like what roles are entitled to post threads, open discussions, etc.

Within the controller layer, the authentication middleware rejects unauthorized controller actions but outside the layer, it prevents invalid requests from ever reaching the application layer. Note that it does not contain any domain logic, only rules on role based access.

Component 5 : Validation Middleware

This component ensures that any incoming HTTP/WS data adheres to the required format. Its responsibilities include validating body/payload structure, enforcing a schema for example a message must not be empty, and preventing a malformed input from reaching controllers.

The validation middleware intercepts requests before reaching the controllers and works closely with the DTO mapper. As regards outside the controller layer, it guarantees that the input port in the application layer receives only safe and validated data.

Component 6 : API Controllers

HTTP controllers handle high level request orchestration for example ThreadController, MessageController, ProfileController, etc.

The API controllers accept validated input from the validation middleware. This input is then converted from an input DTO to a use case input model through the DTO mapper. It calls the appropriate use case/input port and then receives the output models from the application layer. Finally, this data is converted to a HTTP – friendly response.

Component 7 : Websocket Event Controllers

These controllers handle event – based actions over WebSockets such as „sendMessage", „StartTyping", „joinThread".

The main role of the WS event controller is to parse and validate the WS event payloads. They map the payload to the input model, call application use cases for real – time operations and broadcast the resulting output models to subscribed clients.

Component 8 : DTO Mapper

The DTO mapper convert between external API facing data (HTTP/WS DTOs) and internal use – case input models.

It normalizes incoming data, prevents presentation specific data from leaking into the application layer and converts domain output to API response DTOs.

Presentation Layer Components

Component 1 : Web SPA Client

The Web SPA runs in the browser and provides the full UI experience : pages, components, navigation, forms and interactive flows.

Some of the responsibilities of the Web SPA include rendering UI screens (feed, messages, profile, reccomendations, discussion, threads), handle client – side navigation without page reloads, capture user interactions (typing, clicking, form submissions), calling the backend HTTP APIs and WebSocket events and displaying responses received from the backend controllers.

The Web SPA client works with the state manager to store UI state, uses the local storage for authentication tokens and session metadata, sends data to input validation before making requests and uses the input normalizer to convert the input into readable format.

Outside the presentation layer, the Web SPA communicates with the controller layer through HTTP + WebSocket requests and receives DTOs/responses from API/WS controllers.

Component 2 : Mobile SPA Client

A mobile application built using a cross platform SPA framework with the same logical UI flow as the web version.

The role of the Mobile SPA client is to provide mobile-optimized UI for messaging, notifications, profile management, and posting threads, provide push notification support integrated with the backend Notification Adapter, use the same API/WebSocket interface as the web client.

The mobile SPA shares the same components with the Web SPA : State manager, Auth middleware, local storage. It uses mobile storage APIs instead of browser Local storage and it uses WebSocket + HTTP client libraries that are adapted for mobile.

The mobile client also communicates with the HTTP router and WebSocket Gateway in the controllers layer and receives outputs in the same DTO format.

Component 3 : HTTP Client (Frontend Gateway)

This is a thin service inside the SPA responsible for sending HTTP requests to the backend.

The HTTP client is used to handle API URL prefixes (e.g., `/api/thread`, `/api/auth`), attach tokens from Auth Middleware/Local Storage, handle request errors (network issues, 401 unauthenticated, etc.) and serialize/deserialize JSON payloads.

Inside the Layer:

- Used by UI Components and State Manager.
- Accesses tokens via Auth Middleware.

Outside the Layer:

- Sends HTTP requests to the Controllers Layer.
- Receives API Responses mapped by API Controllers.

Component 4 : WebSocket Client (Frontend Gateway)

This is a persistent real – time connection to the backend WebSocket Gateway. Its role is to listen for real – time updates (new messages, thread updates, typing indicators, reccomendations), emit events such as „send_message”, „join_thread”, „start_typing” and manage reconnection, heartbeat messages and connection health.

Inside the Layer:

- Updates the State Manager.
- Sends payloads validated by Input Validation.

Outside the Layer:

- Communicates with WebSocket Gateway and Event Controllers.
- Receives broadcasted events from Application Layer outputs (via WS controllers).

Component 5 : Authentication Middleware (Frontend)

The client – side authentication middleware manages the login state, token availability and UI level permissions. Its responsibilities include storing and refreshing JWT/session tokens, redirect unauthenticated users to the login page, attaching tokens to HTTP and WS requests and enforcing UI – level rules (hiding actions that normal users cannot perform).

