



# NUS

National University  
of Singapore

## CS3219: Project Report

Group 2

No.	Full Name (as in EduRec)	Student Number (Axxxx)
1	Ek Wei Rui	A0234714J
2	Ethan Jed Tan Jia En	A0235290J
3	Kenvyn Kwek Shiu Chien	A0261523M
4	Tan Jia Rong	A0233713M
5	Tan Qin Xu	A0213002J

## Table of Contents

<b>Declaration.....</b>	<b>8</b>
<b>1. Project Introduction.....</b>	<b>9</b>
1.1. Background.....	9
1.2. Our Project and Rationale.....	9
1.3. Generic User Flow.....	9
<b>2. Individual Contributions.....</b>	<b>11</b>
<b>3. Project scope (Product backlog).....</b>	<b>13</b>
3.1. Functional Requirements (+ Fulfillment).....	13
3.2. Non-Functional Requirements.....	18
3.3. Selected Nice-To-Haves.....	20
<b>4. Architecture Overview.....</b>	<b>21</b>
4.1. Tech Stack.....	21
4.2. SPA Model.....	22
4.3. Microservice Architecture.....	23
4.3.1. Microservices.....	24
4.3.2. RESTful APIs.....	24
4.4. Database.....	25
4.5. Design Patterns Employed.....	26
4.5.1. Model-View-Controller (MVC) Pattern.....	26
4.5.2. Publisher-Subscriber (Pub-Sub) Pattern.....	27
4.5.2.1. Socket.io's Pub-Sub Mechanism in collaboration service.....	27
4.5.3. Observer Pattern.....	28
4.5.3.1. Observer Pattern in Collaboration Service.....	28
<b>5. User Service.....</b>	<b>30</b>
5.1. Architecture Design.....	30
5.1.1. Authentication service.....	31
5.2. Internal Microservice Design.....	31
5.2.1. Auth Middleware.....	32
5.2.2. Design Considerations.....	33
5.2.2.1. Handling JWTs on Frontend.....	33
5.2.2.2. User Data Management.....	34
5.2.2.3. Username Validation & Verification.....	37
5.2.2.4. Role Management (Admin Check).....	38
5.2.2.5. Authentication Middleware.....	38
<b>6. Question Service.....</b>	<b>40</b>
6.1. Architecture Design.....	40

6.1.1. Key Architectural Components:.....	40
6.2. Internal Microservice Design.....	41
6.2.1. Core Responsibilities:.....	41
6.2.2. Internal Design Flow:.....	42
6.2.2.1. Internal Design Flow Example:.....	43
6.2.3. User and Admin Privileges.....	43
6.2.3.1. Frontend.....	44
6.2.3.2. Backend.....	45
6.2.4. Design Considerations.....	45
6.2.4.1. Pagination Support.....	45
6.2.4.2. Caching.....	45
<b>7. Matching Service.....</b>	<b>47</b>
7.1. Architecture Design.....	47
7.1.1. Matching Criteria.....	47
7.1.2. Matching Service Tech Stack.....	48
7.1.3. Relevant NFRs.....	49
7.2. Internal Microservice Design.....	50
7.2.1. Design Considerations.....	52
7.2.1.1. Creating a session - choice of session id.....	52
<b>8. Collaboration Service.....</b>	<b>53</b>
8.1. Architecture Design.....	53
8.1.1. Socket.io for Real-Time Synchronization.....	53
8.1.2. Room-Based Collaboration using Session ID.....	53
8.1.3. Monaco Code Editor.....	54
8.2. Internal Microservice Design.....	54
8.2.1. Design Considerations.....	55
8.2.1.1. Integration with Matching Service.....	55
8.2.1.2. Use of Socket.io in Real-Time Communication and Code Editor Synchronization.....	57
8.2.1.3. Monaco Code Editor.....	59
8.2.1.4. Code Execution.....	60
<b>9. History Service.....</b>	<b>62</b>
9.1. Architecture Design.....	62
9.1.1. Key Architectural Components:.....	63
9.1.2. Why Orchestrator.....	63
9.2. Internal Microservice Design.....	63
9.2.1. Core Responsibilities:.....	63






9.2.2. Internal Design Flow:.....	64
9.2.2.1. Internal Design Flow Example:.....	65
9.2.3. Design Considerations.....	66
9.2.3.1. Storing of questions attempted.....	66
9.2.3.2. Handling deleted questions in users history.....	66
9.2.3.3. Criteria for marking a question as attempted.....	67
<b>10. API Documentation.....</b>	<b>68</b>
10.1. User Service.....	68
10.2. Question Service.....	69
10.3. Matching Service.....	70
10.4. Collaboration Service.....	70
10.5. History Service.....	71
<b>11. Project Management.....</b>	<b>72</b>
11.1. Product Backlog.....	72
11.2. GitHub.....	72
11.3. Agile Framework.....	72
11.4. Gantt Chart.....	73
<b>12. Conclusion.....</b>	<b>74</b>
12.1. Future Enhancements.....	74
12.1.1. Test Cases.....	74
12.1.2. Loose Matching Criteria.....	74
12.1.3. Soft Deletion of Questions.....	74
12.1.4. Collaborative Whiteboard and Diagramming Tools.....	74
12.2. Reflections.....	75
12.2.1. Technical Growth.....	75
12.2.2. Collaboration and Communication.....	75
12.2.3. Adaptability and Problem-Solving.....	75
12.3. Acknowledgement.....	76

## Declaration

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
Ek Wei Rui		29/9/24
Ethan Jed Tan Jia En		29/9/24
Kenvyn Kwek Shiu Chien		29/9/24
Tan Jia Rong		29/9/24
Tan Qin Xu		29/9/24

# 1. Project Introduction

## 1.1. Background

Technical interviews have become unavoidable in the job application process, and the range of topics have only become more varied. For the tech industry, these assessments are necessary in securing new employees with strong coding foundations and problem-solving skills. As a result, there has been no shortage of websites that have come about for the sole purpose of technical interview preparation.

## 1.2. Our Project and Rationale

**PeerPrep** is a web application that allows users to practice common and popular programming problems. Like other websites with similar core purposes, users are able to select questions based on their difficulty levels and focus areas. However, in addition to these core features, **PeerPrep** was developed with additional functionality in mind, with the aim of promoting collaborative learning and communication. The application is specifically designed to have two users work together in a common coding environment to solve coding problems. Through real-time interactions, users of various strengths and abilities can come together, exchanging styles and approaches, simulate real-world collaborative problem-solving, and learn to express their ideas to each other clearly.

## 1.3. Generic User Flow

A student, perhaps looking to prepare for upcoming technical interviews, discovers our website while looking for a place to practice.

An account is created and the student logs in.

The student navigates to the Home Page, which lists all the questions available for him/her to practice on.

On the same page, there is a matching menu through which the student can search for a collaboration partner to tackle a programming problem with, based on focus areas and difficulties.

Once the student starts his search, he will be matched with another student with the same search criteria, upon which both students will enter a joint session. If no match is found within 30 seconds, the search will be aborted.

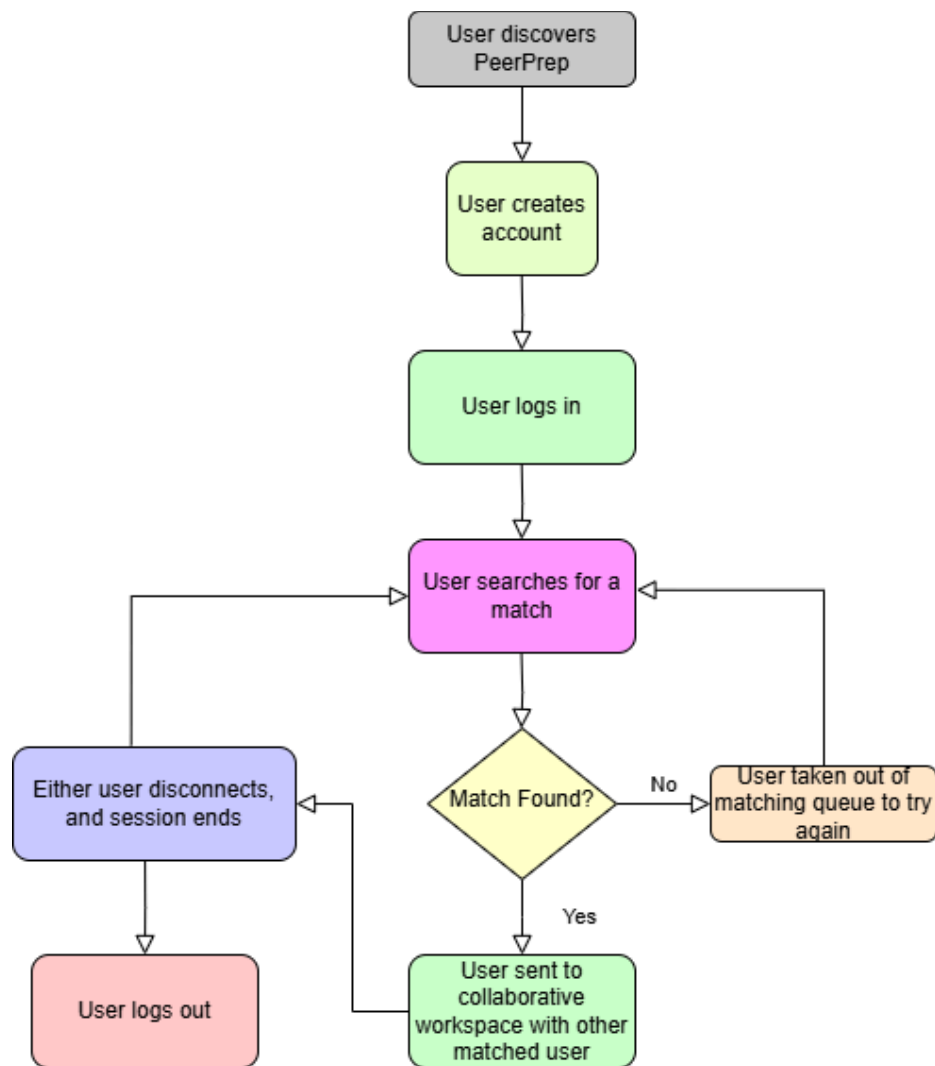
In this collaborative session phase, both students are directed to a page with the question, description, constraints and examples, a chat box, and a coding space and an output terminal from which written code can be executed. The students can then proceed to discuss their ideas and implementations, and tackle the problem together. Attempts run by the students will be saved in their history for future reference.

