NUS
National University
of Singapore

PeerPrep Report

CS3219 Project

Group 18

Name	Matric No.
Lee Sen Wei	A0245789H
Bernice Priscilla Toh	A0237060N
Adriel Soh	A0233032Y
Chung Yunseong	A0219791U
Chew Yew Keng	A0233664A

1. Introduction

1.1 Background

In a world where coding interview questions have become the gold standard for evaluating potential employees and interns, there's an ever-increasing demand for platforms that provide coding practice. While platforms like LeetCode offer a space to practice coding questions, our application stands out by offering a unique set of features that go beyond the norm. We aim to achieve 3 things:

A Collaborative Coding Space:
We recognize that coding is rarely a solitary endeavor in the professional world. Our platform creates a virtual space where users can collaborate on a coding problem with peers, just like they would in a real job context. This collaborative approach allows users to tackle challenges together, brainstorm ideas, and learn from each other. It fosters a sense of camaraderie, making the learning process not only effective but enjoyable.

Real-Time Chat for Clear Communication:
Effective communication is a fundamental skill for any skilled engineer. Our platform goes beyond simply providing coding challenges; it includes a real-time chat feature that allows users to practice articulating their ideas clearly and concisely. When users get stuck, they're not alone; they can reach out to their peers for help, making the learning journey less intimidating and more supportive.

Matching with other users of the same proficiency level:

Many newcomers to coding platforms like LeetCode are often overwhelmed by the complexity of the questions. They may not know where to begin or how to approach problems. Our platform addresses this challenge by effortlessly connecting them with a peer at your proficiency level, so they never feel alone while unraveling coding challenges. Together, they can conquer complexities, support one another, and elevate their coding skills.

In a nutshell, while traditional coding platforms focus solely on questions and answers, our platform goes the extra mile to provide an immersive and collaborative learning experience. We empower users to not only master coding but also become effective communicators and team players—qualities that are highly prized in the tech industry. Join us on this exciting journey where learning is not a solitary path but a collaborative adventure full of growth and fun.

1.2 Features

1. Collaborate real time with other users on a live-code editor with syntax-highlighting, code execution, and even a whiteboard to draw out your ideas.
2. Match with other users based on question difficulty and your proficiency level.
3. Live Chat with other users.
4. Authentication with email and password, *Google OAuth*.
5. See list of questions that you have attempted/submitted, and view or edit the saved code and chat history from the previous attempts.
6. See the details of the questions that you have attempted so you can collate your learning points.
7. Customise personal profile (IP).

1.3 Application Start-Up

Deployed version:

1. To simulate 2 sessions, open the app through the following link: http://34.87.154.192:4173/#/ on 2 different incognito Chrome Tabs.

Local Version:

Software Required
NodeJS
Docker

Ensure that *Docker* is running on your machine. Ensure ports 4173, 3000, 4000, 4001, 4002, 5001, 6001, 7001, 15672 and 5672 are available on your machine.

Installation

1. Navigate into the `ay2324s1-course-assessment-g18` directory from the downloaded release.
   1.1. `cd ay2324s1-course-assessment-g18`
2. From the directory, execute the following commands and rename dockerfile.txt to dockerfile after deleting the dockerfile in web-admin-dashboard
   2.1. `cd ./question-service`
   ```
   docker build -t rgonslayer/peerprep-question:1.0 .
   ```
   2.2. `cd ../user-service`

```
docker build -t rgonslayer/peerprep-user:1.1 .
```

2.3.
```
cd ../auth-service
docker build -t rgonslayer/peerprep-auth:1.0 .
```

2.4.
```
cd ../history-service
docker build -t adrielsoh/peerprep-history:1.0 .
```

2.5.
```
cd ../chat-service
docker build -t adrielsoh/peerprep-chat:1.0 .
```

2.6.
```
cd ../upload-service
docker build -t rgonslayer/peerprep-upload:1.0 .
```

2.7.
```
cd ../matching-service
docker build -t rgonslayer/peerprep-matching:1.0 .
```

2.8.
```
cd ../web-admin-dashboard
docker build -t rgonslayer/peerprep:1.0 .
```

2.9.
```
cd ..
docker compose up
```

3. Visit localhost:4173 on 2 different incognito Chrome Tabs to simulate two sessions

2. Functional and Non-Functional Requirements

Functional Requirements	Priority
F1 User Service *Google Cloud SQL*	
F1.1 Users can edit their own profile details	High
F1.1.1 Users are able to change their username and password	High
F1.1.2 Validation of new password before updating user profile	Medium
F1.2 Admins can change user roles and carry out other CRUD operations on users	High
F1.2.1 Admin changing role on frontend should propagate the changes to backend database as well	High
F1.3 Users can delete their account	High
F1.3.1 Delete user account from backend database	High
F1.3.2 Delete all user related data from the database. E.g user history	High
Non-Functional Requirements	
NF1.1 Usability of User Service	

NF1.1.1 Sessions are persistent for user by using refresh tokens	Medium
NF1.1.2 Set refresh token validity to 30 days to force re-login by user	Medium
NF1.2 Performance	
NF1.2.1 Login and Logout response should be complete in less than 30 seconds	Medium
F2 Matching service	
F2.1 Service should allow users to select the preference of difficulty they want to attempt	High
F2.2 Users should be able to match with other users with similar difficulty levels and put them in the same room	High
F2.2.1 Matching service should be able to check multiple users preferences at once and assign them to each other if they match	High
F2.2.2 The difficulty of the question for the match should be at most the difficulty the user chose	High
F2.3 The system should provide the user with the means to leave a room once matched	High
F2.3.1 Provide confirmation dialogue when trying to exit to prevent accidental leaving of room	Medium
Non-Functional Requirements	
NF2.1 Performance	

NF2.1.1 Users should be matched within 30 seconds if there is a valid match	High
NF2.1.2 Optimize matching algorithm to reduce matching time	Low
NF2.2 Usability	
NF2.2.1 Users should be informed if no match is available if no match can be found within 30 seconds	High
NF2.2.2 Users should be prompted with the same preferences to retry finding a match after there is no match within 30 seconds.	Medium
F3 Question Service *MongoDB Atlas*	
F3.1 Questions need to be indexed in terms of difficulty (Low, Medium, High)	High
F3.1.2 Admins should be able to sort questions by difficulty level	High
F3.2 Questions need to be indexed in terms of category	Medium
F3.2.2 Admins should be able to see the details of the question when clicked	High
F3.3 Ability to store Images and diagrams for questions	Low
F3.3.1 Users must be able to view the related images and diagrams for a question	High
Non-Functional Requirements	

NF3.1 Availability of Questions	
NF3.2 Performance of questions repository	
NF3.2.1 Loading of the question details should not take longer than 5 seconds	High
NF3.2.2 Loading of the questions images and media should not take longer than 30 seconds	Low
NF3.2.3 Loading list of questions should not take longer than 5 seconds	Medium
NF3.3 Access control of questions	
NF3.3.1 Questions should only be allowed to be edited by admins	High
NF3.3.2 Questions should only be allowed to be added by admins	High
NF3.3.3 Questions should only be allowed to be deleted by admins	High
F4 Chat Service	
F4.1 User must be able to see the other user's messages in real time	High
F4.2 User needs to be able to see who sent which message	High
F4.2.1 Chat interface differentiates between messages sent by both parties by showing user with each message	High
Non-Functional Requirements	
NF4.1 Performance	

NF4.1.1 Message does not take more than 3 seconds to send	High
NF4.1.2 Messages should not take more than 3 seconds to appear on the other user's screen	High
NF4.2 Security	
NF4.2.1 Messages are encrypted in transit to ensure data security	Low
F5 Collaboration Service	
F5.1 User must be able to use a code editor to solve the question	High
F5.1.1 Code editor needs to be integrated into the web app	High
F5.1.2 Whiteboard needs to be integrated into the web app	High
F5.2 User must be able to execute code and retrieve the result	Medium
F5.2.1 Integrate interpreter/compiler to execute user's code	High
F5.2.2 Display execution output, errors and warnings to user	Medium
F5.3 User must be able to see the other user's edits in real time	High
F5.3.1 User must be able to see the drawings of other users in real time	High
F5.4 User must be able to benefit from real time code validation	Medium
F5.4.1 Syntax errors should be validated and informed to the user	High

F5.4.2 User must be able to execute the code editor for different programming languages (Top 5 - JavaScript, C, Java, Python, SQL)	Medium
Non-Functional Requirements	
NF5.1 Performance	
NF5.1.1 Code execution environment is secure and isolated to prevent security vulnerabilities.	Low
Other user's inputs should have a delay of no more than 5 seconds (Performance)	Medium
NF5.2 Security	
NF5.2.1 Code written in the editor shouldn't be accessible to users who are not related to the current session.	Low
F6 History Service *MongoDB Atlas*	
F6.1 User must be able to see a record of questions attempted	High
F6.1.1 User can view previous attempts by clicking on history page	High
F6.1.2 System should call backend to retrieve history data from database	High
F6.1.3 History page should display saved code and chat history from previous attempt	Medium
F6.1.4 If the same question is attempted multiple times, only store the latest snapshot even if the question was attempted with different users.	Medium

Non-Functional Requirements	
NF6.1 Security	
NF6.1.1 Users shouldn't be able to view other users' history	High
NF6.2 Usability	
NF6.2.1 History should be available for viewing by the user regardless of the time the question was attempted.	Medium
NF6.3 Performance	
NF 6.3.1 History page should not take more than 5 seconds to load	Medium
F7 Auth Service *MongoDB Atlas*	
F7.1 Users can create a new account with a username, email and password	High
F7.1.1 System should display clear error message for invalid username/password format	High
F7.1.2 System should provide error checking for user password to ensure user password contain at least one uppercase letter, one lowercase letter, one digit, and one special character	Medium
F7.1.3 System should provide error checking for duplicate users by email	High
F7.1.4 System should call to backend to create user in the database	High

F7.2 Users can sign into an existing account with an email and password	High
F7.2.1 System should call to backend to check if email exists in the user database	High
F7.2.2 System should verify password against stored hashed password	High
F7.2.4 System should remember users by default so that user does not have to log in again for subsequent visits	Low
F7.3 Users can log out of the platform	High
F7.3.1 System should prompt user with dialogue before user logs out to prevent accidental logout	Medium
F7.3.2 Generated access and refresh tokens for the user should be cleared upon logout.	High
Non-Functional Requirements	
NF7.1 Security	
NF7.1.1 Passwords should be salted and hashed	High
NF7.1.2 Use strong hashing algo for password hashing	High
NF 7.1.4 Accounts are secured with JWT authentication (Security)	High
NF 7.1.5 Enhanced security by using both refresh token and access token, by setting a short validity for the access token to prevent any unwanted actions due to compromised access tokens.	Medium

3. Technologies and Architecture

3.1 Tech Stack

Frontend	*ShadCN, React, Tailwind, Vite*
User Service	*Google Cloud Platform (GCP) SQL, Nest.JS*
Matching Service	*Socket.io*
Question Service	*MongoDB Nest.JS*
History Service	*MongoDB Nest.JS*
Collaboration Service	*WebRTC with CRDT data storage, YJS*
Chat Service	*Socket.io*
Auth Service	*MongoDB Nest.JS*
Upload Service	*GCP Buckets with Nest.JS*
Code Execution	*judge0*
Inter-Service communication	*RabbitMQ*
Containerization	*Docker*
Orchestration	*GCP Kubernetes Engine*
API Gateway	*GCP Kubernetes Engine Ingress Controller*

3.2 Architectural Decisions

Microservices vs Monolithic Architecture

While both microservices and monolithic architecture are feasible for building this application, we found the microservice architecture more effective for this project for the following reasons.

1. **Scalability**: Microservices allow different components of the application (like user authentication, problem-solving interface, database of questions, etc.) to scale independently. As user traffic grows, particularly during peak times like coding contest events, microservices can be scaled up to handle the increased load through the use of

orchestration such as *Kubernetes*, ensuring better performance and user experience.

2. **Flexibility in Technology Stack**: Each microservice can be built with the most suitable technology stack for its specific functionality. This flexibility enables the use of the best tools for different aspects, such as using a more efficient database management system for storing coding problems and user data, or a more robust real-time engine for coding interviews.
3. **Easier Maintenance and Debugging**: Microservices compartmentalize the application, making it easier to isolate, diagnose, and fix issues. For a complex platform with diverse functionalities, this can significantly reduce downtime and improve overall reliability.
4. **Enhanced Collaboration and Team Productivity**: Different team members can work on different microservices with minimal dependency on each other. This parallel development can speed up the overall development process, which is particularly beneficial in a context where quick adaptation to new programming languages or challenge types is essential. It is also easier to learn how a microservice works if needed compared to a monolithic service.
5. **Resilience**: In a microservices architecture, the failure of one service does not necessarily bring down the entire application. This resilience is crucial for a high-availability service like a coding challenge platform, where downtime can significantly impact user experience.
6. **Better Resource Utilization**: Microservices can be more efficient in terms of resource utilization. Resources can be allocated dynamically based on the demand of individual services, ensuring that the infrastructure is not under or over-utilized.

Although there are also tradeoffs in using a microservices architecture such as inter-service communication and complexity in deployment and management, the team felt that the benefits outweighed the tradeoffs in enabling a rapid development process especially given that we only spent about 10 weeks developing the application. Another factor for choosing this was because we felt that the application size was large enough that a monolithic architecture would become too cumbersome to work with.

Model-View-Controller(MVC) Architecture

The rationale behind choosing an MVC architecture is similar to microservices. It provides easier scalability and maintenance due to its clean separation of concerns which is well suited for a complex and dynamic website like ours with multiple interacting components.

1. **Separation of Concerns**: MVC architecture separates the application into three main components: Model (data), View (user interface), and Controller (business logic). This clear separation facilitates maintenance and scalability, as developers can focus on one aspect without impacting others.
2. **Ease of Development and Maintenance**: With responsibilities clearly divided, different developers can work on the model, view, or controller independently. This parallel development can speed up the process and make maintenance more manageable.
3. **Improved Code Reusability**: Components in MVC, especially the model and controller, can often be reused across different parts of the application, reducing the amount of

redundant code.

4. **Adaptability to Change**: MVC architecture makes it easier to change the user interface or business logic without reworking the entire application. This adaptability is crucial for a dynamic website where user interface trends and business requirements can change frequently.
5. **Support for Asynchronous Technique**: MVC architecture works well with *AJAX* (incorporated into *React.js*), which allows for asynchronous data loading. This can improve the user experience by reducing reloads and making the application more responsive.
6. **Better Performance and Scalability**: Since the view and business logic are separated, the server doesn't need to process data as heavily for each user interaction. This can lead to better performance and easier scalability as user load increases.
7. **Framework and Language Support**: Many popular web development frameworks (like *Nest.JS*) are built around the MVC pattern, offering robust support, extensive libraries, and community resources
8. **Simplifies Complex Applications**: For a website that involves complex functionalities like coding challenges, learning paths, and user dashboards, MVC helps in organizing the application into more manageable pieces.
9. **Consistency in Development**: MVC enforces a certain level of discipline and consistency in application development, which is beneficial in our team setting as multiple developers conform to the same standards.

4. Development process

4.1 User Stories

User Story ID	User Story
AS-1	As a user, I want to create a new account using my email and my own password.
AS-2	As a user, I want to log in to *PeerPrep* with my email and password.
AS-3	As a user, I want to log out of *PeerPrep* so that others are not able to use my account on the same device.
AS-4	As a user, I want to remain logged in until I log out.
US-1	As an admin, I want to be able to change user roles and perform CRUD operations on other users.
US-2	As a user, I want to delete my account so that my information is removed from the *PeerPrep* platform when I no longer wish to use it.
US-3	As a user, I want to be able to change my profile details such as password, username.
MS-1	As a user, I want to select the difficulty of questions I want to do so that I can practice questions that are suitable for my skill level.
MS-2	As a user, I want to match with another user who has chosen the same difficulty level so that we can learn from each other.
MS-3	As a user, I want to be matched with another user within 30s.
MS-4	As a user, I want to be given the option to try and rematch when the matching timer runs out.
MS-5	As a user, I want to leave the room after being matched when I am done with the question.
CS-1	As a user, I want to be able to chat with my peer that I have matched with.
CS-2	As a user, I want to see the messages with my peer in real-time.
CS-3	As a user, I want to differentiate between my messages and my peer’s messages.