Inside the Layer:

- Works with Local Storage for token persistence.
- Wraps HTTP Client and WebSocket Client.

Outside the Layer:

- Interoperates with backend Authentication Middleware (Controllers Layer).
- Provides identity metadata for authenticated requests.

Component 6 : Local Storage/Mobile Secure Storage

Client – side persistent storage for authentication data, UI preferences, session state and cached data. The roles of this component include storing tokens, storing the last selected conversation/thread, and other preferences such as theme preferences, onboarding state, etc.

Within the layer, the local storage exposes the tokens to the authentication middleware and provides the cached state to the state manager. It does not take any part in providing any sort of data to other layers.

Component 7 : State Manager

A global or modular client – side store that holds the UI state which must persist across the components.

Its main role is to store application – wide UI state – the current conversation, user profile, unread notifications, reccomended mentors/student matches, discussion thread posts, etc. It also has to listen to WS events and update the UI reactively and should orchestrate optimistic updates like instant message displays.

Note that this component never communicates with any other layers of the architecture.

Component 8 : Input Validation (Client – side)

Lightweight client – side validation (before sending requests) to avoid bad input and provide instant error feedback. Its responsibility is to validate form – fields (non – empty message, valid email format, etc), provide instant error feedback to users and prevent uneccessary API calls with invalid data.

Outside the presentation layer, the input validation complements the controller side validation middleware however, it does NOT replace it.

Component 9 : Input Normalizer/Input Formatter

Converts raw user input into a readable, normalized form before sending it to the backend. It ensures that input data is clean and consistent, prevents malformed or unreadable messages and improves user experience by formatting content before sending.

This cleaned data is then forwarded to the controllers layer.

Infrastructure Layer Components

Component 1 : Configuration Provider

A centralized provider that loads environment variables, secrets, crednetials, ports, database URLs and all other runtime configuration values.

Its responsibilities include loading configuration from .env, environment variables, secret manager, or config files, validate required configuration (or fail fast if missing), provide typed access to config values (e.g, config.db.url, config.jwt.secret) and ensure consistent configuration across all services.

Inside the Layer:

- Used by all infrastructure components including DB clients, auth service, email adapters, event bus, vector DB client.

Outside the Layer:

- Exposed to the Application Layer through DI when initializing services.

Component 2 : Logger

This is a structured logging system that records all the runtime events, errors, audits and metrics. It produces logs in a structured JSON format, log errors, warmings, info messages, performance timings, it is integrated with montoring tools such as Datadog and it supports request correlation Ids and tracing.

Inside the Layer:

- Used by repositories, external service adapters, event handlers, and WebSocket server.

Outside the Layer:

- Application layer uses logger to record use-case-level events.
- Controller layer uses logger for API/WS request traces.

Component 3 : Authentication Services (Infrastructure Auth Provider)

This is the concrete implementation of the authentication logic such as JWT signing, hashing, token verification, Oauth integration, etc.

The authentication services generate and validate JWT tokens, hash and verify passwords, handle refresh token rotation, integrate with identity providers and provide user identity metadata for controller middleware.

Inside the Layer:

- Uses Config Provider for secret keys.
- Uses Logger for auditing login/logout.

Outside the Layer:

- Called by Authentication Middleware in Controller Layer.
- Used indirectly by Application Layer for login/register use cases.

Component 4 : Repositories (Implementing Domain Repository Interfaces)

This is the concrete implementation of domain-defined repositry interfaces. Examples include UserRepositoryImpl, ConversationRepositoryImply, etc. They map the domain entities to database schemas, execute SQL/ORM queries to load/save aggregates, perform transactions when required and convert persistence models into domain models.

Inside the Layer:

- Uses Relational DB Client for SQL operations.
- Uses Logger for query tracing.

Outside the Layer:

- Application Layer calls repository interfaces.
- Domain Layer interacts with repositories only through interfaces.

Component 5 : Relational Database (Primary data store)

This is the primary database that stroes structured data such as users, messages, threads, profiles, etc.

It is used to persist domain aggregates and entities, enforce schema constraints (Fks, unique keys, indexes), provide transactional guarantees for critical workflows and perform query optimization via indexes and stored procedures (if needed).

Inside the Layer:

- Accessed by repository implementations.
- Uses Config Provider for credentials.

Outside the Layer:

- Never accessed directly — only through repositories.

