CS3219: Project Report

Group 26

No.	Full Name (as in EduRec)	Student Number
1	Andre Sim Jing Xiang	A0257936M
2	Ashley Goh Junting	A0266160J
3	Eunice Koh Shu Ning	A0256986E
4	Rayson Chia Hao Zhe	A0261520U
5	Yong Shi Kang	A0265713A

Table of Contents

Acknowledgment

We would like to extend our gratitude to all individuals that supported and guided us throughout the PeerPrep project. Special thanks to our mentors and lecturers for their invaluable insights, encouragement, and constructive feedback. We would especially like to thank Shufan for his prompt responses and the unwavering support he provided whenever we needed assistance; his guidance was instrumental in keeping us on track.

We also appreciate the hard work and dedication of our team members, whose collaboration and commitment to high-quality work made this project a success.

1 Introduction

1.1 Project Background

PeerPrep is a technical interview preparation platform that aims to help students practice interview questions collaboratively. Many students preparing for technical interviews benefit from practice with peers in a simulated interview environment. This project seeks to address this need by providing a platform where students can connect, and collaborate in real time on coding problems. The design utilises a microservices architecture for flexibility, scalability, and ease of maintenance.

1.2 Project Purpose

The purpose of PeerPrep is to create a structured, supportive platform where students can engage in interactive, whiteboard-style problem-solving sessions. By matching students based on criteria such as difficulty level and topic. The platform enables meaningful peer learning and collaborative preparation for technical interviews. PeerPrep's features are designed to mirror real interview scenarios, fostering familiarity with interview dynamics, improving technical proficiency, and increasing users' confidence in coding under timed, interactive conditions.

1.3 Intended Audience and Reading Suggestions

PeerPrep is primarily designed for individuals who are looking to improve their technical interview skills and ultimately land a job in the software industry. This includes a broad spectrum of users, from beginners to experienced professionals. For students actively preparing for technical interviews, especially those pursuing software engineering or technology-related roles, this platform serves as a valuable tool for collaborative practice.

For technical audiences, such as developers and engineers, it is recommended to focus on sections related to the microservices architecture, backend services including the User, Question, Matching, and Collaboration services. These sections provide detailed insights into design choices and implementations. Non-technical readers can start by exploring the project background, purpose, and key features to understand how PeerPrep helps users prepare for technical interviews. Sections describing the frontend pages and user flow will also provide useful information on how users interact with the platform and navigate its various features.

1.4 Instructions on how to run locally

Step 1: Clone the Repository
Clone the PeerPrep repository to your local machine. Replace <repository-name> with the project folder name of your choice.

git clone https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g26.git <repository-name>
cd <repository-name>

Step 2: Install Docker
Ensure that Docker Desktop is installed on your device. You can do so here.

Step 3: Set Up Environment Variables
Before starting the services, create an environment file (.env) in the root directory of your project. This file will hold all the necessary configuration settings for each service.

- In the root directory of your project, create a new file named `.env`.
- Edit the contents of the .env file. Copy the contents of Appendix section 11.1 below and paste it into the .env file. This will populate your .env file with the required settings for MongoDB, RabbitMQ, email credentials, JWT secrets, and other configurations.

Note: Because .env contains sensitive information, we made sure it's included in .gitignore to prevent it from being committed to version control.

Step 4: Build and Start the Services
From the root directory, use Docker Compose to build and start the containers. This will set up all required services and supporting services. Before you run the command, ensure that docker daemon is running by turning on your Docker Desktop.

docker-compose up --build

This command builds each service's Docker image based on the docker-compose.yml file in the root directory. You should see logs from each service indicating they are starting up.

2 Requirements Specification

2.1 Functional Requirements

Functional Requirements	**Priority**	**Planned sprint**
F1: User Service		
F1.1 Account Creation		
F1.1.1 Allow users to create an account by entering their username, email, and password.	H	Sprint 4
F1.1.2 Ensure that each new account has a unique username and email to prevent duplication and maintain account integrity.	H	Sprint 4
F1.1.3 Store user information securely in the database, including hashed passwords to ensure data security.	H	Sprint 4
F1.1.4 Provide appropriate feedback if issues arise during registration, including password strength, duplicate entries, or missing information.	M	Sprint 4 + 6
F1.2 User Authorization		
F1.2.1 Enable users to log in securely with their registered email and password.	H	Sprint 4
F1.2.2 Provide feedback for invalid login attempts.	H	Sprint 4
F1.2.3 Allow users to initiate a password reset process if they forget their password.	M	Sprint 5
F1.2.4 Handle errors during the password reset process, such as incorrect email formats or unregistered emails.	M	Sprint 5
F1.2.5 Send a password reset link or code to the user's registered email and notify the user to check their inbox.	M	Sprint 5
F1.2.6 Allow users to resend the password reset token if the initial email was not received.	M	Sprint 5
F1.2.7 Redirect users to the password reset page upon successful entry of a valid reset code.	M	Sprint 5
F1.2.8 Redirect users to the login page after a successful password reset.	M	Sprint 5

F1.2.9 Provide feedback for incorrect reset codes or expired tokens.	M	Sprint 5
F1.2.10 Restrict access to internal pages for unauthenticated users, automatically redirecting them to the main page.	M	Sprint 5
F1.3 Account Management		
F1.3.1 Allow users to update their email address, ensuring it remains unique.	M	Sprint 4
F1.3.2 Enables users to change their username, still ensuring uniqueness.	M	Sprint 4
F1.3.3 Allow users to update their password, requiring verification of their current password and ensuring the new password meets confirmation criteria.	M	Sprint 4
F1.3.4 Provide users with a secure logout option, immediately invalidating their session and redirecting them to the login page.	H	Sprint 4
F1.3.5 Allow users to permanently delete their account, deactivating it in the system and confirming with the user before final action.	M	Sprint 5
F1.3.6 Restrict login access to deactivated accounts.	M	Sprint 5
F1.3.7 Prevent registration with an email or username associated with a previously deleted account.	M	Sprint 6
F2 Matching Service		
F2.1 Question Difficulty		
F2.1.1 The system should allow users to select their preferred question difficulty from three options: easy, medium, and hard.	H	Sprint 4
F2.1.2 The matching service should pair users based on their selected question difficulty preference. If the service cannot find a match within 12 seconds, it will attempt to match users with the next closest difficulty level. Example: If user1 chooses easy and user2 chooses medium, they will be matched after 12 seconds.	H	Sprint 4
F2.2 Category Selection		
F2.2.1 The system should allow users to select their preferred categories, and not just limited to one category.	H	Sprint 4

F2.2.2 The matching service should match users based on their selected categories.	H	Sprint 4
F2.3 Matching Request Flow		
F2.3.1 Timeout for Matching If there is no matching user found within a specified time (35 seconds), the system should timeout and inform the user that there is no match and display 'retry' and 'back to home' option for user to choose	H	Sprint 4
F2.3.2 The system should allow users to cancel a match request during the waiting period and return to the dashboard.	M	Sprint 4
F2.3.3 After 5 seconds of the notification indicating a successful match, the user should be automatically directed to the shared collaboration page.	M	Sprint 5
F2.3.4 Once matched, the user should be notified of the matched partner's username, question topic, difficulty.	M	Sprint 5
F3 Question service		
F3.1 Maintain Question Repository		
F3.1.1 The system must keep a collection of questions organised by their difficulty level (such as easy, medium, or hard) and by the topic they belong to (such as algorithms).	H	Sprint 2
F3.1.2 The system should support questions with 3 levels of difficulty (easy, medium, hard)	H	Sprint 2
F3.1.3 The system should allow users to view and filter questions by difficulty level (either hard first or easy first).	H	Sprint 2
F3.1.4 CRUD Permissions The system should only allow admins to create, update, and delete questions to prevent unauthorised modifications and maintain data integrity. Non-admins should only be able to view the list of questions.	H	Sprint 3
F3.1.5 Error Handling for create data: Provide clear error message for various scenarios - Duplicate Question Title - Duplicate Question Description - Missing Fields	H	Sprint 2
F4: Collaboration service		
F4.1 Collaboration Space		

F4.1.1 The system should allow concurrent code editing during the session, real time updates where text changes made by both users are reflected in both users' text editor at the same time.	H	Sprint 5
F4.1.2 The system should automatically save code edits made by the paired users, ensuring that whenever the user reloads, rejoins, or changes language, the existing code is still there.	H	Sprint 5
F4.1.3 For each question, the system should provide default code in a runnable format that users can reference.	M	Sprint 5
F4.1.4 The system should allow users to select and switch between coding languages.	M	Sprint 5
F4.1.5 Provide a mechanism to rejoin their previous session from the dashboard in case of disconnect.	H	Sprint 5
F4.2 Collaboration Termination		
F4.2.1 The system should allow any of the paired users to leave an ongoing collaboration session at any time.	H	Sprint 5
F4.2.2 The system should notify the user when their partner leaves the session. The notification should be present even if the partner leaves while the user is disconnected.	M	Sprint 5
F4.2.3 The system should allow a user to continue editing or submitting code after their partner has left the session.	H	Sprint 5
F5 Chat Service		
F5.1 The service should allow two users in the room to send text messages to each other in real time.	H	Sprint 6
F5.2 The service should notify the other user when a user leaves the session.	M	Sprint 6
F5.3 The service should store messages within each chat session to maintain a chat history.	H	Sprint 6
F5.4 The service should retrieve all messages within a specified session, enabling users to view the chat history when leaving the page and rejoining the session.	H	Sprint 6
F5.5 The service should display their partner's username and include timestamps for each message to provide a clear timeline of the chat history.	M	Sprint 6
F6 History Service		

F6.1 The service should allow the user to see the history of completed questions sorted by most recent.	H	Sprint 6
F6.2 The service should allow the user to filter through their history using provided keywords.	M	Sprint 6
F6.3 The service should allow the user to view the following details of past sessions in the history: - Question details including title, difficulty and category. - Time and date of attempt and time taken. - Code submitted. - Username of partner.	M	Sprint 6
F7 Code Execution Service		
F7.1 The service allows users to execute their code in python, java and javascript within Docker containers.	H	Sprint 6
F7.2 If the code execution fails due to syntax errors, runtime errors, or other issues, the system should capture and return a clear error message.	H	Sprint 6
F7.3 The system should create a new Docker container for each code submission and ensure it is terminated after execution.	H	Sprint 6
F7.4 The system should enforce a timeout for each code execution to prevent indefinitely long or stuck processes.	M	Sprint 6
F7.5 The system should accept user-provided code, inputs, and expected outputs to validate the code execution.	M	Sprint 6
F8 AI Service		
F8.1 The service should generate grammatically correct, relevant, and meaningful responses to the user's query.	H	Sprint 6
F8.2 The service should not take longer than 20 seconds to respond to a query.	H	Sprint 6
F8.3 The service should receive the question context in advance to better assist the users.	M	Sprint 6

2.2 Non-Functional Requirements

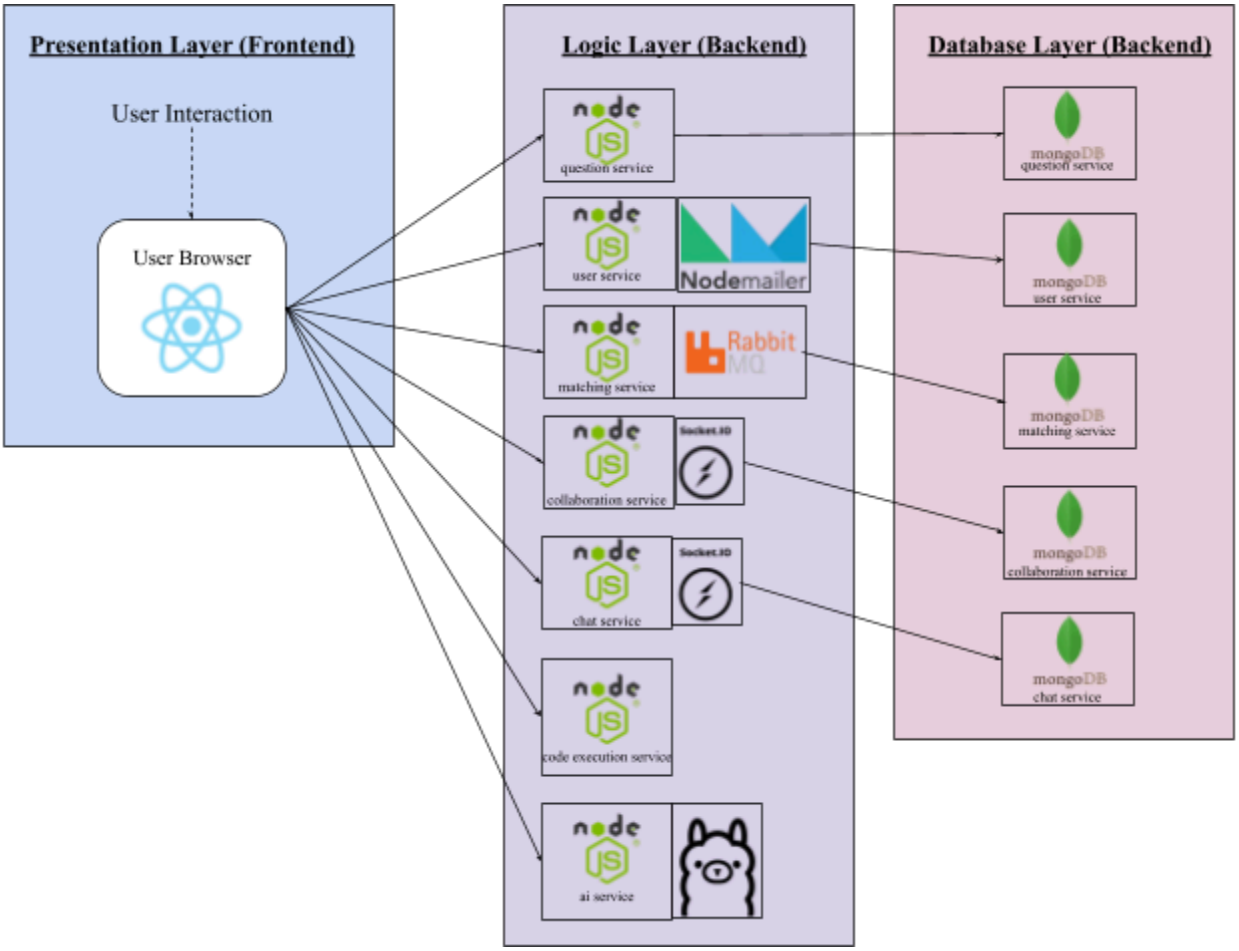
Non-Functional Requirements	Priority	Planned sprint
N1: Performance		
N1.1 Response Time		
N1.1.1 User authorization process should not take more than 3 seconds under normal internet conditions.	H	Sprint 6
N1.1.2 The question should load within 3 seconds after users are matched under normal internet conditions.	H	Sprint 6
N1.1.3 The matching process should be completed within 30 seconds if there is a matching pair.	M	Sprint 6
N1.1.4 Any actions taken by either party should be reflected to the other within 1 second under normal internet conditions.	H	Sprint 6
N1.1.5 The room should be created within 5 seconds after being matched under normal internet conditions.	H	Sprint 5
N1.1.6 The chat service should have an end-to-end delay of at most 1 second when under normal internet conditions.	M	Sprint 6
N1.1.7 The code execution service should ensure quick response times, with code execution and results returned within 5 seconds for standard test cases.	M	Sprint 6
N1.2 Scalability		
N1.2.1 The system should handle at least 100 concurrent users.	M	Sprint 6
N1.2.2 All requests to the application should be facilitated by an API Gateway.	M	Sprint 6
N1.2.3 The backend infrastructure should be decoupled to ensure scalability of individual components (e.g., database, API services, file storage).	M	Sprint 4
N2: Security		
N2.1 Data Protection		
N2.1.1 User data, including login credentials, must be encrypted.	H	Sprint 6
N2.1.2 User password should be at least 8 characters long.	M	Sprint 6

N2.1.3 User password should contain at least one upper/lower case letter and one special character.	M	Sprint 6
N2.1.4 The system should only allow users to access APIs that they are authorised to.	H	Sprint 6
N2.1.5 All code execution should be isolated within secure Docker containers to prevent unauthorised access, data leakage, or interference between user processes.	H	Sprint 6
N2.1.6 All user queries received by the AI service should be processed locally to ensure privacy and security.	H	Sprint 6
N2.1.7 The system must securely manage user authentication within the React application using an auth context to prevent unauthorised access.	H	Sprint 6
N3: Availability		
N3.1 System Uptime		
N3.1.1 The system should have an uptime of 99.9% during operational hours.	H	Sprint 6
N3.2 Accessibility		
N3.2.1 The application should support access from popular web browsers (Chrome, Safari, Firefox, Edge, Opera).	H	Sprint 6
N4: Reliability		
N4.1 Data Reliability		
N4.1.1 The database must not have any data loss in the event of a system failure or session termination.	H	Sprint 6
N4.1.2 Code must not be lost in the event of a system failure.	H	Sprint 6
N4.2 Service Reliability		
N4.2.1 The operational availability of the code execution service must be independent of external services, such as Judge0.	M	Sprint 6
N4.2.2 The operational availability of the AI service must be independent of external providers, such as OpenAI.	M	Sprint 6
N5: Usability		
N5.1 Interface Usability		

N5.1.1 The user interface should be intuitive, allowing users to submit code easily, receive clear feedback on errors, and view execution results in a readable format.	M	Sprint 6
N5.1.2 Clear, intuitive, and consistent navigation menus should be present across the website.	M	Sprint 4
N5.1.3 Visual elements such as fonts, buttons, icons, and colours should be consistent throughout the website.	M	Sprint 4

3 High-Level Architecture

3.1 Architecture Diagram



3.2 Architecture Decisions

3.2.1 Microservice Architecture

We decided that a Microservice architecture suited our needs more than a Monolith architecture. We wanted our system to be easily scalable and loosely coupled. A microservices approach allows us to scale individual services independently, enabling the system to handle varying loads and traffic efficiently.

Additionally, microservices promote loose coupling between components, making the platform easier to maintain and update. Each service is responsible for a specific function, such as user authentication, question management, or matching users, and can be developed, deployed, and scaled independently. This separation of concerns allows for more flexibility in the development process, where different members can work on different services without affecting the overall system. It also enhances fault tolerance, if one service fails, issues can be isolated and would not bring down the entire platform.

3.2.2 Event-Driven Architecture

We chose Event-Driven Architecture (EDA) for its scalability, flexibility, and responsiveness, particularly for our Matching Service. By leveraging RabbitMQ to handle events, we decouple services, allowing them to react asynchronously and independently. This approach enhances fault tolerance, as failures in one service do not impact others, and allows the system to adapt quickly to changing demands. The asynchronous nature of EDA, with RabbitMQ facilitating communication, enables real-time processing in the Matching Service without blocking other tasks, ensuring a more efficient and resilient system.

3.2.3 Client-Server Architecture

We chose Client-Server Architecture for its simplicity and efficiency in handling user requests and data processing. In this setup, the client is responsible for interacting with the user and sending requests, while the server handles business logic, data management, and responses. This separation allows the server to scale independently and ensures a centralised point for resource management. The architecture also improves security, as the server can manage authentication and data storage, while the client remains focused on user interaction. This clear division of responsibilities makes the system easier to maintain and update, promoting efficient handling of multiple client requests.

3.3 Technology Stack

The technology stack for PeerPrep includes React and Vite for the frontend, Node.js with Express for the backend server. For data storage, we use MongoDB, a NoSQL database, which allows for flexible data modelling and quick retrieval of large volumes of data without the constraints of a relational schema. To support real-time communication between users, we use Socket.IO. Additionally, we use RabbitMQ for message queuing and Docker containers for containerization.

4 Design Patterns

4.1 Overview

Design Pattern	Description
Database-per-Service Pattern	Each microservice has a dedicated database, allowing independent data management and fault isolation.
Observer Pattern	React components act as observers, automatically updating the UI in response to data or state changes.
Pub-sub Pattern	Real-time updates using a publisher-subscriber model, enabling synchronisation across clients.
Model-View-Controller Pattern	Separates data, logic, and UI into distinct layers, streamlining data flow and improving maintainability.

4.2 Database-per-Service Pattern

In this project, the Database-per-Service pattern is implemented across multiple microservices that requires persistent storage, each with its own dedicated database and schema. This setup allows each microservice to manage its own data without relying on a shared database. The architecture supports independent scalability, data privacy, and flexible data modelling, enabling each service to evolve without impacting others. Additionally, having dedicated schemas ensures that each service's data structure is optimised for its specific use case. The design also enables fault isolation to prevent issues in one service's database from affecting others. Meanwhile, the frontend operates independently, interfacing with these backend services through API calls. This pattern is applied across various backend services as follows:

Question Service: Performs CRUD operations on question data, storing details such as title, description, category, and complexity.

User Service: Manages user data such as profile information, password, privileges and attempt history. It includes functions for updating profiles, verifying passwords, managing histories, and deleting users.

Matching Service: Operates with a Match schema to store information such as user IDs, categories, complexity, and session details. It includes functions to create, retrieve and delete match records.

Collaboration Service: Handles real-time collaboration sessions via the Session schema, including details about active sessions, code windows, past attempts, and whiteboard data. This service handles session-related data, enabling collaborative coding environments.

Chat Service: Manages chat messages between users during collaborative sessions. The Chat schema includes session-based chat storage, maintaining a history of messages within each session.

4.3 Observer Pattern

The observer pattern is used through React's state management. In this pattern, components act as observers, reacting automatically to changes in the state or data. This keeps the UI in sync with the underlying data and provides a smooth, responsive experience for users.

Props and State: In React, components receive data through props, which are passed down from parent components. Additionally, components manage their own internal state. When props or states change, components re-render, observing and responding to keep the UI up to date. For instance, when the user presses "Run Code" on the coding panel, the state variables such as the output, caseResults and hasError are updated. These changes then propagate down to TestResultTab through props. Upon detecting these updated results, TestResultTab dynamically updates its content based on the output and error states as needed.

Hooks: The useState hook in React manages state in functional components, allowing components to update any state changes. The useEffect hook executes side effects in response to state or prop changes, acting as an observer on specified data changes. For instance, in the waiting page, useEffect is used to detect changes in the matchFound state. When matchFound is set to true, the useEffect hook observes this change and triggers a countdown for the session to begin.

4.4 Pub-sub Pattern

In this project, the pub-sub pattern is implemented using Socket.IO for real-time communication between clients, specifically in the collaboration page. It is applied across these functionalities:

Session Join and Code Changes: When a user joins the collaboration, the client subscribes to the session's unique room by emitting a join event with the session ID. This allows them to receive updates specific to that session. Any code changes are then published by emitting a code change event, which triggers the code update event to broadcast the updated code to all session participants.

Whiteboard Drawing: Whiteboard actions are synchronised by publishing drawing events such as starting to draw a stroke, continuing it, and completing it. Each event updates the whiteboard in real time, maintaining a shared view of the drawing canvas.

Chat Messages: Chat synchronisation occurs as users subscribe to the receiveMessage event, which displays new messages sent by others in the session. When a message is published via the sendMessage event, users receive it instantly and it is displayed in the chat box.