COS-1	As a user, I want to be able to see the edits made on the code editor in real-time.
COS-2	As a user, I want a delay of less than 5 seconds when sending or receiving edits on the code editor.
COS-3	As a user, I want to be able to see the drawing made on the whiteboard in real-time
QS-1	As a user, I want to be issued with a question of the difficulty chosen.
QS-2	As an admin, I want to see a list of all the questions.
QS-3	As an admin, I want to carry out CRUD operations on the questions.
QS-4	As an admin, I want to be able to sort the questions based on difficulty level.
HS-1	As a user, I want to be able to look at my past attempts so that I can collate my learning points.
HS-2	As a user, I would like to be able to work on my previous attempts individually.

Key: AS - auth service, US - user service, MS - matching service, CS - chat service, COS - collaboration service, QS - question service, HS - history service

4.2 Weekly Sprints

Our group adopted the *Agile* software development process, utilizing the Scrum framework with weekly sprints. Each of our sprints usually lasts for a week. Our group dedicated the initial sprint to decide on our framework, tech stacks, user stories and requirements for the product. We also plan for the subsequent sprint during each sprint meeting. We then update and evaluate one another on the tasks completed, issues faced and upcoming tasks during each subsequent sprint. Throughout each sprint, we also make use of our group's *Telegram* chat to make small updates on our progress and issues faced. We feel that the process of weekly sprint meetings and a *Telegram* group chat provides better flexibility for our team while allowing us to keep one another updated regularly.

4.3 Jira

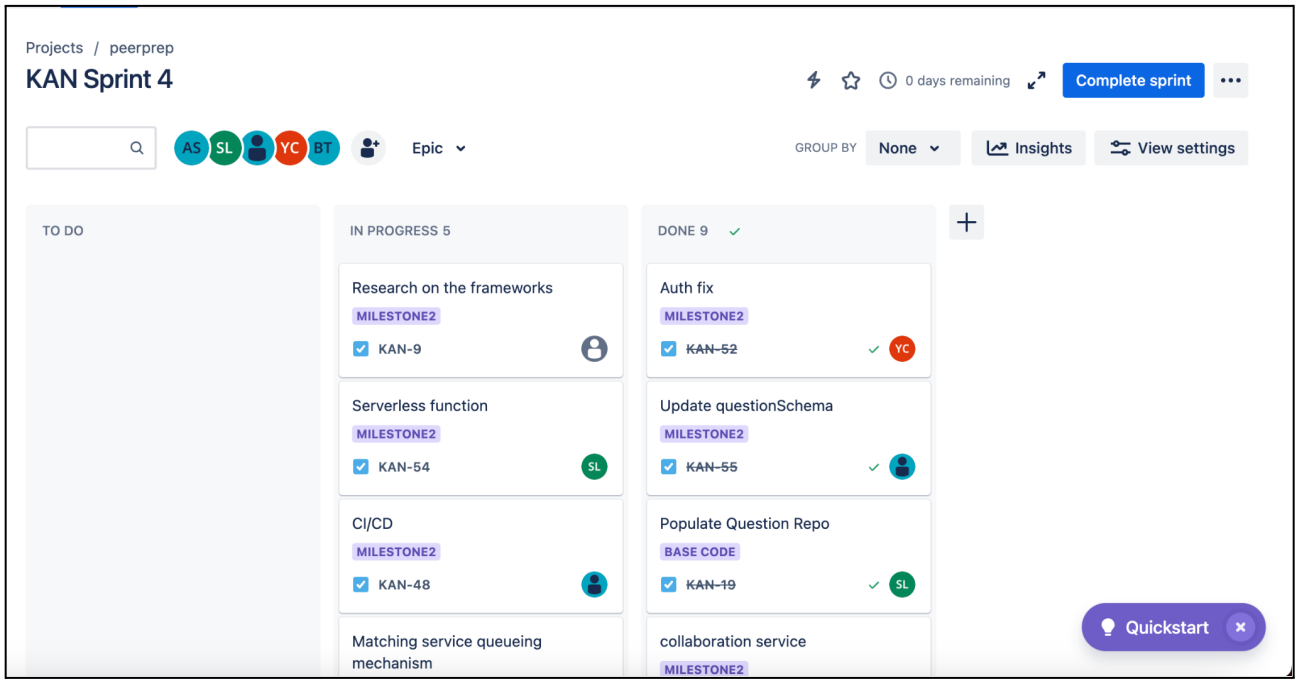


Figure 1: Jira board for task tracking

Our group utilized *Jira* as our weekly task tracker. *Jira* provides us with a broader range of features beyond code-related issues and we used it to manage our entire project lifecycle. Our board is updated during each sprint with the tasks outlined under the “TO DO” column. Throughout each sprint, tasks are updated from “TO DO” to “IN PROGRESS” or “DONE” depending on their progress. We regularly check the *Jira* board to see if our group members require any assistance with their tasks.

4.4 *Github* Issues

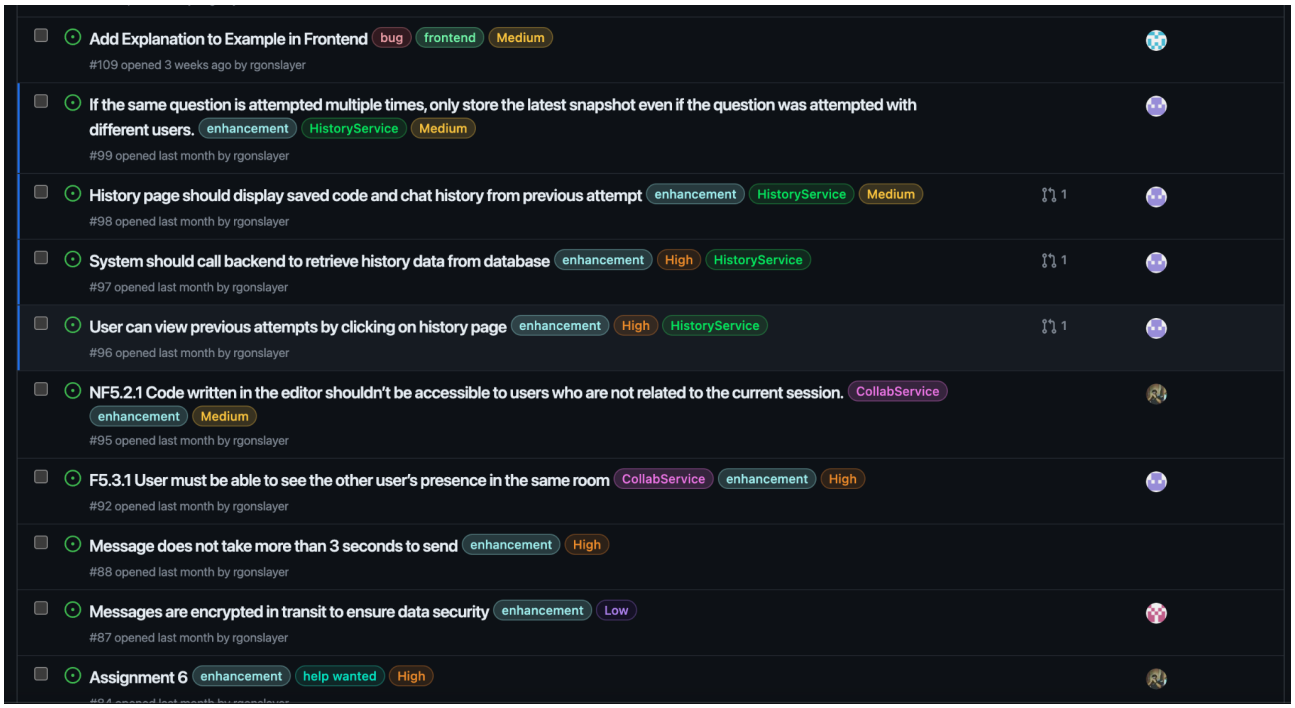


Figure 2: GitHub issues used to track code-related issues

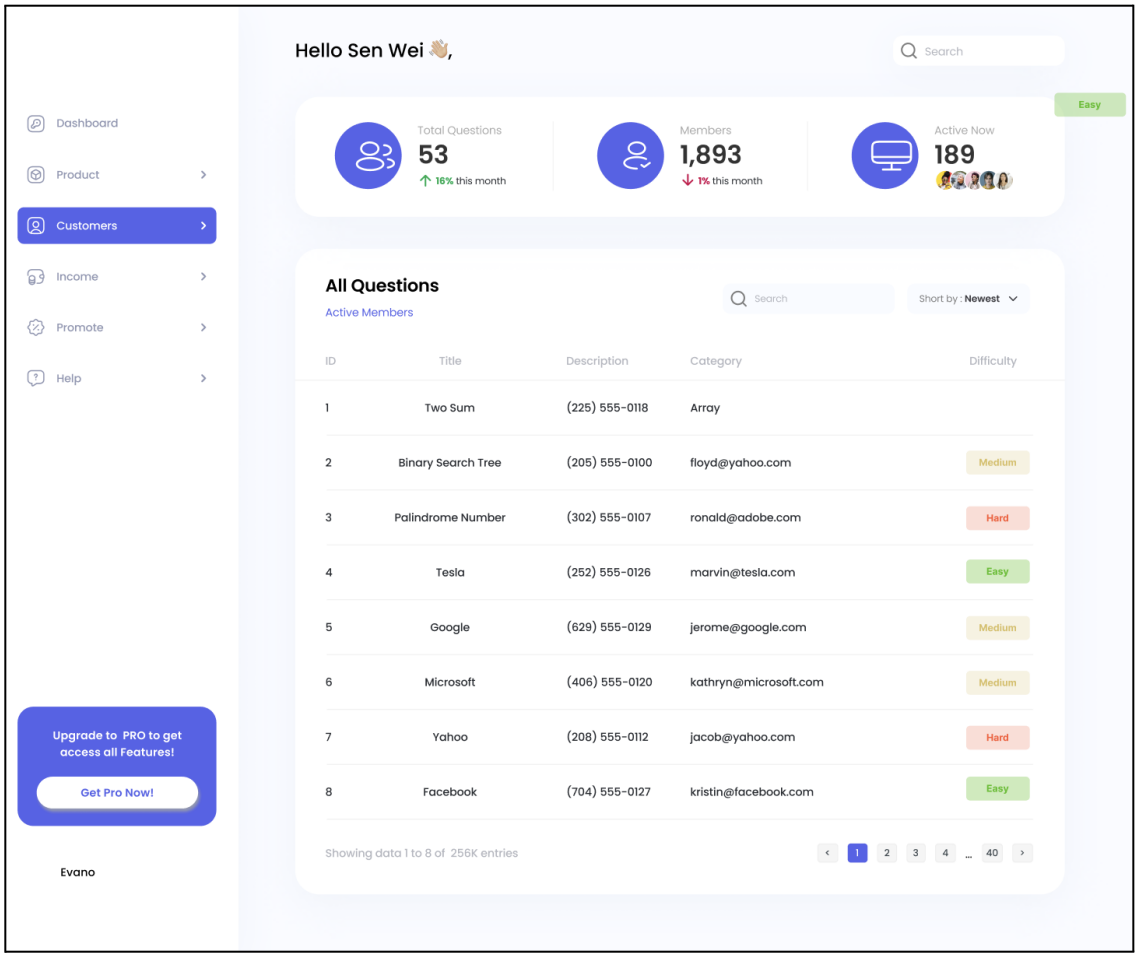
Our group also made use of *GitHub* issues to keep track of our code-related tasks. *GitHub* issues allow for seamless tracking of code-related tasks as it is integrated into our *GitHub* code repository. During each sprint, we update the *GitHub* issues and then assign them respectively. We are able to keep track of the progress of each issue by linking it to our respective pull requests.

4.5 Frontend Design Process

4.5.1 Requirement Analysis and User Stories

At the outset of the project, we conducted a thorough analysis of the requirements and gathered user stories to understand the needs and expectations of our target audience. This phase was critical in shaping the features and functionalities of the frontend. Refer to section 4.1 for all the user stories.

4.5.2 Wireframing and Mockups



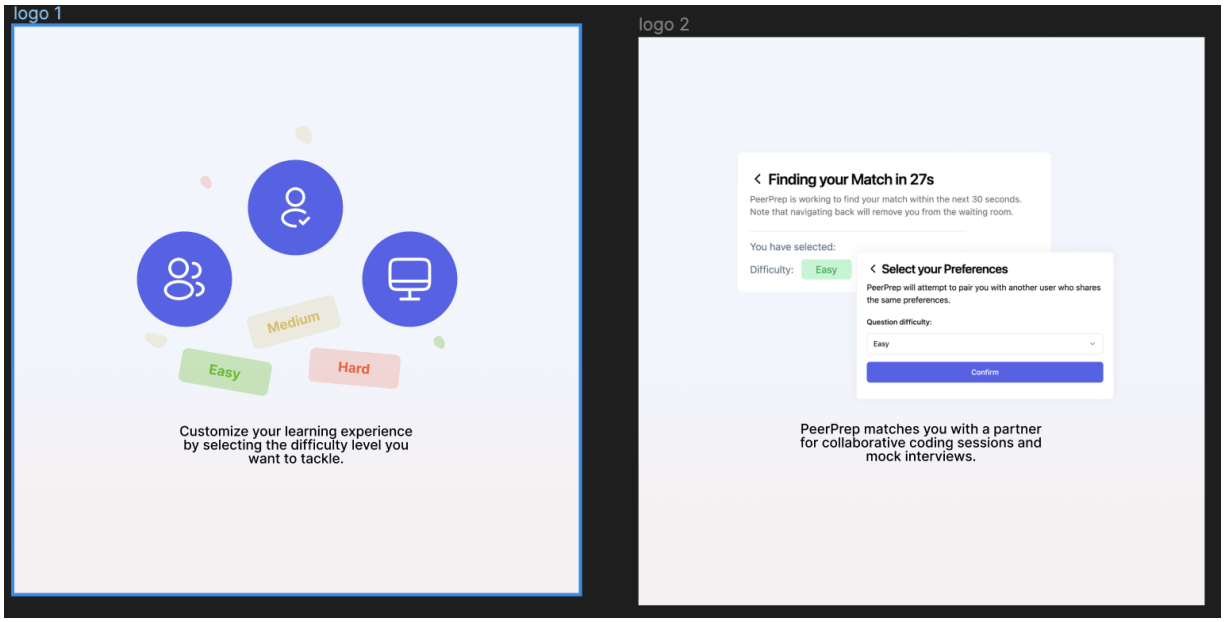


Figure 3: Figma Mockups

With a clear understanding of the requirements, our design team created wireframes and mockups using figma to visualize the user interface. This step allowed us to iterate on the layout and design to ensure a user-friendly and aesthetically pleasing interface. It also served as a guideline for us to design and develop the frontend UI, referring to the mockup as a guideline as we develop.

By following this structured frontend design process, we successfully created an interactive user-friendly, and feature-rich web application that caters to the collaborative learning and problem-solving needs of our users on PeerPrep

4.5.3 Technology Selection

We chose a stack that aligned with our project requirements. Our frontend was built using the following technologies:

1. *React*: To create dynamic and interactive user interfaces.
2. *TypeScript*: To ensure type safety and improved code quality.
3. *Axios*: To handle HTTP requests to the server.
4. *TailwindCss*: For efficient and responsive styling.
5. *ShadCN*: For reusable components that are well designed.
6. *Vite*: For more efficient development experience.

***React*:** We opted for *React* due to its component-based architecture, making it easier to build and maintain complex user interfaces. *React's* virtual DOM ensures efficient updates, providing a seamless and performant user experience. Its extensive community and ecosystem also meant access to a wealth of libraries and tools.

***TypeScript*:** To elevate code quality and maintainability, we embraced *TypeScript*. The static typing offered by *TypeScript* caught potential errors during development, reducing bugs and enhancing the overall robustness of our codebase. The type safety provided by *TypeScript* also improved collaboration among team members.

***Axios*:** Handling *HTTP* requests efficiently is crucial for any web application. *Axios*, with its simplicity and flexibility, provided a clean way to manage asynchronous data fetching. Its promise-based API and robust error handling made it a suitable choice for managing communication with the server.

***Tailwind* CSS**: For styling, we chose *Tailwind* CSS, a utility-first CSS framework. *Tailwind's* approach to styling using utility classes allowed us to build a responsive and visually consistent UI rapidly. Its flexibility and ease of customization aligned well with our design needs.

***ShadCN*:** To streamline our development process and ensure consistency in design, we leveraged *ShadCN*, a UI library for reusable components. We primarily used these components as the foundation for our tables, dialogs, forms, etc. These components are not only well-designed and

visually appealing, but also follow best practices, reducing redundancy and promoting a modular and maintainable codebase.

***Vite*:** To optimize the build time for our application, we integrated *Vite*, a build tool to provide a faster and more efficient development and build process by optimizing the generation of highly efficient bundles and taking advantage of ES Module imports to pre-bundle dependencies during development.