The student can choose to leave the session at any time and he will be directed back to the homepage where he can continue searching for other partners and questions. Meanwhile, the other user remains in the session until he chooses to leave himself.

The student logs out when he is finished.

Through this core usage loop, the students slowly accumulate their abilities and techniques, and can eventually better approach technical interviews.

Below is a diagram of this user flow.

**Fig 1: User Flow Diagram**



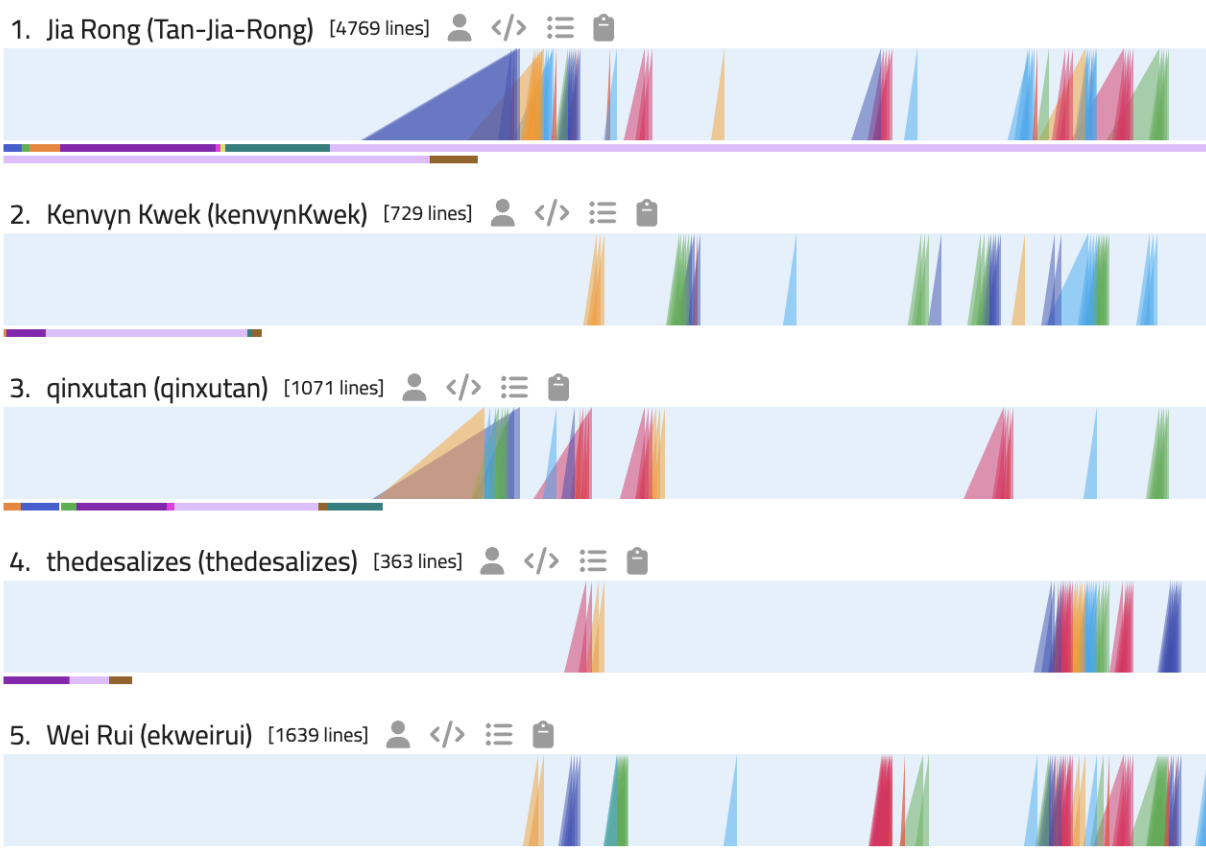
## 2. Individual Contributions

The allocation of responsibilities among team members is based on their expertise and interest in the services to be developed. We identify individual microservices of the project, ensuring clear ownership to minimize conflicts and streamline the development process. This enables each developer to work on their assigned tasks independently, fostering progress without significant disruptions. As the development advances, team members working on related services collaborate closely to integrate their work and ensure compatibility.

No	Full Name (as in Edu Rec)	Contributions ( <i>write point wise for different components</i> ). <i>Extend the table as needed.</i>
1	Ek Wei Rui	a. Frontend: <ol style="list-style-type: none"> <li>1. Matching Page: Timer and User Alerts</li> </ol> b. Backend: <ol style="list-style-type: none"> <li>1. Question Service: Lead implementation</li> <li>2. Matching Service: Matching of Users &amp; Question to Session, Integrate in Redis &amp; RabbitMQ</li> <li>3. User Service: Authentication of Token</li> <li>4. History Service: Lead implementation</li> </ol> c. Dockerize entire application d. Deploy Website
2	Ethan Jed Tan Jia En	a. Frontend: <ol style="list-style-type: none"> <li>1. User Service: Hide administrative components from regular users, User greeting message</li> </ol> b. Backend: <ol style="list-style-type: none"> <li>1. User Service: Users collection, isAdminUser middleware</li> </ol>
3	Kenvyn Kwek Shiu Chien	a. Frontend: <ol style="list-style-type: none"> <li>1. Create Account Page: Lead implementation</li> <li>2. Collab Page: Lead implementation</li> </ol> b. Backend: <ol style="list-style-type: none"> <li>1. User Service: Firebase Account Creation</li> <li>2. Collab Service: Code editor and Code Execution</li> </ol>
4	Tan Jia Rong	a. Frontend: <ol style="list-style-type: none"> <li>1. Question Page: Lead implementation</li> <li>2. History Page: Lead implementation</li> </ol> b. Backend: <ol style="list-style-type: none"> <li>1. Question Service: Firebase backend</li> </ol>

		2. Matching Service: Firebase backend
5	Tan Qin Xu	a. Frontend: <ol style="list-style-type: none"> <li>1. Home Page: Lead implementation</li> <li>2. Login Page: Lead implementation</li> </ol> b. Backend: <ol style="list-style-type: none"> <li>1. User Service: Firebase authentication</li> <li>2. Collab Service: Socket io implementation, Chatbox</li> <li>3. Matching Service: Linked matching service to collab service</li> </ol>

Details of our individual code contributions are available in this [dashboard](#)



**Fig 2: Reposense Code Contributions**

### 3. Project scope (Product backlog)

#### 3.1. Functional Requirements (+ Fulfillment)

Functional Requirements	Priorit y	Planne d sprint/i teratio n	PR
F1: User Service – responsible for user profile management.			
F1.1: Creation of User Accounts			[Link to PR or Commit]
The system should allow users to sign up for an account with the following information. • Username • Email • Password	High	29 Sep 2024	<a href="#">[PR30]</a>
The system should ensure that every account created has a unique email and username to prevent duplicate accounts.	High	29 Sep 2024	<a href="#">[PR30]</a>
The system should validate the password during account creation to ensure that it meets security requirements • Minimum length: 8	High	29 Sep 2024	<a href="#">[PR30]</a>
The system should validate for correct email format during account creation	Low	29 Sep 2024	<a href="#">[PR30]</a>
F1.2. Logging in of User Accounts			
The system should allow users to login with their registered email and password.	High	29 Sep 2024	<a href="#">[PR30]</a>
The system should prevent users from logging in with invalid credentials. • Incorrect Email • Incorrect Password	High	29 Sep 2024	<a href="#">[PR30]</a>
The system should allow users to logout of their account.	High	29 Sep 2024	<a href="#">[PR30]</a>
The system should maintain the user's login status until: • User logs out manually • Login session expires	Low	29 Sep 2024	<a href="#">[PR86]</a>
The system should provide visual feedback for unsuccessful login attempts.	Low	29 Sep 2024	<a href="#">[PR30]</a>