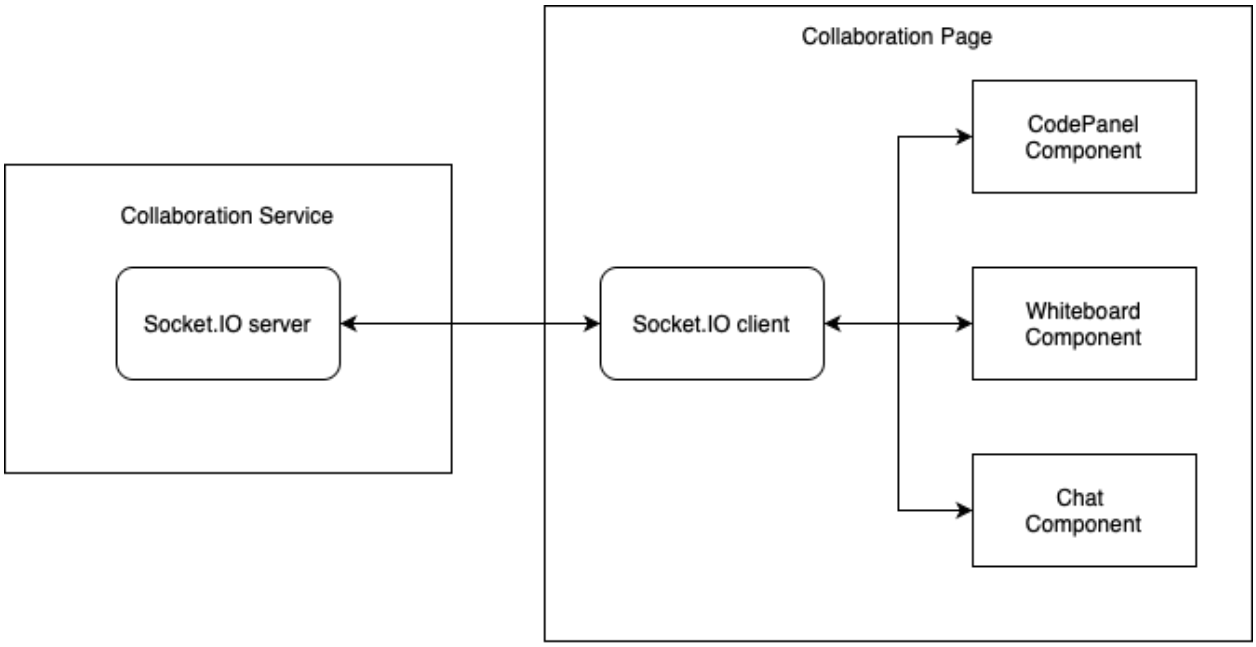


Diagram 4.4.1 Implementation of Socket.IO for real-time communication

4.5 Model-View-Controller Pattern

Our project backend and frontend adheres to a Model-View-Controller (MVC) pattern, with the backend serving as a structured API, and the frontend consuming and presenting the data. This setup helps to streamline data handling, logic processing, and user interface rendering, ensuring a clear separation of concerns and enhancing maintainability.

Model: The backend uses MongoDB schemas for entries such as Chat, User, Question and Match. Each schema defines data structures and interactions with the database.

Controller: Processes incoming requests, interacting with the models to retrieve or update data as necessary. For example, the question controller manages question related CRUD operations.

View: The frontend consumes backend responses, such as question data and match details, and displays them in a user-friendly format. Frontend components will render information based on API responses, updating UI elements such as the question table.

Example of MVC Flow:

1. **Frontend (View):** The user initiates a request to add a new question. This request, containing details like the questions's title, description, category and complexity, is sent to the backend API.
2. **Backend (Controller):** The controller function createQuestion in the backend receives the request. It validates the input fields and interacts with the Model.
3. **Model:** The question schema model is used to create and store a new question entry in the database.
4. **Backend (Controller)**: Once the question is added, the controller packages this data into a JSON response and sends it back to the frontend.
5. **Frontend (View):** The frontend receives the response and updates the QuestionTable component, completing the MVC cycle.

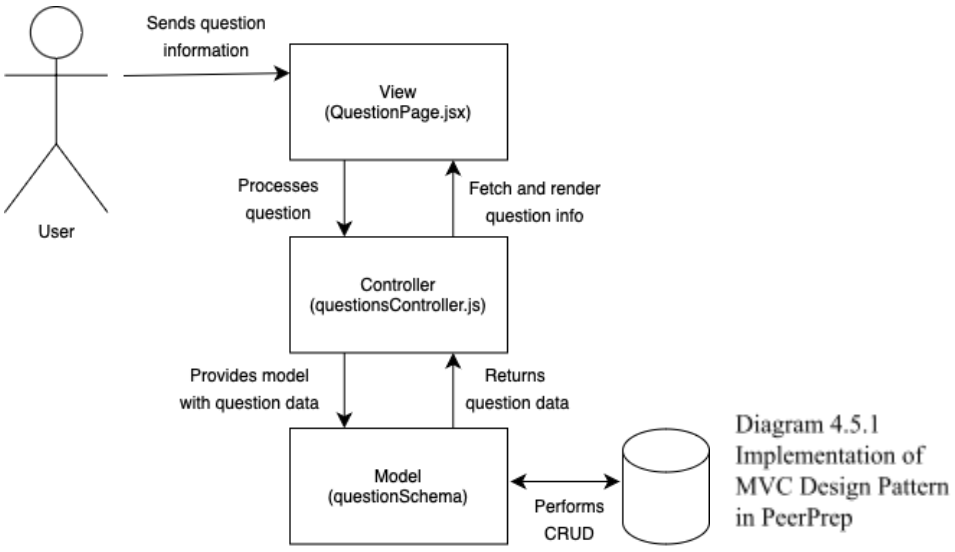


Diagram 4.5.1 Implementation of MVC Design Pattern in PeerPrep

5 Frontend

5.1 React + Vite Framework Choice

React is a popular JavaScript library developed by Meta for building user interfaces, particularly single-page applications (SPAs). It allows developers to create reusable UI components, manage the state of applications efficiently, and implement a virtual DOM that optimises rendering performance. React's component-based architecture makes it easy to maintain and scale applications, promoting a modular approach to development.

Vite is a modern build tool that significantly enhances the frontend development experience. It leverages native ES modules in the browser and provides a fast and efficient development environment with features like hot module replacement (HMR) and instant server start. Vite focuses on speed and performance, ensuring that developers can iterate quickly during the development process.

In this project, we leverage **React's component-based structure alongside Vite's rapid build** and serve capabilities to create a performant, maintainable, and user-friendly application.

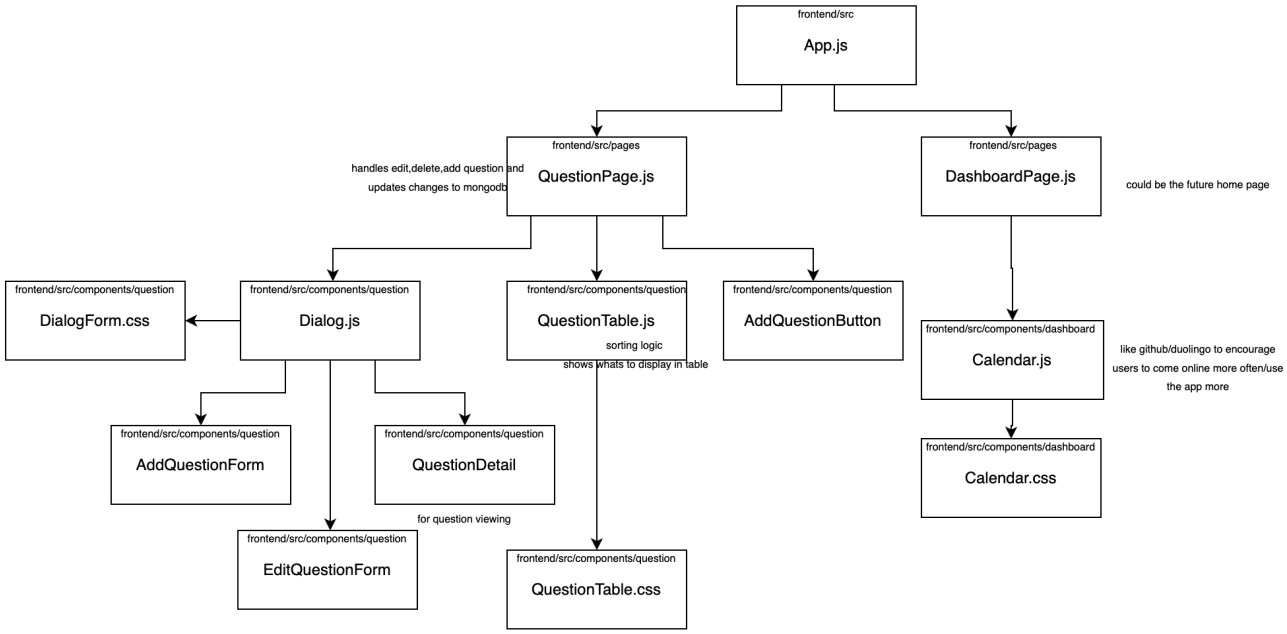
Alternatives	**Reasons for Exclusion**
TypeScript	Although TypeScript offers enhanced type safety and could improve code reliability, we opted to use JavaScript in this project to streamline the learning curve for team members who are less familiar with TypeScript. Given the project timeline, prioritising rapid development in JavaScript allowed us to focus on delivering features efficiently without the added complexity of TypeScript integration and configuration.
Create React App	While Create React App (CRA) is a common starting point for React applications, it lacks the performance optimizations and fast development features of Vite. CRA's build times can become slower as the application grows, which was a key consideration for our project.
Vue.js	While Vue.js is a strong contender with its simplicity and flexibility, it lacks the extensive ecosystem and enterprise support that React offers, making it less suitable for large-scale applications that may require a more robust solution.
Angular	Angular has a steeper learning curve compared to React, which can slow down development for new team members. Additionally, its size and complexity may lead to longer load times, which is a critical factor for performance-focused applications.

5.2 Frontend Architecture Diagram

When designing the frontend, we naturally create multiple classes for each page and its components. To keep the codebase understandable for everyone, architecture diagrams are generated at every second milestone, providing a clear overview of where to find specific code elements. Additionally, we focused on keeping class sizes manageable by adhering to the Single Responsibility Principle (SRP). This led us to create separate classes for individual components within each page, ensuring that each has a distinct role.

For the next three architecture diagrams, please note that the arrows indicate either a dependency on another class or a navigational link, rather than a strict class diagram structure.

5.2.1 Milestone 2 Architecture Diagram (link here)



In Milestone 2, our frontend user flow was structured with App.js serving as the main homepage, offering two navigation buttons to access the QuestionPage and DashboardPage.

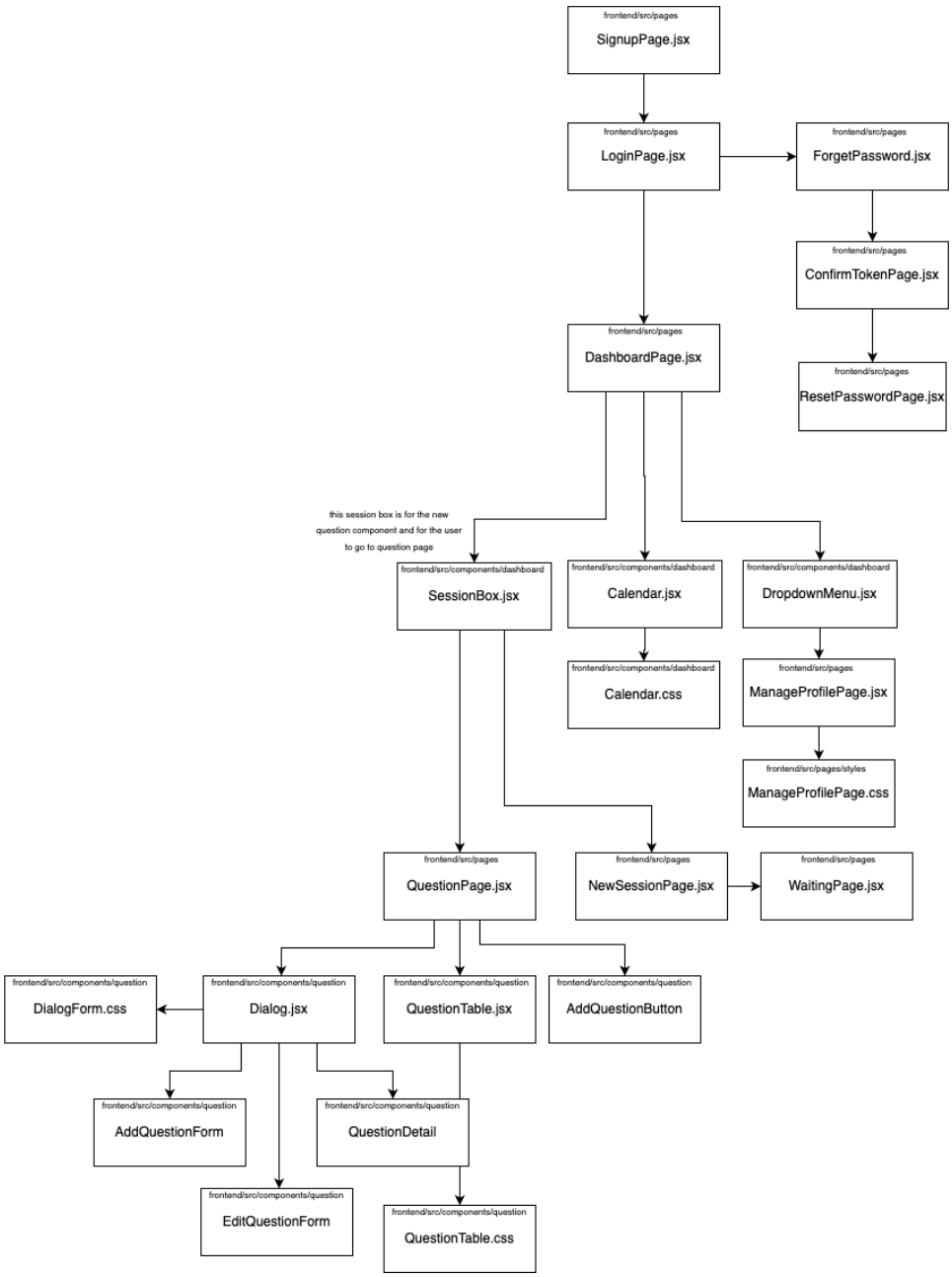
On the QuestionPage, we implemented three main components:

- Dialog.js for managing pop-up dialogs,
- QuestionTable.js for displaying the question table, and
- AddQuestionButton for adding new questions.

Dialog.js handles pop-up forms for adding, editing, and viewing questions, with styling provided by DialogForm.css.

The DashboardPage includes a single component: the Calendar, which tracks user activity by marking each day the user logs in. If the user was online, that day is shaded; if not, it remains blank. Online activity dates are stored in MongoDB and updated every time the user logs in or visits the dashboard page.

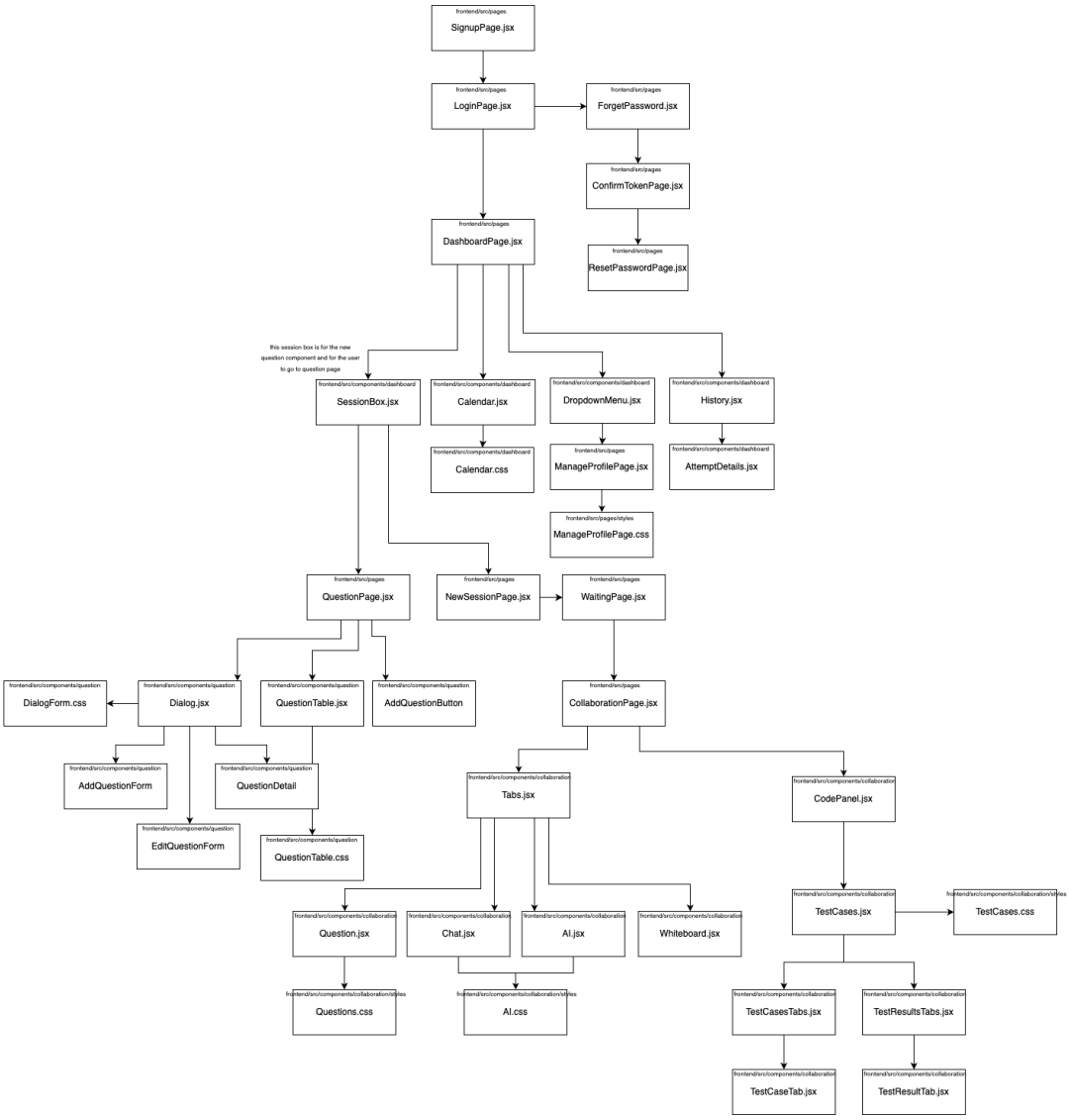
5.2.2 Milestone 4 Architecture Diagram (link here)



In Milestone 4, we created a way for users to sign up, login and start a session. Our frontend user flow was structured with DashboardPage.js serving as the main homepage, offering two navigation buttons to access the QuestionPage and NewSessionPage. A dropdown menu containing a Manage Profile button was provided for users to edit their profile. Users can also reset passwords at the forgot password page.

QuestionPage and NewSessionPage uses SessionBox to ensure that code is not repeated (Don't Repeat Yourself) and to ensure a uniform look, NewSessionPage directs to WaitingPage, which will later inform users whether the match succeeded or failed.

5.2.3 Final Architecture Diagram (link here)



By Milestone 6, our class diagram had expanded to include new features on the dashboard and a dedicated collaboration page.

On the dashboard, we added a History component, allowing users to view completed questions along with an "Attempt Details" pop-up for more in-depth information on each attempt.

The collaboration page is now divided into two main sections:

- On the left, Tabs.jsx provides a space-saving interface that lets users switch between Question, Chat, AI, and Whiteboard tabs. This font layout keeps the design clean and minimalistic.
- On the right, the CodePanel component includes a Test Case section that displays the results of question test cases (pass/fail status) and allows users to switch coding languages, run code, and interact with a code editor powered by CodeMirror.

5.3 Design and Implementation Decisions

Through extensive user feedback and testing, the interface went through several iterations and improvements. One key objective for us was to create a platform that felt like a true coding environment, which is why we chose the FigTree font for the entire application. We also wanted the platform to be visually engaging, as users often spend long hours coding, so we aimed for a dynamic and stimulating user experience.

We made sure the dashboard was informative and motivational, with features designed to keep users engaged. This led to the introduction of the Calendar and a streak system, where the colour changes based on daily usage, encouraging users to return regularly.

Simplicity and clarity in code were also crucial. Each component, like Chat.jsx, is focused solely on its specific function (Single Responsibility Principle) — in this case, the chat logic — ensuring the code remains concise and easy to understand.

We also wanted the interface to feel personal and welcoming. By greeting users with their username and playful phrases like "We hope you are having a great day," we aimed to establish a connection with our users. Additionally, phrases like "You have no active sessions" were designed to create a sense of direct engagement with the user.

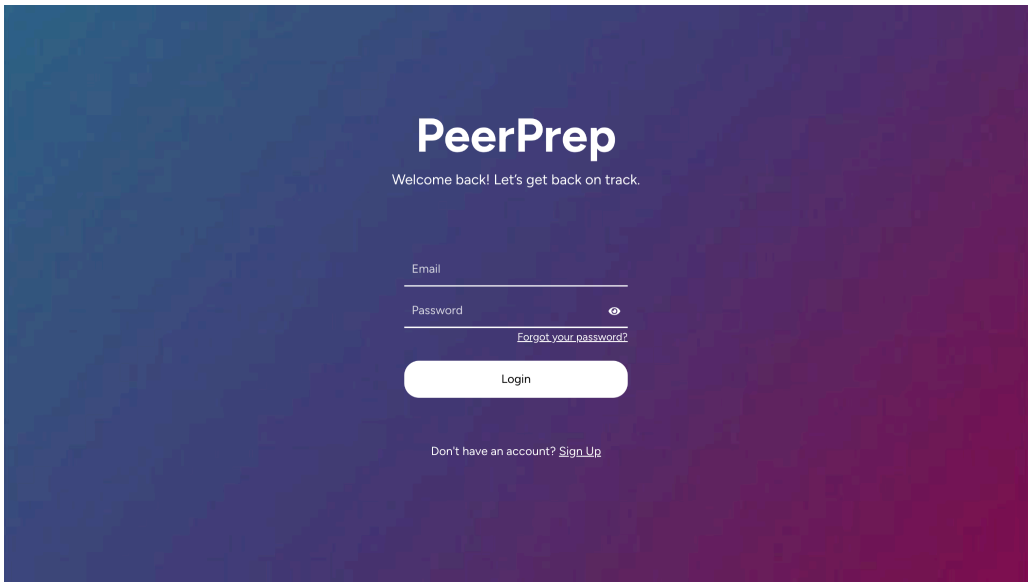
To prevent abrupt session endings, we added a summary page to give users a moment to reflect on their experience and view key statistics before returning to the dashboard.