These technology choices collectively empowered our development team to efficiently deliver a feature-rich, responsive, and maintainable frontend that aligns seamlessly with our project goals.

4.5.4 Component Architecture

We adopted a component-based architecture, a fundamental design principle in modern web development. This approach involves breaking down the user interface into smaller, reusable building blocks known as components. Each component is responsible for a specific piece of the user interface and its associated functionality. This architecture offers modularity, allowing us to work on specific components independently, which keeps the codebase organized and manageable. The reusability of components reduces redundancy and promotes efficiency. Collaboration is streamlined, enabling parallel development by different team members. As the application evolves, the architecture ensures scalability, making it adaptable to changing needs. Isolation within components minimizes the scope of potential bugs, aiding in troubleshooting and debugging. Maintainability is enhanced as updates can be focused on specific components, reducing the risk of unintended side effects.

By following this structured frontend design process, we successfully created an interactive, user-friendly, and feature-rich web application that caters to the collaborative learning and problem-solving needs of our users on PeerPrep.

4.6 Code Organisation

4.6.1 Frontend

We applied modularisation with hierarchy to make the codebase easier to understand by setting boundaries between features and minimizing code coupling and side effects. A good folder structure allows us and other developers to find files faster and manage them more easily. A general high level overview is that we separated the files into a few building blocks:

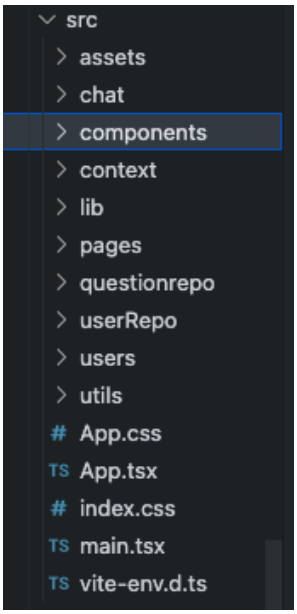


Figure 4: Frontend File Structure

Components: All shared components that are used across the entire application.

Assets: The assets folder contains all images, icons, css files, font files, etc. that we used in our web application. Custom images, icons, paid fonts are being placed inside this folder.

Context: The context folder stores all our react context files that are used across components and multiple pages.

Pages: The pages of our application, like login page, admin dashboard, user dashboard, etc.

utils: Shared utility functions.

Repo: Module responsible for managing interactions for a specific entity like questions and users. Acts as a bridge between the frontend and backend, handling HTTP calls and data management associated with the entity.

4.6.2 Backend

As we were implementing a microservices architecture, it made sense for our backend services to be organized into their individual folders. This clearly separates the code between different services and keeps them from interfering with each other.

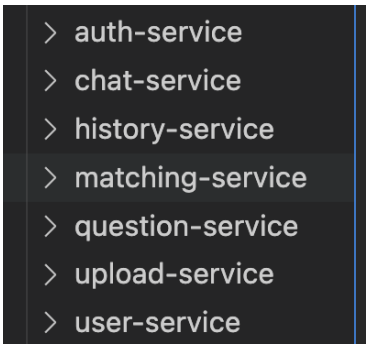


Figure 5: Backend Services File Structure

Within each service, files were typically organized into the *Nest.JS* main application files and our service files. We placed our main files within our service folder while any supporting documents would be placed in additional folders such as decorators, guards and strategies.

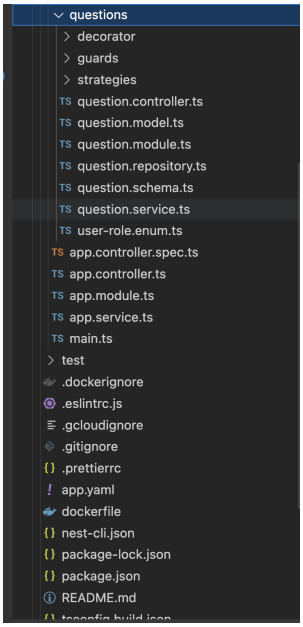


Figure 6: QuestionService File Structure

As we were following the MVC architecture, every service generally has the following, a controller file or a gateway file, a service file, a repository file if a database is linked to the service and a schema or a DTO file.

1. **Controller**: The controller is responsible for handling incoming HTTP requests and sending responses to the client. Controllers define routes, handle requests, and return responses.
2. **Gateway**: The gateways are used primarily for handling WebSocket connections, enabling real-time, bi-directional communication between clients and the server.

3. **Service**: Services are part of the service layer and are responsible for business logic and data manipulation. Services handle tasks like data validation, database interactions, and other business rules, and are injected into controllers or other services.
4. **Repository**: Repository is an abstraction that encapsulates the logic required to access data sources. It provides a collection of methods for performing operations on the data source, such as create, read, update, and delete (CRUD) operations. Repositories work with entities and help in isolating the data layer, making the code more maintainable and testable.
5. **DTO (Data Transfer Object)**: A DTO is a simple object that defines the shape of data for a specific case, like a request body or a response format. DTOs are used for data transfer between different parts of the application, particularly between clients and servers. They help in validating and transforming incoming data, ensuring that the data adheres to the specified structure and type before being processed by the application.

5. Services

5.1 General architecture

Microservices architecture

Peerprep follows a microservices architecture where each feature is encapsulated within a service. The services are mostly independent of each other, but in the case where communication between two services is required, a message broker is used to facilitate the communication, which in our application is *RabbitMQ*. By following the microservices architecture, we are able to decouple different services, and each service independently communicates with the client to handle requests. This makes future extensions to the application easier as we can easily add a new service without modifying existing services, as they are all independent of each other.

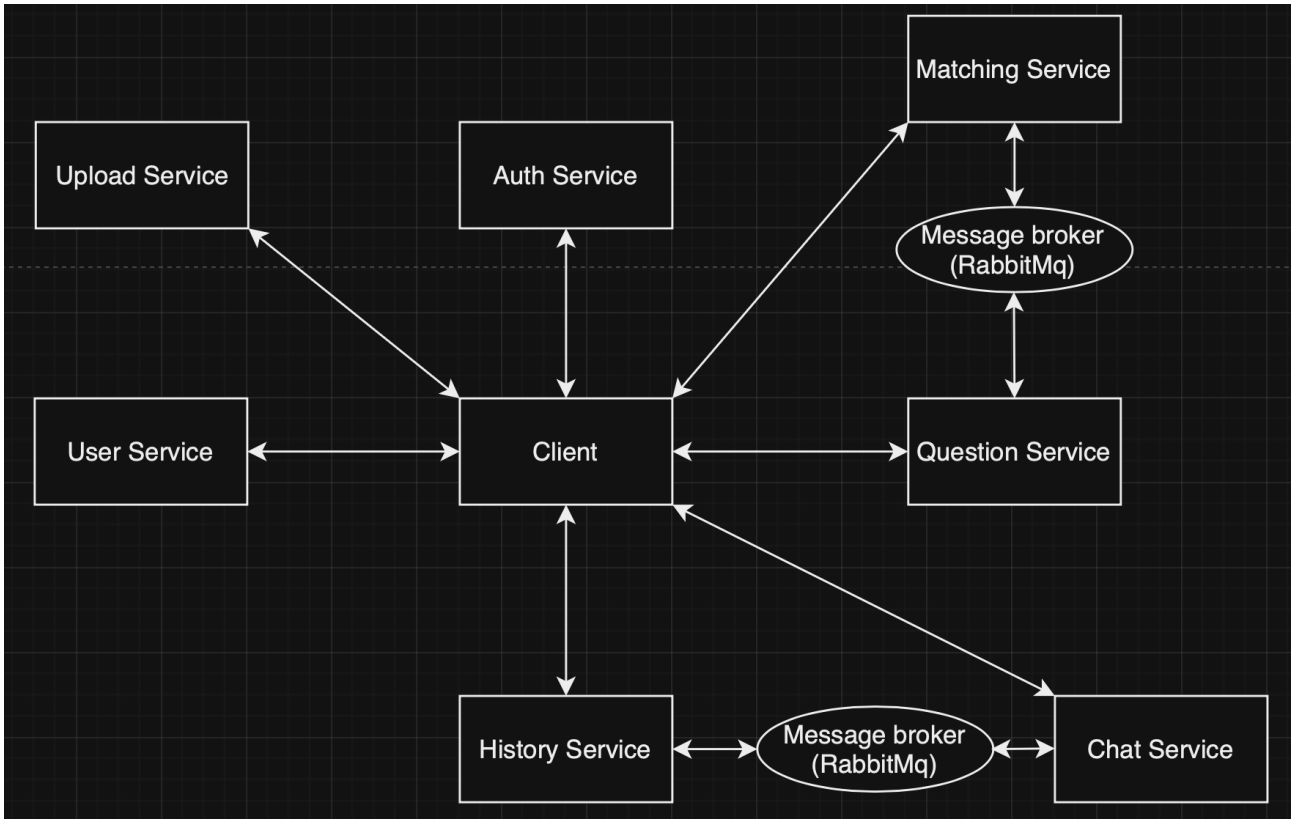


Figure 7: High level architecture of peerprep

Model View Controller (MVC) pattern

Our team has adopted the MVC pattern in each of the services. MVC encourages a modular design, where the service is divided into 3 components, Model, View and Controller as a whole, although all services share the same view. This modularity aligns with the modularity of microservices architecture, which makes MVC a natural pattern to adopt in individual services. Furthermore, it allows the services to achieve separation of concerns, as each component in the MVC pattern has a specific responsibility, reducing the overall complexity of the service. Below are the MVC pattern diagrams for 2 components, where one contains a controller to accept *HTTP* requests, and the other contains a gateway to emit and accept *Socket.io* events.

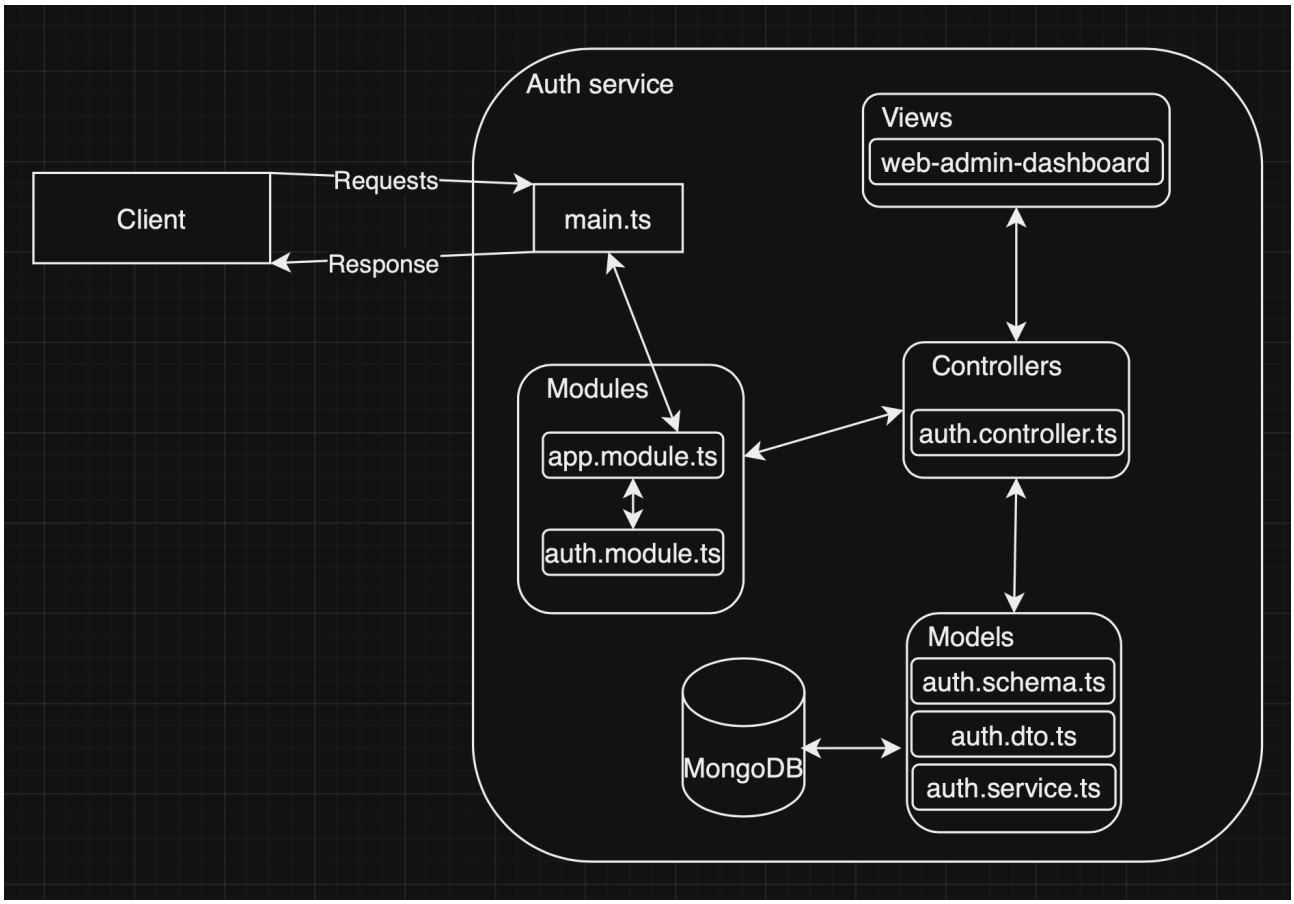


Figure 8: MVC architecture of auth service

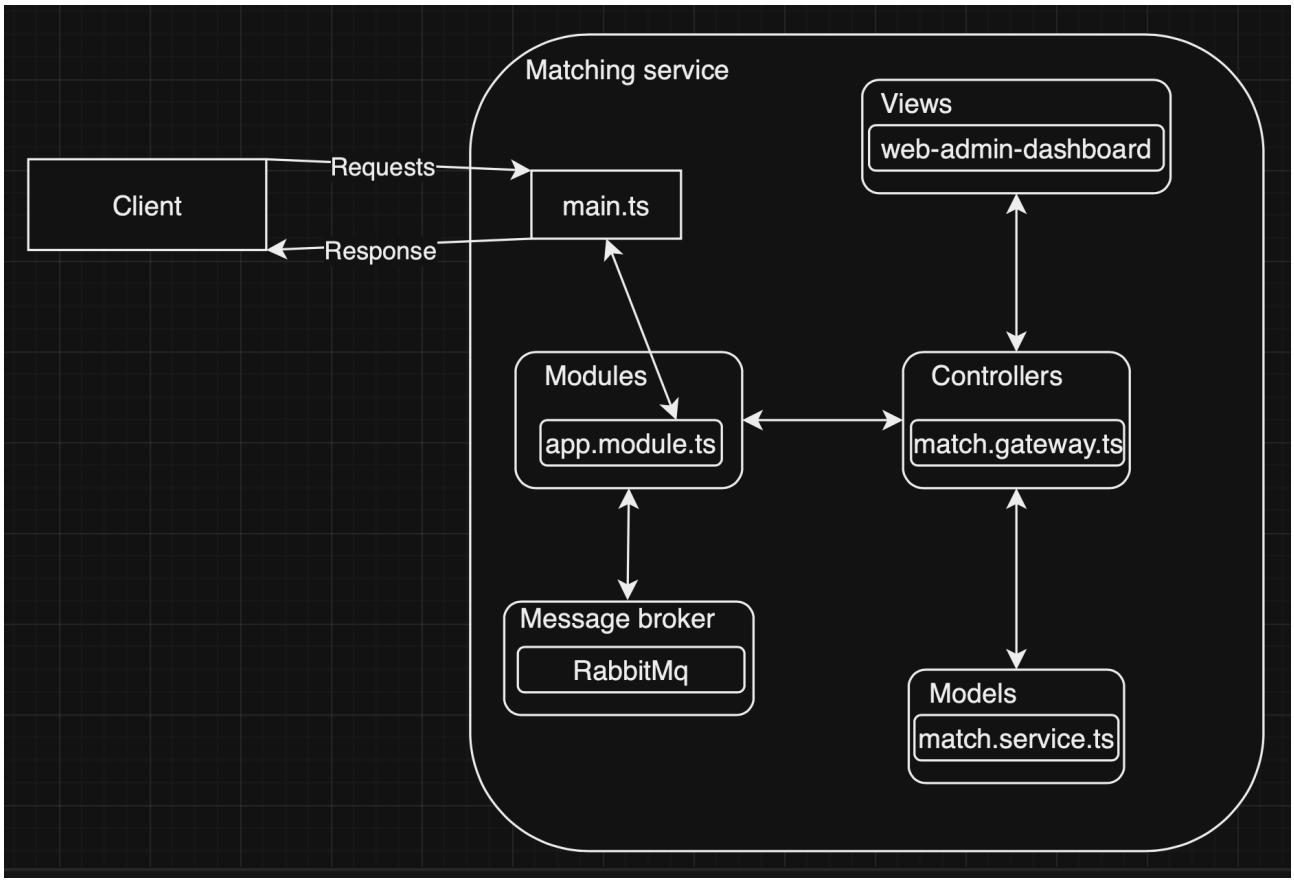


Figure 9: MVC architecture of matching service

5.2 Auth Service

The auth service is responsible for authentication and authorization of the user. It handles requests such as login and signup of users, and issues them relevant access and refresh tokens for sending requests to the server. It stores user credentials, namely email and password in a *NoSQL* database.