F1.3. Deletion of User Accounts			
The system should allow users to delete their accounts.	High	29 Sep 2024	<a href="#">[PR30]</a>
The system should prompt the users for confirmation before deletion warning that the action is an irreversible one.	High	29 Sep 2024	<a href="#">[PR30]</a>
The system should delete any associated data from the database upon account deletion.	High	28 Oct 2024	<a href="#">[PR82]</a>
The system should automatically log out the user upon successful deletion.	High	29 Sep 2024	<a href="#">[PR30]</a>
F2. Matching Service – responsible for matching users based on some criteria			
F2.1. User defined matching criteria			
The system should allow users to select multiple criteria for matching, namely difficulty level and topic.	High	20 Oct 2024	<a href="#">[PR47]</a>
The system should require users to select a difficulty level (easy, medium, hard) as a mandatory field for the matching process.	High	20 Oct 2024	<a href="#">[PR47]</a>
F2.2. Matching based on multiple criteria			
The system should consider all criteria (difficulty and topic) during the matching process, matching users with exact match in the criteria.	High	20 Oct 2024	<a href="#">[PR47]</a>
F2.3. Matching process			
The system should show how long a user has been waiting in the queue.	High	20 Oct 2024	<a href="#">[PR47]</a>
The system should timeout a user if the matching process exceeds 30 seconds.	High	20 Oct 2024	<a href="#">[PR47]</a>
The system should allow users to cancel the matchmaking process at any time while still in the queue.	Medium	20 Oct 2024	<a href="#">[PR47]</a>
F2.4. Post-Match Interaction			
The system should allocate the same question to matched users to ensure consistency during the session.	High	20 Oct 2024	<a href="#">[PR47]</a>
The system should inform the user if a match is found.	High	20 Oct 2024	<a href="#">[PR47]</a>
The system should inform the user if no match is found after 30 seconds.	Medium	20 Oct 2024	<a href="#">[PR47]</a>

F3. Question Service – responsible for maintaining a question repository indexed by difficulty level and specific topics			
F3.1. Question repository maintenance			
The system should be seeded with a list of questions.	High	29 Sep 2024	<a href="#">[PR18]</a>
The system should classify each question into the respective topics, and then into the different difficulty levels (easy, medium, hard).	High	29 Sep 2024	<a href="#">[PR18]</a>
F3.2. Creating Questions			
The system should allow only administrators to upload a question via the website, along with fields such as: <ul style="list-style-type: none"> <li>• title</li> <li>• description</li> <li>• difficulty</li> <li>• topics</li> </ul>	High	29 Sep 2024	<a href="#">[PR83]</a>
F3.3. Reading Questions			
The system should allow users to get question data from the database.	High	29 Sep 2024	<a href="#">[PR18]</a>
F3.3. Updating Questions			
The system should allow only administrators to update the following fields of existing questions in the database: <ul style="list-style-type: none"> <li>• title</li> <li>• description</li> <li>• difficulty</li> <li>• topics</li> </ul>	High	29 Sep 2024	<a href="#">[PR83]</a>
F3.4. Deleting Questions			
The system should allow only administrators to delete questions from the question repository.	High	29 Sep 2024	<a href="#">[PR83]</a>
F3.5. Question assignment			
The system should select a random question that satisfies the condition provided during the matching process.	High	20 Oct 2024	<a href="#">[PR47]</a>
F3.6. Question Bank and Filtering			
The system should have the question bank visible for the users.	High	29 Sep 2024	<a href="#">[PR22]</a>
The system should have the question bank with administrative control visible only for the admin.	High	28 Oct 2024	<a href="#">[PR80]</a>

The system should allow users to filter questions by • Difficulty Level, and/or • Topics	High	29 Sep 2024	<a href="#">[PR22]</a>
The system should allow users to view question descriptions from the question bank.	Low	29 Sep 2024	<a href="#">[PR29]</a>
F4. Collaboration Service – provides the mechanism for real-time collaboration (e.g., concurrent code editing) between the authenticated and matched users in the collaborative space			
F4.1. Real time, concurrent code editing in a collaborative environment			
The system should allow multiple users to edit the same codebase concurrently.	High	27 Oct 2024	<a href="#">[PR37]</a> & <a href="#">[PR#51]</a>
The system should provide real-time updates to the codebase when it is edited by either party.	High	27 Oct 2024	<a href="#">[PR37]</a> & <a href="#">[PR#51]</a>
The system should display the question in the collaborative space.	High	27 Oct 2024	<a href="#">[PR55]</a>
The system should enable each user to run their code independently and display the results separately.	Medium	27 Oct 2024	<a href="#">[PR52]</a>
F4.2. Real time communication between matched users			
The system should support text messaging between matched users for better collaboration	High	27 Oct 2024	<a href="#">[PR45]</a>
F4.3. Termination of Collaboration Space			
The system should allow either party to terminate the session for himself/herself.	High	27 Oct 2024	<a href="#">[PR71]</a>
The system should inform the user when the other user chooses to terminate the session.	High	27 Oct 2024	<a href="#">[PR71]</a>
F5. History Service – responsible for tracking and maintaining a record of all questions attempted by each user			
F5.1. Creation of Question History			
The system should create a new record for each question attempted during a collaboration session, including the following fields: • questionUid • dateAttempted • codeWritten	High	7 Nov 2024	<a href="#">[PR81]</a>
F5.2. Reading of Question History			
The system should allow users to retrieve a list of all questions they have attempted from the database.	High	7 Nov 2024	<a href="#">[PR81]</a>
F5.3. Updating Question History			

The system should update the codeWritten field for the corresponding question attempt record each time the "Run Code" button is pressed.	High	7 Nov 2024	<a href="#">[PR81]</a>
F5.4. Deleting Question History			
The system should automatically delete question attempt records that are older than 180 days.	High	7 Nov 2024	<a href="#">[PR81]</a>

## 3.2. Non-Functional Requirements

Non-Functional Requirements	Priority	Planned sprint/iteration
N1: Performance		
N1.1. Performance of Matching System		
The system should match users based on their criterias within 5 seconds if a match is found.	High	27 Oct 2024
The system should return the uid of the matched user also within that 5 seconds.	High	27 Oct 2024
N1.2. Performance of Question Repository		
The system should search and get a question for the specified topic and then the specified difficulty within 1 second.	High	29 Sep 2024
The system should return the retrieved question to the users within 1 second.	High	29 Sep 2024
The system should search for the specified topic and then the specified difficulty to store the question within 1 second.	High	29 Sep 2024
The system should return the successfully stored message to administrators within 1 second.	High	29 Sep 2024
N2: Usability		
N2.1. Satisfaction		
The system should enable users to initiate matchmaking with a single click from the homepage.	High	27 Oct 2024
The UI should be intuitive and easy to understand, allowing users to quickly grasp its functionality.	High	29 Sep 2024
The UI should be fully responsive and adapt seamlessly to a 13' inch screen size.	High	29 Sep 2024
The UI should adhere to a consistent and visually appealing color scheme throughout the application.	High	29 Sep 2024
N3: Security		
N3.1. Confidentiality of User information		
The system should be encrypted for confidential data in the database or in the browser itself to protect sensitive information. • Password	High	29 Sep 2024



N4: Scalability		
N4.1. Scalability in terms of users		
The system should be able to handle a huge number of users of up to 10000 without performance degradation.	High	29 Sep 2024
The system should be able to match users within 3 seconds even with a huge number of users of up to 10000 without performance degradation.	High	29 Sep 2024
N4.2. Scalability in terms of question repository		
The system should be able to support the addition of up to 1 million questions into the question repository without performance degradation.	High	29 Sep 2024
The system should be able to search and retrieve the required question from a database of up to 1 million questions within 1 second.	High	29 Sep 2024
N4.3. Scalability in terms of collaborative space		
The system should be able to support up to 5000 concurrent collaborative sessions without performance degradation.	High	27 Oct 2024

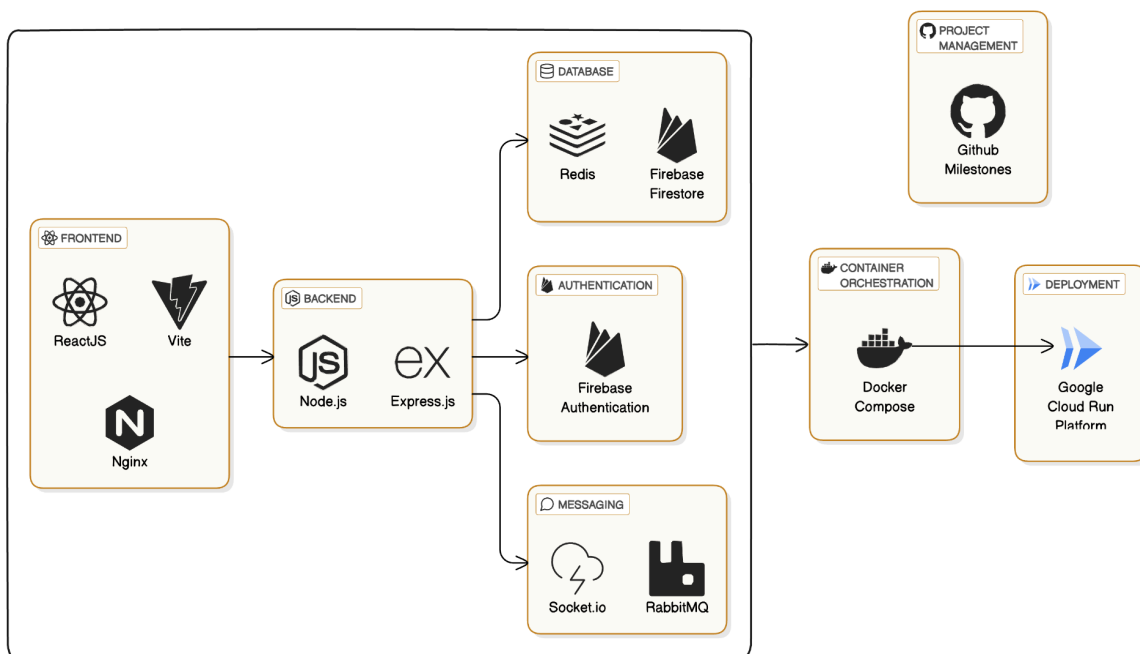
### 3.3. Selected Nice-To-Haves

Nice to have features	Description (Based on project document)	What the team has accomplished
N1: Communication	Implement a mechanism to facilitate communication among the participants in the collaborative space (other than the shared workspace) e.g., text-based chat service and/or video (+voice) calling service.	Implement a mechanism to facilitate communication among the participants in the collaborative space (other than the shared workspace) via a <b>text-based chat service</b> .
N2: History	Maintain a record of the questions attempted by the user e.g., maintain a list of questions attempted along with the date-time of attempt, the attempt itself and/or suggested solutions.	Maintain a record of the questions attempted by the user by maintaining a <b>list of questions attempted</b> along with the <b>date-time of attempt</b> , and the <b>attempt itself</b> .
N3: Code execution	Implement a mechanism to execute attempted solution/code in a sandboxed environment and retrieve and present the results in the collaborative workspace.	Implement a mechanism to execute attempted solution/code in a sandboxed environment and retrieve and present the results in the collaborative workspace.
N4: Enhance collaboration service	Providing an improved code editor with code formatting, syntax highlighting for one language, syntax highlighting for multiple languages.	Provide an improved code editor with code formatting, and syntax highlighting for <b>multiple languages</b> .
N7: Deployment	Deploy the app on a production system (AWS/GCP cloud platform).	Deploy the app on <b>GCP cloud platform</b> .