Component 6 : Vector Database

This database stores semantic search records, alumni – student mentor matching reccomendations and user – interest matching. The vector database stores embeddings for users, interests, profiles and conversations, performs vector similarity search (cosine distance, dot product), supports AI powered reccomendations – mentor matching and thread discovery and is integrated with a LLM/embedding model pipelines.

Inside the layer :

- Accessed by a Vector Repository or Vector Adapter.
- Uses Logger and Config Provider.

Outside the Layer:

- Application Layer calls vector search through a domain interface abstraction.
- Domain Layer never knows the vector DB exists.

Component 7 : External Services (Adaptors to 3rd – party APIs)

These are adapters for communicating with external systems such as LLM services, storage services, push notification systems, etc.

They convert domain/application requests to external API calls, handle authentication with third – party services and retry logic, timeout handling and circuit breaking.

Inside the Layer:

- Uses Logger + Config Provider.
- May use caching layer for optimization.

Outside the Layer:

- Application Layer calls these via interface abstractions.

Component 8 : Application Event Handler (Infrastructure subscribers)

Infrastructure level subscribers that respond to domain/application events and perform side effects. They receiver the published events such as MessageSent, UserRegistered, etc, trigger notifications, update search indexes or push WS events, persist event logs for auditing (optional) and execute async tasks using worker queues.

Inside the Layer:

- Uses Event Bus, Logger, Email/Notification Adapters, Vector DB, or External APIs.

Outside the Layer:

- Events are published by Application Event Publisher (Application Layer).
- Domain emits events → Application → Infrastructure handles them.

Component 9 : Email and Notification Adapter

This is the adapter responsible for sending emails, push notifications, or in – app notifications. It formats and sends verification emails, mentor matches, unread – thread alerts, etc.

Inside the Layer:

- Used by Event Handlers when responding to domain events.
- Uses Config for API keys and Logger for delivery logs.

Outside the Layer:

- Application Layer triggers notifications indirectly via events.

Component 10 : WebSockets Server (Real – Time gateway)

This is a low – level WebSocket transport layer that enables real-time bi-directional communication between clients and the backend. Maintains active WebSocket connections, authenticates WS clients, broadcasts domain events and application outputs to connected users and provides channels/rooms e.g per thread, per conversation.

Inside the Layer:

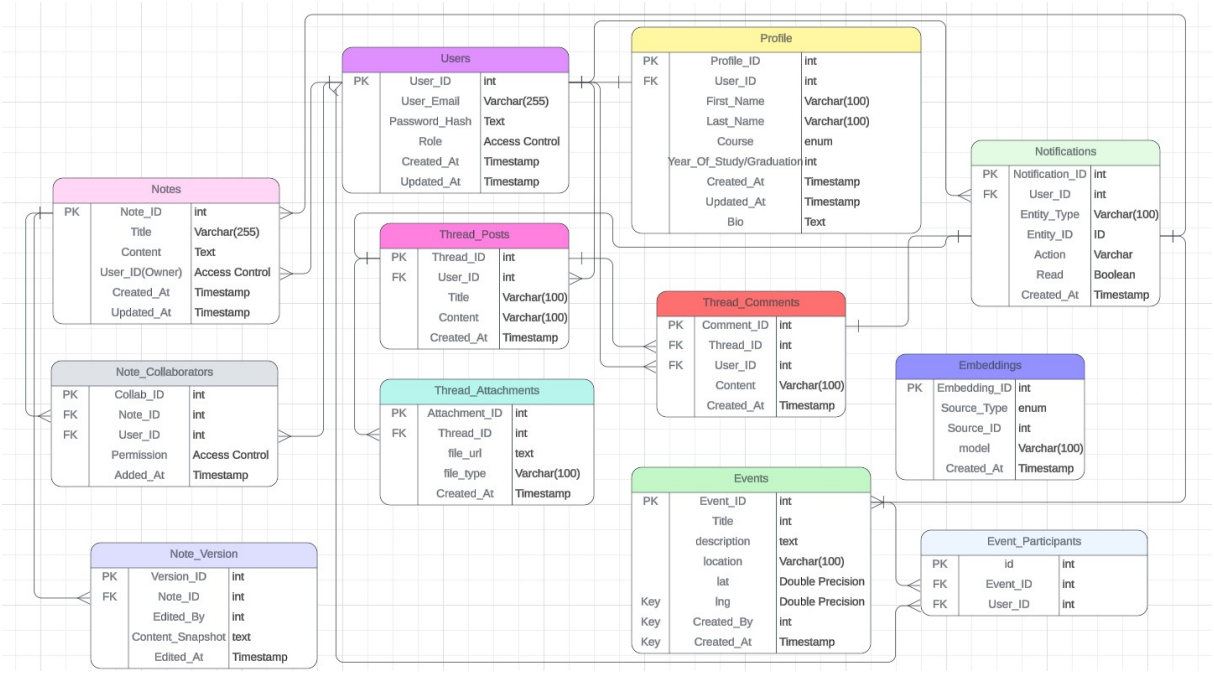
- Works with Controller Layer WebSocket Gateway controllers.
- Uses Logger for connection events.
- Receives events from Event Handler to broadcast.