Combining User and Auth service vs Separating User and Auth service

One of the main design decisions we had to make was whether to separate or combine user and auth service, as they are both concerned about the same entity, which in this case is the user. However, to strongly adhere to the microservices architecture, we have decided to separate these two services as although these two services serve the same entity, their purpose is quite distinct in

the sense that auth service is responsible for storing user credentials, and user service is responsible for storing other user information like username.

Authentication and authorization

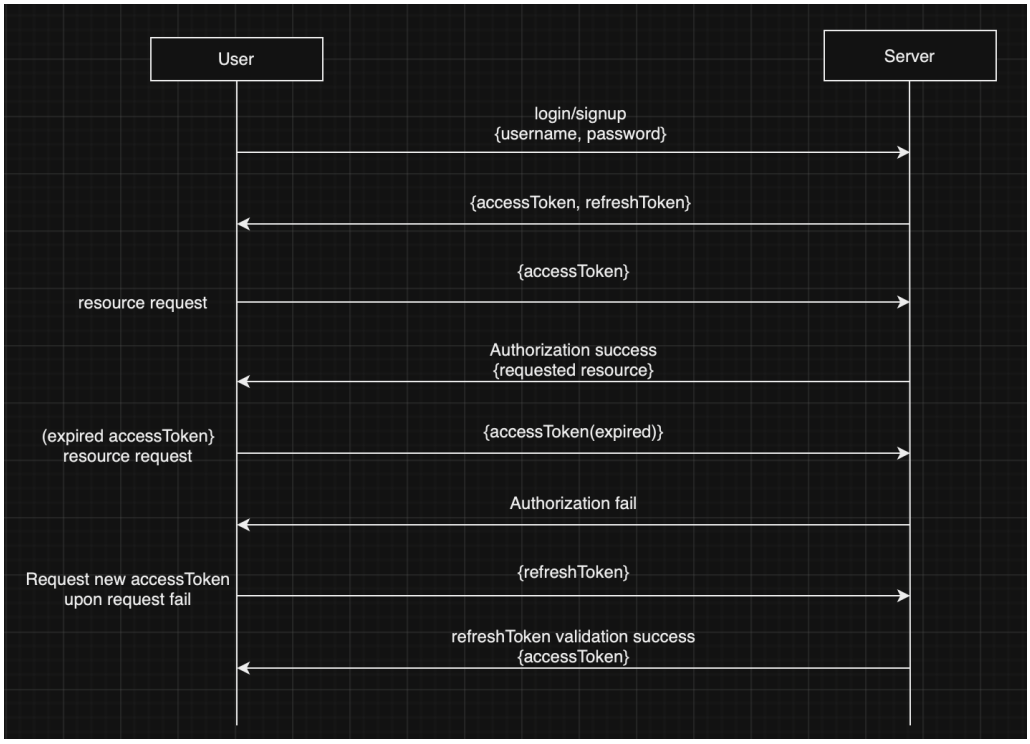


Figure 10: Authentication and authorization using JWT tokens

Authentication and authorization is implemented using *JWT* tokens. There are two types of tokens used in our application, which are access token and refresh token. The access token is the main token which is sent when making requests to the protected routes of our server. It also contains confidential information about the user such as user Id, email and role. The server validates the access token, checks that it's still valid, and only lets the request pass through upon successful validation. One of our design choices regarding authentication and authorization was the number of tokens to be used. An alternative approach to the current one would be using a single access token, but with a longer validity, for example 15 days. However, this would have serious security issues in the case where the access token is leaked, or stolen, as the intruder would then be able to make valid requests to our system until the access token expires. Therefore, it is essential that the access token is valid only for a short time, which in our case is only 5 minutes.

Session persistence

One main disadvantage of having an access token which expires quickly is that it interrupts the user experience as the user would have to log in again every 5 minutes to receive new access tokens in order to make requests to the server. We have migrated this issue with the introduction of

refresh tokens, whose sole purpose is to refresh the access token without logging in again. Therefore, we have set interceptors on the frontend, which will intercept unauthorized responses returned from the server, send a request to refresh access token using refresh token, and use the new access token to continue with the original request. This is all carried out automatically, and the user won't notice that all this process is going on behind the scene, providing them a seamless user experience.

Google OAuth

To further incline user experience, we have incorporated *Google OAuth* to our application to allow users to login or signup with their *Google* accounts. One of the design decisions that was made in the progress of implementing *Google OAuth* was choosing whether to handle *OAuth* requests from the frontend or the backend. We initially created *Google OAuth* endpoints in the backend and called them from the frontend, but this made fetching responses from the backend very troublesome. Therefore, our team has decided to migrate this responsibility from the backend to the frontend, allowing us to take a similar approach to existing login/signup to handle *Google OAuth* login/signup.

5.3 User Service

The user service is responsible for storing and retrieving information related to the user such as username, refresh token, role and email. User information is stored in a *PostgreSQL* database on Google Cloud Platforms.

User roles

In order to facilitate user management, different roles are given to different users present in the system. The two roles present are 'Admin' and 'User'. Users are directed to different screens depending on their role. For example, users with the role 'User' will be directed to the user-dashboard where they are able to view their question attempt history, along with a match button to find a peer to collaborate with. On the other hand, users with the role 'Admin' will be directed to the admin-dashboard where they can view the entire list of questions and users, and perform actions like deletion and modification.

User management

Basic user profile management can be carried out by respective users once they log in. Some of the actions they can perform include changing their username or password. Regarding user management as a whole, users with 'Admin' role can view an exhaustive list of users, and perform actions like deletion or modification of roles of the users.

Security concerns

Our application strictly prohibits users with role ‘User’ to navigate to admin dashboard, as it is only the admins who can manage questions and users. Users who try to manually route to the admin dashboard from the frontend will be directed to an unauthorized page. However, in the case where a user somehow manages to access the admin dashboard, related routes in user service are equipped with a guard which prevents non-admin users from sending requests. The guard in this case is the *RolesGuard*, which should be used together with *AccessTokenGuard* which validates access tokens, and only allows requests with valid access tokens to pass through. When a request arrives, *AccessTokenGuard* validates the access token and extracts user information such as role, email and user Id from the provided token. This information is used by the *RolesGuard* to check if the current user’s role is in the list of allowed user roles, and only lets the request through if the user’s role is in the list. Having two layers of security, one layer blocking the access of unauthorized pages directly from the frontend, and one layer blocking non-authorized users to send requests from the backend, makes sure that only the users with a valid role are able to perform certain actions in the system.

5.4 Matching Service

Matching service is responsible for matching two users if they chose the same difficulty. As there is no data that needs to be persistent after matching, no database was required for matching service. Socket io was used to handle events related to matching service, such as joining a match, or being informed of a successful/unsuccessful match.

Events in matching service

Events received by server - *‘match’, ‘matchCancel’, ‘leaveSession’*

Events emitted by server - *‘partnerLeaveSession’, ‘matchSuccess’*

‘match’ - This event is emitted by the client, which in this case is the frontend. When the user chooses the difficulty and presses the matching button, this event is emitted from the frontend, and received at matching service, which then proceeds to enqueue the user in the queue with the corresponding difficulty.

‘matchCancel’ - This event is emitted by the client when matching was unsuccessful or the client deliberately decides to cancel matching. Once this event is received in matching service, the user is dequeued from the queue they were previously in.

‘leaveSession’ - This event is emitted by the client when one of the two matched users decide to leave the session. The purpose of this event is to inform the remaining user that their peer

has left the session by emitting ‘partnerLeaveSession’ event.

‘partnerLeaveSession’ - This event is emitted by the matching service once the ‘leaveSession’ event is received. It informs the remaining user that their peer has left.

‘matchSuccess’ - This event is emitted by the matching service upon a successful match between two peers. It informs the clients that a successful match has been made, and the client consumes this event to navigate to their respective sessions.

Design choices

One of the design choices we had regarding matching service was whether we should allow the user to choose to stay or leave the session when their peer left the session. It seemed sensible to forcefully make the remaining user leave the session as the whole point of collaboration is lost when a user is inside a session alone. However, at the same time, there may be users who may be willing to work further on their collaboration to come up with a more defined solution. Hence, we have left this choice to the users, allowing them to decide on whether to leave or stay in the session once their peer leaves.

Random question dispatching

In order to incorporate a randomness to the question that is being dispatched to a session, matching service calls an endpoint in question service to fetch a random question of chosen difficulty and dispatches the question to the users in the session. An alternative choice to fetching the question from matching service was to hand over this role to the frontend. However, this would mean that users in the same session may be viewing different questions as they would be fetching the random questions independently of the other user. Therefore, we have taken the current approach where fetching of random questions will only occur once per session by the matching service, and hence allowing the users to receive the same question.

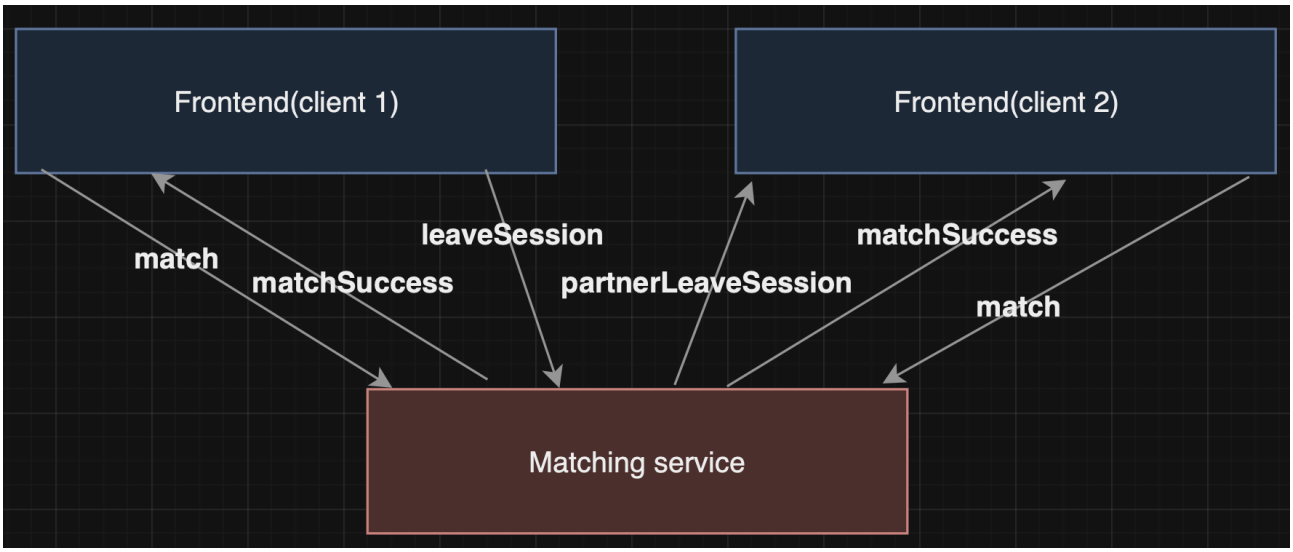


Figure 11: Matching service's architecture

5.5 Chat Service

The chat service is responsible for allowing seamless communication between the matched users so that they can effectively communicate their thought process during the session.

One of the most crucial implementations in chat service was to make sure that the message is propagated to only the intended individuals. To achieve this, we have made use of rooms in Socket.io to make sure that only the users within the same room receive the message.

Upon being matched, matching service generates an unique uuid which is used as the room Id for the current session. This approach makes sure that the room Id is unique for every session, migrating the issue of two sessions having the same room Id. The matched users subscribe to this room, and sending or receiving a message is strictly contained within the subscribers of the room.

Subscribers of the room emit a ***'message'*** event whenever they send a message, with the event payload being the message content. Upon receiving this event, the server emits a ***'sendMessage'*** event to propagate the message to other users in the same room.

This is the publisher subscriber model, where each user takes turns to both be the publisher as well as the subscriber.

Design Choices

Publisher Subscriber Pattern

Both peers act as both a publisher and a subscriber. They subscribe to the current room, which is the roomId for the current session. The peer sending the message becomes the publisher and the peer receiving the message becomes the subscriber. The diagram below contains only 2 publishers/subscribers as there are only 2 peers in a session.

The main benefit of the Publisher Subscriber pattern is that it is easily scalable. In the current implementation, only 2 peers join a single session, but it can easily be scaled to more than 2 users by allowing more users to join the same room. This does not require much modifications to the current implementation.

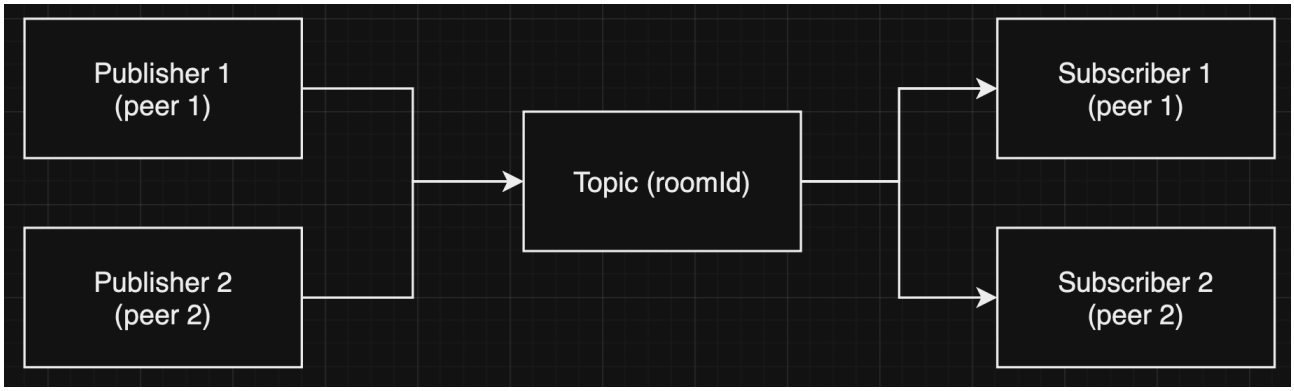


Figure 12: PubSub design pattern in chat service

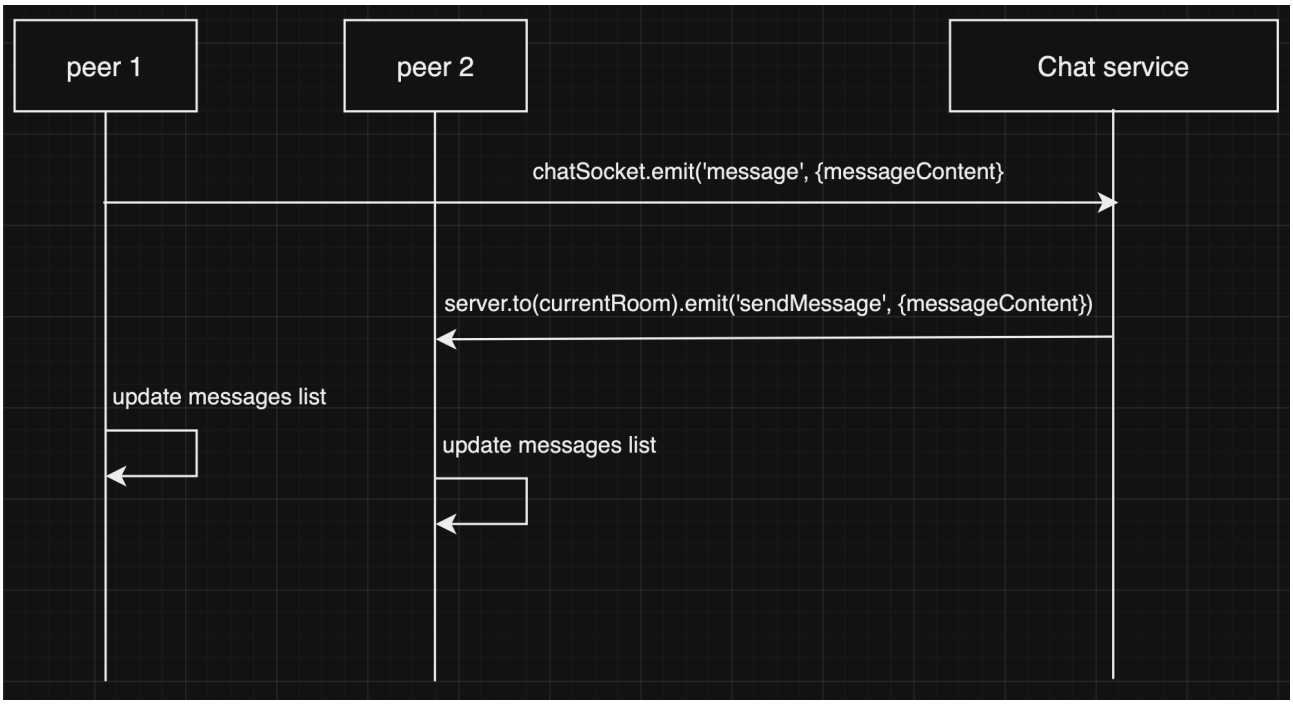


Figure 13: Sequence diagram for message sending/receiving

5.6 Question Service

The question service is responsible for management of the questions such as *CRUD* operations on the questions. Furthermore, one of the main logics within question service includes random question fetching, which dispatches a random question with the chosen difficulty.