## 4. Architecture Overview

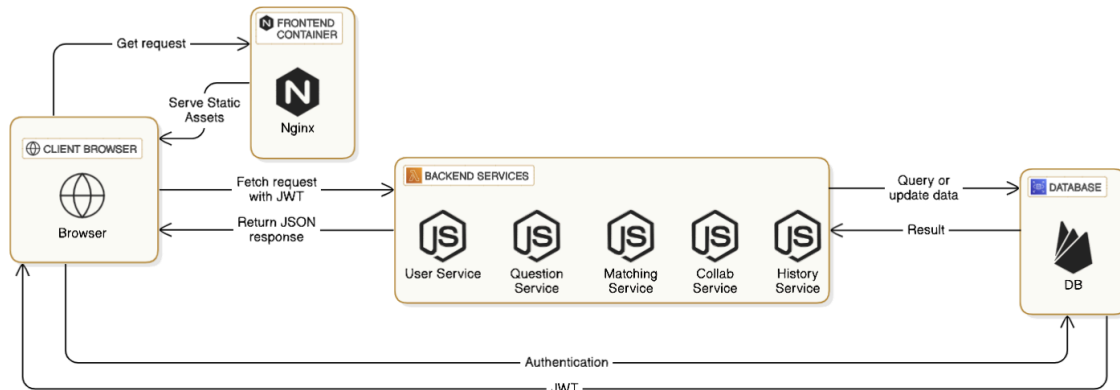
### 4.1. Tech Stack

Component	Technologies
Frontend	ReactJS, Vite, Nginx
Backend	Node.js, Express.js
Database	Firebase Firestore, Redis
Deployment	Google Cloud Run Platform
Authentication	Firebase Authentication
Messaging	Socket.io, RabbitMQ
Container Orchestration	Docker Compose
Project Management	Github Milestones



**Fig 3: Tech Stack Architecture Diagram**

## 4.2. SPA Model

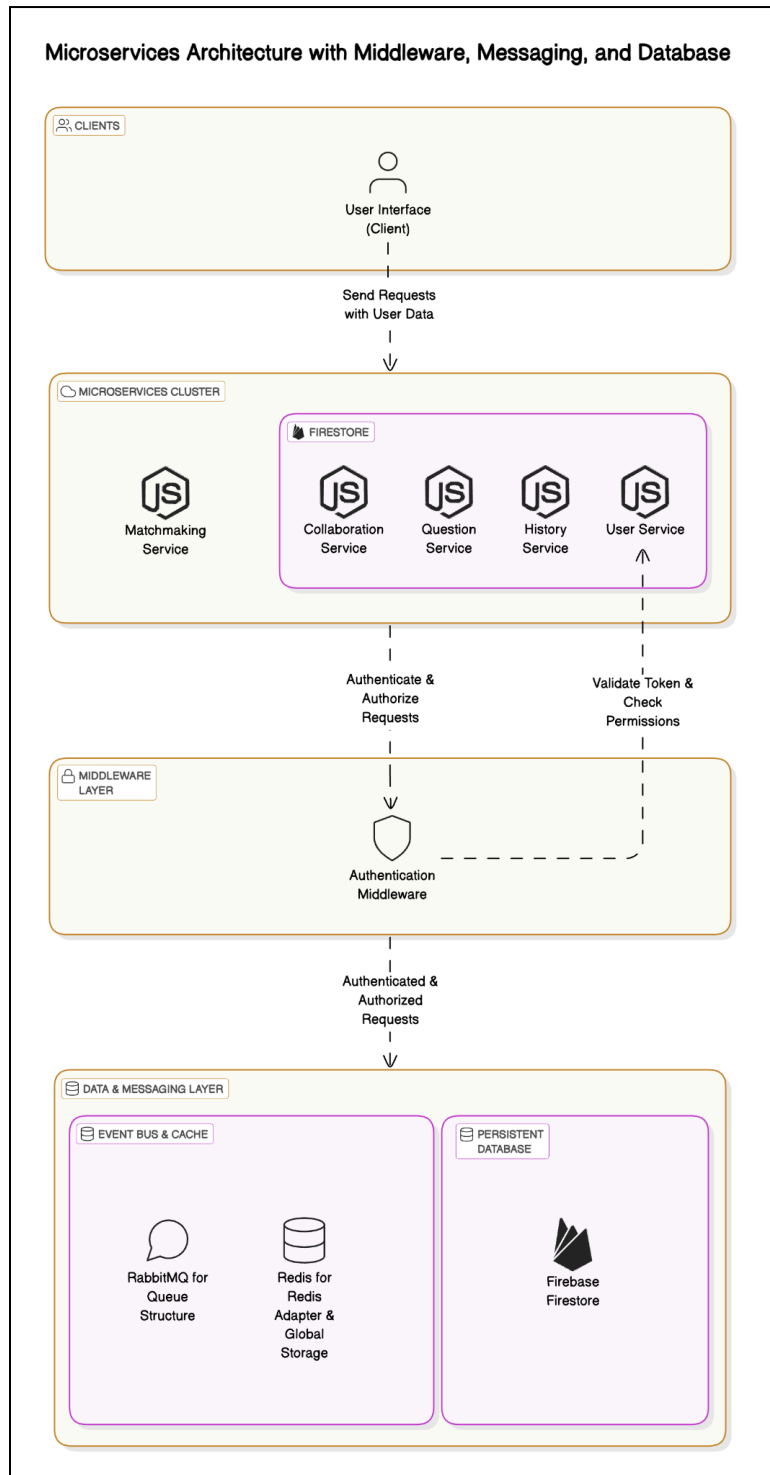


**Fig 4: SPA design diagram**

Our architecture follows a single-page application (SPA) model, where Nginx serves as the web server within the frontend container, handling HTTP requests and delivering the static assets generated by the build process to the client.

Once the SPA is loaded in the client browser, it relies on client-side routing and JavaScript to manage functionality. Authentication is handled by sending requests from the frontend to Firebase, which returns a JWT for subsequent backend requests. When backend services are needed, the frontend makes fetch requests to the relevant service, passing along the JWT for authentication. The backend responds with JSON data, which JavaScript in the client's browser then processes to display or interact with as needed.

## 4.3. Microservice Architecture



**Fig 5: Peerprep's Microservice Architecture**

### 4.3.1. Microservices

Our architecture adopts a microservices approach, where the system is broken down into smaller, independent services, each dedicated to a specific problem domain.

The modular structure of microservices also enabled us to develop, deploy, and scale each service independently based on its specific demand, enhancing maintainability, flexibility and fault tolerance.

Additionally, this decoupling also enabled our team to work effectively in parallel in our respective services — such as the Matching and Collaboration services — knowing that we can simply call the predefined API endpoints to retrieve the necessary data.

### 4.3.2. RESTful APIs

Our architecture utilizes RESTful APIs to facilitate communication both between the client and server, as well as between microservices.

RESTful APIs offer a stateless, scalable, and uniform interface for interacting with services, enabling communication in a simple and consistent manner.

This allows for seamless integration of new services and systems. By adhering to REST principles, we ensure that our system remains flexible and scalable, allowing it to evolve and accommodate future changes without disrupting existing functionality.

## 4.4. Database

We chose Firebase Firestore as our database over other databases based on the following reasons:

### NoSQL

Firebase Firestore is a NoSQL database, which is ideal for our needs due to its flexibility in handling unstructured data and supporting rapid development cycles. We prefer using a NoSQL database because our data doesn't have many relationships, and we don't need to perform complex queries. Firestore stores data in a JSON-like format, allowing for dynamic schema management and scalability without rigid table structures. This flexibility is particularly beneficial as our data requirements evolve over time. For example, when creating user accounts, we initially overlooked the need for a field to manage administrative permissions. With Firestore's NoSQL structure, we were able to easily update existing user documents to include this new field without requiring complex migrations or downtime.

### Scaling

Perhaps the most important reason for choosing a NoSQL database like Firebase Firestore is its ability to support horizontal scaling, which is a key strength of NoSQL databases due to their support for database sharding.

This contrasts with SQL databases, which are not designed for horizontal scaling. SQL databases often require multiple join operations when performing queries, making it inefficient to share data across multiple servers. The need to join data from different servers before executing queries significantly reduces performance, making it hard to horizontally scale SQL databases.

### Security

Apart from databases, Firebase provides robust built-in security features, including Firestore Security Rules and Firebase Authentication, which handle critical aspects of securing user data and managing access control. This allows us to focus our efforts on core application development, rather than spending time building and maintaining custom security systems.