Lastly, when selecting the colour scheme, we kept our team's favourite colours in mind, ensuring a visually appealing design that everyone could enjoy.

5.4 Pages and Screenshots

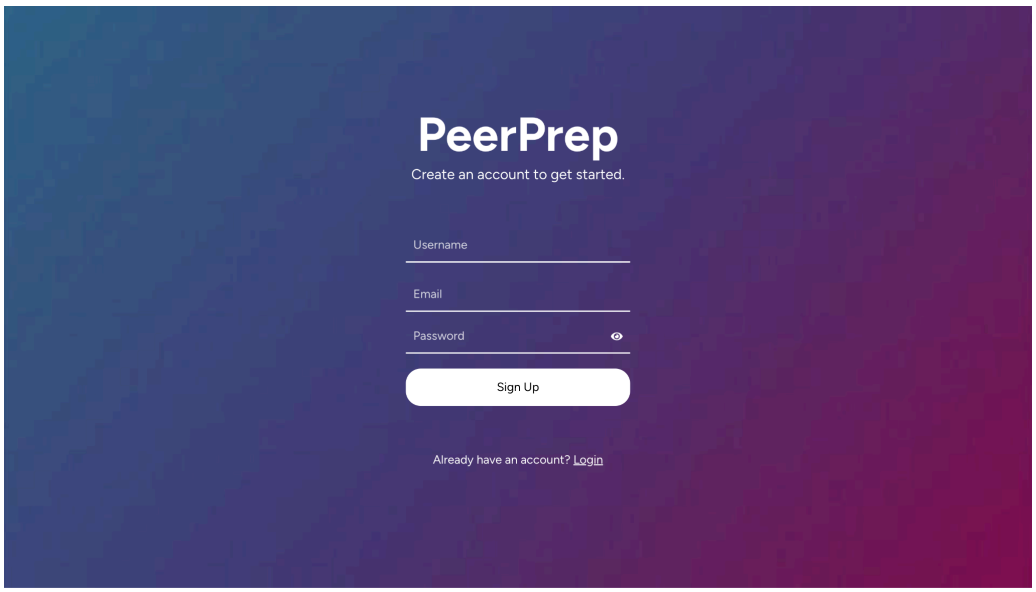
This was our initial User Interface Design: CS3219 User Interface

5.4.1 LoginPage



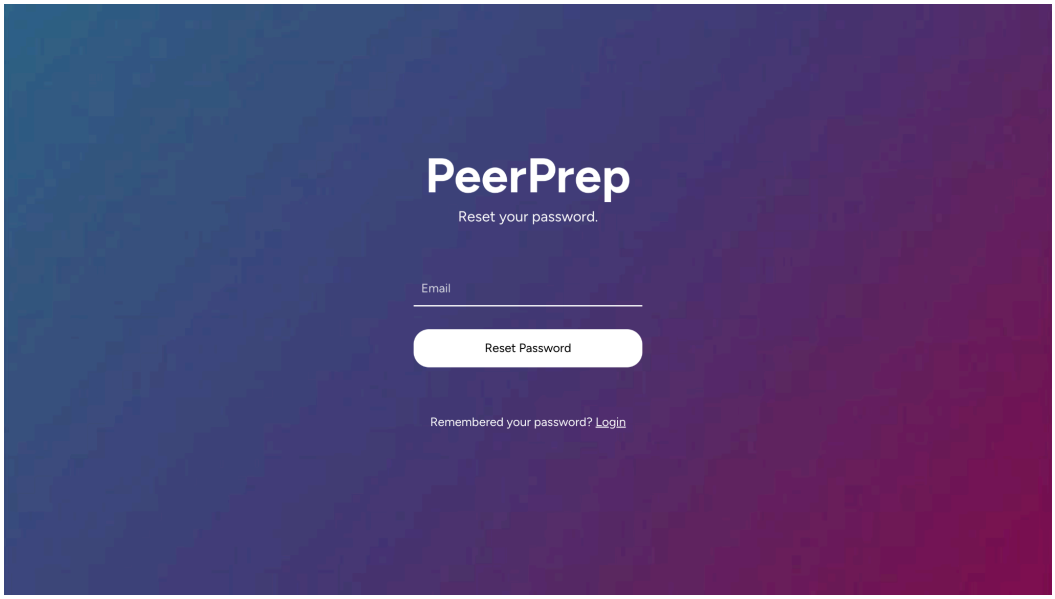
Where users login to our application.

5.4.2 SignUpPage



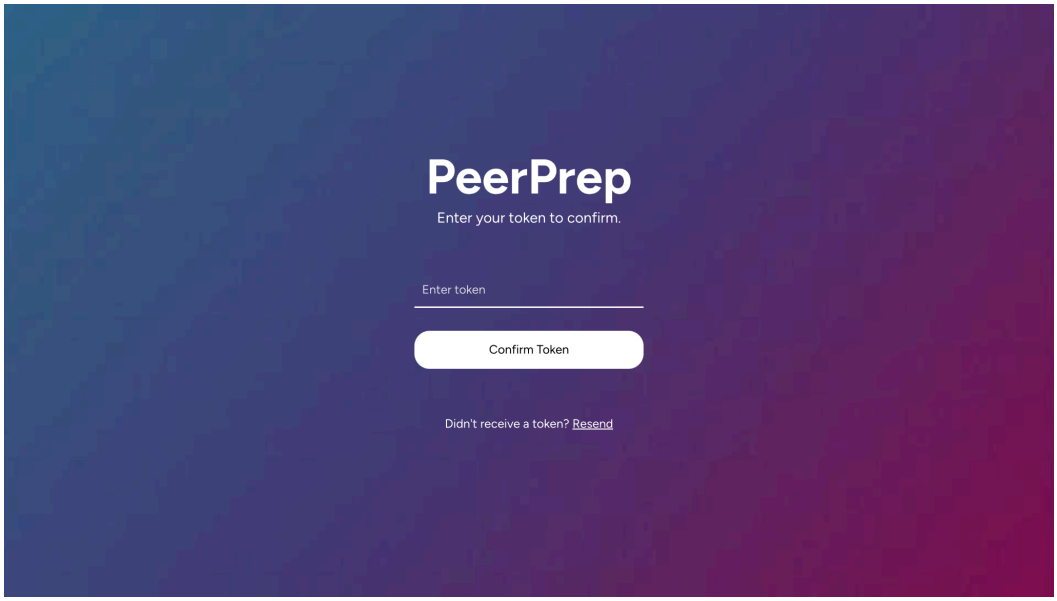
Where users sign up for our application.

5.4.3 ForgetPasswordPage



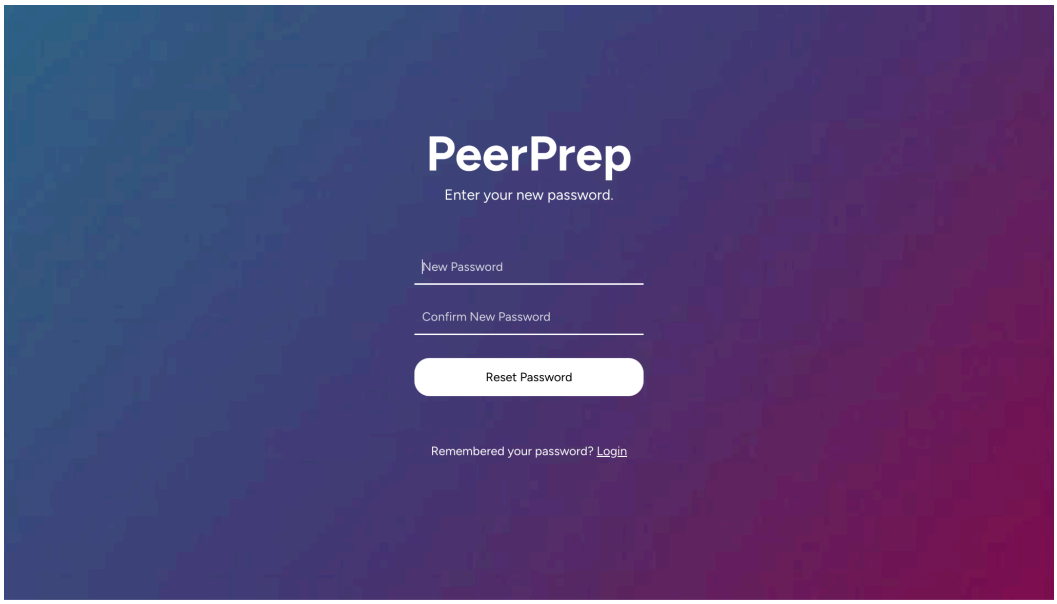
When users forget their password they can reset it by entering their email.

5.4.4 ConfirmTokenPage



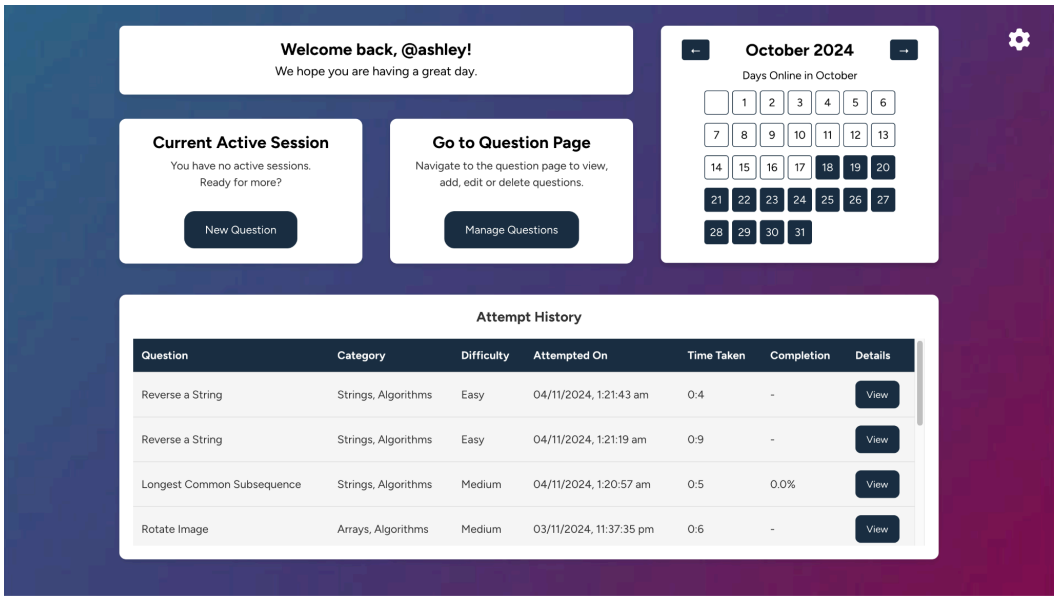
A token will be sent to their email, and this page will take in the token.
If users did not receive their token, they can request for a new one.

5.4.5 ResetPasswordPage



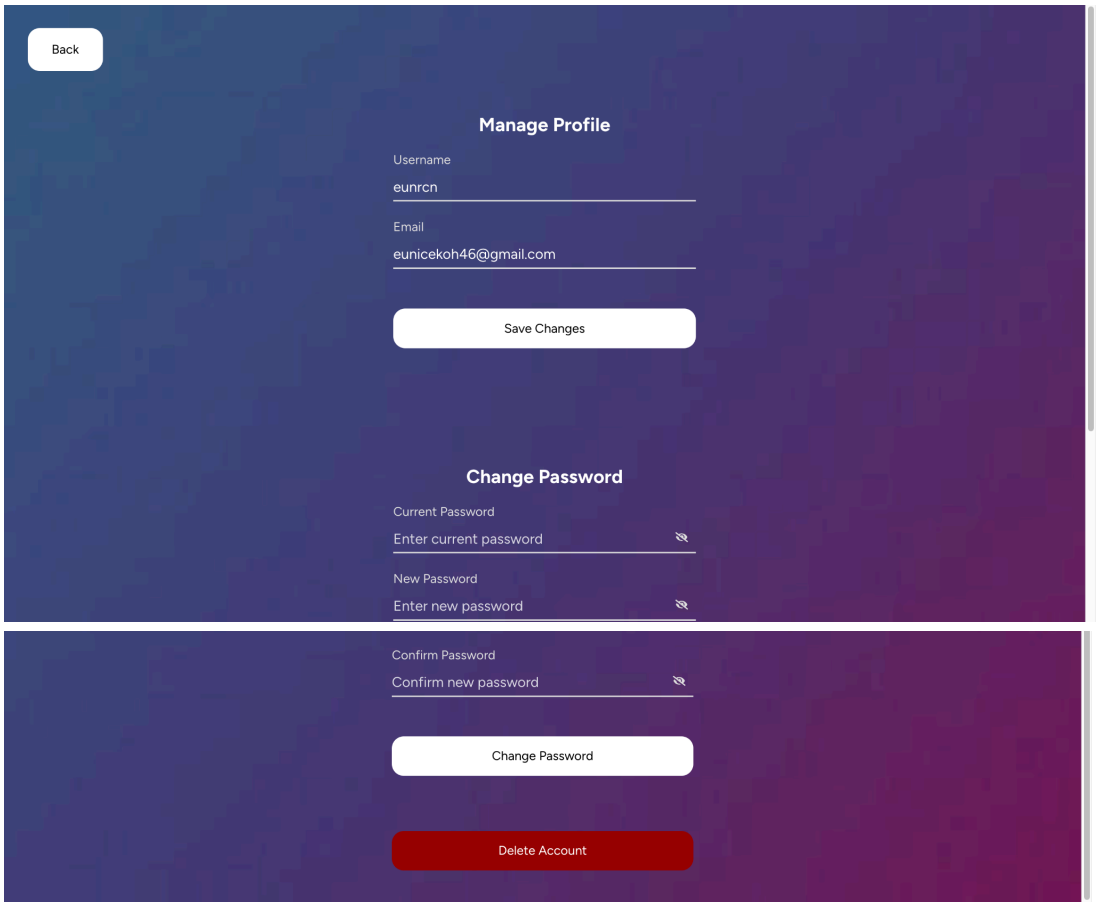
After users get their reset password token from their email, if they input it to our user interface correctly, they will be directed to this page to set their new password.

5.4.6 DashboardPage



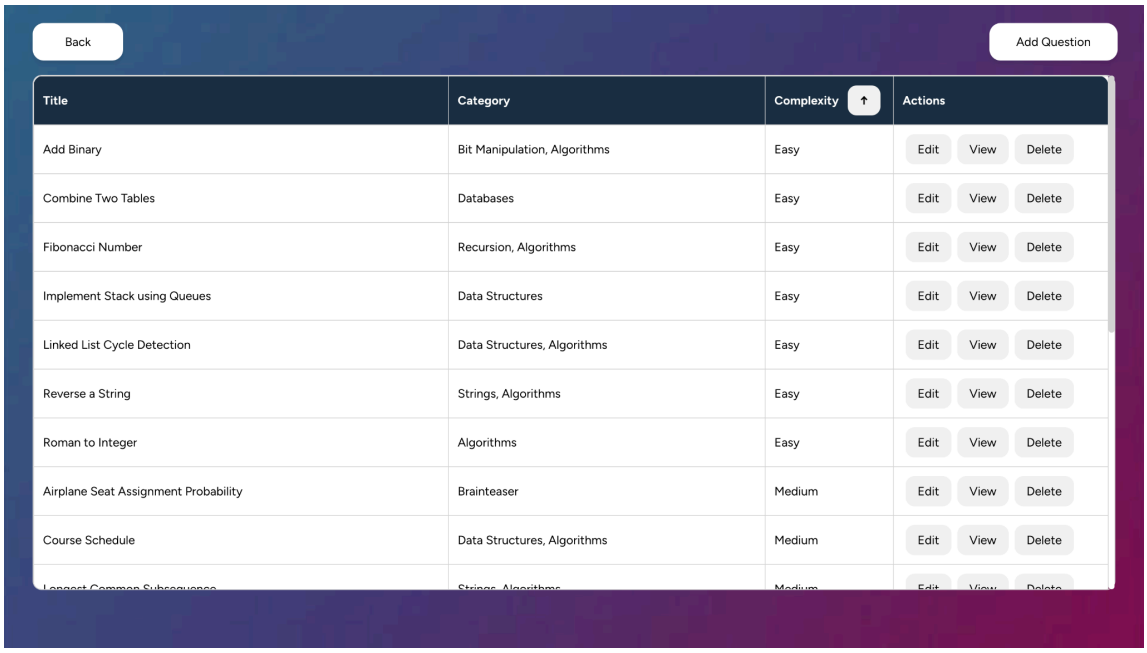
Upon successful login, users will be directed to the dashboard page.

5.4.7 ManageProfilePage



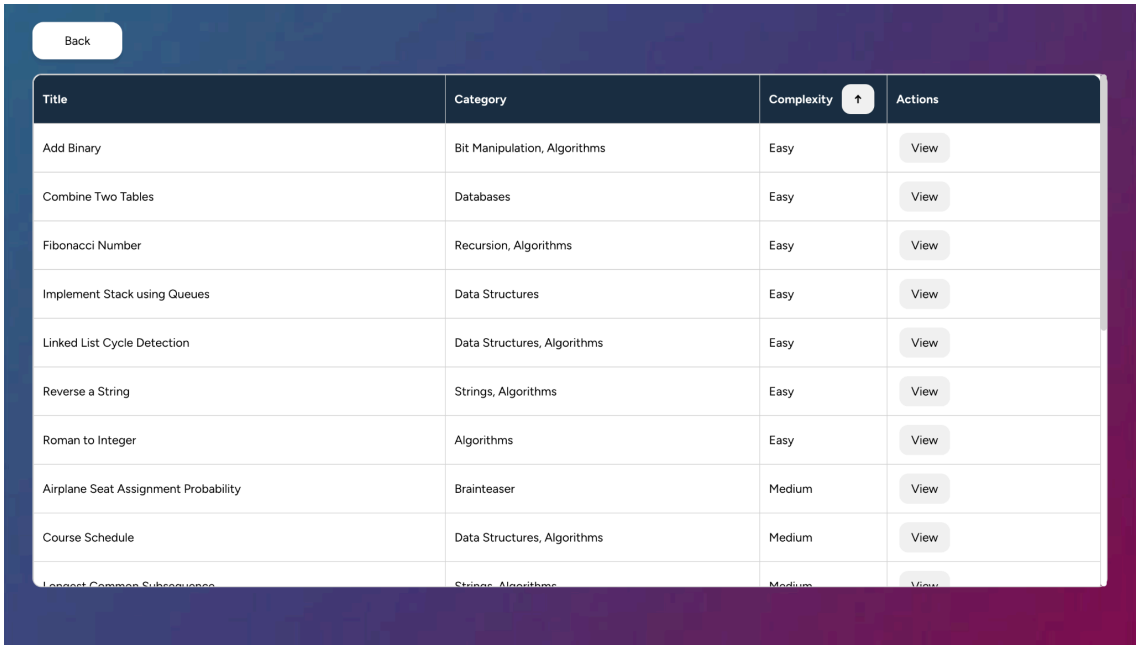
Where users get to edit their username, email and change their password, this page can be accessed by clicking the settings icon on the dashboard page.

5.4.8.1 QuestionPage (for Admins)



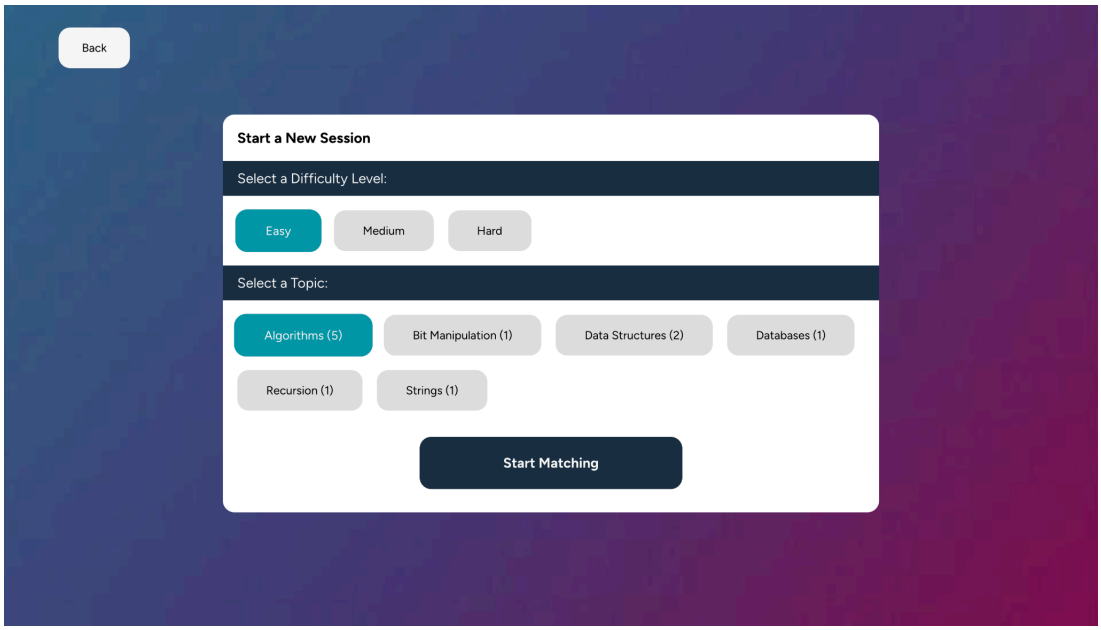
This page shows the available questions from the database, this page can be accessed by clicking the “manage questions” button on the dashboard page. This is the admin view.

5.4.8.2 QuestionPage (for Non-Admins)



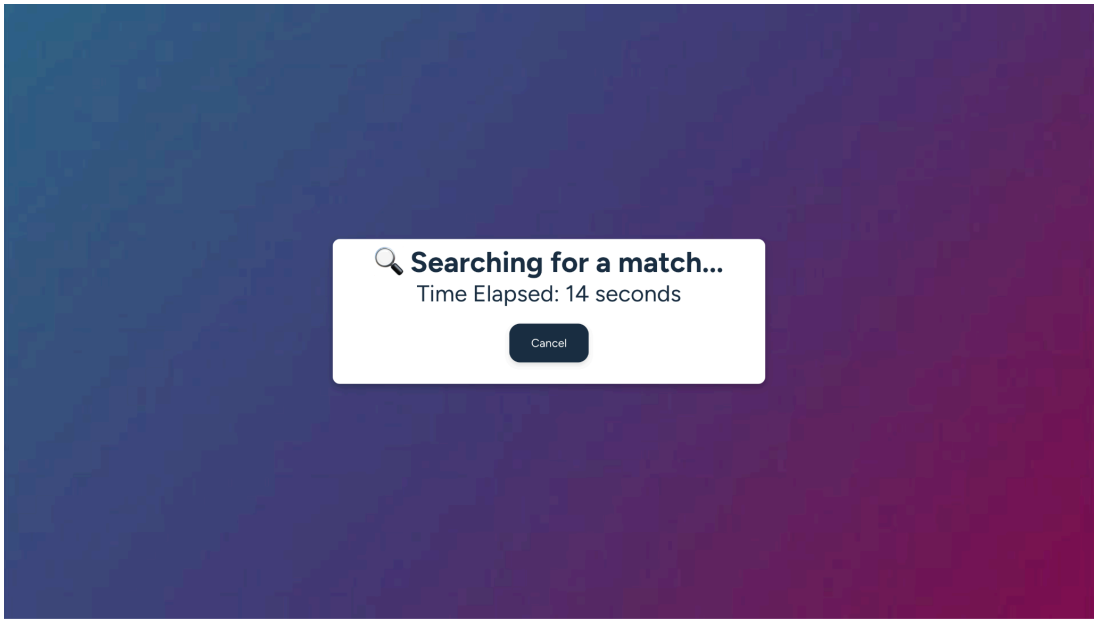
This page shows the available questions from the database, this page can be accessed by clicking the “manage questions” button on the dashboard page. This is the non-admin view.