Question Management

Just like user service, only admin users are allowed to perform *CRUD* operations on the questions. This is achieved with the use of *RolesGuard* which is identical to the one used in user service. These operations can only be performed on the admin-dashboard, which ordinary users cannot access. Questions are always tagged with difficulty, which allows random question fetching by difficulty.

The question schema contains fields such as ***questionTitle, questionCategories, questionDifficulty, questionDescription, questionExamples, questionConstraints and questionImages***. To allow for future schema extensions, a *NoSQL* database was used to incorporate schema flexibility.

Random question fetching

One of the main logic in question service is random question fetching. Whenever a successful match is made, the matching service sends a message to the question service through *RabbitMQ* to obtain a random question. Through a custom tailored query to *MongoDB*, we are able to obtain a random question with a selected difficulty within the question repository. This question is then returned to the matching service where it is dispatched to the matched users.

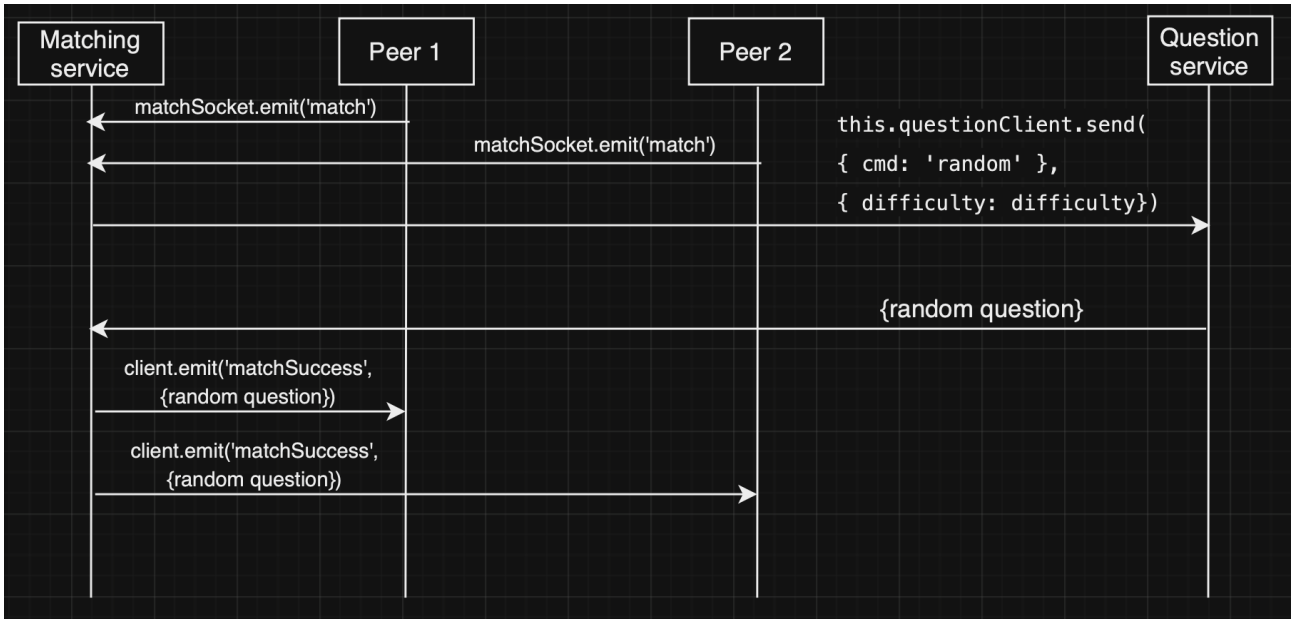


Figure 14: Sequence diagram for random question fetching

5.7 Collaboration Service

The collaboration service is implemented using *WebRTC*, which enables real time communication between web browsers without the implementation of a server to transmit data. To ensure that there are no race conditions (conflicts in text edits), we use an implementation of *CRDT* (Conflict-free Replicated Data Types) with a *Yjs* wrapper library in order to transmit text data that is able to resolve itself through internal implementations in the library. CRDTs are decentralized, they don't require a single server to work, instead you can sync your devices via any kind of network that happens to be available. We chose *CRDT* for this project as there are many open-source implementations available with well written documents. *Yjs* also works with *WebRTC* technology out of the box so creating a real time collaborative text-editor was a matter of putting the two together.

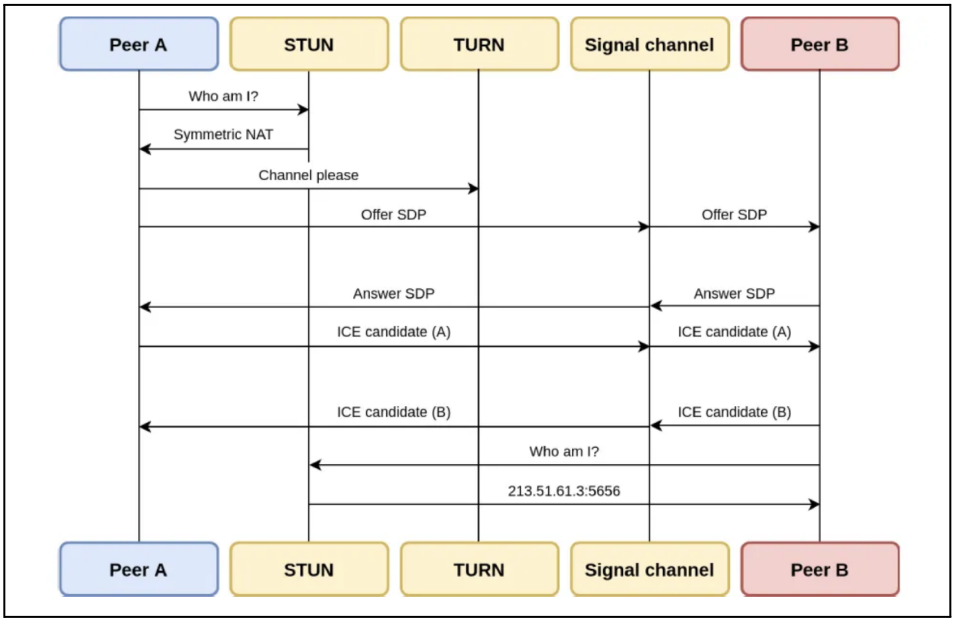


Figure 15: A traditional WebRTC setup

Traditionally, *WebRTC* requires signaling servers that match a waiting connection to another waiting connection, but this also requires setting up of *STUN* or *TURN* servers, which are used to establish peer connections between two wanting peers which, in our case, are two PeerPrep users.

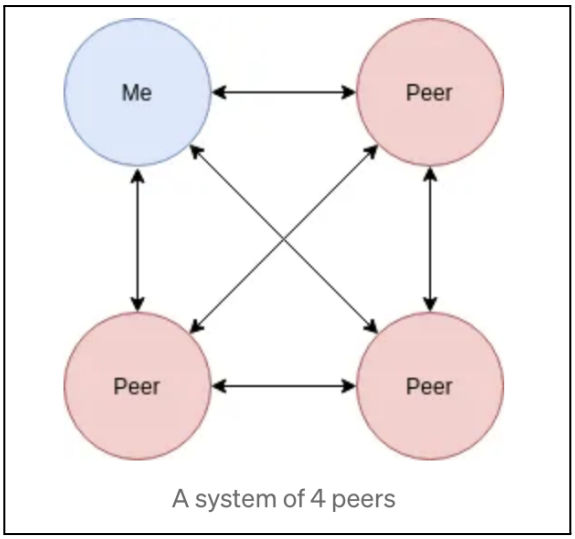


Figure 16: P2P Architecture

However, an easier method involves all peers connecting to all other peers. To distinguish the specific code within the editor that each user is working on, we employ a dictionary. This dictionary assigns unique room IDs as keys, with the corresponding text from the code editor

serving as the associated content. This ensures that only the peers belonging to the same room are able to edit their code. The chance of key collision is impossible, since our roomIds are generated as unique ids, so they can never be the same. The process can be found below from matching to establishing a connection for the editor:

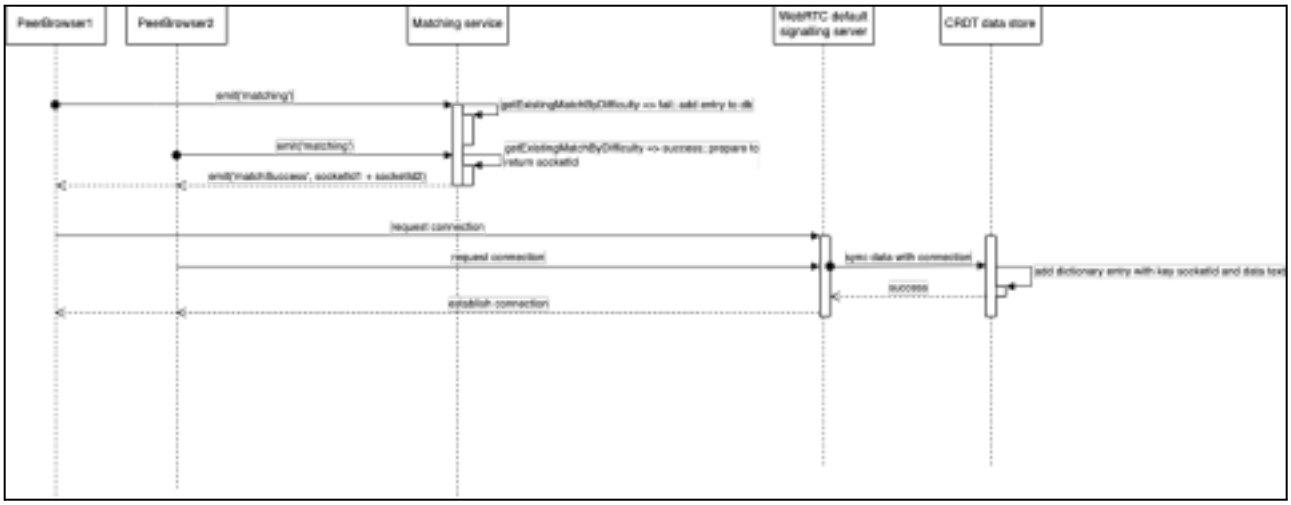


Figure 17: Activity diagram for matching and collaborative editing

5.8 History Service

History service is implemented using *Nest.JS* and *MongoDB*. History service maintains a record of the questions attempted by the users, maintaining information such as room id, the question attempted, the peer that the user was matched with, code history, chat history and date of submission.

When users are successfully matched, matching service makes a call to history service and a pair of history documents, one for each user, is added to history service. During the session, any messages sent through chat or edits made on the code editor are updated in the history documents for both users in real-time.

While the history service stores the attempt history of all the users, users are only able to view their own past attempts. A user's history is rendered on the user dashboard page by making an API call to the history service backend. The history service returns all the history documents pertaining to the current user and they are displayed on the user dashboard.

Users are also able to view their individual history session where they are able to make changes to the code. The edits made to the code will be updated in their respective history document.

Below is the sequence diagram for the flow of successful matching to creating a history document, along with saving chat in real-time. Edits made on the code editor are saved in a similar manner to how chat is being saved.

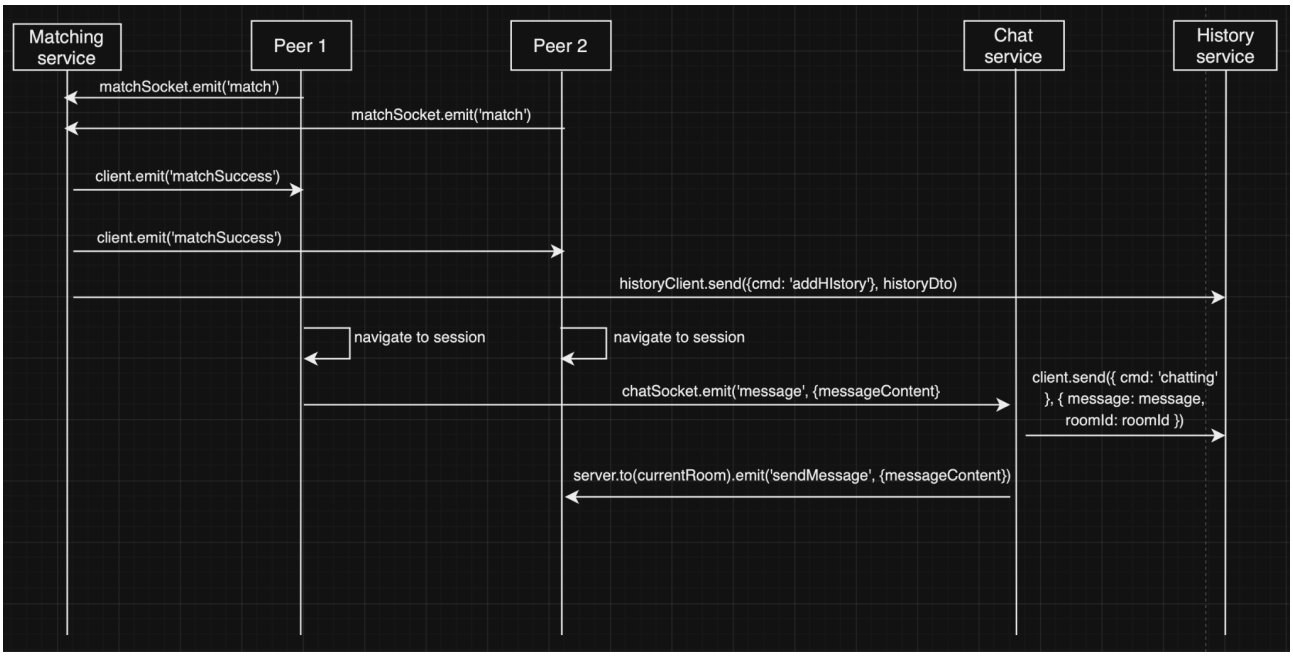


Figure 18: Sequence diagram for automatic addition of history document

6. Deployment

Our deployment is executed entirely on *Google Cloud Platform's Kubernetes Engine (GKE)*. This decision was driven by *GKE's* robust features that align well with our microservices architecture. Below, we detail the factors contributing to our choice of *GKE* and how it accommodates the complexities of our application.

1. **Scalability and Performance**:
   GKE demonstrates exceptional scalability and performance capabilities, essential for our application, which is subject to variable traffic and workload. Features like the *Cluster Autoscaler* and *Horizontal Pod Autoscale*r enable our microservices to scale dynamically in response to demand, ensuring resource-efficient operation.
2. **Container Orchestration**:
   As a powerful container orchestrator, *Kubernetes* under *GKE* simplifies the deployment, scaling, and management of our containerized microservices. This orchestration is fundamental to maintaining the continuous integration and deployment flow of our complex application.
3. **Advanced Networking and Security Features**:
   GKE provides advanced networking options, including sophisticated load balancing and network policies which is another one of our nice to haves.
4. **Integrated Google Cloud Services**:
   The seamless integration with other Google Cloud services, such as *Cloud SQL*, provides a comprehensive and cohesive infrastructure, simplifying the management of our microservices ecosystem.

6.1 Backend Deployment

Our deployment began with the backend, utilizing *GCP's Kubernetes Engine.* The choice was influenced by features like automated horizontal pod deployment and integrated ingress controllers. Already using *GCP's Cloud SQL* for our databases, *GKE* provided a unified platform for backend management.