### Serverless Architecture

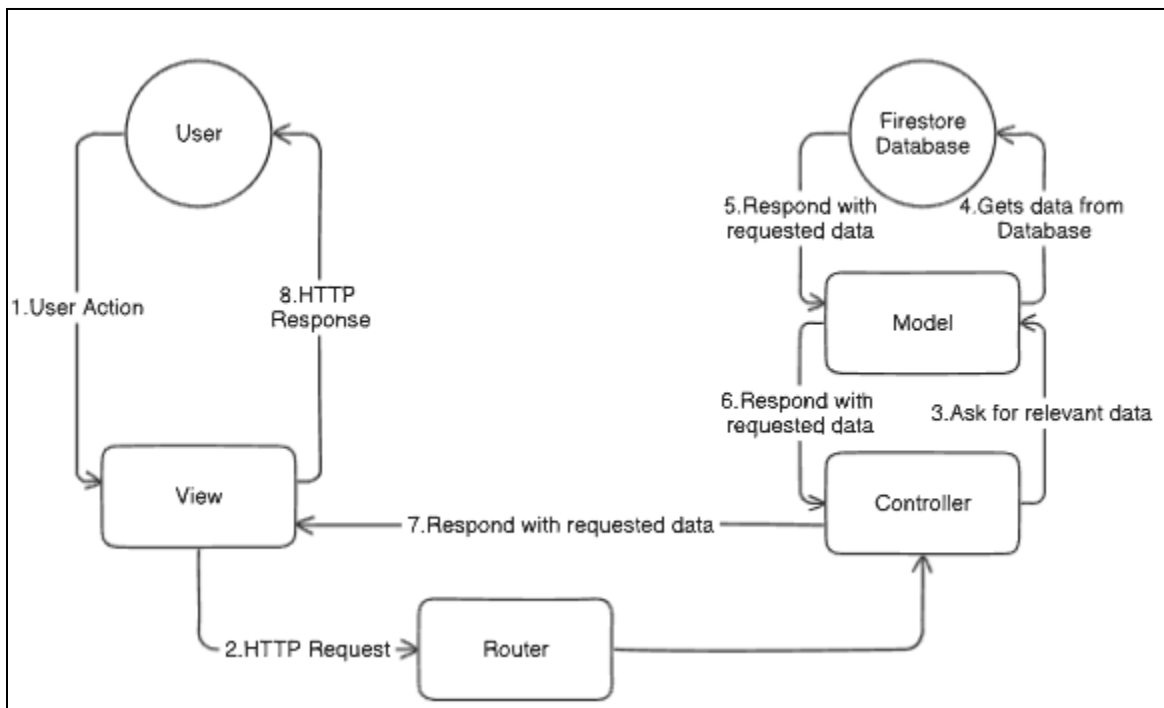
Firebase operates on a serverless architecture, meaning we do not have to manage or maintain servers. This greatly simplifies infrastructure management and allows for automatic scaling as the app grows. This helps to reduce operational overhead, increases reliability, and allows our team to focus on core functionality.

## Google Cloud Credits

The module provides \$50 Google Cloud credits for each team member. Since Firebase is part of the Google Cloud ecosystem, we decided to use it since it supports our use case and it can seamlessly integrate it with other GCP services. The credits also allows us to experiment not only with Firebase's capabilities but also with the Google Cloud tools without worrying about upfront costs.

## 4.5. Design Patterns Employed

### 4.5.1. Model-View-Controller (MVC) Pattern



**Fig 6: Model-View-Controller (MVC) Pattern**

In our backend microservices, we follow the Model-View-Controller (MVC) design pattern. This pattern aims to achieve a clear separation of concerns by dividing the application into three interconnected components: Model, View, and Controller.

### Model-Controller

In our application, the Model and Controller layers are grouped together because Firebase's built-in functions abstracts away many of the responsibilities typically handled by the Model.

Firebase handles data access and transformation, allowing the Controller to directly interact with the database without the need for an explicit Model class. Our application's



logic is also simple enough that we do not need a separate Model to manage complex data operations or transformations. Instead, the Controller performs the necessary CRUD operations using Firebase's API, processing the data, and formats the response to be sent back to the View for presentation to the user.

This simplification enhances the maintainability and efficiency of our system while still adhering to the core principles of the Model and Controller pattern.

## View

The View layer is responsible for presenting data to the user interface. In our application, the View is represented by the frontend, which communicates with the backend through HTTP requests. The Controller sends data to the View as an HTTP response (if any), typically in JSON format, and the View (frontend) is responsible for rendering this data for the user.

This separation allows the frontend to focus on rendering the interface, while the backend handles the data and business logic.

## Router

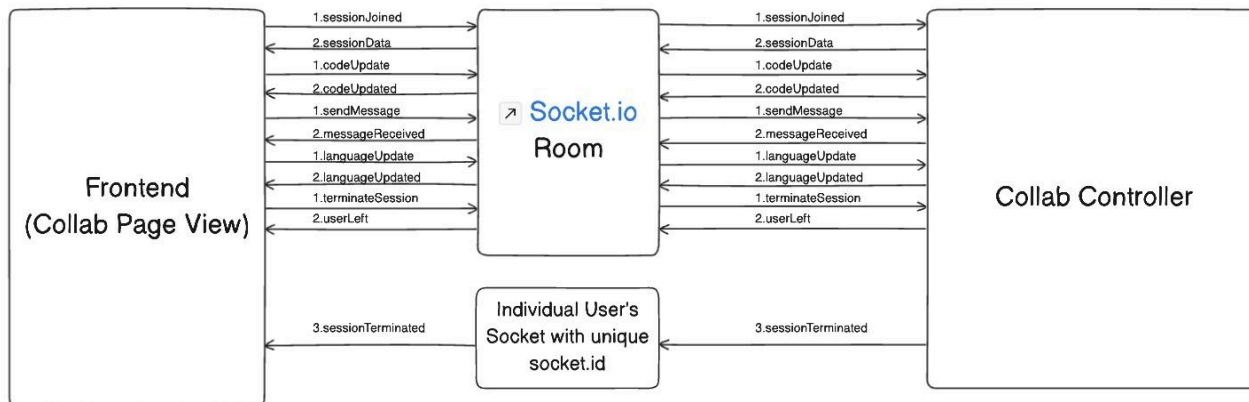
The Router handles incoming HTTP requests by directing them to the appropriate Controller methods based on the request URL and HTTP method. In our application, it associates each endpoint with its corresponding function in the Controller, ensuring efficient routing of requests to the correct logic for processing. Additionally, the Router can include middleware for specific routes, such as an `isAdminCheck` middleware for managing admin-actions like adding, deleting, or editing questions.

### 4.5.2. Publisher-Subscriber (Pub-Sub) Pattern

#### 4.5.2.1. Socket.io's Pub-Sub Mechanism in collaboration service

The collaboration service adopts the Pub-Sub pattern where events, such as `codeUpdate`, `messageSent`, `languageUpdate`, are published to specific rooms with the specified session id, and users in those rooms are subscribed to receive those events. This pattern decouples event producers from consumers and allows efficient, asynchronous communication without tightly coupling the components, promoting scalability and modularity.

Except for the `sessionTerminated` event that is sent to the individual user's socket in order to navigate only the user who chose to terminate session back to the homepage, all the other events in the collaboration service is sent to the rooms with the specified session id so that both users in the room can receive the events.



**Fig 7: Diagram of Events in Collaboration Service**

### 4.5.3. Observer Pattern

#### 4.5.3.1. Observer Pattern in Collaboration Service

Socket.io is used to handle real-time communication between the observers and the observable. The observers could be the backend (collaboration controller) and the observable could be the frontend (collaboration page view), and vice versa, depending on the event flow. The observer listens for specific events (e.g. code updates, messages sent) and reacts by updating their state or triggering specific actions.

a. Observable:

- Emits events (e.g. codeUpdate, messageSent)
- Notifies all users in the specific session about changes

b. Observers:

- Listens to events from backend
- Reacts by updating the UI or perform actions based on the received data

For example:

Backend (collabController.js):

```
socket.to(sessionIdObj).emit('codeUpdated', { code });
```

Frontend (CollabPageView.tsx):

```
newSocket.on("codeUpdated", (data) => {  
    console.log("Code update received from server:", data);  
    setCode(data.code);  
});
```

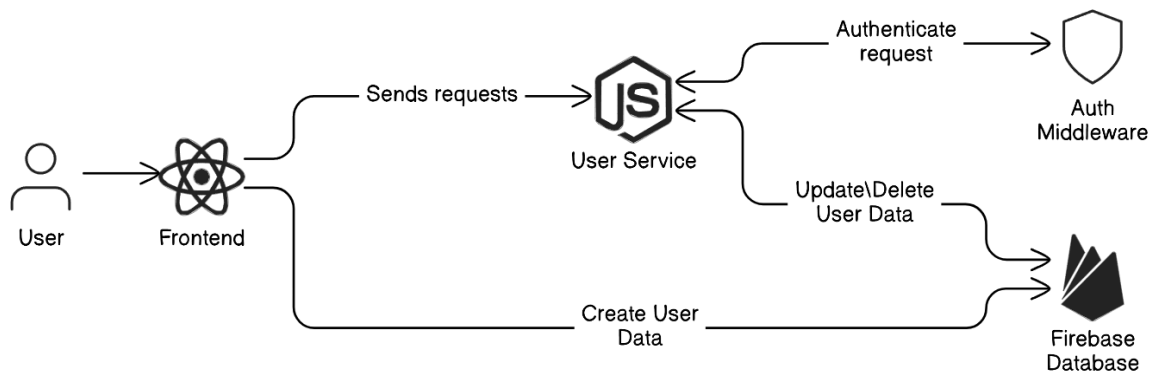
In this case, the frontend (CollabPageView.tsx) is the observer while the backend (collabController.js) is the observable. Hence the frontend listens for the event `codeUpdated` and updates the state of the date accordingly.