5.4.9 NewSessionPage



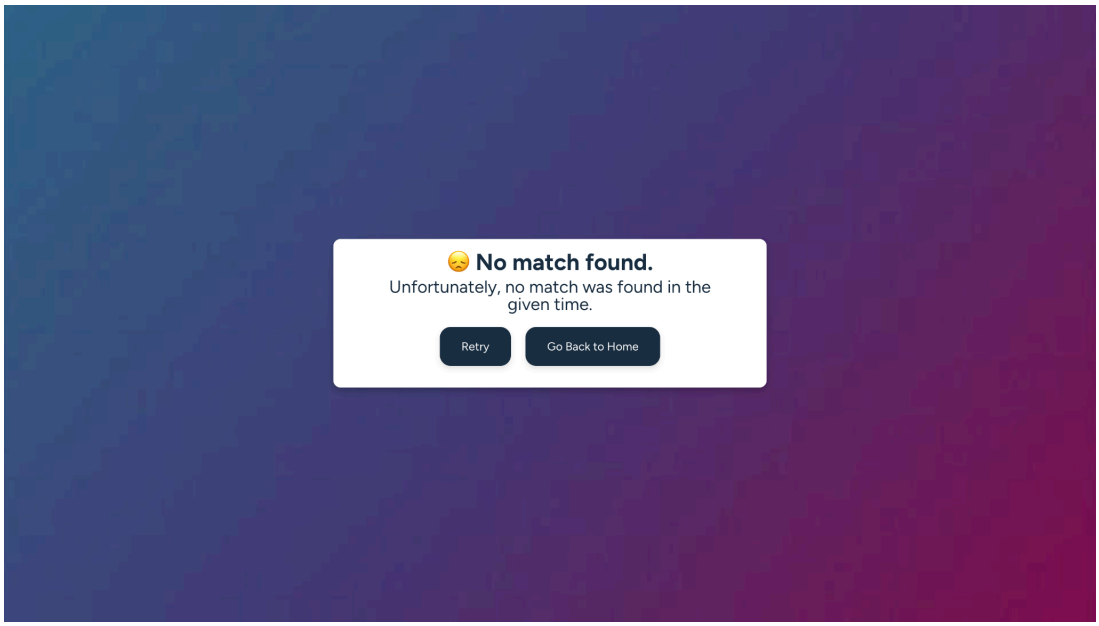
After users click on a new session on the dashboard, they are navigated to this page to select their preferred difficulty and topic.

5.4.10.1 WaitingPage (Timer)



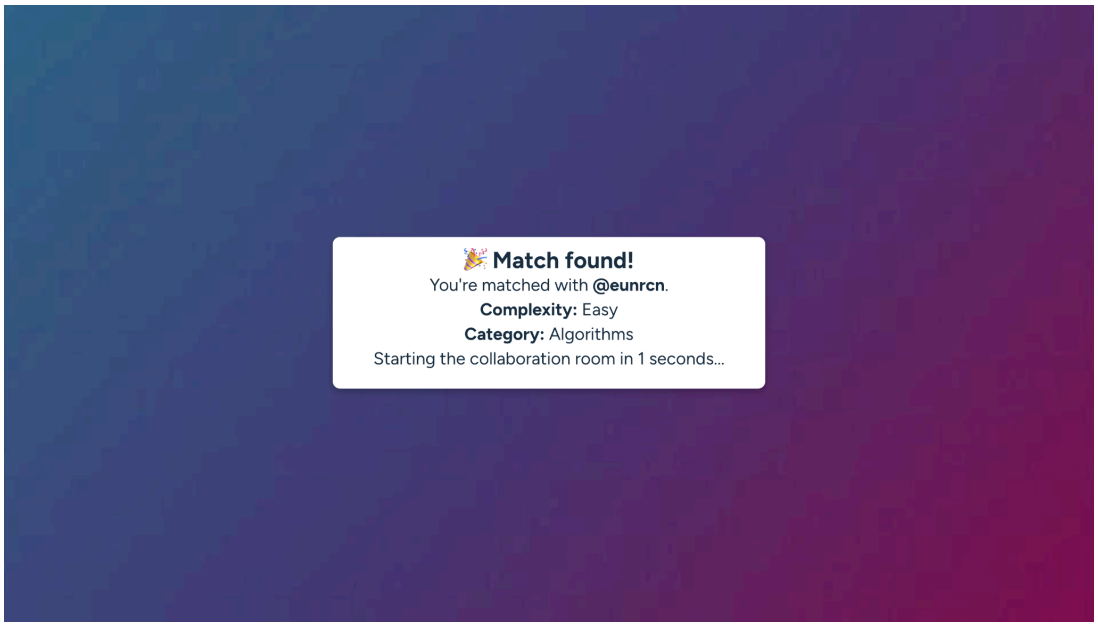
While users wait for a match, there will be a timer showing how much time has passed.

5.4.10.2 WaitingPage (No Match Found)



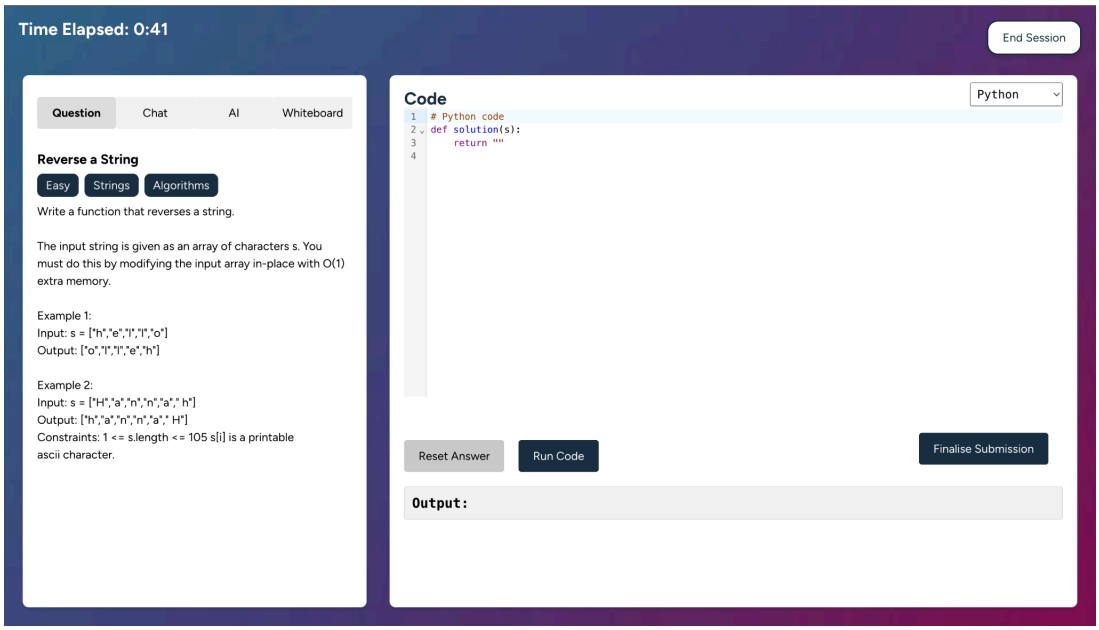
After 35 seconds, if no match was found, this is the interface that the user will see.

5.4.10.3 WaitingPage (Match Found)



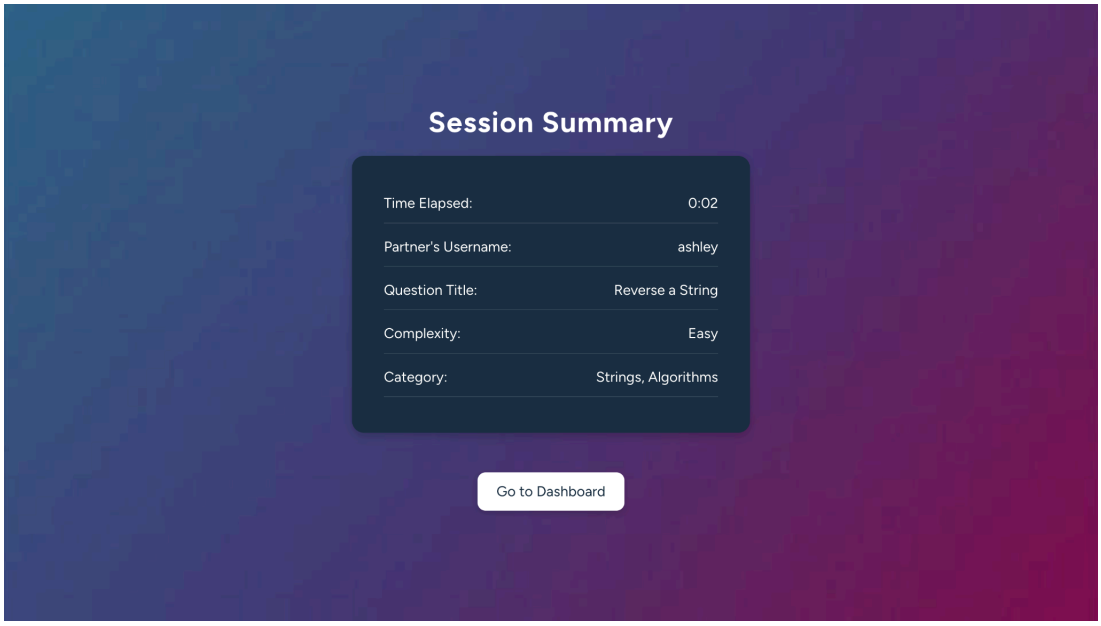
If a match was found, this is the interface that the user will see for 5 seconds before being directed to the collaboration page.

5.4.11 CollaborationPage



Collaboration page, where both users can code together.

5.4.12 SummaryPage



After users end the session, they will see this session summary page.

6 Core Backend (Microservices Architecture)

6.1 Overview

Service	Description	Port Number	Key Technologies
User Service	Manages user data, authentication, and profiles.	8081	Node.js, Express, Docker, MongoDB
Question Service	Handles question-related operations and CRUD.	8080	Node.js, Express, Docker, MongoDB
Matching Service	Matches users based on topic and difficulty.	8082	Node.js, Express, Docker, RabbitMQ
Collaboration Service	Manages collaboration features, code sharing	8084	Node.js, Socket.IO, Docker, MongoDB
RabbitMQ	Message broker for asynchronous communication between services.	5672, 15672	RabbitMQ

6.2 User Service

6.2.1 Overview

In building our user-service, we utilised MongoDB, Express, and Docker.

- MongoDB serves as our NoSQL database, providing flexible document-based storage that supports rapid data retrieval and scalability.
- Express, a flexible Node.js framework, streamlines API handling, making it easier to manage routes and middleware.
- Docker containers ensure portability and consistency across development and production environments.

6.2.1.1 AuthContext

AuthContext acts as a centralised store for managing authentication state within a React application. It leverages the Context API to provide authentication-related data, such as accessToken and userId, to any component that needs it.

The AuthProvider component encapsulates the authentication logic and offers methods for logging in and out. These methods update both the context state and persistent storage. The useAuth hook enables easy access to this context, allowing components to interact with the authentication state seamlessly.

By combining AuthContext with the withAuth higher-order component (HOC), the application creates a robust authentication flow that enhances security, user management, and the overall user experience. This design pattern facilitates scalable and maintainable code by decoupling authentication logic from UI components, promoting better organisation and readability.

6.2.1.2 withAuth

The withAuth HOC serves as a crucial mechanism for enforcing user authentication across various components within a React application. By wrapping components that require authentication, it checks for the presence of an accessToken stored in the application's context. If the token is absent or invalid, the component triggers a redirect to the login page, ensuring that unauthorised users cannot access protected routes.

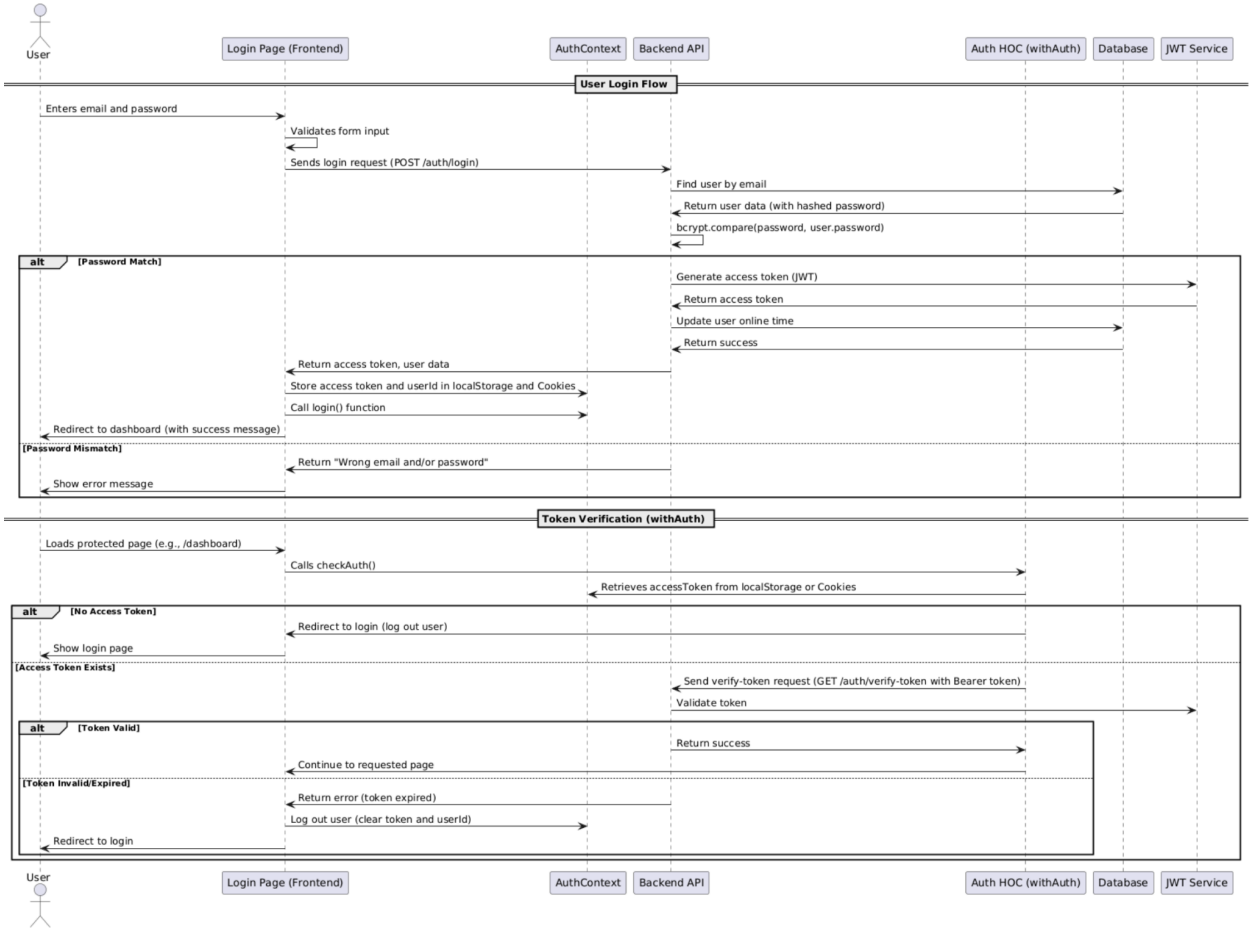
This approach enhances both security and user experience by preventing invalid logins that might occur due to changes in the website domain or session expiration. By centralising authentication logic, withAuth simplifies the implementation process, allowing developers to easily protect any component that requires authentication simply by wrapping it with this HOC.

Example of how to wrap a page/component:

```
const WrappedDashboardPage = withAuth(DashboardPage);
export default WrappedDashboardPage;
```

6.2.1.3 Sequence Diagram to show how **AuthContext** and **withAuth** are used

The sequence diagram illustrates the user login process, including authentication, token generation, session management via **AuthContext**, and token verification for accessing protected pages using **withAuth**.

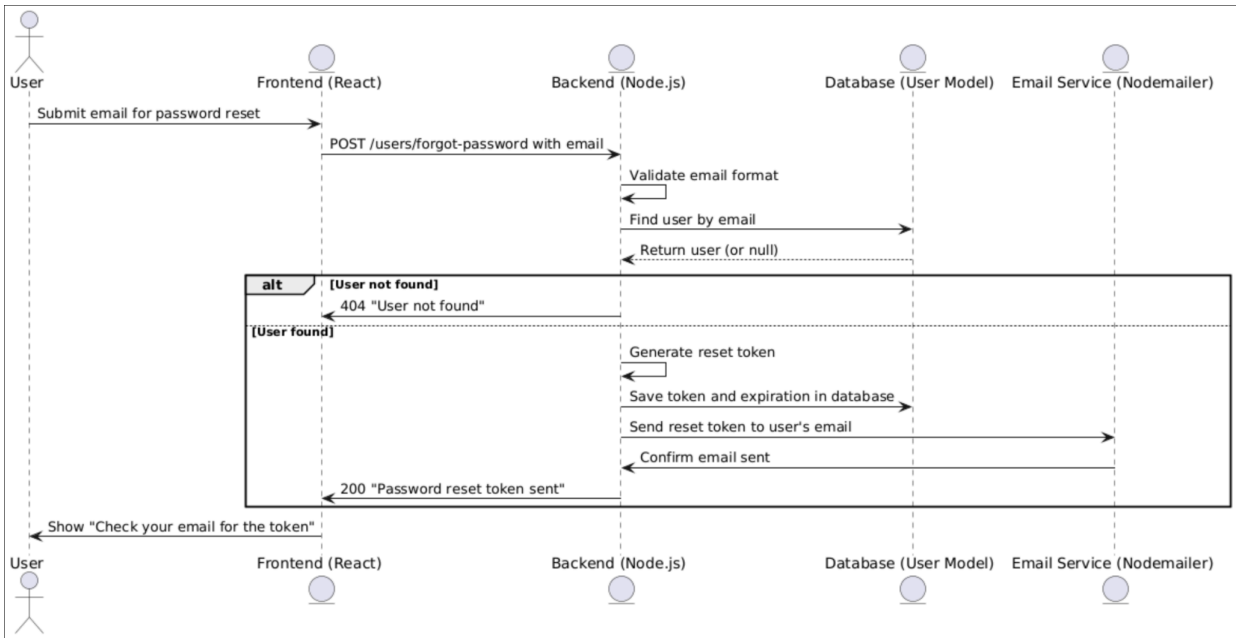


6.2.1.4 Forget Password

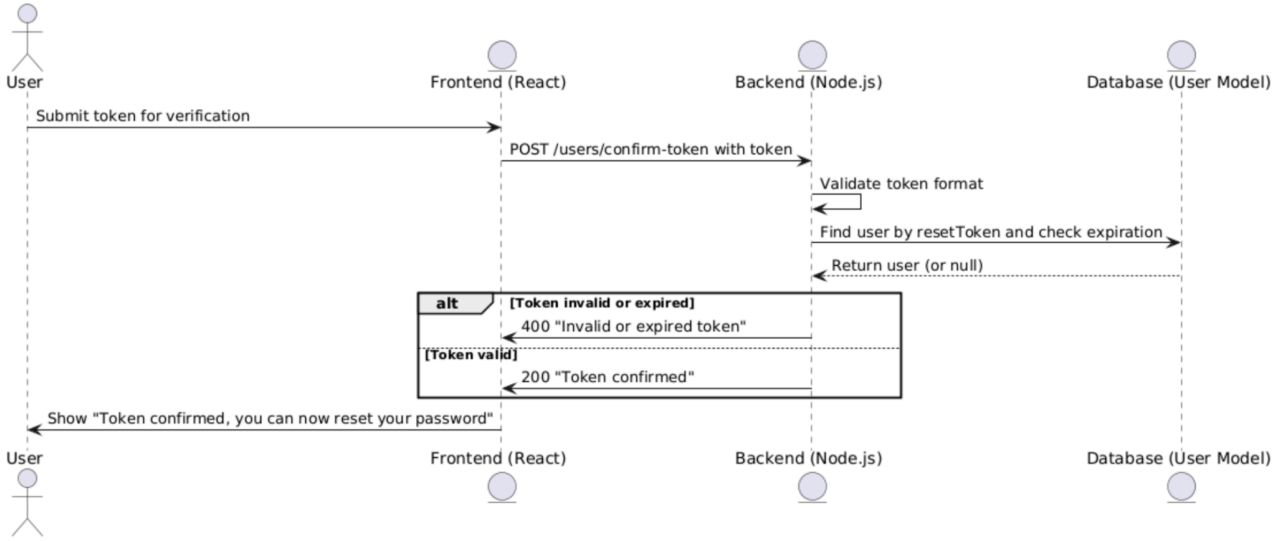
While the existing user service provides comprehensive features for user management—such as login, signup, and account deletion—it lacks a vital functionality: the ability for users to reset their passwords. To fill this gap, we implemented a password reset feature that significantly improves user experience and enhances security.

The process begins by requesting the user's email address. We then utilise Nodemailer to send a password reset token via a dedicated Gmail account (raesa2648@gmail.com). Once the user receives the email, they are prompted to enter the token. If the provided token is valid, the user is allowed to set a new password. However, if the token is incorrect or has expired, an error message indicating "invalid token" is displayed. Additionally, we offer users the option to resend the token if needed. Below, we have 3 sequence diagrams to illustrate this process.