Outside the Layer:

- Controller WS layer pushes events into it.
- Clients (Web SPA & Mobile SPA) connect through it.

6. Data Design

Data Model: Entity-Relationship Diagrams (ERD) or other data models.



Database Schema: Description of the database schema, including tables, fields, and relationships.

Entity	Attributes	Relationships
Users	User_ID, User_Email, Password_Hash, Role, Created_At, Updated_At	Profile, Notes, Thread_Posts, Note_Collaboration, Note_Version, Notifications, Thread_Comments, Event_Participants, Embeddings
Profile	Profile_ID, User_ID, First_Name, Last_Name, Course, Year_Of_Graduation, Created_At, Updated_At, Bio	Users
Notes	Note_ID, Title, Content, User_ID (owner), Created_At, Updated_At	Users, Note_Collaboration, Note_Version
Note Collaboration	Collab_ID, Note_ID, User_ID, Permission, Added_At	Notes, Users
Note_Version	Version_ID, Note_ID, Edited_By,	Notes

	Content_Snapshot, Edited_At	
Notifications	Notification_ID, User_ID, Entity_Type, Entity_ID, Action, Read, Created_At	Users, Notes, Threads, Events
Thread_Posts	Thread_ID, User_ID, Title, Content, Created_At	Users, Thread_Attachements, Thread_Comments
Thread_Attachements	Attachment_ID, Thread_ID, file_url, file_type, Created_At	Thread_Posts, Thread_Comments
Thread_Comments	Comment_ID, Thread_ID, User_ID, Content, Created_At	Thread_Posts, Thread_Attachments, User
Events	Event_ID, Title, Description, location, lat, lng, Created_By, Created_At	Event_Participants
Event_Participants	ID, Event_ID, User_ID	Events, User_ID
Embeddings	Embedding_ID, Source_Type, Source_ID, model, Created_At	

7. Interface Specifications

User Interface (UI): Wireframes, mockups, and descriptions of the user interface components.

UI Component	Description	Key Interactions
Login and Registration	Institutional login for students & professors; alumni use previous Neptun email	Email domain validation, error handling
Home Dashboard	Personalized sections, search bar, recommended mentors	Uses semantic search + recommendation engine
Question Posting UI	User types question, shows similar threads	Calls `/questions/similar` API in real time
Thread Page	Full view of question, answers, related threads	Uses similarity search for related content
Alumni/Professor Profiles	Experience, skills, matching score	Uses embeddings for match scoring

Admin Dashboard	User/thread management, AI performance	Uses admin API endpoints

Application Programming Interface (API): Detailed descriptions of APIs, including endpoints, request/response formats, and authentication methods.

Category	Purpose	Examples
Authentication	Login, registration, session handling	/auth/login, /auth/register
User Management	Profile update, role – based functions	/users/me, /users/update
Questions and Threads	CRUD operations for threads	/questions, /questions/:id
Similarity Search	Semantic lookup of similar threads	/questions/similar
Mentor Clustering	Reccomending alumni clusters to students	/mentors/reccomend
AI Embedding Services	Generate embeddings, serach vectors	/ai/embedded, /ai/search

Key REST API Endpoints

Endpoint	Method	Description	Auth Required
/auth/register	POST	Registering a new user	No
/auth/login	POST	Generate JWT	No
/users/me	GET	Fetch current user	Yes
/users/update	PATCH	Update profile	Yes
/questions	POST	Create a question + generate embeddings	Yes
/questions/:id	GET	Fetch full question and replies	No

/mentors/ reccomend? studentId = x	GET	Reccomend alumni clusters	Yes
/ai/embedded	POST	Generate embedding using Cohere	Internal Only
/ai/search	POST	Perform vector similarity search	Internal Only

External Interfaces: Specifications for any external interfaces or integrations.