## 5. User Service

This section overviews a detailed examination of the User Service, highlighting its role in the microservice architecture and the functions it performs. Inclusive of how the User Service integrates with other components in the application to support authentication, user data management and user role management.

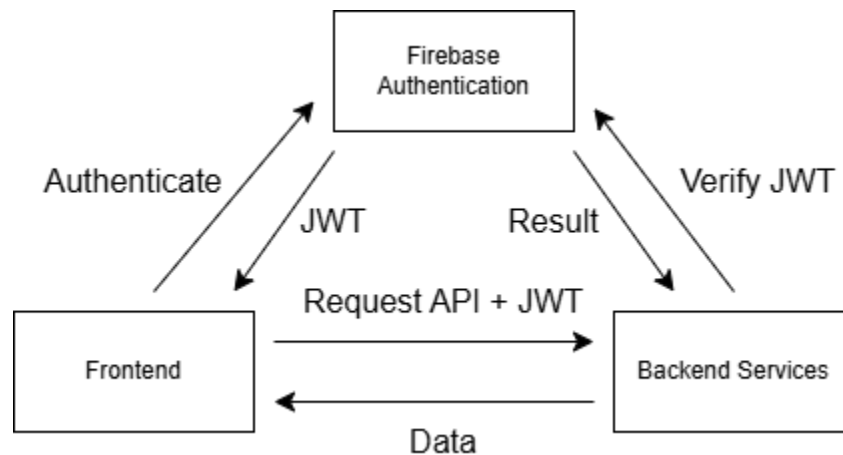
### 5.1. Architecture Design

The User Service is designed as a RESTful microservice, providing HTTP endpoints that allow for efficient and secure interaction with user data. Operating independently of other services, interfacing with both the frontend and backend as needed to support user-related operations. While core functions are managed within this backend service, leveraging the Firebase database, offering a seamless integration that simplifies frontend data handling. Thus, some aspects of user data management, such as real-time data access and authentication, are handled partially within the frontend. This hybrid approach balances backend control with Firebase's client-side capabilities, creating a streamlined and responsive experience for user-related functionalities.



**Fig 8: Architecture Design of User Service**

### 5.1.1. Authentication service

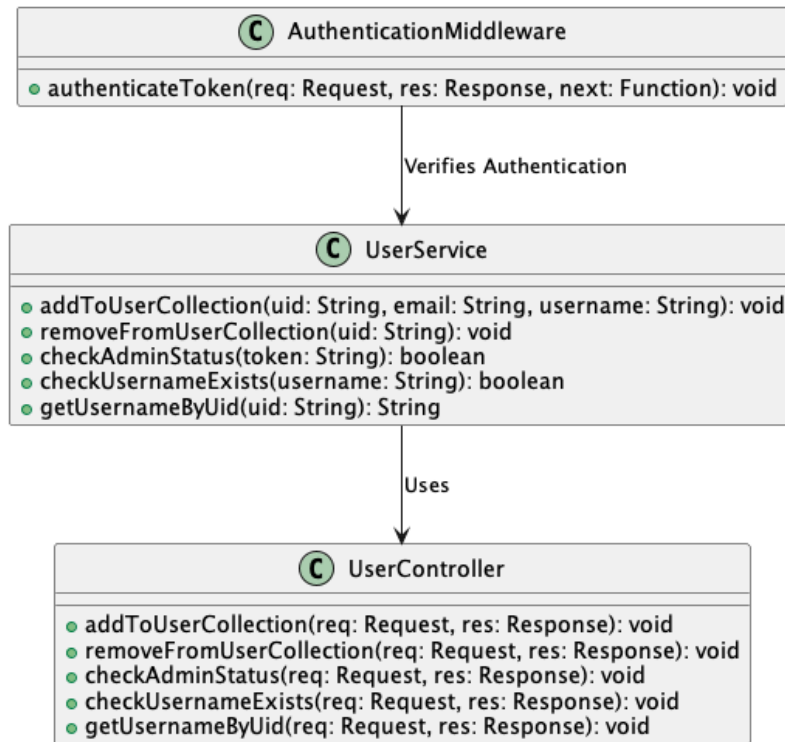


**Fig 9: Firebase Authentication Diagram**

The Authentication service leverages Firebase Authentication as the primary authentication provider. Firebase's secure token generation and validation provide an easy solution for user verification. Tokens issued by Firebase are JWTs, which allows for interoperability across client and server. Firebase's custom claims feature allows integration of Role Based Access Control, directly storing admin roles within the token, reducing complexity on the server. By delegating these processes to Firebase, our application benefits from Firebase's security features without implementing custom authentication.

## 5.2. Internal Microservice Design

The user service is a backend microservice process responsible for managing user account and authentication operations. Users are able to create a new account, log in, and delete their accounts. Authentication of the users is handled through JWTs, which ensures secure session management and user verification upon each request.



**Fig 10: Class Diagram of User Service**

### 5.2.1. Auth Middleware

The `authenticateToken` middleware verifies JWTs to secure routes by validating user identity on every request. The middleware extracts the token from the `Authorization` header, then verifies its authenticity using Firebase Authentication's `verifyIdToken` function. If the token is valid, the decoded token data, which includes user details like `uid`, is attached to the `req` object, making it accessible to route handlers downstream. If the token is invalid or missing, the middleware returns a 401 or 403 error, respectively, preventing unauthorized access to protected resources. This ensures secure handling of user sessions.

For admin verification, an additional layer checks the `isAdmin` custom claim, stored in Firebase when a user has admin privileges. Upon accessing an admin-only route, the middleware not only verifies the token but also inspects the decoded token's claims. If `isAdmin` is set to `true`, access is granted; otherwise, the user receives an authorization error. This design supports secure role-based access control (RBAC), ensuring that only authenticated admins can perform privileged operations.

Additionally, this authentication middleware is integrated into every microservice, providing a consistent, secure authentication layer across distributed services. This approach ensures that each microservice is protected by the same verification process, simplifying the management of secure user sessions and role-based access in a microservices architecture.

### 5.2.2. Design Considerations

1. User Data Management: User information, including uid, email and username, is stored directly within Firebase Authentication. This approach centralizes user credentials and data, simplifying authentication flow and removing the need for a separate “users” collection in firebase.
2. Username Validation & Verification: The `isValidUsername` function checks if the username meets the specific criteria of containing only alphanumeric characters. The `doesUserExist` function checks if the username already exists in Firebase. This helps to maintain consistency, username uniqueness and prevent injection attacks.
3. Role Management (Admin Check): Admin roles are stored in Firebase Authentication as custom claims, with a boolean field of `isAdmin`. This allows for role based access control. The service includes an endpoint `/admin/checkAdminStatus` that checks if the user has admin status by examining the custom claims.
4. Authentication Middleware: The `authenticationToken` middleware function validates JWTs on protected routes. The function extracts the token from the `Authorization` header, verifies it with Firebase Auth, and attaches the decoded token data to the request object to be used by subsequent handlers.

#### 5.2.2.1. Handling JWTs on Frontend

Firebase authentication tokens automatically expire after 1 hour, so storing them in `sessionStorage` or `localStorage` isn't ideal, as they become invalid after expiration. Thus, each time a token is needed, it is retrieved directly from Firebase, which manages token expiration automatically.

However, because tokens are not stored in `sessionStorage` or `localStorage`, a page refresh requires the user to log in again. While this may be less convenient, it's

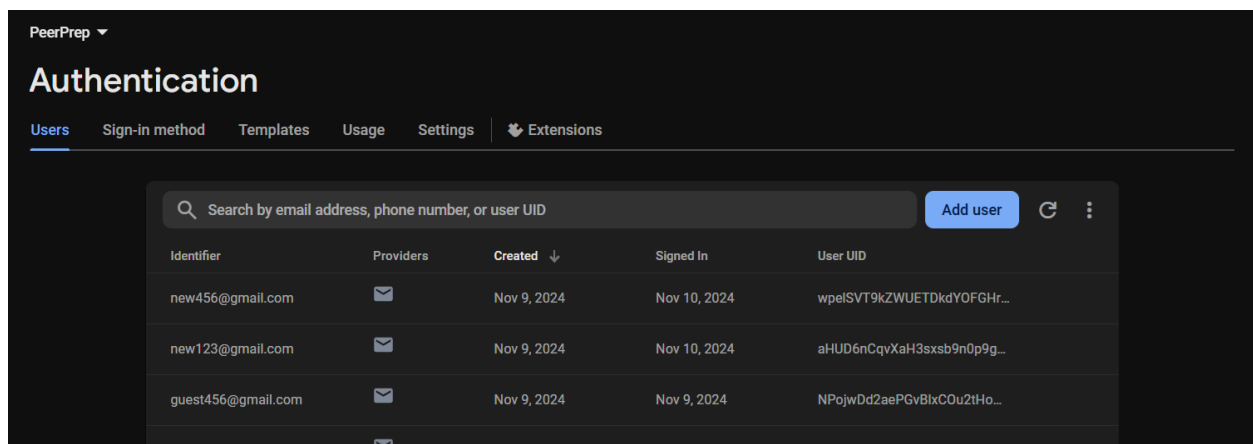
manageable as our application is a single-page application (SPA) that generally doesn't require frequent refreshing.