Ideally, each backend service would be deployed as its own workload, enabling independent scaling. However, at this moment, each workload requires its own *Kubernetes* service to serve as a load balancer and each load balancer requests a new IP address. Due to quota limits set by *GCP*,

we were unable to request more than 4 IP addresses, which limits the number of workloads we can deploy down to 4.

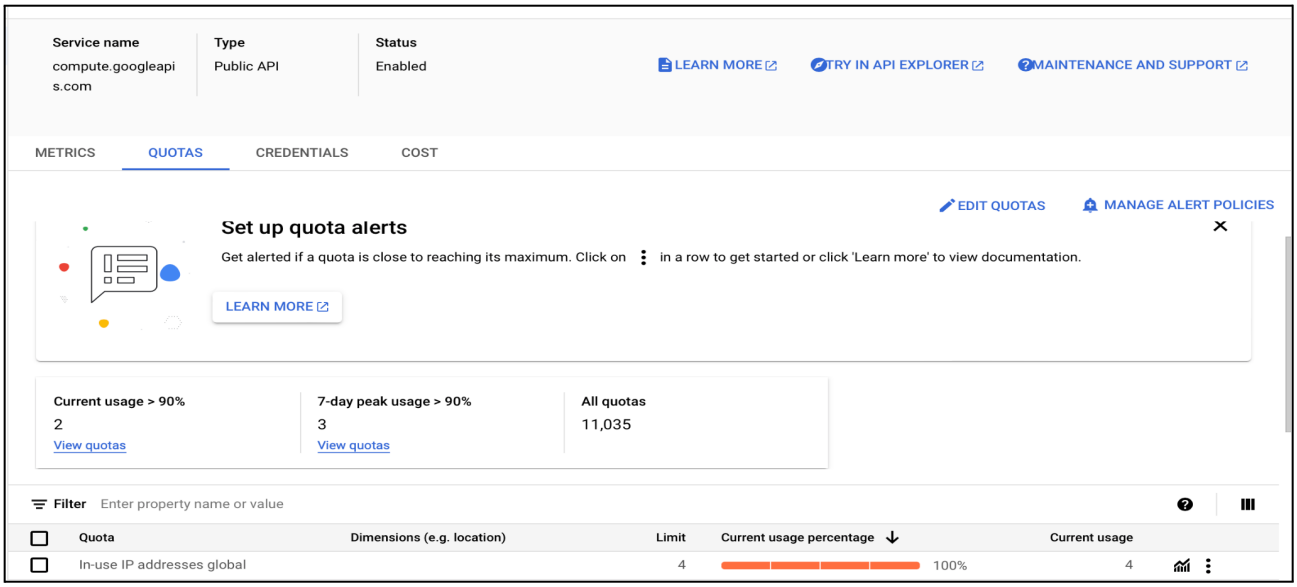


Figure 19: Quota for IP addresses

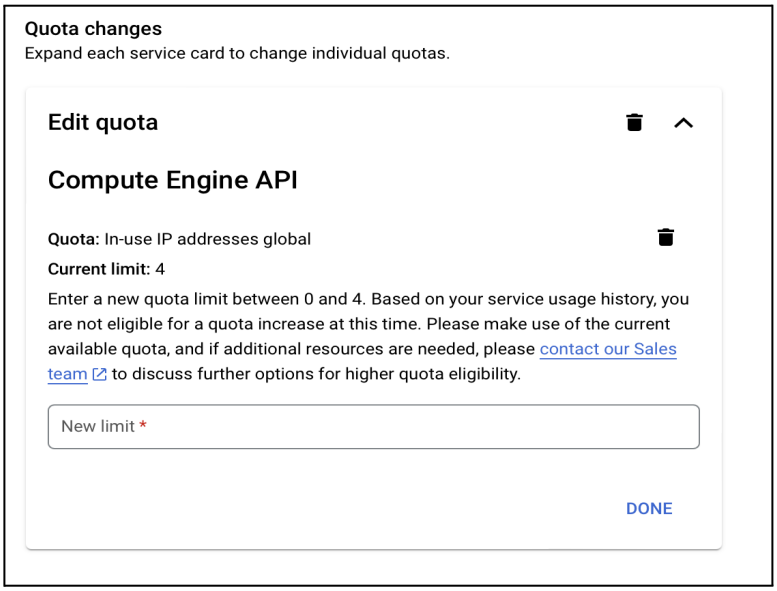


Figure 20: Quota change request

Hence, for now we will deploy the frontend container as one workload, and split the other services into 3 workloads, one to handle matching(ChatService, MatchingService and RabbitMQ), one for authentication (UserService and AuthService) and one for the rest (QuestionService, UploadService, HistoryService).

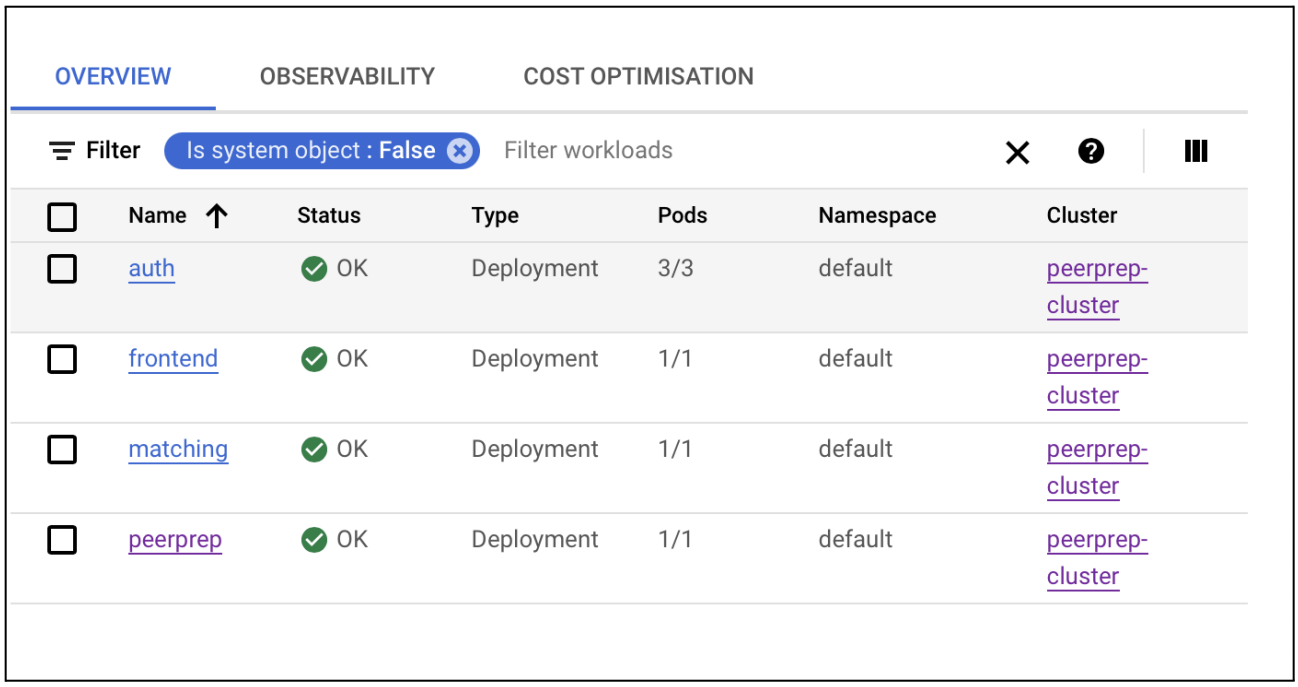


Figure 21: Kubernetes Workloads

Each workload has a Horizontal Pod Auto-scaler to scale up the number of pods available when the workload increases.

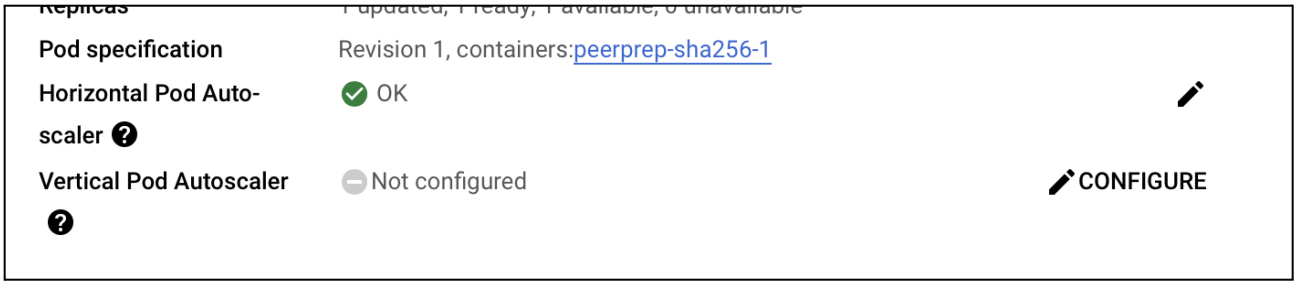


Figure 22: Kubernetes HPA

When evaluating our matching workload, we took into account the presence of stateful services like *RabbitMQ* and MatchingService. We recognized a potential issue under the *Horizontal Pod Autoscaler (HPA)* setup: users might connect to different pods, leading to possible mismatches. However, we prioritized scalability during high traffic periods, considering it more crucial for overall system performance. Additionally, the likelihood of having a single user in one pod during production is minimal. With these factors in mind, we concluded that maintaining our current deployment architecture was the most pragmatic approach, balancing scalability with user experience.

6.2 Frontend Deployment

In our search for an optimal frontend deployment solution, we considered Vercel, GitHub Pages, and Firebase Hosting for their convenience in deployment and inherent scalability. However, we faced a significant challenge: our backend was deployed on *Google Kubernetes Engine (GKE)* without *SSL/TLS* encryption, thus limiting us to *HTTP* traffic. Since these frontend platforms mandate *HTTPS* for enhanced security, they were incompatible with our *HTTP-only* backend API endpoints.

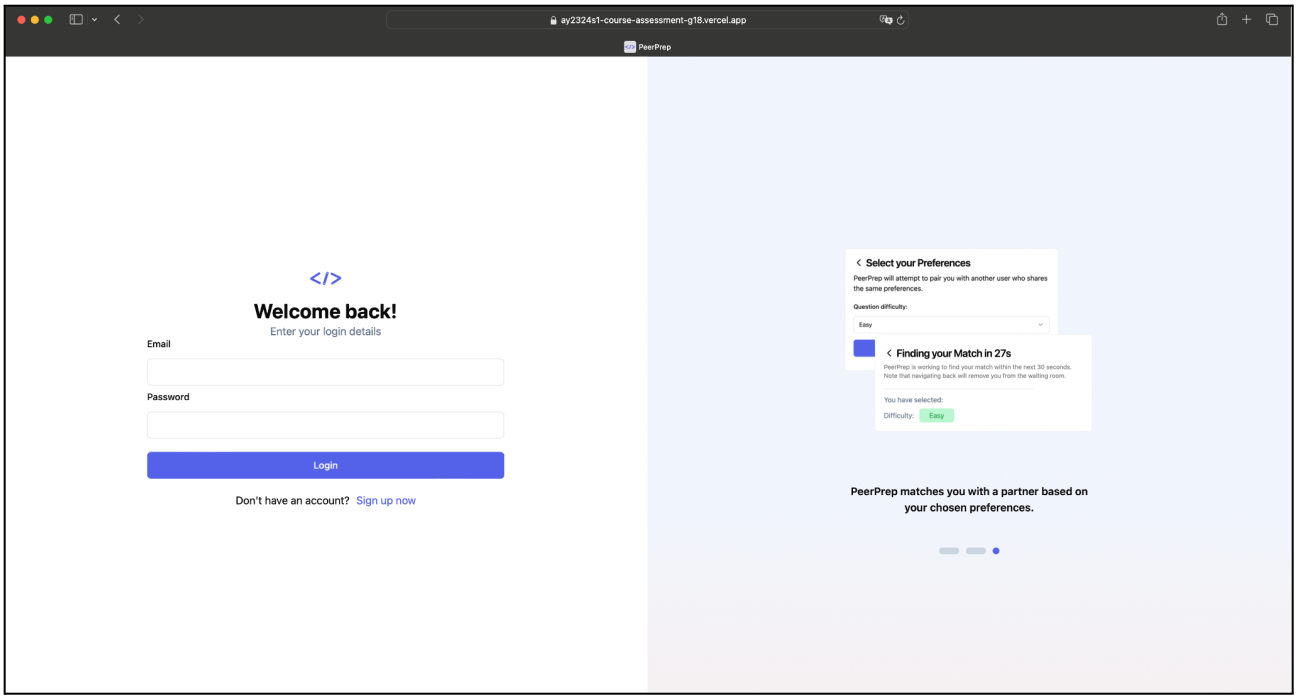


Figure 23: Deployment on Vercel with HTTPS enabled

Configuring *SSL/TLS* on *GKE*, particularly with a self-signed certificate and a custom domain, presented a complex task. Considering the time constraints and the steep learning curve involved in this setup, we opted for a more immediate solution. We chose to dockerize our frontend application and deploy it directly on *GKE* using *HTTP*. This approach, albeit a temporary compromise on security, was necessary to maintain operational functionality. We do, however, plan to transition to *HTTPS* in the future to bolster security.

This decision to utilize *GKE* for frontend deployment, though initially unplanned, proved advantageous. *GKE's* strengths in handling complex microservices architectures, its efficient backend logic processing, and superior container management capabilities made it an unexpectedly robust and fitting solution for our project's requirements.

6.3 API Gateway (Ingress Control)

GKE's architecture includes a sophisticated *Ingress Controller* that interfaces with the external load balancer. This setup is pivotal for managing incoming traffic. As traffic reaches the *Ingress Controller*, it scrutinizes each request, applying defined routing rules to direct the traffic efficiently. This process ensures that incoming requests are forwarded seamlessly to the appropriate *Kubernetes* services or workloads within our system.

Once a request is received, the *Ingress Controller's* routing mechanism ensures it is directed to the correct pod within the specified workload. These pods, in turn, are responsible for relaying the requests to the relevant microservices. This streamlined process facilitates efficient handling of requests, ensuring quick and accurate responses.

The API Gateway configuration in our setup is intentionally straightforward. Each workload is linked to a *Kubernetes* service equipped with load balancing capabilities. This service is associated with an externally accessible IP address. We have strategically exposed each service through a designated port on this external IP. This approach not only simplifies the architecture but also enhances the manageability and scalability of our system. By minimizing complexity, we ensure a robust and responsive infrastructure capable of handling varying traffic loads with ease.

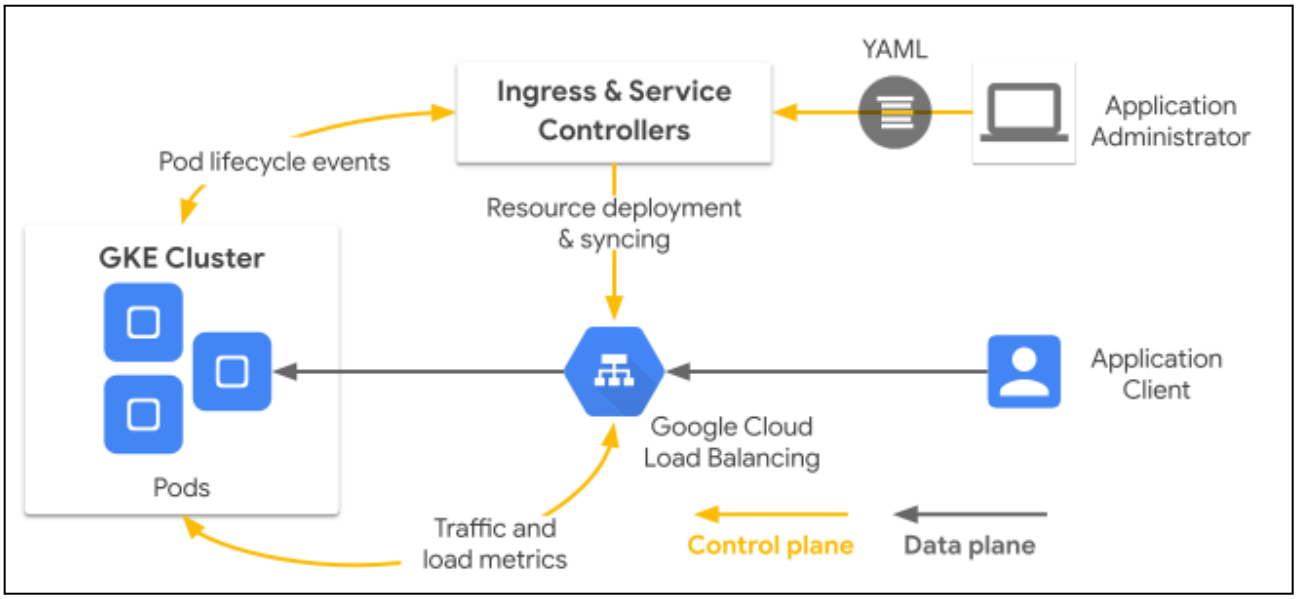


Figure 24: Google Cloud Load Balancing
(https://cloud.google.com/kubernetes-engine/docs/concepts/service-networking)

7. Future improvements

Integration with video/voice call

Recognizing the inherent challenges of relying solely on text-based chat for users during a whiteboard session, we aim to provide a more dynamic and interactive communication channel for users by seamlessly integrating video or voice calls in the future. This addition not only alleviates the tedium of typing but also fosters a richer and more nuanced form of interaction.