Cohere Embedding API

Item	Description
Provider	Cohere
PurposeModel	Embeded (Encoding Model) – english – v3.0 or newer
Input	Generate semantic embeddings for questions, user profiles, alumni experience
Output	Raw text
Used for	Vector array
Authentication	Similarity search, clustering, mentor recommendations, duplicate prevention
Required	Yes – core AI feature

Vector Database Integration

Option	Type	Pros	Cons	Fit for app
Pinecone	Managed SaaS	Fast, auto scaling	Costly at scale	Good but expensive
Qdrant Cloud	Managed/self - hosted	Cheaper, great recall	Some ops needed	Recommended
PostgreSQL + pgvector	Self - Hosted	Cheapest, simple	Not best for huge scale	Best for student budget

Email Authentication Provider

Item	Description
Purpose	Validate Neptun email domains
Integration	Simple SMTP or institutional Oauth (optional)
Required	Yes(for institutional access rules)

File Storage

Item	Description
Purpose	Store attachments, profile images
Options	AWS
Required	Yes

Geo – Location Service

Item	Description
Purpose	Convert user IP or coordinates into country/city/region for personalization and analytics
API Options	**IP-based:** ip-api.com, ipwhois.io, MaxMind GeoLite (free) **GPS reverse geocoding:** Google Maps Geocoding API, Mapbox, OpenStreetMap
Data Returned	Country, City, Region, Latitude/Longitude, optionally ISP
Used for	Showing region-specific alumni clusters, analytics, validating alumni signup, improving recommendations
Authentication	API Key (for Google/Mapbox), none (ip-api free tier)
Requirements	Not strictly required for MVP, but strongly

	recommended for matching quality

8. Non-Functional Requirements

Performance: Performance requirements, such as response time and throughput.

- REST API responses should return in **< 300 ms** for 95% of requests (excluding AI endpoints).
- WebSocket message delivery should occur in **< 150 ms** normally and **< 300 ms** under peak load.
- AI mentor-clustering and semantic search should produce results within **1–2 seconds**.
- Relational database queries should execute in **< 50 ms** on average.
- Vector search queries should execute in **< 150 ms**.
- System must support **5,000+ concurrent users** initially, scalable to **20,000+**.
- System should process **100 chat messages per second** during peak hours.
- SPA initial load time should be **< 2 seconds** on broadband and **< 5 seconds** on slower mobile networks.

Security: Security requirements, including authentication, authorization, and data protection.

- Use secure authentication (JWT + HttpOnly cookies, or secure mobile tokens).
- Enforce strict authorization using **Role-Based Access Control** (Student, Alumni, Professor, Admin).
- All communication must use **TLS/HTTPS** (WebSockets must use wss://).
- Passwords must be hashed using **bcrypt**.
- Sensitive fields should be encrypted at rest where necessary.
- Input validation must occur in the controller layer before hitting use cases.
- Apply API rate limiting (e.g., 200 requests/min per user).
- Maintain audit logs for sensitive operations (login, profile updates, admin actions).
- Use short-lived access tokens and secure refresh token storage.

Usability: Usability standards and requirements.

- The SPA must maintain consistent UI/UX across web and mobile.

- The app must comply with **WCAG 2.1 AA accessibility standards**.
- Users should be able to complete onboarding (setup + profile) in **< 3 minutes**.
- Key actions should require **≤ 3 clicks/taps**.
- Provide visual feedback for all actions (loading indicators, success/error messages).
- SPA should support graceful offline behavior with cached data (optional for MVP).

Reliability: Requirements for system reliability and availability.

- Target **99.5%** system availability for MVP.
- WebSockets should automatically reconnect after interruptions.
- HTTP requests should include automatic retry logic for transient failures.
- Perform daily backups of the relational database; perform weekly backups of vector DB.
- System must support regional failover or recovery within 30 minutes.
- Chat system should guarantee **at-least-once delivery** through queued or buffered messages.

Scalability: Requirements for system scalability.

- API servers, WebSocket gateway, vector search service, and queue processors must scale horizontally.
- Traffic must be distributed using load balancing (sticky sessions optional).
- Relational DB should support read replicas; vector DB should support sharding.
- AI services must support batching, caching, and parallel processing of embeddings.
- Image/object storage must scale automatically with uploads.
- Application servers should be stateless to support scaling in/out.

Compliance: Legal and regulatory compliance requirements.