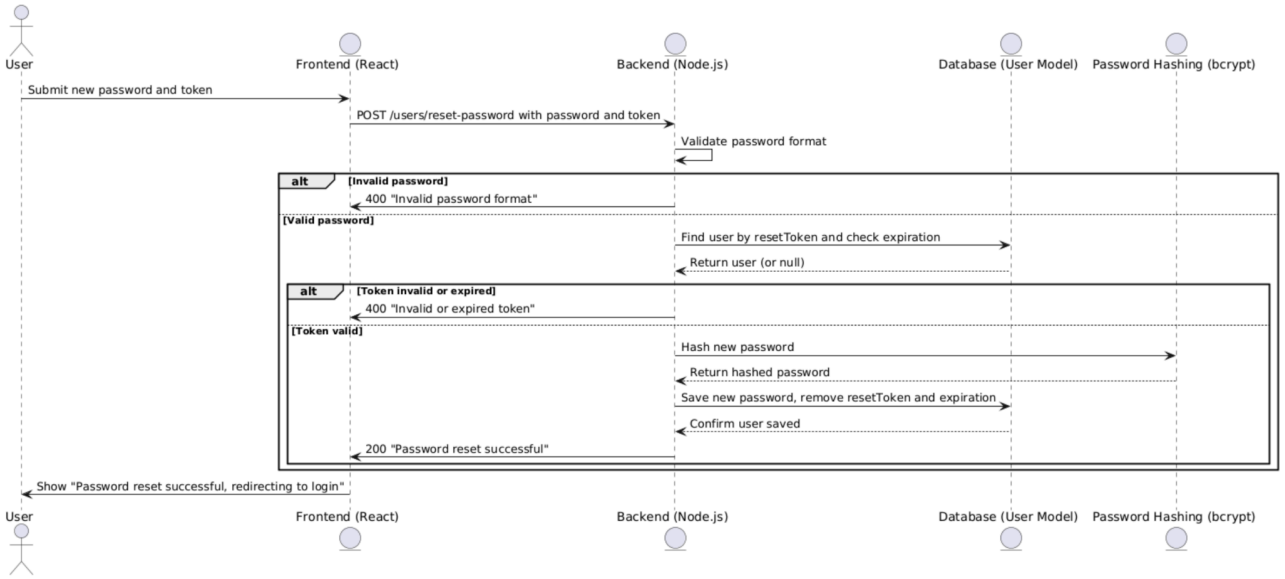
6.2.1.4.1 Sequence Diagram for sending password reset token



6.2.1.4.2 Sequence Diagram for sending confirming password reset token



6.2.1.4.3 Sequence Diagram for changing password



6.2.2 Core Functionalities

6.2.2.1 Authentication and Authorization

Functionality	Description
handleLogin	Validates user credentials using bcrypt; generates JWT access token for authenticated sessions (24-hour expiration). Logs user login time with updateOnlineTime.
handleVerifyToken	Verifies JWT token validity, ensuring only authenticated users access secure services.
verifyAccessToken	Parses JWT from request header, validates it, and attaches user info to the request object for role-based operations.
verifyIsAdmin	Ensures the user has admin privileges before accessing certain functionalities. (This is used when deciding whether to allow users to have access to AddQuestion, EditQuestion and ViewQuestion.)
verifyIsOwnerOrAdmin	Checks if the user owns the resource they're accessing or has admin rights.

6.2.2.2 User Registration and Profile Management

Functionality	Description
createUser	Registers new users by accepting username, email, and password; enforces uniqueness; hashes passwords with bcrypt before storage.
getUser	Retrieves user details by user ID, also updating their online status.
getAllUsers	Provides a list of all users in the system, formatted for structured responses.
getAllActiveUsers	Retrieves a list of active users only, formatted for structured responses.
updateUser	Updates user profile (username, email, password) with validation checks to prevent duplicate usernames or emails.
updateUserHistory	Updates the user's history with new question attempts, partners, and statistics about their performance.
deleteUser	Soft deletes a user by user ID, preventing access to their data while retaining it in the database for future reference.

6.2.2.3 Password Management and Reset

Functionality	Description
verifyPassword	Validates current password before allowing account changes, ensuring security by matching the hashed password.
sendPasswordResetEmail	Generates a secure reset token and sends it to the user's registered email for password recovery.
confirmToken	Confirms the validity of the password reset token, ensuring the reset request is secure and authorised.
resetPassword	Allows users to set a new password after token confirmation, storing the new password securely after hashing.

6.2.3 API Calls

6.2.3.1 Authentication and Authorization

HTTP Method	Endpoint	Handler	Middleware
POST	/login	handleLogin	-
GET	/verify-token	handleVerifyToken	verifyAccessToken

6.2.3.2 User Registration and Profile Management

HTTP Method	Endpoint	Handler	Middleware
POST	/	createUser	-
GET	/public	getPublicProfile	verifyAccessToken
GET	/active	getAllActiveUsers	verifyAccessToken, verifyIsAdmin
GET	/	getAllUsers	verifyAccessToken, verifyIsAdmin
GET	/:id	getUser	verifyAccessToken, verifyIsOwnerOrAdmin
PATCH	/:id/privilege	updateUserPrivilege	verifyAccessToken
PATCH	/:id	updateUser	verifyAccessToken, verifyIsOwnerOrAdmin

PATCH	/:id/matched	updateUserMatchedStatus	-
PATCH	/questions/:id	addQuestionToUser	verifyAccessToken, verifyIsOwnerOrAdmin
PATCH	/history/:id	updateUserHistory	verifyAccessToken, verifyIsOwnerOrAdmin
DELETE	/:id	deleteUser	verifyAccessToken, verifyIsOwnerOrAdmin

6.2.3.3 Password Management and Reset

HTTP Method	Endpoint	Handler	Middleware
POST	/forgot-password	sendPasswordResetEmail	-
POST	/confirm-token	confirmToken	-
POST	/reset-password	resetPassword	-
POST	/verify-password	verifyPassword	-

6.2.4 User Database Schema

Field	Type	Required	Default Value	Description
username	String	Yes	N/A	Unique identifier for the user.
email	String	Yes	N/A	Unique email address of the user.
password	String	Yes	N/A	Encrypted password for user authentication.
createdAt	Date	No	Date.now	The date and time when the user account was created.
isAdmin	Boolean	Yes	false	Indicates whether the user has admin privileges.
isActive	Boolean	No	true	Indicates whether the user's account is active or deleted.
questionDone	Array	No	[]	Stores IDs of questions that the user has completed.
history	Array	No	[]	More details in 6.2.4.1
onlineDate	Array	No	[]	Stores timestamps of when the user was online.
isMatched	Boolean	No	false	Indicates whether the user has been matched with another user.
matchData	Object	No	{}	More details in 6.2.4.2
resetToken	String	No	null	Token used for password reset functionality.
resetTokenExpiration	Date	No	null	Expiration date for the reset token, indicating when the reset token becomes invalid.

6.2.4.1 History Schema

Field	Type	Description
question	Object	Contains full details about the question attempted during the session, as of the time of the attempt. More details can be found at 6.3.4.
partner	String	Username of the partner associated with the question attempt.
startDateTime	String	The date and time at which the session was started.
attempt	Object	Contains details about the latest and highest scoring (by number of test cases passed, if available) code submission attempt.
- language	String	Programming language used for the attempt (e.g., "python", "java", “javascript" or "None").
- content	String	Content of coding panel submitted during the attempt.
- testCases	Array	Boolean array indicating which test cases were passed by the code submitted in this attempt.
timeTaken	String	Duration of the session (e.g., "0:4" indicates 4 seconds. Displays as minutes:seconds by default but displays as hours:minutes:seconds if required).
completion	String	Indicates the percentage of test cases completed, if available (e.g., "-" if no attempt was made or test cases are not available, or "x.x%" if an attempt was made and test cases are available).

6.2.4.2 Match Data Schema

Field	Type	Description
matchData	Object	
- matched	Boolean	Indicates if the user successfully matched with another user (e.g., true if matched).
- matchedUserId	String	Unique identifier of the matched user.
- matchedUserName	String	Username of the matched user.
- category	Array of Strings	Array listing relevant categories for the match session (e.g., ["Bit Manipulation"]).
- complexity	String	Difficulty level of the question provided in the session (e.g., "Easy", "Medium", "Hard").
- sessionId	String	Unique identifier for the current session.
- question	Object	Object containing details of the coding question for the match session.
-- testcase	Object	Object containing details for test cases associated with the question.
-- _id	String	Unique identifier for the question.
-- title	String	Title of the question (e.g., "Add Binary").
-- description	String	Detailed description of the question.
-- category	Array of Strings	Array listing categories associated with the question (e.g., ["Bit Manipulation", "Algorithms"]).
-- complexity	String	Difficulty level of the question (e.g., "Easy", "Medium", "Hard").
-- __v	Int32	Version number or internal versioning field for the question data.
- userNo	Int32	Integer indicating if this is the first or second user involved in the matching.

6.3 Question Service

6.3.1 Overview

In building our question-service, we utilised MongoDB, Express, and Docker.

- MongoDB serves as our NoSQL database, providing flexible document-based storage that supports rapid data retrieval and scalability.
- Express, a flexible Node.js framework, streamlines API handling, making it easier to manage routes and middleware.
- Docker containers ensure portability and consistency across development and production environments.

Question service's purpose is to enable users to view the questions from the question bank and also obtain questions during collaboration sessions to practise programming with their partner. It also allows admins to create, update and delete questions.

6.3.2 Core Functionalities

Functionality	Description
getAllQuestions	Provide a list of all questions in the system
createQuestion	Add new questions by taking in question title, description, category,and complexity.
updateQuestion	Update questions details such as question title, description, category, and complexity.
deleteQuestion	Delete a question by question ID,
getQuestion	Retrieves a single question details by question ID
getQuestionsByID	Retrieves multiple questions based on an array of question IDs provided.
getQuestionOnMatch	Retrieves a single question based on the given category and complexity.
addTestCase	Update question details to add test cases.

6.3.3 API Calls

HTTP Method	Endpoint	Handler	Middleware
POST	/ids	getQuestionsByID	verifyUser
POST	/specific	getQuestionOnMatch	verifyUser

PATCH	/	addTestCase	verifyAdmin
GET	/	getAllQuestions	verifyUser
POST	/	createQuestion	verifyAdmin
PUT	/:id	updateQuestion	verifyAdmin
DELETE	/:id	deleteQuestion	verifyAdmin
GET	/:id	getQuestion	verifyUser

6.3.4 Question Database Schema

Field	Type	Required	Default Value	Description
_id	String	Yes	N/A	Unique identifier for the question.
title	String	Yes	N/A	Title of the question.
description	String	Yes	N/A	Detailed description of the question.
category	Array	Yes	N/A	Categories associated with the question (e.g., "Recursion").
complexity	String	Yes	N/A	Difficulty level (easy, medium, hard)
testcase	Object	No	N/A	Object containing test case details.
- isAvailable	Boolean	No	false	Indicates if the question includes test cases
- python	Object	No	N/A	Python-specific test case details.
- params	String	No	""	Parameters for the function.
-input	Array	No	[]	Input values.
-output	Array	No	[]	Expected output
- java	Object	No	N/A	Java-specific test case details.
- params	String	No	""	Parameters for the function.
-input	Array	No	[]	Input values.
-output	Array	No	[]	Expected output

6.4 Matching Service

6.4.1 Overview

In building our matching-service, we utilised MongoDB, RabbitMQ, Express, and Docker.

- MongoDB serves as our NoSQL database, providing flexible document-based storage that supports rapid data retrieval and scalability.
- Express, a flexible Node.js framework, streamlines API handling, making it easier to manage routes and middleware.
- Docker containers ensure portability and consistency across development and production environments.

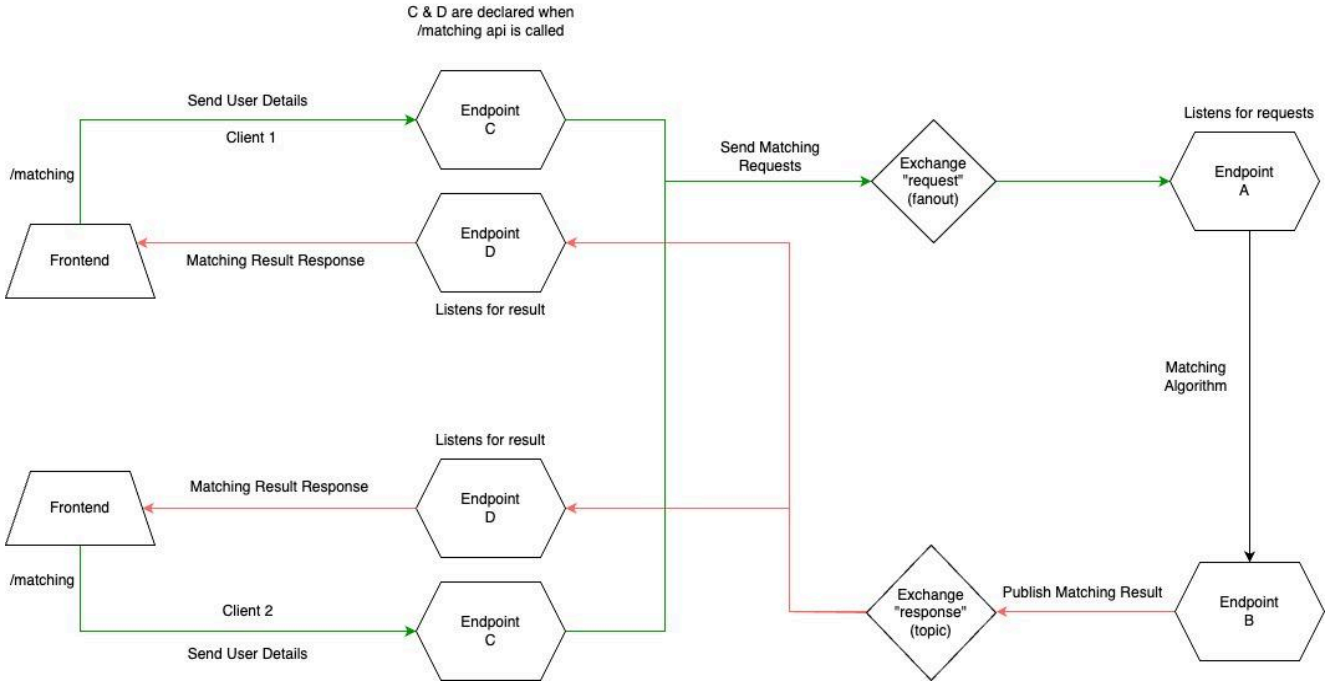
For the Queue System, we chose RabbitMQ instead of Kafka and Redis.

Feature	RabbitMQ	Kafka	Redis
Persistency	Persistent message storage, configurable.	Persistent message storage.	Data persistence primarily for caching but supports streams.
Consumer Capability	Good for message queuing, supports multiple consumers through exchanges.	Good for one-to-many messaging but lacks the advanced routing capabilities of RabbitMQ.	Limited consumer handling, more cache-focused.
Performance / Scalability	Scalable with plugins, suitable for high-throughput.	High performance for streaming	High performance, more optimised for data caching and in-memory operations.
Use Case	Suitable for real-time analytics, background tasks.	Good for high-volume data streaming.	Great for caching data, not ideal for complex message routing.

Considering the strengths and weaknesses of each message broker listed above, we found RabbitMQ to be the best option for us due to its routing capabilities and simplicity of implementation.

6.4.2 Matching Flow Explanation

6.4.2.1 Matching Flow Diagram



Upon initialization of the matching service, two RabbitMQ exchanges are declared: a “request” (fanout) exchange for matching requests and a “response” (topic) exchange for matching results. Endpoint A subscribes to the “request” exchange to listen for incoming matching requests, while endpoint B publishes the matching results to the “response” exchange.

The matching process begins when the /matching endpoint is called by the frontend, endpoints C and D, are set up. Endpoint C publishes the matching requests via the “request” exchange to endpoint A, and endpoint D subscribes to the “response” exchange to listen for matching results.

A match between two matching requests is determined by calculating a “Match Score” between each request, determined by the number of the following conditions satisfied:

1. At least one category is selected in both requests (+2 to score).
2. The difficulties of the two requests differ by at most 1 (easy and medium or medium and hard) (+1 to score).
3. The difficulties of the two requests match exactly (+1 to score).

Once a matching request is received in endpoint A, the matching algorithm is run to match the new request with the requests in the queue. If there is no match, the matching request will be added to the queue. The matching algorithm is then run again every 6 seconds with the minimum match score required for a match starting at 4 and decrementing by 0.5 on each run, up to a minimum of 2.5. The algorithm runs a total of 5 times including the initial run, requiring a minimum match score of 2.5 on the last two

runs, allowing for matches between requests with at most 1 difference in difficulty. If a match is found during any of the runs, the matching data from the two requests is used to form a successful matching result. If a match is not found in 30 seconds, the algorithm times out, returning an unsuccessful match result.

Endpoint B then publishes the matching result with the user ID as the topic key, so only the intended recipient, endpoint D can receive it. Finally, the match data will be sent to the frontend for the match details to be displayed.

6.4.3 Core Functionalities

6.4.3.1 Match (connect to MongoDB)

Functionality	Description
createMatch	Create a new match to be stored in the database
getMatchById	Retrieve a single match by its ID
getAllMatches	Retrieve all matches (optional: filter by category, complexity, etc.)
deleteMatchById	Optionally delete a match by ID
getMatchBySessionId	Retrieve a single match by its sessionID

6.4.3.2 Match Request (connect to rabbitMQ)

Functionality	Description
createMatchRequest	Sends a request to RabbitMQ to initiate a matching process. This function queues a new match request, allowing the system to process and pair participants based on predefined criteria.
cancelMatchRequest	Sends a cancellation request to RabbitMQ to terminate an existing match request. This function removes the specified match from the queue, ensuring that no further processing occurs for that match request.

6.4.4 API Calls

HTTP Method	Endpoint	Handler	Middleware
GET	/	getAllMatches	-
GET	/:id	getMatchById	-
GET	/session/:sessionId	getMatchBySessionId	-
POST	/	createMatchRequest	-
DELETE	/:id	cancelMatchRequest	-

6.4.5 Match Database Schema

Field	Type	Required	Default Value	Description
sessionId	String	Yes	N/A	Unique identifier for the matched session
user1Id	String	Yes	N/A	Unique identifier for user 1
user1Name	String	Yes	N/A	Username of user 1
user2Id	Array	Yes	N/A	Unique identifier for user 2
user2Name	String	Yes	N/A	Username of user 2
category	Array	Yes	N/A	Categories associated with the question (e.g., "Recursion", "Algorithms")
complexity	String	Yes	N/A	Difficulty level (easy, medium, hard)
createdAt	Date	Yes	Date.now	The date and time when the match was created

6.5 Collaboration Service

6.5.1 Overview

The Collaboration Service handles the communication of changes in the coding panel between the two users using Socket.IO. It also manages data storage and saving of session data while a session is in progress. This includes the content of the code window for each individual language, data of code submissions, data of the users involved in the session, and content of the whiteboard.

In building our matching-service, we utilised MongoDB, RabbitMQ, Express, and Docker.

- MongoDB serves as our NoSQL database, providing flexible document-based storage that supports rapid data retrieval and scalability.
- Express, a flexible Node.js framework, streamlines API handling, making it easier to manage routes and middleware.
- Docker containers ensure portability and consistency across development and production environments.
- Socket.IO allows real-time communication, it enables bidirectional event-based communication between clients and servers. This is essential for our collaboration service, as it allows for instantaneous updates and notifications, enhancing user experience during live interactions.

6.5.2 Communication between clients

Communication between clients is handled by Socket.IO. When a user logs in, the collaboration service connects them to the server via Socket.IO. When two users are matched through the matching process, a session is created with a unique session ID (included in the match data previously detailed in section 6.4.5). Upon entering the collaboration page, the coding panel is initialised and a join event is emitted to the Socket.IO server which prompts it to join the session via the session ID. This enables users connected to the same session to receive events emitted by their partners during the course of the session. Events emitted include the following:

Event Emitted	Event triggered when	Response to event
join	A user is matched and a unique session ID is allocated.	The user joins the session, serverside.
codeChange	The code in the coding panel is changed for any language by either user.	Changes in code by a user's partner for this session are reflected in the user's coding panel. A timeout is also started to save the code in session data after 0.5 seconds.
languageChange	A user changes the language they are coding with locally.	A notice is displayed in chat for the user when their partner changes the language they are coding in.

endSession	A user clicks the “Leave Session” button.	The user ends the session and navigates to the summary page and emits the ‘partnerLeft’ event to the user’s partner. If the user’s partner has already left, also deletes session data.
partnerLeft	A user’s partner leaves.	The user receives a notification displaying that their partner has left.
getDrawing	A user clicks on the whiteboard tab.	Saved drawing from the database will be reflected on the whiteboard.
startDrawing/drawing	A user begins drawing on the whiteboard.	The drawing is reflected on the partner’s whiteboard as well.
endDrawing	A user stops drawing on the whiteboard.	The drawn stroke is saved to the session data.
clearWhiteboard	A user clicks on the “Clear Canvas” button.	The empty whiteboard is reflected on the partner’s whiteboard as well.
sendMessage	A user sends a message in the chat.	The message is sent to the other user.

6.5.3 Handling Leaving of Sessions

Several design considerations were made when deciding how to handle users leaving the session. The following options were considered:

1. When one user leaves the session, forcibly end the session for both users. Since the goal of the collaboration space is to practise both coding skills and collaboration, the latter element is forgone upon a user’s partner leaving. Therefore, users should not be allowed to continue the session independently.
2. When one user leaves the session, inform the other user but allow them to stay in the session until they decide to leave. This allows the other user to continue writing and submitting code. This option prevents sessions from ending prematurely in cases where one user has to leave early.
3. When one user leaves the session, prompt the other user for confirmation to end the session. If the other user accepts, the session ends for both users. Otherwise, the session continues for both users. This option presents a middle ground between the other two options.

Ultimately, we decided to choose option 2 due to the following reasons:

- Option 1 allows for a user to forcibly end the session at any time for the other user against their will and removes the option for a user to continue solving a question if their partner has to leave, especially when they have already found a solution but have yet to type it out.
- Option 3 potentially allows for a user to prevent their partner from leaving indefinitely, allowing users to effectively take their partner’s accounts hostage as they are unable to start a new session without ending the current active one.

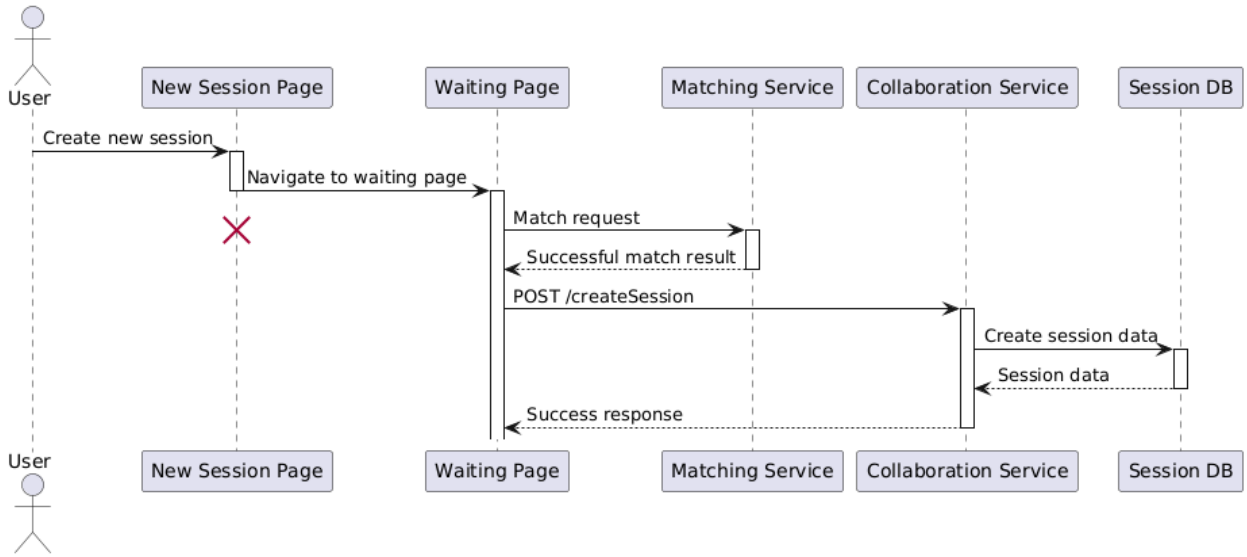
Some design consequences of selecting option 2 is as follows:

1. When a user leaves, their partner is notified with a pop up at the top of the screen that remains until it is manually closed. A notice is also displayed in the chat.
2. The same pop up and notice is also displayed when a user rejoins the session after leaving the page. This ensures the user is aware of their partner having left even if the partner had left while the user was disconnected.
3. When a user leaves the session, their history is updated with a new entry containing details of the session at the time of leaving. This allows only attempts and statistics relevant to the user to be saved (attempts submitted after the user has left will not be saved into the history of that user).