#### 5.2.2.2. User Data Management

User data, which includes the uid, email, and username, is centrally managed within Firebase Authentication. By using Firebase Authentication, sensitive user information like passwords and verification tokens are handled securely without needing to store or directly manage this data in our own database.

Firebase's Authentication feature provides a streamlined, API-based approach to user identity and session management. When a user creates an account, Firebase generates a uid (unique user ID) and allows us to link it with a user email and username. These identifiers are later used in our database to associate users with their respective data without needing to handle or store sensitive password data ourselves.

This design minimizes the potential attack surface in our application, as we rely on Firebase for secure storage and handling of user credentials, reducing our need to build custom, complex user authentication logic. Furthermore, Firebase tokens can be easily integrated across different services, ensuring a unified identity framework that simplifies user data consistency and security across both the user and server.



**Fig 11: Firebase Authentication**

While Firebase Authentication provides a convenient and secure way to manage user login with minimal configuration, it handles only the email and password fields, primarily focusing on authenticating credentials. To meet our app's specific requirements for additional user data, we attached a separate Firebase Firestore collection to



supplement user profiles with custom fields. This setup enables a flexible and scalable way to extend user information without compromising security or performance.

For example, creating a Firestore collection to store personalized attributes such as **Username** and an **isAdmin** flag, which allows for differentiating between normal and administrative users. Linking this collection to each authenticated user's unique ID, we maintain a seamless and organized structure that keeps additional data accessible and secure.

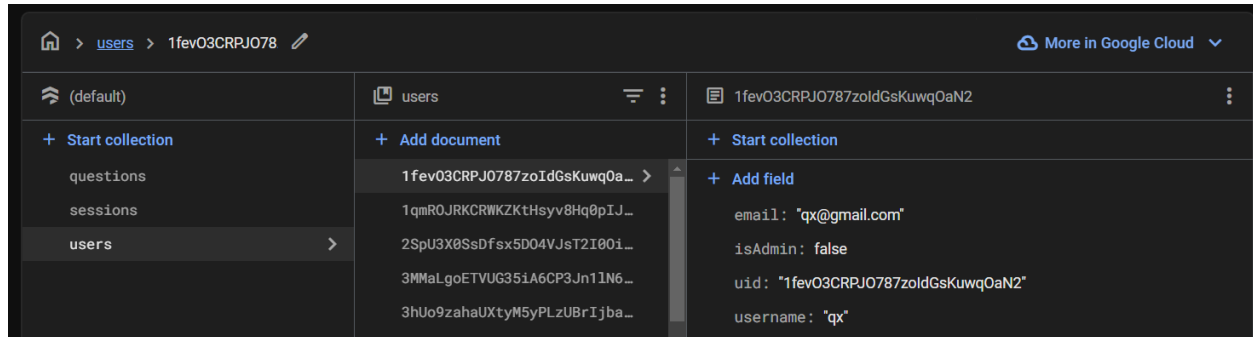


Figure 12: Firebase Firestore 'Users' Collection

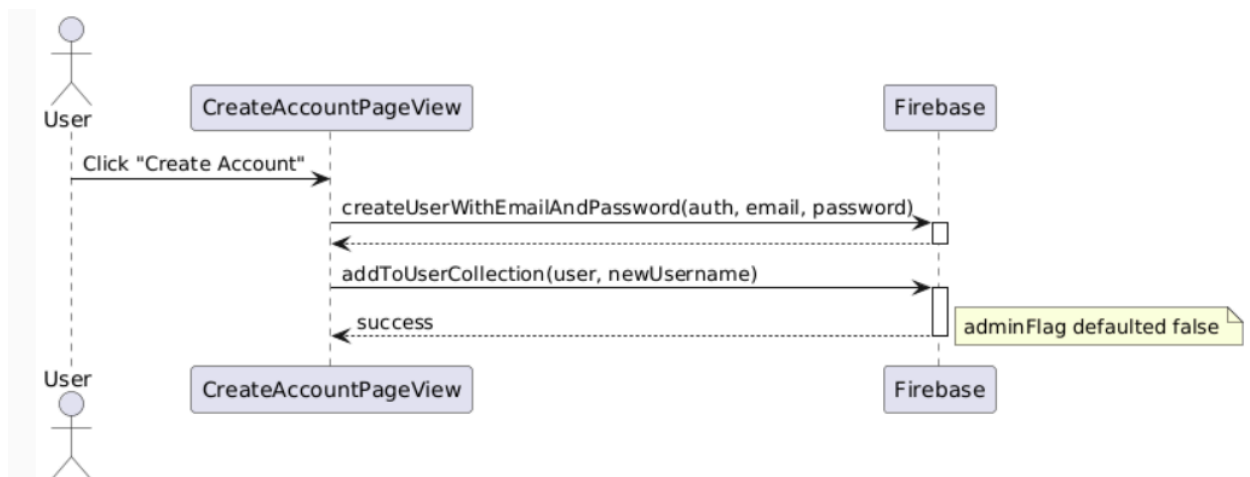
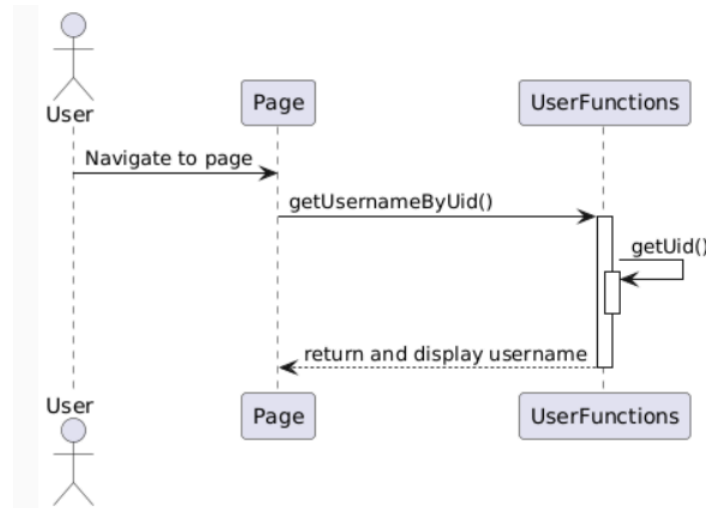


Fig 13: Sequence Diagram of Creating an Account

Retrieving user information plays an integral part in providing visual feedback to users at the Questions Page and especially for the collaboration chat box feature. After navigating to a page requiring username(s), the **getUsernameByUid** function is invoked forwarding to **getUid**, a utility helper function which retrieves the current user's UID from the Firebase Authentication instance, maintaining security and accuracy in user identification. After which, a call to the server endpoint **user/username/{uid}**

is made and returns the username associated with the UID and is then displayed appropriately on the respective page. This layered approach ensures that usernames are retrieved from authenticated users protecting user data and displaying personalized information within the application.

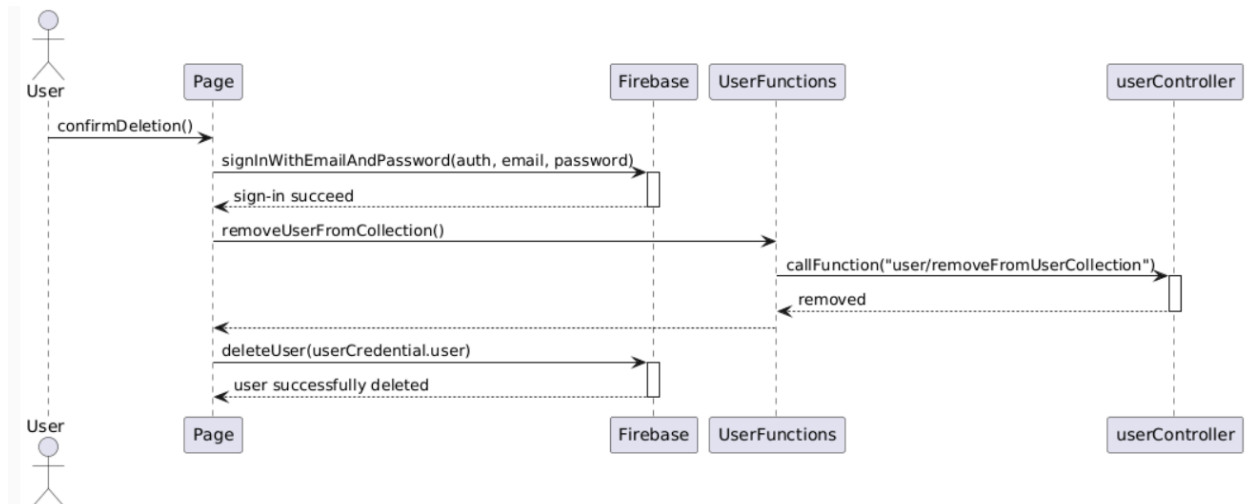


**Fig 14: Sequence Diagram of Retrieving Username**

Deleting an account involves a secure, multi-step process to ensure user authentication and comprehensive removal. When a user confirms and clicks the option to delete their account, `confirmDeletion` is triggered to initiate the deletion process. For security, the user is re-authenticated by calling Firebase Authentication's `signInWithEmailAndPassword` function, ensuring that the account deletion request is made by the verified user.

If re-authentication is successful, `removeUserFromCollection` is called to delete additional user information stored in the Firebase user-collection such as `username` and `isAdmin` flag, attached on top of the core Firebase Authentication data.