This feature could potentially be done with *WebRTC*.

Enhance users' matching preferences

In the future, we aspire to elevate the precision and personalization of our matching process by introducing additional criterions—Question Category and Coding Language. Recognizing the significance of tailoring user experiences to specific interests and preferences, this enhancement aims to create a more refined and personalized matching process. This enhancement would be particularly beneficial for users who wish to prioritize and delve deeper into specific question categories, providing them with a more targeted and meaningful interaction tailored to their distinct areas of interest while practicing specifically for the language of their choice.

This improvement potentially involves enhancing the frontend to allow users to select a question category, on top of selecting the question difficulty as well as sending the chosen category to the backend matching service.

Enhance management of question

Currently, we only provide the option to sort questions by difficulty. As such, a future implementation would be to give users the ability to sort by difficulty and categories. We should also provide them with the options to filter questions based on categories and difficulty. This would empower admins with greater flexibility and customization, allowing them to tailor their question management based on specific criteria and for easy viewing of questions.

Whiteboard state

Currently the whiteboard state is not preserved and the updates to the board are only received on the client when the whiteboard tab is selected. The board is also cleared if another tab is selected. A potential improvement is to store all updates in the background even if the whiteboard tab is not selected. This allows the state to be preserved as well.

Pagination of questions and users from Backend

Lastly, we hope to implement the technique of pagination for questions and users data. Backend

pagination is beneficial for optimizing resource usage, improving performance, and enhancing the user experience. By dividing the data into smaller, manageable chunks, backend pagination minimizes the load on our servers, contributing to better scalability. Users will experience a more responsive interface as only the necessary data is fetched and transmitted at any given time. This implementation aligns with our commitment to providing a seamless and efficient user experience on PeerPrep.

Enable HTTPS on GCP KBE deployment

In order to enhance security when using our web application, a future enhancement would be to enable *HTTPS* on our *GCP KBE* deployment through the use of *TLS/SSL* certificates. By providing a *HTTPS* connection, communication between *GCP KBE* and the client would be private and the data would not be readable by middle-men, thus preserving the confidentiality of any session or user data.

Collaboration Service Fix

Currently, collaboration only works on the same browser. We realize in real world use cases, many people may not be using the same browser, so there is a need to design the collaboration service in such a way that it has browser compatibility with various browsers, making sure the collaboration function works even with different browsers.

8. Work Distribution

Name	Contributions
Sub Group 1 - YunSeong - Yew Keng	Overall Responsibilities - Assignment 2 - Assignment 3 - Assignment 4 - Assignment 5 - N1 - Communication - N9 - Deployment - N10 - Scalability - N11 - API Gateway
Chung Yunseong	**Non-Technical** - Contributed to weekly sprint meetings **Technical** - Initialized *UserService* and connected it with *MongoDB* - Implemented the entire *QuestionService.* - Implemented the entire *ChatService* - Implemented the entire *AuthService* - Contributed in integrating *AuthService* with frontend - Contributed in integrating *UserService* with frontend - Contributed in enhancing *MatchingService* to make random room id generation, along with connecting it with question service to fetch random questions, and connecting it with history service to automatically generate history documents upon successful match. - Implemented *RoleGuards* and *JWT* tokens for authenticating requests to *UserService*, *QuestionService* and for persisting user sessions - Contributed in integrating *MatchingService* and *ChatService* with frontend with the initialization of sockets in frontend - Configure *RabbitMQ* in *MatchingService*, *ChatService*, *HistoryService* and *QuestionService* to facilitate communication between relevant services - Implemented *Google OAuth* in frontend and connected it with backend to allow users to sign in with their *Google* account

	- Connected chat service with *HistoryService* so that chat between users in the session is recorded in their history documents in real-time
Chew Yew Keng	**Non-Technical** - Contributed to weekly sprint meetings - Updated weekly *Jira* sprints and assigned tasks to individual developers - Populated question repository - Set up initial *GitHub* issues and assigned to individual developers **Technical** - Co-developed *UserService* with Yunseong - Created the *MatchingService* that uses sockets to connect two users that select the same difficulty, and times out otherwise after 30 seconds - Enhanced *QuestionService* through the creation of a new *UploadService*. This service provides image management and facilitates upload/deletion of images to the GCP *Peerprep-question* bucket and the return of the URL. Using this URL, diagrams and images are rendered together with questions, enabling a better understanding to users. - Dockerized *UserService, QuestionService, MatchingService, AuthService, UploadService* services and ensured they could run on Kubernetes Engine. - Created *GCP SQL* database for *UserService* and *MongoDB* database for QuestionService - Designed Schema for *GCP SQL database* to be used with *UserService* and for *MongoDB database* to be used with *QuestionService* - Connected *QuestionService* to *MongoDB* and *UserService to GCP SQL* - Add Email, Username and Password validators to *Sign Up* and *Profile* pages to ensure details specified are valid. - Update Question Schema to new requirements for images and examples format - Deployed *Docker images* to *GCP Kubernetes Engine (KBE)* for all services and Frontend - Set up *Horizontal Pod* auto-scaling for *GCP KBE* - Set up *Ingress Controller* with *GCP KBE* built in *Ingress controller* to redirect requests to relevant microservices

Sub Group 2 - Adriel - Bernice - Sen Wei	Overall Responsibilities: - Assignment 1 - Assignment 2 - Assignment 3 - Assignment 5 - N2 - History - N3 - Code Execution - N4 - Question Enhancement - N5 - Collab Enhancement
Lee Sen Wei	**Non-Technical** - Contributed to weekly sprint meetings - Created *GitHub* issues and *Jira* cards to facilitate project management - *Figma* Mockups for the frontend design - Brainstormed and developed Logo with Bernice **Technical** - Developed and designed the frontend admin dashboard with Bernice - Implemented delete question function for admins, integrated with backend - Created the code editor and whiteboard, providing support for multiple languages, and also different color and sizes for the whiteboard paint brush. - Implemented syntax highlighting for few languages with Bernice - Implemented code validation for 2 languages with Bernice - Implemented a collaborative code editor with *WebRTC* and *CDRT* on the frontend to ensure no race conditions when editing simultaneously - Implemented a collaborative whiteboard by creating a socket connection that taps onto matching service and listens for changes on other users' whiteboards - Implemented Code Execution for users to run their code, showing the error status codes when failed to compile or runtime error, the memory usage and the time taken to run - Enhanced question service by implementing sorting of questions based on difficulty level and ID - Implemented simple user session persistence, routing them to the login page if they logged out, and routing them to the dashboard if they were already previously logged in

	- Implemented the logging in, signing in and logging out, integrating it with the frontend and the backend with YunSeong - Developed the logic for storing the access and refresh tokens on the frontend, to facilitate multiple operations such as logging in, crud operations on questions and users - Developed *Axios* interceptor to handle *HTTP* errors
Bernice Priscilla Toh	**Non Technical** - Brainstormed and developed Logo with Senwei - *Figma* Mockups - Participated in weekly sprints **Technical** - Enhanced question service by implementing search for questions based on name on the frontend - Implemented add and edit question function for admins, integrated with the backend - Enhanced model of question for the frontend with Dylen - Implemented syntax highlighting for few languages with Sen Wei - Implemented code validation for 2 languages with Sen Wei - Implemented updating user profile functionality for the frontend and integrated with backend APIs (Change password, change username, ability to delete account) - Developed frontend matching flow (choosing difficulty -> waiting for match -> no match/ match found -> rematch) with YunSeong - Integrated matching flow with the appropriate backend API calls - Enhanced usability of application by implementing user feedback features, such as matching timer, no match found, exit matching, match found, error handling of frontend adding question, peer leaves room - Enhanced overall UI of application with Framer-motion - Implemented Sidebar navigation for admins and Navbar navigation for users for intuitive routing within the application. - Created Chat function for communication during a code session, integrating it with the backend API. - Developed overall structure of Session Page on the frontend

Adriel Soh	**Non Technical** - Contributed to weekly sprint meetings **Technical** - Implemented frontend *UserPage* for admins to be able to view all users - Implemented *UserList* to display all users in a nicely formatted manner - Implemented functionality on frontend for admins to edit/delete users - Implemented functionality to sort and search for users on frontend - Integrated frontend and backend for *UserService* - Developed *HistoryService* to store past attempts of users - Create *HistorySchema* for *HistoryService* to store relevant details regarding user's history - Implemented functionality on *HistoryService* to allow saving, editing and deleting of history documents - Implemented frontend *UserDashboardPage* for users to view their past attempts - Implemented *HistoryList* to display past attempts of questions for a user in a clean and neat manner - Worked together with YunSeong to modify *HistoryService* functions to save a new history document for each user once matched - Worked together with YunSeong to implement saving of chat history in real-time - Implement functionality to save code in real-time - Dockerized *HistoryService* - Integrate frontend and backend for *HistoryService* - Create *HistoryDialog* for user to have a summarized view of their past attempt - Implement *HistorySessionPage* for user to make edits to their previous submissions

9. Reflections

Reflecting on CS3219 Project *PeerPrep*, it is common consensus among all team members that we have gained valuable insights into both technical and interpersonal aspects. From a technical perspective, this project provided an immersive learning experience bridging the gap between theoretical knowledge and practical execution. An illustrative example of this transformative learning journey was our encounter with Microservice architecture—a concept unfamiliar to the majority of the team at the project's inception. The lecture on Microservice architectures not only introduced the theoretical underpinnings of microservice architectures but also guided us through the intricate process of implementing these principles in our project.

This project became a dynamic laboratory where we not only comprehended the importance of microservices but actively engaged in the process of architecting and implementing them. Implementing Microservice architecture highlighted the significance of decoupling our various services, allowing us to witness the benefits of scalability, flexibility, and resilience. Each service acted as a self-contained unit, contributing to a modular and maintainable codebase. This experience not only solidified our understanding of the theoretical advantages of Microservices but also illuminated the practical challenges associated with their implementation.

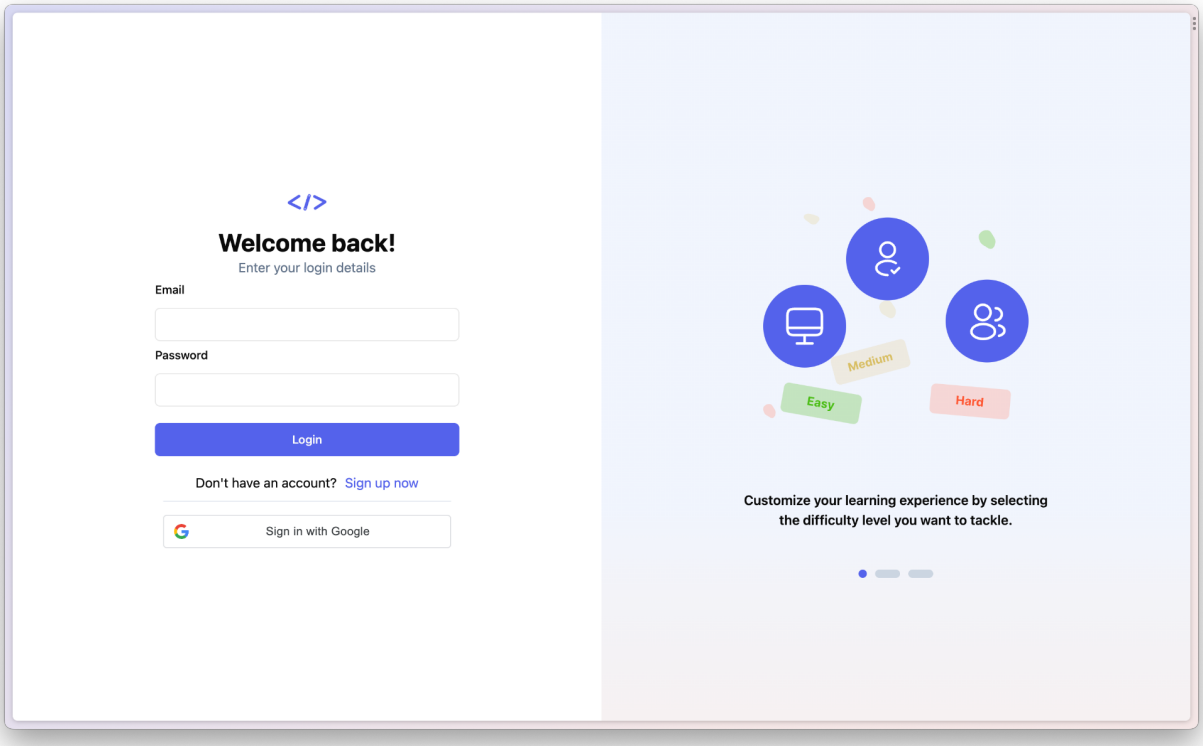
From an interpersonal perspective, the collaborative nature of the *PeerPrep* fostered effective communication and teamwork. Regular meetings provided a platform for us to update each other on our progress, brainstorm ideas, address our concerns and align our efforts towards shared goals. The importance of clear communication became evident in resolving challenges, ensuring that everyone's perspectives were considered and integrated into the decision-making process. For instance, the initial meetings to work on the Functional and Non-Functional Requirements proved to be important as it allowed us to collectively define the project scope, features we want to prioritize and set clear expectations.

During the initial meetings to establish functional and non-functional requirements, our team found itself at a crossroads regarding the priority of certain features. Some team members emphasized the immediate impact on user experience, while others underscored the long-term scalability of the proposed functionalities. This diversity of thought forced us to critically evaluate our assumptions, balance the pros and cons, and ultimately arrive at a consensus that balanced immediate needs with the overarching project vision. The collaborative nature of these discussions fostered an environment where dissent was valued, constructive dialogue was encouraged, and decisions were made collectively with a well-rounded perspective.

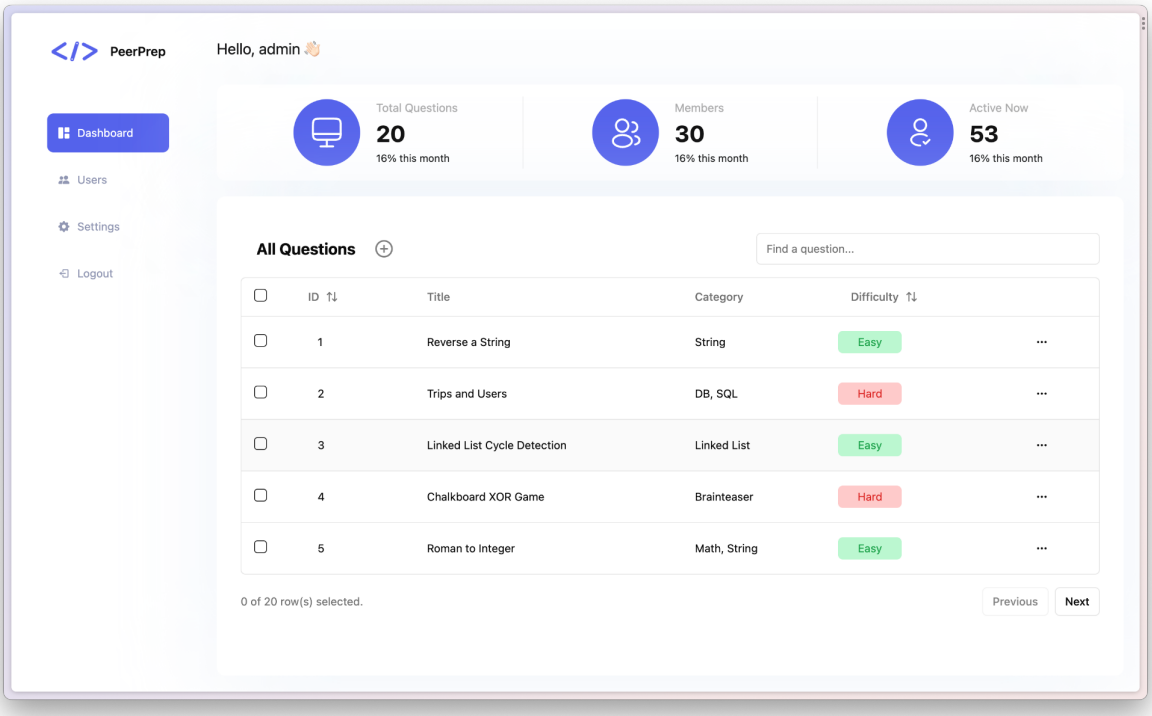
Project *PeerPrep* not only expanded our technical expertise but also honed our collaborative and communication skills. The journey from theory to execution was a transformative experience that

will undoubtedly shape our approach to future projects and foster continued growth as aspiring software engineers.