6.5.4 Saving of Session Data

Session data for each individual session is saved using MongoDB. When two users are matched through the matching process outlined in the previous section, the match data sent back to the client of both users involved assigns one user as user 1 and the other as user 2. The client for user 1 then attempts to create the session data for this session, which includes code for each language, code submission attempts, users involved in the session, and the content of the whiteboard (more details in section 6.5.5). The default code for each language is initialised depending on whether test cases are available for the question being attempted (to provide code in a format that can be run to produce an output given available test cases), while the code submission attempts and whiteboard data are initialised as empty arrays.

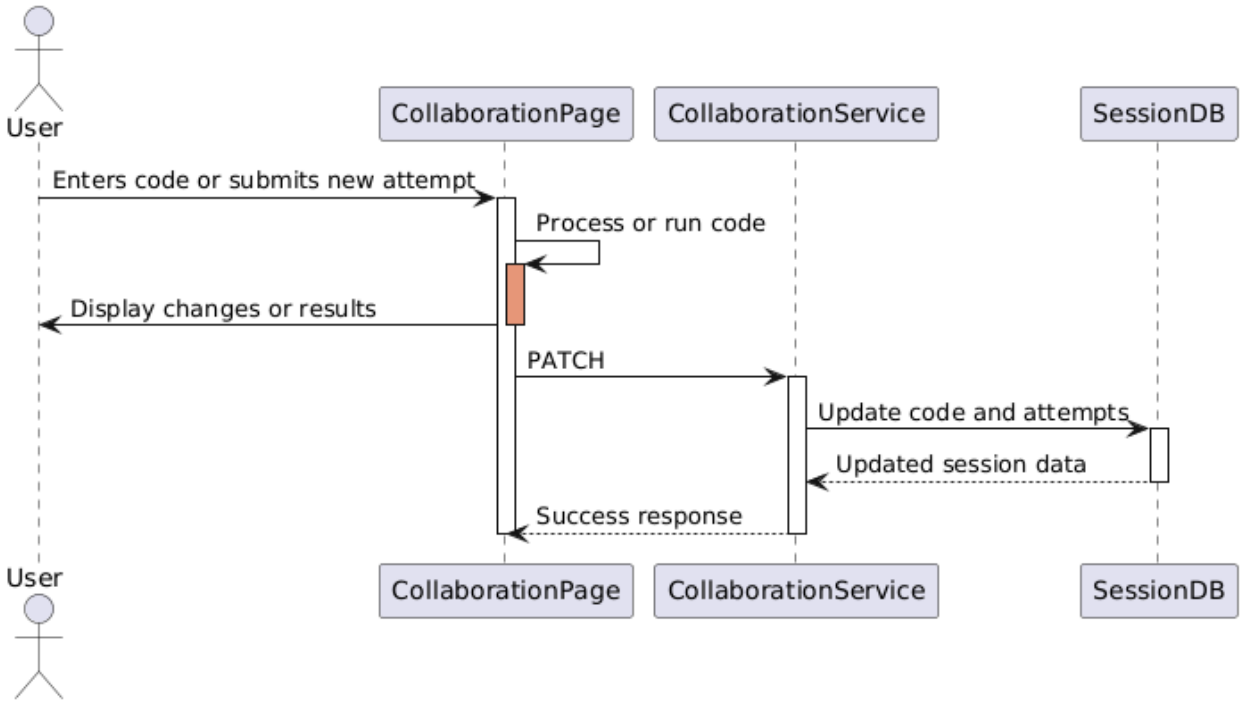
6.5.4.1 Sequence diagram for creation of session data



Upon a user entering the collaboration page, the code to be displayed is loaded from the session data on MongoDB. If it is the first time a user loads the code panel, the code displayed will be the default code as initialised above when the session data was created. This interaction ensures that the correct code is displayed on a user entering the collaboration page, even in the case of unforeseen disconnection or closing of the page.

When code is input into the code panel by one user, an event is emitted using Socket.IO to communicate the change to the other user, reflecting the same input on the code panel of the other user. The content of the coding window is saved to MongoDB 0.5 seconds after either user stops inputting code into the code panel by calling the PATCH API call, which updates the session data according to the data provided.

When code is executed (more details in section 7.5), the code being used in the execution is recorded as an attempt. An attempt includes the code being run, a boolean array indicating the test cases passed, as well as the coding language being used. An attempt number is also generated depending on the number of previous attempts recorded. The attempt is saved into the array of past attempts by calling the same PATCH API call after each code submission.



When the user clicks on the whiteboard tab, the initial saved stroke data will be retrieved from the session data on MongoDB and displayed on the user's whiteboard. As the user draws on the whiteboard, the change is communicated to the other user using Socket.IO. Each time a user stops drawing (whenever they complete a stroke), details such as starting positions, line width, and colour are saved to MongoDB. The whiteboard data is also saved whenever it is cleared, resetting the data to an empty array.

When one user leaves the session, the "partnerLeft" field is updated to indicate that at least one user has left.

When both users leave the session, the DELETE API call is called to delete the session data from MongoDB.

6.5.5 Core Functionalities

Functionality	Description
getSession	Retrieves a session from the database based on the provided sessionid in the request parameters.
createSession	Creates a new session in the database by taking in sessionid, userid1, userid2, username1 and username2.
updateSession	Updates data for an existing session in the database. It checks for a session ID in the request body and updates the code for a specified programming language, adds a new attempt to the session's past attempts or updates whether one of two users have left the session.
deleteSession	Deletes a session based on the sessionid provided.
getWhiteboard	Retrieves whiteboard data from the database based on sessionid.
saveWhiteboard	Saves whiteboard data to the database associated with the provided sessionid.
clearWhiteboard	Clears whiteboard data in the database based on sessionid.

6.5.5 API Calls

HTTP Method	Endpoint	Handler	Middleware
GET	/:id	getSession	-
POST	/:id	createSession	-
PATCH	/:id	updateSession	-
DELETE	/:id	deleteSession	-

6.5.6 Session Database Schema

Field	Type	Required	Default Value	Description
sessionid	String	Yes	N/A	Unique identifier for the matched session.
codewindow	object	No	N/A	Contents of the code window for each language.
- python	String	No	defaultCodes.python	Python code. Pre-populated with default Python code.
- java	String	No	defaultCodes.java	Java code. Pre-populated with default Java code.
- javascript	String	No	defaultCodes.javascript	Javascript code. Pre-populated with default Javascript code.
pastAttempts	Array	No		Contains previous attempts made by the user. Attempts are stored in the database whenever the user runs the code.
- attemptNo	Number	No		Attempt number.
- language	String	No		The programming language used for the code in the attempt.
- content	String	No		The code ran in the attempt.
- testCases	Array	No		An array of test cases associated with the attempt, used for validating the code execution.
users	Array	No	N/A	Array containing information about the users involved in the session.
- userId	String	No	N/A	The unique identifier for a user.
- username	String	No	N/A	Username of the user.
whiteboard	Array	No	[]	An array that stores the drawing made on the whiteboard.
partnerLeft	Boolean	Yes	false	Indicates whether at least one user has left the session.

7 Nice to Haves

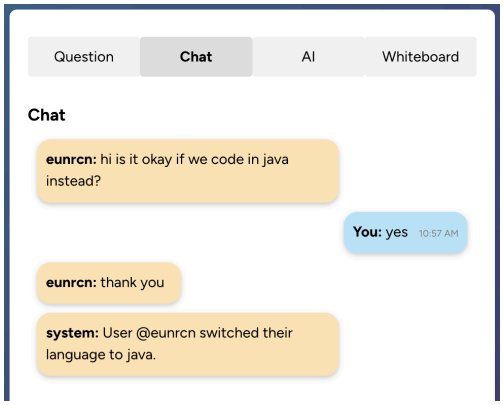
7.1 Overview

Nice to Have	Description	Port	Key Technologies
Communication	Manages chat history between two users in the same session, facilitates text communication between the two users during collaboration.	8085	Node.js, Express, Docker, MongoDB
History	Allows users to monitor progress, and revisit past solutions. Displayed in the dashboard, including question titles, results, timestamps, and code.	-	Node.js, Express, Docker, MongoDB
Code Execution	Executes code submissions securely in isolated environments. Includes Test Case Input/Output Panel.	8083	Node.js, Docker, Java, Python
Enhanced code editor with syntax highlighting	Provides language switching between Python, Java and Javascript. CodeMirror also allows Auto-completion, Indentation and Syntax Highlighting.	-	CodeMirror
Incorporate generative AI	Provides generative AI-based communication. Ollama platform used to run and manage language models locally.	9680 11434	FastAPI, Ollama
Whiteboard	Manages whiteboard communication between two users in the same session, allowing them to draw while collaborating.	-	Sockets, MongoDB

7.2 Assignees

No.	Description	Person-In-Charge	User Testing
1	Communication	Rayson	Andre Ashley Eunice Rayson Shi Kang
2	History	Andre	Andre Ashley Eunice Rayson Shi Kang
3	Code Execution	Eunice	Andre Ashley Eunice Rayson Shi Kang
4	Enhanced code editor with syntax highlighting	Ashley	Andre Ashley Eunice Rayson Shi Kang
6	Incorporate generative AI to assist during the preparation	Shi Kang	Andre Ashley Eunice Rayson Shi Kang
13	Whiteboard	Andre Ashley Eunice Rayson Shi Kang	Andre Ashley Eunice Rayson Shi Kang

7.3 (N1) Communication



7.3.1 Functionalities

This feature allows users to communicate in real-time during their collaborative coding sessions through a text-based chat box. In the chat tab, users can view each message along with their partner's username, content and timestamp.

Real-Time Messaging: Users can send and receive messages instantly during a session.

Message History: Previous messages within a session are retrieved and displayed when a user joins, allowing participants to view past conversations when rejoining after a disconnect.

Session-Based Chat: Each chat session is associated with a specific ID, ensuring that messages are stored and accessible only for that session.

Notifications and Alerts: The chat includes notifications to keep users informed about important session events. For example, users are notified when their partner changes languages or exits the session, ensuring participants stay updated on any changes.

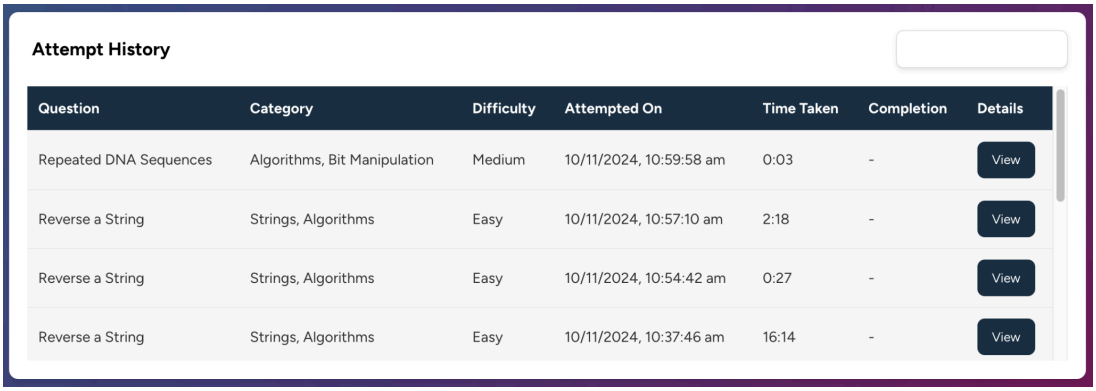
7.3.2 Implementation Details

Frontend: The Chat component handles the messaging UI. It uses useState for message storage and input handling, and useEffect to fetch message history upon loading the collaboration page. Messages are sent and received via axios API requests and real-time socket updates.

WebSocket Integration: Socket.IO was used to manage real-time communication. Events such as sendMessage and receiveMessage allow the frontend to update in real-time when a user sends a new message.

Backend: We use a chat service to manage sessions. MongoDB is used to store the chat history, containing details such as session ID, user ID and messages.

7.4 (N2) History



7.4.1 Functionality

The history of questions attempted in past sessions and related statistics is recorded to provide the user the ability to review past attempts of questions in general or in specific.

Each entry in the history records, for one session, the following fields:

Field	Description
Question	Title of the question attempted at the time of the attempt..
Category	Categories of the question attempted at the time of the attempt.
Difficulty	Difficulty of the question attempted at the time of the attempt.
Attempted On	The date and time at which the session began, in SGT time zone.
Time Taken	The duration of the session, displayed in minutes and seconds if the time does not exceed an hour and displayed in hours, minutes and seconds otherwise.
Completion	The proportion of test cases passed during the session by the highest-scoring and latest attempt (in that order of importance).
Details	Contains a button that can be clicked to view more details. Details will be shown in a pop up window on the same page.

By default, the latest 10 entries recorded are displayed in order of latest to earliest sessions.

When clicking on the "View" button in the details column, a pop up window appears on the same page, showing the details listed above, as well as the following additional details:

- Username of the partner for the session.
- Description of the question attempted at the time of the session.
- Language of the code submitted in the highest-scoring and latest attempt, if any.
- Code submitted in the highest-scoring and latest attempt, if any.
- Test cases passed in the form of [Test Cases Passed]/[Total Test Cases Available].

History Filtering: A search bar is implemented at the top of the history that allows users to filter their history by keywords they input. They can also filter by multiple keywords by separating the keywords with commas. The search is not case sensitive and searches in the following fields:

- Question title, categories (individually), and difficulty.
- Date and time of session.
- Time taken for the session.
- Completion %.
- Username of partner.
- Content of the code in the highest-scoring and latest attempt.
- Language of the highest-scoring and latest attempt.

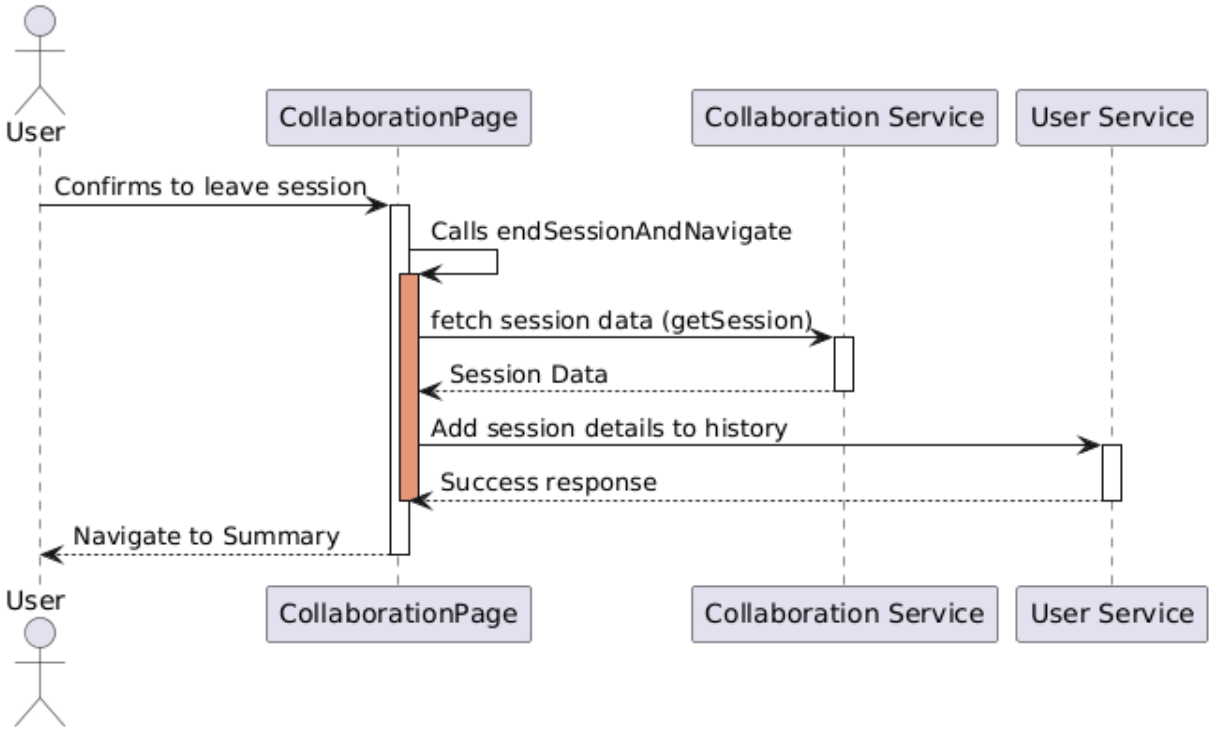
When filtering, the number of entries displayed is not limited to 10 unlike the default display to allow users to view all entries that match the keywords provided. Using this, users are able to search for past attempts of specific questions and review the code they had submitted to improve their coding. Being able to view the partner's username may also be useful for the moderation of users should such functions be implemented in the future.

7.4.2 Implementation

7.4.2.1 Updating of New Entries in History

Upon leaving a session, the match data for this session is queried from MongoDB, providing the question data and username of the partner. The starting date and time for the session is then obtained from local storage, along with the elapsed time for the session, which is used to calculate the time taken. Lastly, the best attempt is searched for in the session data. This data is then used to push a new entry into the history of the user using a PATCH API call to the user service. More details of the schema for each entry into the history can be found in section 6.2.4.1.

7.4.2.1.1 Sequence diagram for saving of history entry

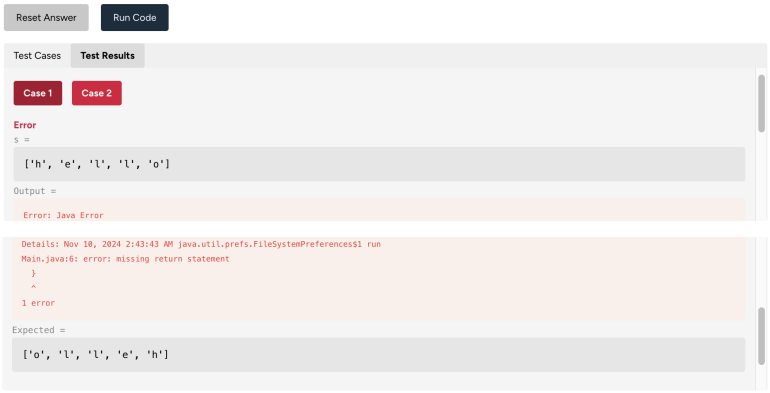


7.4.2.2 Display and Filtering

During the initialization of the dashboard, user details are retrieved to be displayed. The user's history is similarly extracted from user details and passed to the History component. With no input in the search bar, the history component displays the latest 10 entries in history in order of newest to oldest. This is done by reversing the order of the array and taking the first 10 entries, given that the history is by default sorted oldest to newest due newer entries being pushed to the end of the array.

When input is entered into the search bar, the input string is split by commas, then trimmed, before the history array is filtered by entries that have at least one field containing each keyword.

7.5 (N3) Code Execution & Test Cases



7.5.1 Thought process while creating code-execution-service

While many built-in libraries, such as Judge0, utilise Docker to execute code in isolated, secure, and sandboxed environments, we felt it was too convenient to depend solely on a library that manages all aspects for us—ranging from compiling and running code to providing test cases and handling the sandboxing. Instead, we opted to develop our own backend code execution service to deepen our learning experience and gain a better understanding of the underlying processes.

Consequently, we designed a comprehensive process for executing code while ensuring it functions within a controlled sandboxed environment. This involved implementing our own logic for code compilation, execution, and error handling, along with creating the necessary infrastructure to support multiple programming languages. Here's how we achieved this:

Docker Integration: We utilised Docker to create isolated environments for each programming language, allowing us to run user-submitted code securely without affecting the host system.
Dynamic Code Execution: By generating temporary files for user code and constructing appropriate execution commands, we ensured that our service could handle various languages effectively.
Custom Test Case Implementation: Instead of relying on pre-existing test case libraries, we designed our own test cases tailored to different scenarios. This approach not only enhanced our understanding of testing methodologies but also provided flexibility in how we validated user submissions.
Error Handling and Feedback: We implemented robust error handling to capture and log any issues that arose during code execution. This allowed us to provide informative feedback to users, facilitating easier debugging.
Performance Optimization: Our implementation included cleanup processes to remove any residual files after execution, optimising resource usage and maintaining system performance.

Additionally, we took the initiative to create our own test cases from scratch, enabling us to thoroughly evaluate the correctness of user solutions in various scenarios. This hands-on experience was invaluable in reinforcing our understanding of both software development and testing principles.

7.5.2 Technical Design and Workflow

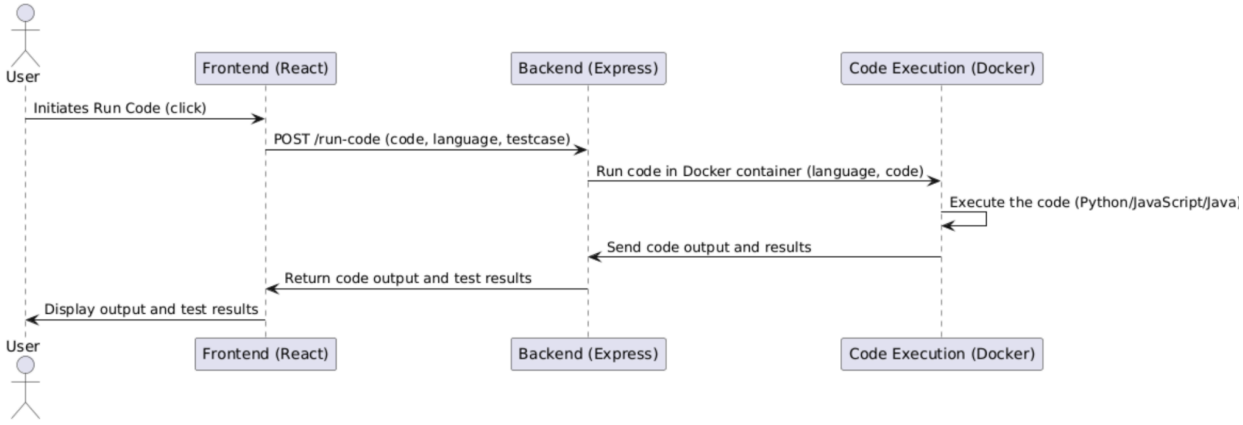
Request Handling: The API layer receives the request and parses the request body to extract the necessary information (code, language, and test cases).
Code Execution: For each programming language, a temporary file is created to store the submitted code. The service then invokes the corresponding language function (e.g., runPython, runJava, runJavaScript), which compiles and/or executes the code using system commands through the `exec` function from Node.js's `child_process` module. Each execution is executed inside a Docker container, making it a secure and controlled environment, allowing the code to run without affecting the host system.