- System must comply with **GDPR** (right to delete, export data, data minimization).
- If educational records are stored, treat them according to **FERPA-like principles**.
- Users must be informed when AI is used (matching, recommendations).
- Logs must avoid storing sensitive personal data.
- All data must be stored within **EU regions** if required by the university.

- Accessibility guidelines (WCAG 2.1 AA) must be followed.

9. Technology Stack

Programming Languages: List of programming languages to be used.

Layer	**Language**
Frontend (Web)	TypeScript + React
Mobile	TypeScript + React Native
Backend	TypeScript (Node.js + Nest.js)
AI/Embeddings Scripts	TypeScript

Frameworks and Libraries: Description of frameworks and libraries to be used.

Option	Benefits	Drawbacks	Decision
NestJS (with TypeScript)	- Ideal for Clean Architecture - Built-in modules (WS, Auth, Config) - Decorators make controllers clean - Works well with DTOs	- Slight learning curve	**Selected** — perfect match based on architectural style.
Express.js	- Simple - Flexible	- No built-in structure - Requires manual layering	Not chosen — clean, layered architecture needed.
Fastify	- Very Fast	- Less documentation	Not chosen — lacks the ecosystem NestJS provides.
Django REST	- Batteries included	- Not ideal for WS real-time	Not chosen.

Selected Backend Libraries

Purpose	**Library**
WebSockets	NestJS Gateway/Socket.IO
ORMs	Prisma (SQL)
Validation	Class - Validator

Authentication	Passport + JWT
AI Calls	Axios + OpenAI/Anthropic APIs
Logging	Winston or Pino

Selected Relational Database

Option	**Benefits**	**Drawbacks**	**Decision**
PostgreSQL	- Best relational DB - Great JSON support - Strong indexing	- Slightly more complex than MySQL	**Selected** — best overall choice for structured profile + messaging data.
MySQL	- Simple	- Weaker JSON support	Not chosen — PostgreSQL fits modern apps better.
SQLite	- Lightweight	- Not Scalable	Not chosen — insufficient for real-time multi-user app.

Selected Vector Database

Option	Benefits	Drawbacks	Decision
PostgreSQL + pgvector (Chosen)	- Uses your existing relational DB (no third database) - No external paid services needed - Easy to maintain, host, and scale on one system - Supports similarity search (cosine/L2/inner product) - Cheaper than Pinecone, Qdrant, Weaviate - Works perfectly for AI semantic search & alumni-student matching	- Slightly slower than Pinecone at massive scale (millions of vectors) - More manual tuning needed (indexes, performance)	Selected — best balance of cost, simplicity, and integration for this project.
Pinecone	- Fully managed - Very fast	- Paid service	Reliable managed vector

	- Good metadata filtering		search for recommendations + matching.
Qdrant	- Open-source - Fast	- Requires hosting & scaling	Not chosen — adds DevOps overhead.
Weaviate	- Feature rich	- More complex to operate	Not chosen — Pinecone is simpler for a prototype.

Development Tools: List of development tools and environments.

Tool	Purpose
VS Code	Main Editor
Postman	API Testing
Docker	DB + Backend Containers
Github	Version Control
Github Actions	CI/CD
Expo	React Native mobile development
Prisma Studio	DB Browser
Swagger/OpenAPI	API Documentation
ESLint + Prettier	Linting and Formatting

Deployment Environment: Description of the deployment environment, including hardware and software.

Option	Benefits	Drawbacks	Decision
Dockerized Deployment (Docker Compose + VPS)	- Full control - Cost efficient - Easy to scale gradually	- Requires DevOps knowledge	**Selected** — perfect balance of cost + flexibility.
AWS (ECS, RDS, API	- Fully scalable	- Expensive for	Not chosen — cost

Gateway, etc.)	- Highly reliable	student projects	too high for a prototype.
Vercel (Frontend) + Railway/Render (Backend)	- Very easy deployment - Automatic scaling	- Limitations on WebSockets - Vendor lock-in	Partial — can deploy frontend here but backend needs stable WS.
Heroku	- Easy	- Expensive now / no free tier	Not feasible.

Final Deployment Architecture

Component	Deployment
Web SPA	Vercel/Netlify
Mobile App	Expo build + app stores/APK
Backend API + WS Gateway	Dockerized Node.js service on VPS
PostgreSQL	Managed DB or Docker container
Vector DB	PG Vector
Storage (images)	AWS S3
CI/CD	Github Actions