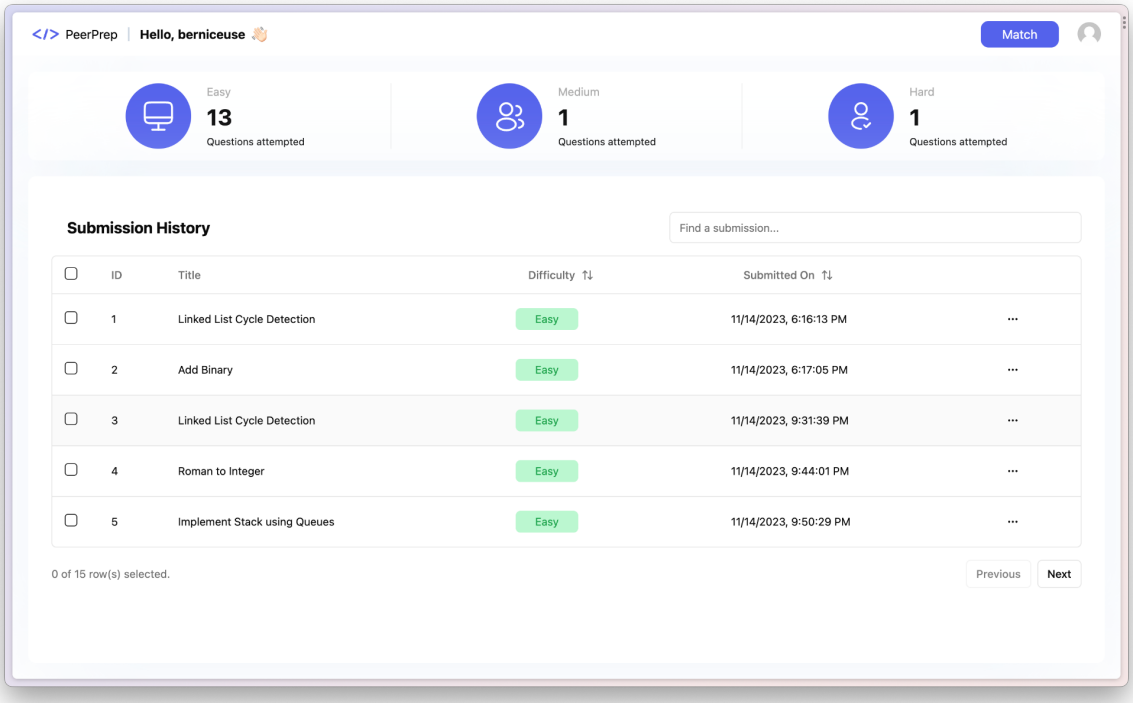
10. Appendix



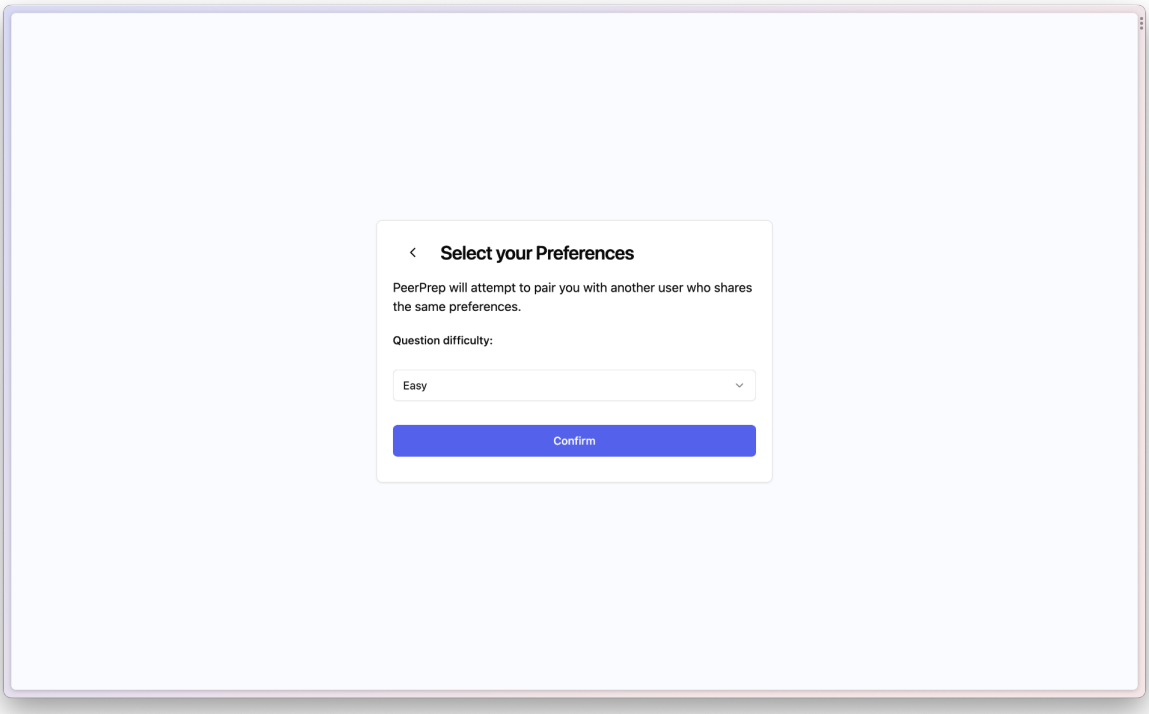
Page1: Login/Register Page



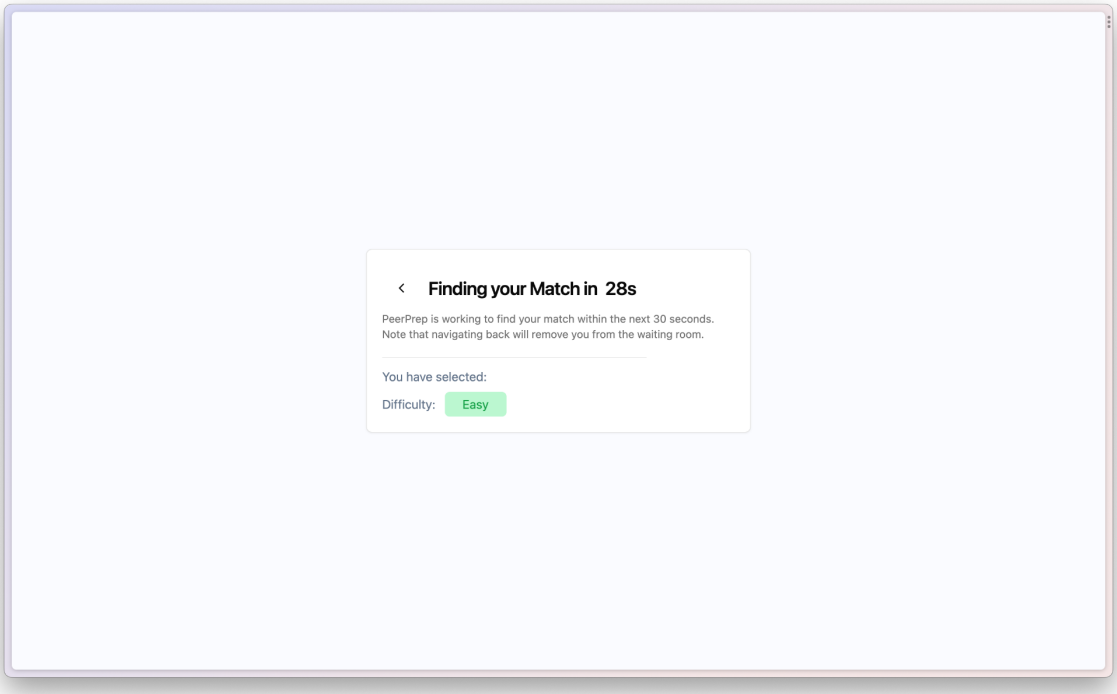
Page 2: Admin Page



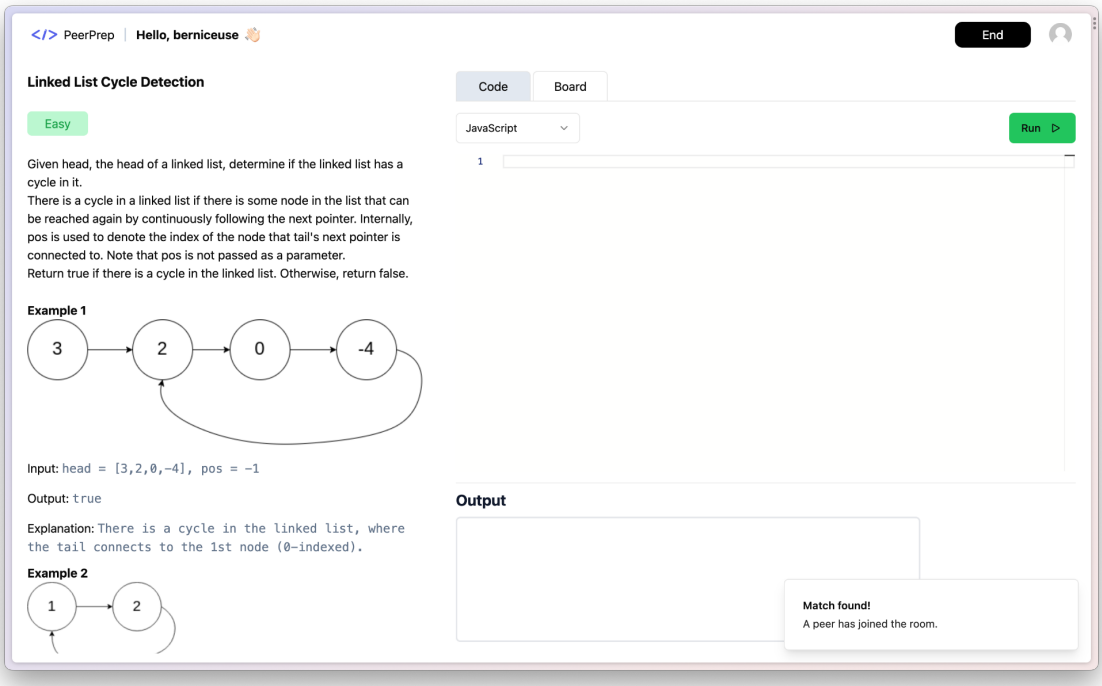
Page 3: User Dashboard Page



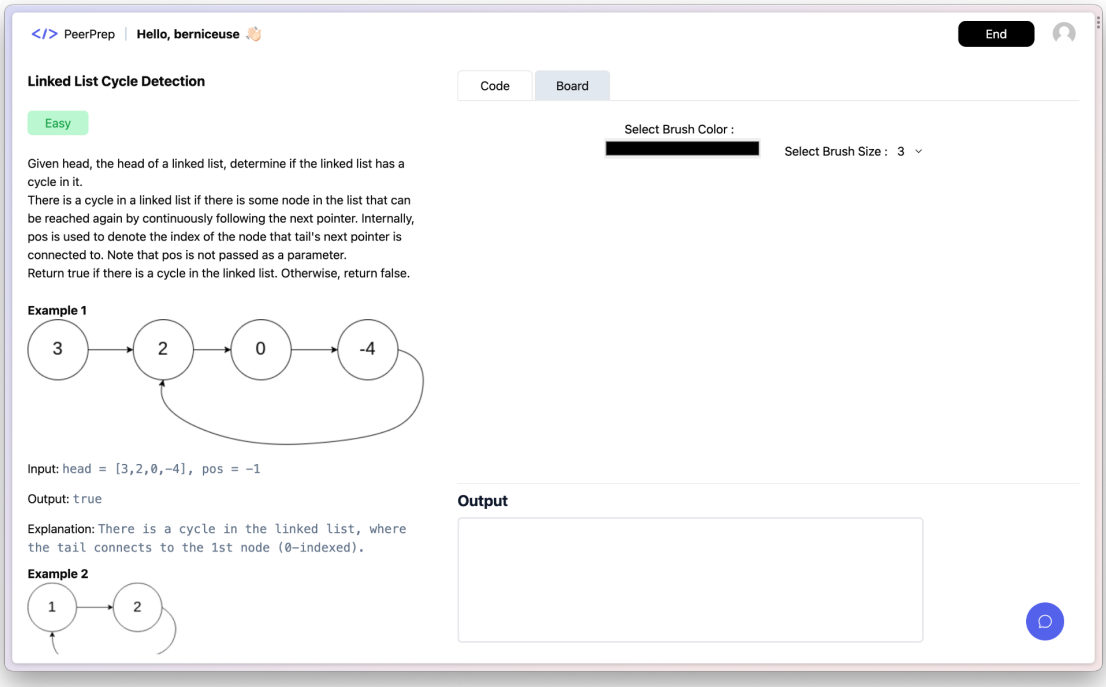
Page 4: Selecting Question Preferences Page



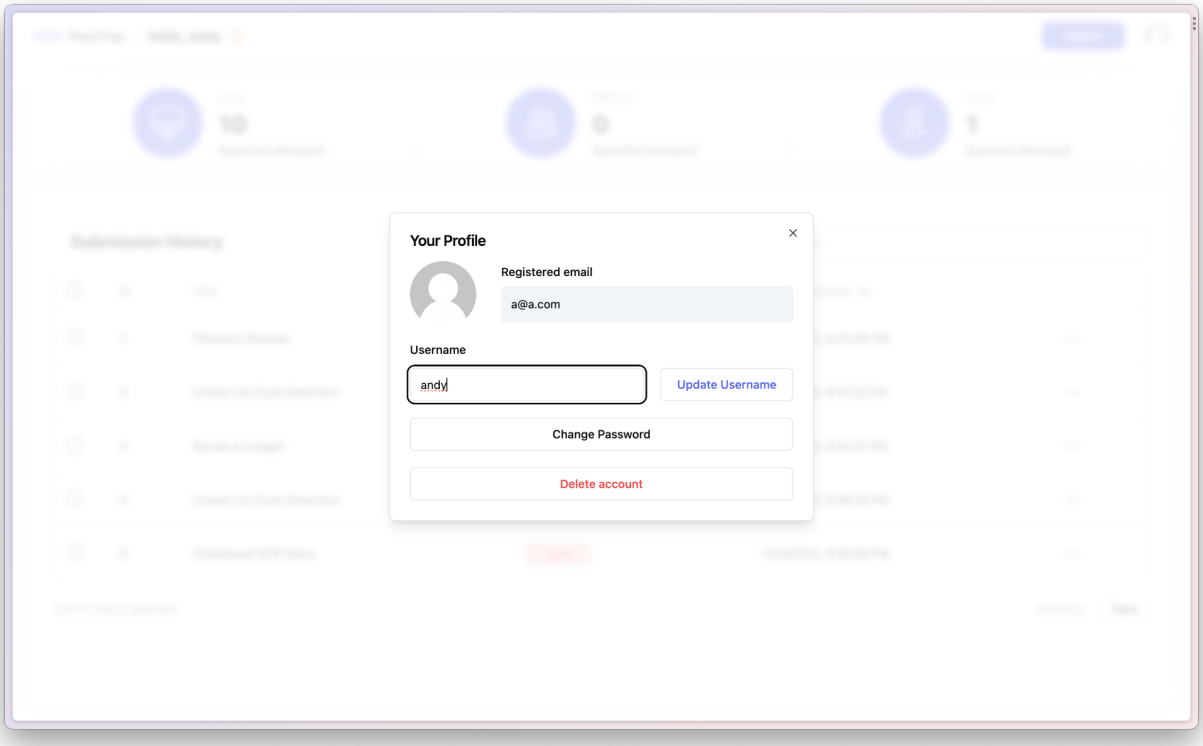
Page 5: Finding Match Page



Page 6: Session Page



Page 7: Whiteboard Page



Page 8: Settings Page