Wrapping the runPython, runJava and runJavascript functions around runInDocker to ensure code execution occurs in a sandboxed environment:

```
const runPython = (code) => runInDocker("python", code);
const runJava = (code) => runInDocker("java", code);
const runJavaScript = (code) => runInDocker("javascript", code);
```

Cleanup: In the command that starts the Docker container, the --rm flag is used. This flag automatically removes the Docker container after it finishes executing, ensuring no lingering containers take up resources or clutter the system.

7.5.2.1 Sequence Diagram for running the code



7.5.3 How runInDocker Works

1. **Creating a Promise**: The function returns a Promise, which is a JavaScript object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. This allows the function to handle asynchronous code execution, meaning it can run without blocking the main thread.
2. **Choosing the Docker Image**: Based on the specified programming language, the images are standard Docker images for each language, providing the runtime environment needed to execute code in that language.

```
const dockerImage =
  language === "python"
    ? "python:3.13"
    : language === "java"
    ? "openjdk:17"
    : "node:18";
```

3. **Setting the File Name**: Similarly, based on the language:

```
const fileName =
  language === "python"
    ? "code.py"
    : language === "java"
    ? "Main.java"
    : "code.js";
```

4. **Constructing the Command:** The command variable is constructed as a string that contains the shell commands to be executed. Here's what happens:
    a. The echo command is used to write the provided code into a file.
    b. The docker run command is prepared to execute the code inside the Docker container:
        i. --rm: Automatically removes the container once it finishes executing, preventing clutter from leftover containers.
        ii. -v $(pwd):/usr/src/app: Mounts the current working directory into the container, allowing the container to access files from the host.
        iii. -v /tmp/.java:/root/.java: This line is specific for Java, allowing Java to use the system's preferences directory for running.
        iv. -w /usr/src/app: Sets the working directory inside the container where the command will be executed.
        v. Finally, depending on the language, the appropriate command to run the code is appended:
            1. For Python, it runs the Python script directly.
            2. For Java, it compiles the Java file and runs the compiled class.
            3. For JavaScript, it runs the Node.js script.
5. **Executing the Command:** The commands are executed within a Docker container using the exec function from Node.js's child_process module. It takes a callback function that handles the result:

```
exec(command, (error, stdout, stderr) => {
```

7.5.4 Creating a code running service like judge0

In developing our code execution feature, we closely considered Judge0's extensive functionality and aimed to replicate as many of its features as possible. While we successfully implemented several comparable features, it is valuable to reflect on how our solution measures up to an established platform like Judge0 and to understand the additional capabilities that a comprehensive library like Judge0 offers.

Feature	Our Run Code	Judge0
Language Support	Python, Java, JavaScript	over 40 languages
Execution Environment	Uses Docker containers with specific images for Python (python:3.12.7-slim), Java (openjdk:17), and JavaScript (node:18)	Uses Docker, with custom images for each language; provides detailed control over runtime settings, limits
Execution Command	Uses shell commands via exec() to compile and run code	Uses REST API to send code and configuration for remote execution
Error Handling	Catches and cleans error messages	Returns structured error messages including error type, line, and stderr output
Testing Capabilities	Python and Java have helper functions (pythonTestcase, javaTestcase) for input/output and expected results	Supports input/output test cases and allows configurable test cases directly via API
Code Isolation	Each code execution is isolated within a Docker container, ensuring code isolation and security	Each execution is also containerized, ensuring code isolation and security
System Independence	Runs independently within our infrastructure, ensuring availability even if Judge0 is down	Requires external API availability; system outage or API downtime affects access to code execution

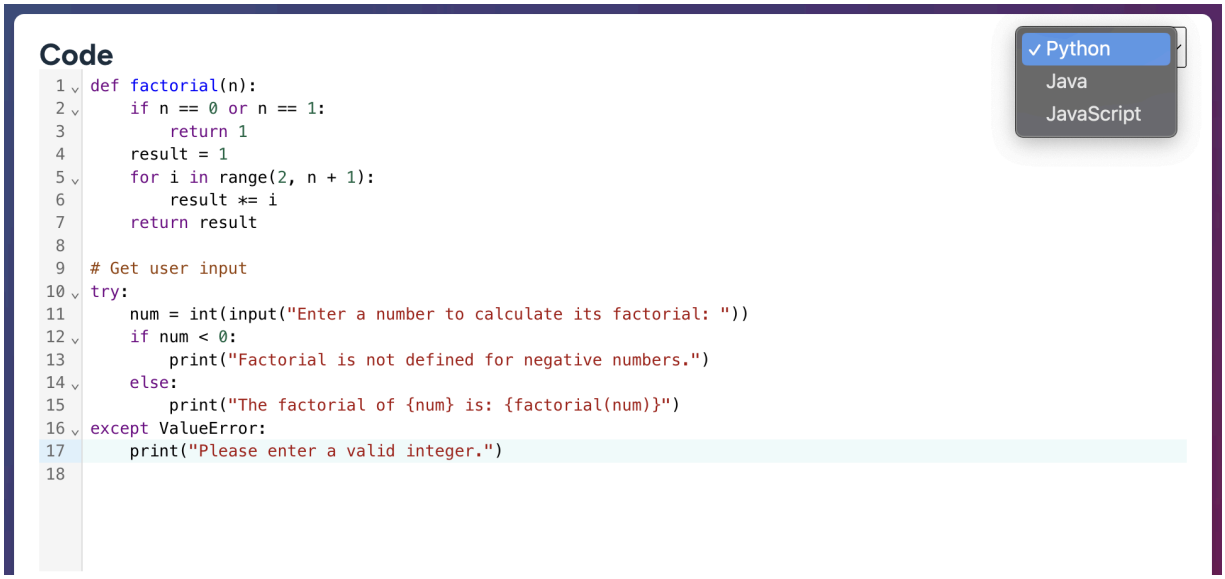
7.5.5 API Calls

Endpoint	Method	Description	Request Body
/run-code	POST	Executes submitted code in the specified programming language and returns the output or error.	{ "code": "<code>", "language": "<programming language>", "testcase": { "isAvailable": <boolean>, "<language>": { "input": [<input values>], "output": [<expected output values>] } } }

7.5.6 Responses

Status Code	Description	Response Example
200 OK	Indicates successful execution of the code, providing the output and the results of test case evaluations.	{ "output": <output>, "result": [<boolean_results>] }
400 Bad Request	Returned when an unsupported programming language is specified in the request.	{ "error": "Unsupported language" }
400 Bad Request	Indicates that the input data is malformed or missing required fields in the request.	{ "error": "Invalid input data", "details": <details> }
500 Internal Server Error	Returned when an error occurs during code execution, such as compilation errors or runtime exceptions. The error message provides insight into the nature of the failure.	{ "error": <error_message>, "details": <error_details> }

7.6 (N4) Enhanced code editor with syntax highlighting



7.6.1 Core Functionalities

We made use of CodeMirror as it is a well-established library that supports multiple programming languages, allowing users to switch between languages seamlessly. The enhanced code editor provides users with an intuitive interface for writing and editing code across multiple programming languages, including JavaScript, Python, and Java.

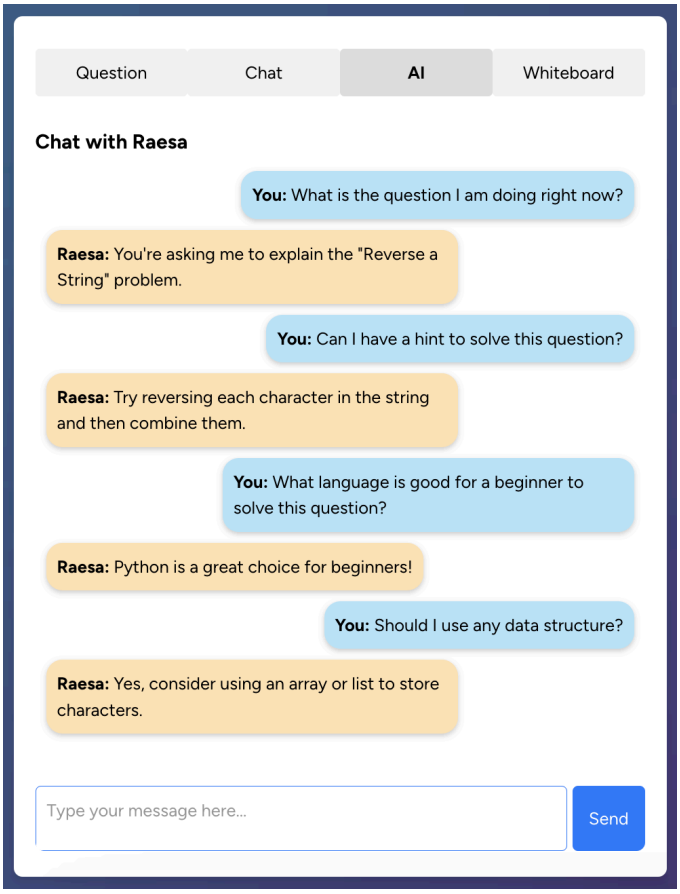
Key Features:

Syntax Highlighting: Different elements of the code are colour-coded based on the programming language, making it easier for users to read and understand the structure of their code. Additionally, the editor listens for language changes to update the highlighting appropriately, ensuring a consistent user experience.

Error Highlighting: The editor can highlight syntax errors, helping users identify and correct issues as they code. This feature not only improves coding efficiency but also reduces frustration by providing immediate feedback.

Overall, CodeMirror enhances the coding experience by combining a robust feature set with an easy-to-use interface, making it an ideal choice for our enhanced code editor.

7.7 (N6) Incorporate generative AI



7.7.1 Core Functionalities

Peerprep features a built-in AI model named Raesa, specifically designed to assist users to efficiently prepare and answer coding questions. Raesa is powered by Meta's Llama 3.2 model, a state-of-the-art, lightweight small language model known for its advanced capabilities. Our AI model runs locally, via the use of Ollama, a platform that manages and orchestrates various AI models.

To enhance user experience, Raesa is designed to receive the current question's context in advance, allowing it to better assist users by providing responses that are relevant and effective. The context enables Raesa to act like a real-time coding assistant that can assist with debugging and code optimisation. Furthermore, by using a local model, we can fine-tune the model as needed and ensure better data privacy.

7.7.2 Technical Design and Workflow

Streaming endpoint: The endpoint /stream expects POST requests from the frontend containing the user query and question in the body. The stream() function is the main entry point for handling requests to this endpoint, where it parses and verifies that both the query and question fields are provided.

Prompt Formatting: The query and question are then sent into the prompt_formatter function where a structured prompt for the language model is created. Various information will be part of the prompt such as the AI's name "Rasea", the user query and the question context. It also includes instruction for the model to provide a concise and context-aware answer.

Query the Model: The structured prompt will be sent to the language model through the get_response function. The function will send a POST request to the Ollama API (/api/generate) with the prompt. The model is configured to respond in a streaming format, which will be yielded back to the client for continuous, readable stream of the model's response, allowing for a real-time interactive experience.

7.7.3 Local Model vs OpenAI API

One of the key decisions was to opt for a local model over the widely popular OpenAI API. The table below shows some key considerations for the choice. A local model was selected as it was significantly safer, incurs no cost and offers fast performance. Given that the model's primary purpose is to assist students with coding questions rather than providing highly advanced features, hence opting for a local model is a more suitable choice for our application.

Considerations	Local Model	OpenAI API
Privacy & Security	Data processed locally, ensuring privacy and security	Data sent to OpenAI servers, no guarantees for privacy and security
Cost	No cost	Pay-per-use model
Speed	Typically fast, no latency	Dependent on latency and API server load
Capabilities	Limited by model size & hardware	Latest and most capable models
Usage Limits	No rate limits	Subject to API usage
Availability	Always available	Requires internet and dependent on the availability of OpenAI servers

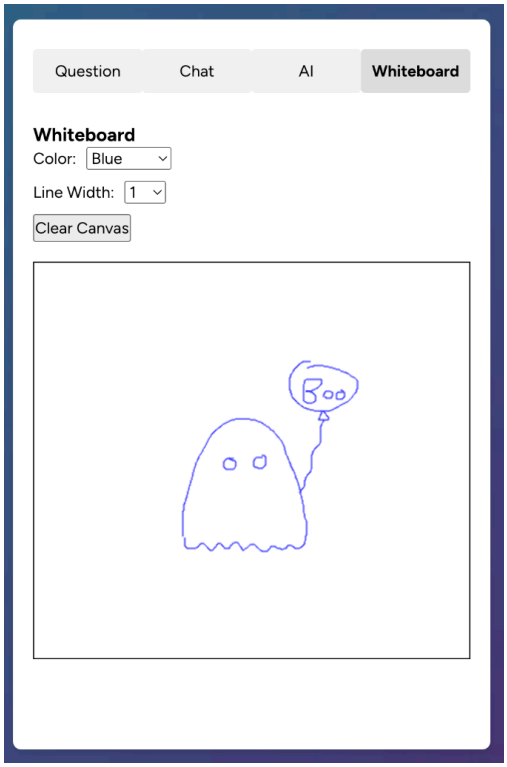
7.7.4 API Calls

Endpoint	Method	Body Fields	Description
/	GET	*	Health check endpoint.
/stream	POST	query: String question: json	Send a query and question to the language model. The response will be in a stream.

7.7.5 Responses for /stream endpoint

Status Code	Description	Response Example
200 OK	Response by the language model to the user query.	StreamingResponse
400 Bad Request	User query and/or question are not in the request body.	{"message": "Query and question must be provided in body."}
500 Internal Server Error	Error occurs while waiting for the response from the language model.	{ "error": <error_message>, "details": <error_details> }

7.8 (N13) Whiteboard



7.8.1 Core Functionalities and Components

This is a collaborative whiteboard application where users will be able to write, edit, and visualise code on a shared digital whiteboard. This allows users to simplify and communicate complex ideas using drawings, making it ideal for collaborative tasks like brainstorming, teaching, and pair programming. This application is built with React for the frontend and a Socket.IO-based backend for real-time communication.

Component	Description
Drawing Operations	- startDrawing: Initiates drawing and emits start event - finishDrawing: Finalizes drawing and emits end event - draw: Handles drawing on canvas
Socket Events	- getDrawing: Fetches initial drawing data - startDrawing: Emits drawing start event - drawing: Emits drawing updates - endDrawing: Finalises drawing session - clearWhiteboard: Clears canvas for all users
Canvas Features	- Adjustable width and height - Responsive design - Drawing with rounded line caps

8 Project Management

8.1 Development Methodology

Our team adhered to the Scrum framework, conducting weekly sprints every Sunday or Monday. Usually on Sunday, this is where we share about completed tasks, pending work, challenges encountered, and conduct final checking before submitting the required documents for milestones 2, 4, 5. On Monday, this is where we plan the task we would each work on for the week until our next meeting. Tasks are assigned based on each member's preferences ensuring that everyone gets the opportunity to work on tasks we are interested in.

8.2 Project/Milestones Timeline, Individual Contribution

For a full view of the gantt chart, you can refer to this Google Sheets: CS3219 Gantt Chart
Below, we included gantt chart screenshots of every milestone.

1	**[Milestone 1]** **Product backlog and mockup**					
1.1	UI Mockup	Andre Eunice	9/2/24	9/14/24	12	Completed
1.2	Functional, Non-functional Requirements and Nice to haves	Ashley Rayson Shi Kang	9/2/24	9/12/24	10	Completed
1.3	Submission	Andre Ashley Eunice Rayson Shi Kang	9/13/24	9/16/24	3	Completed

2	**[Milestone 2] SPA + REST API development**					
2.1	**Design Single Page Application (SPA) for Question Repository**					
2.1.1	Develop frontend for CRUD operations (Question Id, Title, Description, Category, Complexity)	Eunice, Rayson	16/9/2024	22/9/2024	6	Completed
2.1.2	Implement table view for questions (show Title, optionally Complexity/Category)	Eunice, Rayson	16/9/2024	22/9/2024	6	Completed
2.1.3	Add click-to-view description functionality	Eunice, Rayson	16/9/2024	22/9/2024	6	Completed
2.1.4	Style the page using CSS for usability and organization	Eunice, Rayson	16/9/2024	22/9/2024	6	Completed
2.2	**Create REST API for Backend (NoSQL Database)**					
2.2.1	Design REST API endpoints for CRUD operations	Ashley, Shi Kang	16/9/2024	22/9/2024	6	Completed
2.2.2	Set up communication between frontend and backend	Andre, Ashley, Shi Kang	16/9/2024	22/9/2024	6	Completed
2.2.3	Split question topic by commas, store as array in databse	Ashley	17/10/2024	17/10/2024	0	Completed
2.2.4	Implement functionality for Add, Edit, Delete operations to be updated to mongoDB database	Ashley	16/9/2024	22/9/2024	6	Completed
2.2.5	Error handing on mongoDB database for duplicate names, empty fields, etc.	Andre, Shi Kang	16/9/2024	22/9/2024	6	Completed
2.3	**Test Application**					
2.3.1	Implement basic error handling (e.g., duplicate check)	Andre	23/9/2024	28/9/2024	5	Completed
2.4	**Prepare Deliverables**					
2.4.1	Take 1-3 screenshots of SPA	Andre, Ashley, Eunice, Rayson, Shi Kang	29/9/2024	29/9/2024	0	Completed
2.4.2	Write instructions for setting up and running the code locally	Andre, Ashley, Eunice, Rayson, Shi Kang	29/9/2024	29/9/2024	0	Completed
2.4.3	Submission	Andre, Ashley, Eunice, Rayson, Shi Kang	16/9/2024	29/9/2024	13	Completed
3	**[Milestone 3] Containerization**					
3.1	**Set up User Service**					
3.1.1	Implement given user-service	Ashley	10/2/24	10/2/24	0	Completed
3.1.2	Containerize user-service (and question-service)	Eunice	10/2/24	10/2/24	0	Completed
3.2	**Documentation**					
3.2.1	Write instructions for setting up the repository locally, Document commands for spinning up the services using Docker	Andre, Ashley, Eunice, Rayson, Shi Kang	10/2/24	10/6/24	4	Completed
3.2.2	Merge all code to the master branch and tag the commit as "Milestone-D3"	Andre, Ashley, Eunice, Rayson, Shi Kang	10/6/24	10/6/24	0	Completed
3.3	**Final Submission and Review**					
3.3.1	Submit PDF with setup instructions and Docker commands	Andre, Ashley, Eunice, Rayson, Shi Kang	10/6/24	10/6/24	0	Completed
3.3.2	Submission	Andre, Ashley, Eunice, Rayson, Shi Kang	9/30/24	10/7/24	7	Completed

4	**[Milestone 4 Prereq] User service**					
4.1	**Frontend for user-service**					
4.1.1	Log In, Sign Up, Forget password (email)	Eunice	06/10/2024	06/10/2024	0	Completed
4.1.2	Manage profile (Change username, email, password)	Eunice	13/10/2024	13/10/2024	0	Completed
4.1.3	Forget password (token, new password)	Eunice	21/10/2024	21/10/2024	0	Completed
4.1.3	Password restrictions	Ashley	26/10/2024	26/10/2024	0	Completed
4.1.3	Calendar UI	Ashley Eunice	06/10/2024	06/10/2024	0	Completed
4.2	**Backend for user-service**					
4.2.1	Link user login, account creation, manage profile to user-service	Eunice	06/10/2024	06/10/2024	0	Completed
4.2.2	Cookies 🍪 and auto login	Eunice Ashley	17/10/2024	17/10/2024	0	Completed
4.2.3	Implement deactivated account logic in given user-service, adding an extra isActive field to the user schema	Shi Kang	17/10/2024	17/10/2024	0	Completed
4.2.4	Edit user service schema for user online days, make user online days linked to account instead of local storage	Shi Kang Ashley	18/10/2024	18/10/2024	0	Completed
4.2.5	Link user online days from frontend to backend	Eunice Ashley	19/10/2024	28/10/2024	9	Completed
4.2.6	Edit user service schema for user question done	Shi Kang	18/10/2024	18/10/2024	0	Completed
4.2.7	Create backend api for forget password to send reset password email	Eunice	19/10/2024	19/10/2024	0	Completed
4.2.8	Create backend api for token confirmation and reset password	Eunice	21/10/2024	21/10/2024	0	Completed
4.2.8	Create backend api for password verification (When user wants to change password)	Eunice	27/10/2024	27/10/2024	0	Completed
4	**[Milestone 4] Message queues and matching service**					
4.1	**User Initiation**					
4.1.1	Create a dashboard landing page where users can start a new session	Eunice	12/10/2024	14/10/2024	2	Completed
4.1.2	Create new session page where user is able to select difficulty and topic and press start matching	Rayson	12/10/2024	14/10/2024	2	Completed
4.1.3	Filter the question topic to only show the ones available in the database. Allow multiple topics to be selected.	Rayson	18/10/2024	18/10/2024	0	Completed
4.1.4	Allow multiple topics to be selected.	Rayson	18/10/2024	18/10/2024	0	Completed
4.1.5	Install and configure RabbitMQ on the backend. (running on port http://localhost:15672/ username: guest password: guest)	Ashley	11/10/2024	12/10/2024	1	Completed
4.1.6	Implement a function that triggers on the "Start Matching" button click (e.g., /startMatch). Serialize the user's details (userId, difficulty, topic) into a JSON payload and add user into rabbit queue	Andre	14/10/2024	17/10/2024	3	Completed
4.1.7	Call the add user to queue api so that frontend can connect to rabbit MQ (via an intermediary backend API).	Ashley	18/10/2024	19/10/2024	1	Completed
4.2	**Queues**					
4.2.1	Define and initalize a request exchange A (fanout) in rabbitMQ to listen for matching requests.	Shi Kang	14/10/2024	17/10/2024	3	Completed
4.2.2	Define and initalize a response exchange B (topic) in rabbitMQ to return matching results to exchange D.	Shi Kang	14/10/2024	17/10/2024	3	Completed
4.2.3	Set up request exchange C (fanout) for every client matching request to send to matching requests to exchange A for matching.	Shi Kang	14/10/2024	17/10/2024	3	Completed
4.2.4	Set up response exchange D (topic) for every client matching request to receive matching result.	Shi Kang	14/10/2024	17/10/2024	3	Completed

4.3	**Matching Check (matching-service)**					
4.3.1	Create matching-service and connect to mongoDB database	Ashley, Shi Kang	09/10/2024	12/10/2024	3	Completed
4.3.2	Link matching-service so that it listens for incoming match requests from rabbit's matchingQueue.	Shi Kang	14/10/2024	17/10/2024	3	Completed
4.3.3	Implement a function in this service to check the queue for other users with the same difficulty and topic.	Ashley, Andre	11/10/2024	17/10/2024	6	Completed
4.3.4	Add prioritization logic (e.g., match users who've waited longer first/difficulty/topic).	Ashley, Andre	11/10/2024	17/10/2024	6	Completed
4.3.5	When a match is found, package the details (user1Id, user2Id, difficulty, topic) into a new message and send to matching-service.	Ashley, Shi Kang	11/10/2024	17/10/2024	6	Completed
4.3.6	When a match is found, remove user from queue and notify user.	Ashley, Shi Kang	11/10/2024	19/10/2024	8	Completed
4.4	**Storing Match Data**					
4.4.2	Implement logic to add, store, delete the match details in database	Ashley	11/10/2024	12/10/2024	1	Completed
4.4.3	Create an API endpoint (e.g., /getMatchById) that checks if the user has been matched.	Ashley	11/10/2024	12/10/2024	1	Completed
4.5	**User Feedback**					
4.5.1	Use a timer or status update while the user waits.	Eunice	13/10/2024	13/10/2024	0	Completed
4.6	**Room Creation**					
4.6.1	Timer page shows ui change to reflect match success, remove timer and buttons	Eunice	13/10/2024	13/10/2024	0	Completed
4.6.2	When a match is confirmed (via API response), trigger the frontend to transition to match success page.	Ashley	19/10/2024	19/10/2024	0	Completed
4.7	**Handling No match / Cancellation / Rematch**					
4.7.1	Set a fixed wait time (e.g., 30 seconds) for users to find a match. If no match is found, show options for rematch or cancel.	Eunice	13/10/2024	13/10/2024	0	Completed
4.7.2	Implement fixed wait time (e.g., 30 seconds) for users to find a match, remove user from queue after 30 seconds and handle as "no match found"	Ashley, Shi Kang	11/10/2024	17/10/2024	6	Completed
4.7.3	Implement a "Rematch" logic that triggers the same logic as "Start Matching" to resubmit the user to the queue.	Andre	14/10/2024	17/10/2024	3	Completed
4.7.4	Link rematch logic to the button	Ashley	19/10/2024	19/10/2024	0	Completed
4.7.5	Implement a "Cancel" button to allow users to stop the match request while waiting.	Eunice	13/10/2024	13/10/2024	0	Completed
4.7.6	On clicking "Cancel", send a request to the backend to remove the user from the queue (e.g., /cancelMatchRequest API).	Andre	14/10/2024	17/10/2024	3	Completed
4.7.7	Create an API to handle cancel requests by removing the user from the RabbitMQ queue	Ashley	19/10/2024	19/10/2024	0	Completed
4.8	**Containerize Matching Service**					
4.8.1	Create Dockerfile to containerize the matching service	Ashley	11/10/2024	12/10/2024	1	Completed
4.8.2	Ensure frontend and all services (user service, question service, matching service) are properly packaged and managed with Docker Compose.	Eunice	12/10/2024	12/10/2024	0	Completed
4.9	**Restrict Access to Question Service**					
4.9.1	Only authenticated users are able to access the read (get) the questions	Shi Kang	07/10/2024	09/10/2024	2	Completed
4.9.2	Only admin users are able to access the create, update and delete (post, put, delete) questions	Shi Kang	07/10/2024	09/10/2024	2	Completed

4.1 i cant make this 4.10	**Restrict Access to Frontend Pages**					
4.10.1	Stores accesstoken and userid upon login to ensure access to inner pages only for active, logged-in users. Handles expired tokens	Eunice	08/10/2024	08/10/2024	0	Completed
4.10.3	Only admin users can have access to the manage question tab & page	Rayson	12/10/2024	14/10/2024	2	Completed
4.11	**Submission**					
4.11.1	Submit a PDF with the following: Screenshot of frontend with timer and feedback messages Log messages showing the status of the queue	Andre Ashley Eunice Rayson Shi Kang	20/10/2024	20/10/2024	0	Completed
4.11.2	Merge code to the master branch, tag as "Milestone-D4"	Andre Ashley Eunice Rayson Shi Kang	20/10/2024	20/10/2024	0	Completed
4.11.3	Sign up for a live demonstration slot	Andre Ashley Eunice Rayson Shi Kang	20/10/2024	20/10/2024	0	Completed
4.11.4	Submission	Andre	07/10/2024	21/10/2024	14	Completed
5	**[Milestone 5 & 6] Collaboration Service and Nice-to-Haves**					
5.1	**Room Creation**					
5.1.1	Matching service creates a sessionId for collaboration service	Shi Kang	20/10/2024	20/10/2024	0	Completed
5.1.2	When a match is confirmed (via API response), trigger the frontend to transition to match success page, then navigate to shared collaboration page.	Eunice	17/10/2024	17/10/2024	0	Completed
5.1.3	Matching Service creates session data in Mongodb database.	Andre	30/10/2024	30/10/2024	0	Completed
5.2	**CodePanel (right side)**					
5.2.1	Text field to input code (start with basic starting code) and dropdown to change language (Python, Java, Javascript)	Eunice	24/10/2024	24/10/2024	0	Completed
5.2.2	When user change language, previous code should not reset	Andre	28/10/2024	30/10/2024	2	Completed
5.2.3	Restore code when user rejoins/disconnect using code stored in database	Rayson Andre	27/10/2024	30/10/2024	3	Completed
5.2.4	[N4] Configure the editor to support multi-language syntax highlighting and basic code editing features.	Ashley Eunice	25/10/2024	25/10/2024	0	Completed
5.2.5	[N3] Run code functionality and Output	Eunice	24/10/2024	25/10/2024	1	Completed
5.2.6	[N3] Containerize each run code in its own container	Eunice	04/11/2024	04/11/2024	0	Completed
5.2.7	[N3] Run code user testing (please try to break the run code function, if you can >:))	Andre Ashley Eunice Rayson Shi Kang	25/10/2024	30/10/2024	5	Completed
5.2.8	Reset Answer Button & Submit Button	Shi Kang	28/10/2024	30/10/2024	2	Completed
5.2.9	Question test cases and test cases grader	Shi Kang	28/10/2024	30/10/2024	2	Completed
5.2.10	Test Cases Frontend Tab	Shi Kang Rayson	30/10/2024	01/11/2024	1	Completed

5.3	**Tabs (left side)**					
5.3.1	Design the tabs to switch between Question, Chat, AI and Whiteboard	Eunice	24/10/2024	24/10/2024	0	Completed
5.3.2	[Question] Display question that user selected in the question tab	Rayson	24/10/2024	24/10/2024	0	Completed
5.3.3	[Question] Implement backend route in question service to get specific question	Shi Kang, Rayson	18/10/2024	18/10/2024	0	Completed
5.3.4	[Chat] [N1] Frontend	Rayson	02/11/2024	02/11/2024	0	Completed
5.3.5	[Chat] [N1] Link Frontend to Backend	Ashley, Rayson	02/11/2024	02/11/2024	0	Completed
5.3.6	[Chat] [N1] Chat service running on port 8085 and connected to mongoDB	Ashley	29/10/2024	30/10/2024	1	Completed
5.3.7	[Chat] [N1] Routes to create a chat, retrieve chat by session ID, and add messages to existing chats	Ashley	29/10/2024	30/10/2024	1	Completed
5.3.8	[Chat] [N1] Design the chat-service schema (messageSchema and chatSchema)	Ashley	29/10/2024	30/10/2024	1	Completed
5.3.9	[Chat] [N1] Containerize chat-service	Ashley	29/10/2024	30/10/2024	1	Completed
5.3.10	[Chat] [N1] Chat User Testing	Andre, Ashley, Eunice, Rayson, Shi Kang	02/11/2024	03/11/2024	1	Completed
5.3.11	[AI] [N6] Ai-service backend	Shi Kang	22/10/2024	29/10/2024	7	Completed
5.3.12	[AI] [N6] Chat with Raesa frontend	Shi Kang, Eunice	24/10/2024	29/10/2024	5	Completed
5.3.13	[AI] [N6] AI User testing (Try to break Raesa before she breaks you with her 1 minute 40 seconds response time)	Andre, Ashley, Eunice, Rayson, Shi Kang	25/10/2024	30/10/2024	5	Completed
5.3.14	[Whiteboard] Frontend	Eunice	26/10/2024	26/10/2024	0	Completed
5.3.15	[Whiteboard] Backend using sockets	Eunice, Shi Kang	26/10/2024	30/10/2024	4	Completed
5.3.16	[Whiteboard] Save and load whiteboard data in MongoDB	Shi Kang	04/10/2024	05/10/2024	1	Completed
5.3.17	[Whiteboard] User Testing (Mercy please she's just a baby)	Andre, Ashley, Eunice, Rayson, Shi Kang	26/10/2024	03/11/2024	7	Completed
5.4	**Disconnect/End Session**					
5.4.1	Add isMatched and matchData to user-service schema, include POST api to edit the two fields (**updateUserMatchedStatus** in user-service/controller/ user-controller.js)	Rayson	27/10/2024	27/10/2024	0	Completed
5.4.2	Update dashboard to use hasActiveSession and matchedUsername (this uses localstorage)	Rayson	27/10/2024	27/10/2024	0	Completed
5.4.3	Allow user to go directly to collaboration page from dashboard, display "Current meeting with ____"	Rayson	27/10/2024	27/10/2024	0	Completed
5.4.4	Update user matched status when user ends session	Rayson	27/10/2024	27/10/2024	0	Completed
5.4.5	Frontend to show that your partner wanted to end the session, do you also want to end as well?	Ashley	06/11/2024	06/11/2024	0	Completed
5.4.6	Delete session data when session ends.	Andre	30/10/2024	30/10/2024	0	Completed
5.4.7	Summary page	Eunice	27/10/2024	27/10/2024	0	Completed
5.5	**Collaboration Service (Backend)**					
5.5.1	Create and Setup collaboration service to run on port 8084	Ashley	24/10/2024	25/10/2024	1	Completed
5.5.2	Connect collaboration service to mongoDB	Andre	28/10/2024	30/10/2024	2	Completed
5.5.3	Develop a RESTful API to manage user sessions, including features to create, retrieve, update, and delete sessions with error handling and data validation.	Andre	28/10/2024	30/10/2024	2	Completed
5.5.4	Design the collaboration-service schema	Andre	28/10/2024	30/10/2024	2	Completed
5.5.5	User Testing (Start and End sesion correctly updates session database)	Andre, Ashley, Eunice, Rayson, Shi Kang	01/11/2024	01/11/2024	0	Completed

5.6	**WebSocket (In collaboration-service)**					
5.6.1	Set up the WebSocket server (using Node.js with Socket.IO) to manage real-time communication. (**join, disconnect** in collaboration-service /index.js)	Ashley	24/10/2024	25/10/2024	1	Completed
5.6.2	Establish a WebSocket connection to listen for changes in the code editor (**codeChange, languageChange** in collaboration-service/index.js)	Ashley Andre	24/10/2024	30/10/2024	6	Completed
5.6.3	Handle real-time updates within the editor, ensuring smooth synchronization of code between both users (**codeUpdate, languageUpdate** in components/collaboration/ codepanel.jsx)	Ashley Andre	24/10/2024	30/10/2024	6	Completed
5.6.4	Use sockets to know when a user has ended session (**endSession** in collaboration-service/index.js)	Rayson	26/10/2024	26/10/2024	0	Completed
5.6.5	Use sockets to update partner that the other person ended session (**partnerLeft** in components/collaboration/ codepanel.jsx)	Rayson	26/10/2024	26/10/2024	0	Completed
5.6.6	Use sockets to know when a user is drawing (**startDrawing, endDrawing** in collaboration-service/index.js)	Eunice	26/10/2024	26/10/2024	0	Completed
5.6.7	Use sockets to update the whiteboard for the other person (**beginDrawing, drawingUpdate, finishDrawing, clearWhiteboard** in components/collaboration/Whitebo ard.jsx)	Eunice	26/10/2024	26/10/2024	0	Completed
5.6.8	Update whiteboard sockets to use array to store the strokes, so that even if both users aren't on the wb tab, it will still sync	Shi Kang	31/10/2024	31/10/2024	0	Completed
5.6.9	Use sockets for chat (**sendMessage** and **receiveMessage**)	Rayson	02/11/2024	02/11/2024	0	Completed
5.6.10	Use sockets on backend for end session (**endSession** and **responseEndSession**)	Ashley	06/11/2024	06/11/2024	0	Completed
5.6.11	Use sockets on frontend for end session (**confirmEndSession**, **endSessionBoth** and **userContinues**)	Ashley	06/11/2024	06/11/2024	0	Completed
5.7	**[N2] History**					
5.7.1	When user end session, save session data to user-service schema	Andre	30/10/2024	31/10/2024	0	Completed
5.7.2	Link user history frontend to backend	Andre	30/10/2024	31/10/2024	0	Completed
5.7.3	Frontend for user history	Andre Eunice	30/10/2024	31/10/2024	0	Completed
5.7.4	Search/filter for history	Andre Ashley	06/11/2024	06/11/2024	0	Completed
5.7.5	User Testing	Andre Ashley Eunice Rayson Shi Kang	01/11/2024	01/11/2024	0	Completed
5.8	**Docker**					
5.8.1	Write a Dockerfile to containerize the collaboration service	Ashley	24/10/2024	25/10/2024	1	Completed
5.8.2	Add the collaboration service to the docker-compose.yml file to ensure it can be launched alongside other services (user, matching, question)	Ashley	24/10/2024	25/10/2024	1	Completed
5.8.2	Use single .env file in root directory	Eunice	09/11/2024	09/11/2024	0	Completed

7	[Milestone 7] Project Report					
7.0	**Pre-Introduction**					
7.0.1	Table of Contents	Andre Ashley Eunice Rayson Shi Kang	29/10/2024	29/10/2024	0	Completed
7.0.2	Acknowledgment	Andre Ashley Eunice Rayson Shi Kang	30/10/2024	09/11/2024	9	Completed
7.1	**Introduction**					
7.1.1	Project Background/Purpose	Eunice	30/10/2024	09/11/2024	9	Completed
7.1.2	Intended Audience and Reading Suggestions	Ashley Shi Kang	08/11/2024	09/11/2024	1	Completed
7.1.3	Steps on to run locally	Eunice	30/10/2024	09/11/2024	9	Completed
7.2	**Functional Requirements**					
7.2.1	FR	Andre Ashley Eunice Rayson Shi Kang	26/10/2024	10/11/2024	14	Completed
7.3	**Non-functional Requirements**					
7.3.1	NFR	Andre Ashley Eunice Rayson Shi Kang	04/11/2024	10/11/2024	6	Completed
7.4	**Architecture**					
7.4.1	Architecture Overview	Ashley Eunice	08/11/2024	10/11/2024	2	Completed
7.4.2	Architecture Design	Ashley	08/11/2024	10/11/2024	2	Completed
7.4.3	Technology Stack	Ashley	08/11/2024	08/11/2024	0	Completed
7.5	**Design Patterns (What it is + examples in code)**					
7.5.1	Overview	Rayson	10/11/2024	10/11/2024	0	Completed
7.5.2	Database-per-Service Pattern	Rayson	09/11/2024	10/11/2024	1	Completed
7.5.3	Observer Pattern	Rayson	09/11/2024	10/11/2024	1	Completed
7.5.4	Publish-Subscribe Pattern	Rayson	09/11/2024	10/11/2024	1	Completed
7.5.5	Model-View-Controller Pattern	Rayson	09/11/2024	10/11/2024	1	Completed
7.6	**Frontend**					
7.6.1	React/Vite Framework choice	Eunice	29/10/2024	30/10/2024	1	Completed
7.6.2	Frontend Code Structure (Architecture Diagram)	Eunice	29/10/2024	09/11/2024	10	Completed
7.6.3	Design and Implementation Decisions	Eunice	09/11/2024	09/11/2024	0	Completed
7.6.4	Pages Screenshots	Eunice	30/10/2024	09/11/2024	9	Completed
7.7	**Backend (Microservices)**					
7.7.1	Overview	Eunice	29/10/2024	29/10/2024	0	Completed
7.7.2	User Service	Eunice	31/10/2024	09/11/2024	9	Completed
7.7.3	Question Service	Ashley	04/11/2024	10/11/2024	6	Completed
7.7.4	Matching Service	Ashley Shi Kang Andre	04/11/2024	10/11/2024	6	Completed
7.7.5	Collaboration Service	Ashley Andre Shi Kang	04/11/2024	10/11/2024	6	Completed
7.8	**Nice to Haves (lets bring all the nice to haves here)**					
7.8.1	N1 Communication	Rayson	07/11/2024	10/11/2024	3	Completed
7.8.2	N2 History	Andre	07/11/2024	10/11/2024	3	Completed
7.8.3	N3 Code Execution Service	Eunice	29/10/2024	09/11/2024	10	Completed
7.8.4	N4 Enhanced code editor with syntax highlighting	Ashley	04/11/2024	04/11/2024	0	Completed
7.8.5	N6 AI Service	Shi Kang	03/11/2024	08/11/2024	5	Completed
7.8.6	N13 Whiteboard	Eunice	04/11/2024	04/11/2024	0	Completed
7.9	**Project Management**					
7.9.1	Development Methodology (Describe the approach used, such as Agile, Scrum, or Waterfall.)	Ashley	08/11/2024	08/11/2024	0	Completed
7.9.2	Project Timeline, Resource Allocation, Milestones and Deliverables (use gantt chart)	Andre Ashley Eunice Rayson Shi Kang	29/10/2024	10/11/2024	11	Completed
7.10	**Conclusion**					
7.10.1	Reflections and Learning Points	Andre Ashley Eunice Rayson Shi Kang	09/11/2024	10/11/2024	1	Completed
7.10.2	Suggestion for Improvement and Enhancement	Andre Ashley Eunice Rayson Shi Kang	07/11/2024	09/11/2024	2	Completed
7.10.3	Reading out the entire project report on a 3 hour zoom call	Andre	10/11/2024	10/11/2024	0	Completed

9 Conclusion

9.1 Reflections and Learning Points

Human	Sentence
Andre	Personally, I felt that this project presented many new challenges and experiences, both in developing software using a new language and working with a team with working styles I was not accustomed to. Each member on my team was much more hardworking and completed their work earlier than I had expected on most milestones. I usually prioritised working on other modules first to ensure that I could work on project tasks without having other issues on my mind. This often led to situations where most of the work would be done before I had begun working on the project for the week. To me, this experience emphasised the importance of communication. Communication was also pivotal in ensuring that all of us were on the same page for each function implemented as each of us had to work on different portions of the code. Overall, I have gained a greater appreciation of software development beyond just the coding aspects.
Ashley	I started working on the D7 project report at milestone 5. I think it would have been more beneficial if I started working on the project report earlier, around milestone 2. I could have documented the new features and their implementation details as they were developed. This would have provided a reference for my team members, helping them understand the changes made and how the features functioned, minimise communication gaps and the need for private clarifications.
Eunice	Even though we may have differing opinions on how things should be done, one common objective we had was that we all wanted to do well, and that to me is the most important. Thank you all for putting in so much effort into the group project, reading and checking each other's work. It really feels like I'm working in a team rather than alone. Thank you for the open communication, I appreciate the directness, very cool
Rayson	The project taught me a lot about effective teamwork and technical skills. We worked through complex features and improved our process by splitting tasks based on our strengths. Building a functional web application while learning new technologies like React was challenging but rewarding. Despite some challenges, particularly with collaboration features, working and communicating together helped us complete the project.
Shi Kang	I think that the team’s overall progress and completion for the project is great. We have developed a MVP for a collaboration coding platform. Through this project, I’ve learnt many new frameworks and technologies such as RabbitMQ. Most importantly, I had a chance to experience working together on a web application in a team setting, which will prove valuable in my future endeavours. One thing I thought we could improve on is the creation of test cases for the project. As it is not part of the requirement and due to time constraints, we did not have the capacity to create them. I would make sure to include them next time I work on a software project.

9.2 Future Improvements and Enhancements in consideration

1. Matching service could be enhanced to match users not only based on category, difficulty but also language and experience level. This would ensure more relevant and effective pairings, improving collaboration and the overall user experience.

2. Chat notification that alerts users when they receive a new message. We realise that users might be busy solving the questions and might not toggle to the chat tab frequently. Thus by having a pin or alert icon whenever there is any new incoming message, it can prompt users to check the new messages, ensuring that communication remains timely and that users are not missing important updates.

3. A private session feature for users to specifically match with other users that they know. The current matching implementation is based on the user preferences, however users might want to tackle programming questions with someone they know. This feature is also particularly useful in a technical interview where the interviewer can observe how the interviewee works through the various programming problems online.

4. Adding markdown support in chat messages would enable users to format text with elements like inline code, bold, and headings, making messages clearer and more expressive. Markdown would be particularly beneficial for technical discussions, allowing for easy sharing of code snippets and structured information, improving readability and user experience.

5. Implementing end-to-end encryption for chat messages would enhance the security and privacy of user conversations. By ensuring messages are encrypted during transmission and storage, sensitive information is better protected from authorised access. This improvement would be essential for maintaining user trust, especially in collaborative environments where private information may be shared.

10 The End

We, the undersigned, declare that:

1. The work submitted as part of this project is our own and has been done in collaboration with the members of our group and no external parties.
2. We have not used or copied any other person's work without proper acknowledgment.
3. Where we have consulted the work of others, we have cited the source in the text and included the appropriate references.
4. We understand that plagiarism is a serious academic offense and may result in penalties, including failing the project or course.
    1. We have read the NUS plagiarism policy and the Usage of Generative AI.

Group Member Signatures:

Full Name (as in Edu Rec)	Signature	Date
Andre Sim Jing Xiang		1st November 2024
Ashley Goh Junting		1st November 2024
Eunice Koh Shu Ning		1st November 2024
Rayson Chia Hao Zhe		1st November 2024
Yong Shi Kang		1st November 2024

11 Appendix

11.1 .env file contents

```
# MongoDB credentials (shared across services)
MONGODB_USERNAME=admin
MONGODB_PASSWORD=g26password
MONGODB_ENDPOINT=peerprep.xvavl.mongodb.net

# Matching Service-specific MongoDB database
MATCHING_DB=peerprepMatchingServiceDB

# Question Service-specific MongoDB database
QUESTION_DB=peerprepQuestionServiceDB

# User Service-specific MongoDB database
USER_DB=peerprepUserServiceDB

# Chat Service-specific MongoDB database
CHAT_DB=peerprepChatServiceDB

# Collab Service-specific MongoDB database
COLLAB_DB=peerprepCollaborationServiceDB

# RabbitMQ Configuration
RABBITMQ_URI=amqp://guest:guest@rabbitmq:5672
RABBITMQ_REQ_CH=request
RABBITMQ_RES_CH=response
RABBITMQ_DEFAULT_USER=guest
RABBITMQ_DEFAULT_PASS=guest

# JWT Secret for User Service
JWT_SECRET=your-jwt-secret

# Email credentials (for forget password)
EMAIL_USER=raesa2648@gmail.com
EMAIL_PASS=jhyf dgub whti bjwy
```