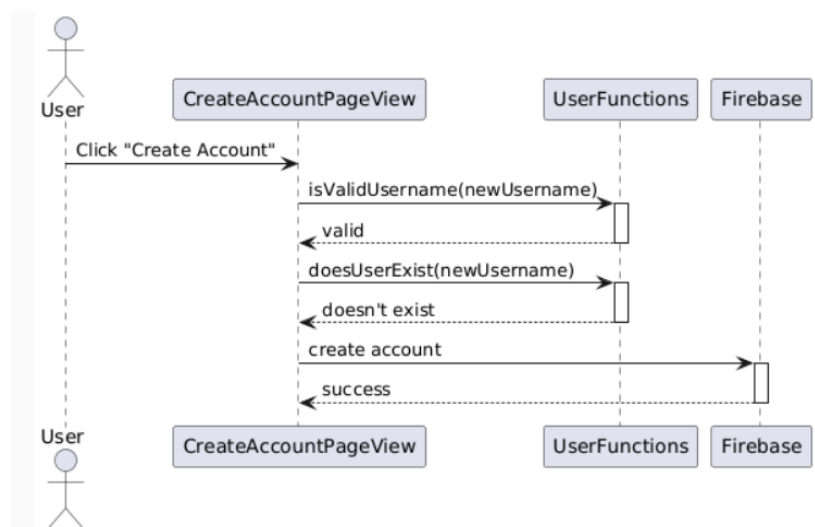
Finally the Firebase Authentication's `deleteUser` function is called, removing the user's credentials and invalidating any active sessions and navigating back to the homepage. This approach ensures that both the authentication credentials and any associated user data is thoroughly removed from Firebase.



**Fig 15: Sequence Diagram of Deleting an account**

### 5.2.2.3. Username Validation & Verification

When a user creates an account, the application checks the provided username using the `isValidUsername` function and verifies if the username already exists in Firebase using the `doesUserExist` function. The `isValidUsername` function ensures that the username only includes alphanumeric characters, preventing any special characters, non-standard or potentially harmful input. By restricting inputs to a safe character set, this function effectively mitigates risks of SQL injection, and other input-based vulnerabilities.



**Fig 16: Sequence Diagram of Username Validation & Verification**

#### 5.2.2.4. Role Management (Admin Check)

Role-based access control is achieved through Firebase's custom claims feature. For admins, the `isAdmin` claim is added to their JWT by setting this value in Firebase's user management system. This way, when an authenticated user accesses a route, the `authenticateToken` middleware can check the token for an `isAdmin` property, instantly identifying the user's role without needing additional database queries.

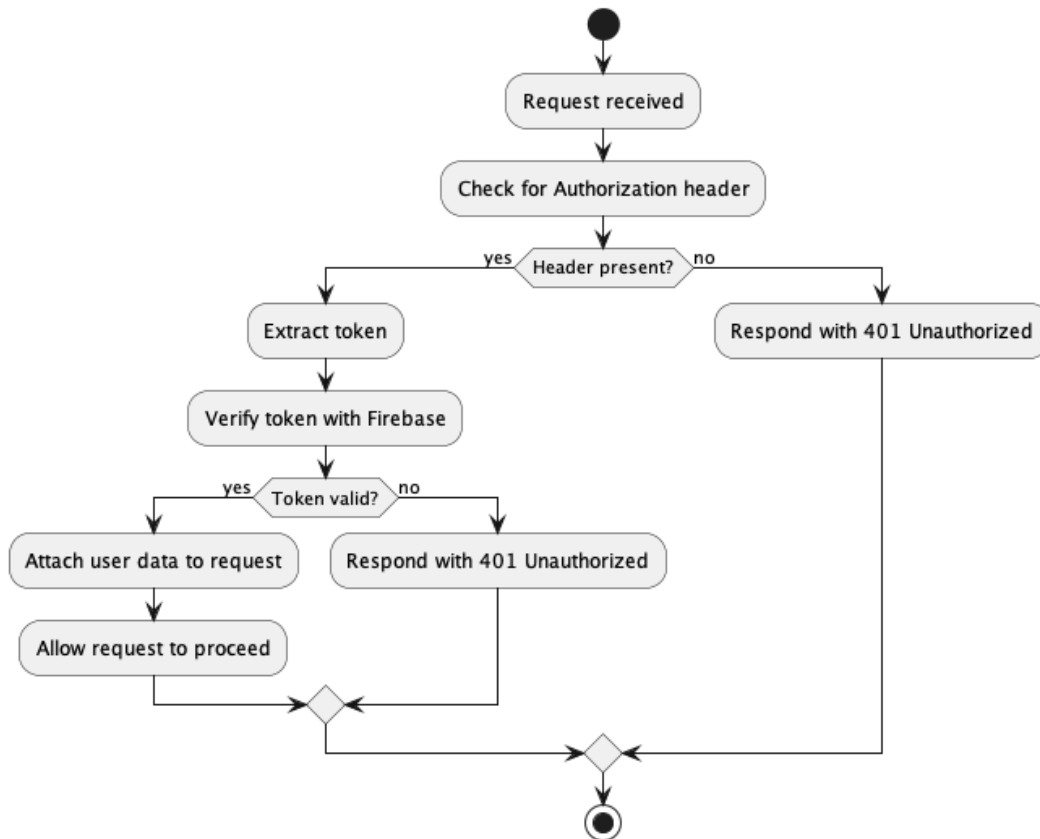
Admin-only routes can thus securely filter access by checking if `isAdmin` is set to `true`, allowing only privileged users to execute admin actions such as creating, editing and delete questions. For normal users, the `isAdmin` claim is set to false by default.

Firebase's custom claims, stored within the token, allows us to scale this design to include various roles by setting additional claims for different access levels. This system not only provides security but also optimizes performance by reducing the need for enhanced database lookups.

#### 5.2.2.5. Authentication Middleware

The `authenticateToken` middleware is a critical part of securing routes and services in the application. Each time a request is made to a protected route, the middleware intercepts it and checks for a valid JWT in the `Authorization` header. If the token is valid, the middleware decodes it to retrieve user information, attaching details like `uid` and `isAdmin` (if applicable) to the request object. This allows downstream route handlers to access user-specific data securely.

In the absence of a valid token, the middleware responds with a `401 Unauthorized` error, stopping the request from proceeding. This structure ensures that only authenticated users can interact with the application's protected resources, safeguarding sensitive data and preventing unauthorized access.

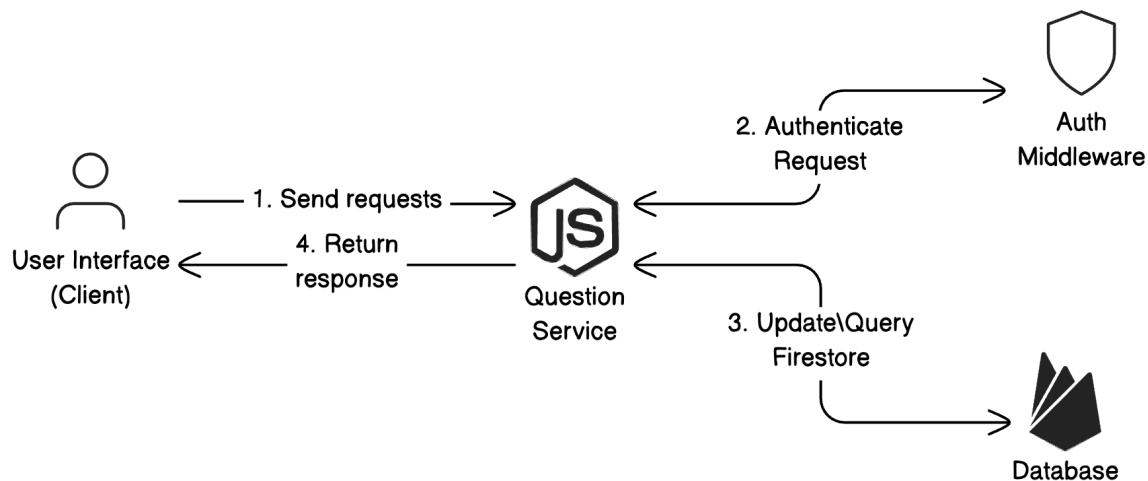
**Fig 17: Activity Diagram of Authentication Middleware**

## 6. Question Service

This section provides an overview of the Question Service, detailing both its place in the overall system architecture and the specific internal design decisions that govern its behavior and operations. It covers both the high-level architecture of how the service interacts with other components in the system, as well as the internal mechanics that define its core functionality.

### 6.1. Architecture Design

The Question Service is a key component in the microservices architecture, providing functionality for managing questions within the application. This service is designed as a RESTful microservice that exposes a set of HTTP endpoints for interacting with question data. It operates independently of other services, communicating with the frontend and other microservices as necessary.



**Fig 18: Architecture Design of Question Service**

#### 6.1.1. Key Architectural Components:

##### User interface (Client)

The frontend (client) communicates with the Question Service via a RESTful API. Through this interface, users can perform various operations on questions, including retrieving, creating, updating, and deleting question records.

## Auth Middleware

Before any operation is performed on question data, the Question Service first authenticates incoming requests using the Auth Middleware ([Section 5.2.1](#)). By intercepting requests early in the process, the middleware acts as a security layer that prevents unauthorized access and ensures the integrity of the data.

## Database

Once the request is authenticated, the Question Service interacts with the database (Firebase Firestore) to either retrieve or update question data.

## 6.2. Internal Microservice Design

The Question Service is responsible for managing all question-related operations within the application. It is built to handle typical CRUD operations (Create, Read, Update, Delete), and provides functionality for filtered queries.

### 6.2.1. Core Responsibilities:

- **Create:** Allow administrators to create new questions.
- **Read:** Allows all users to retrieve questions or a filtered list of questions based on various parameters (e.g., difficulty, topics).
- **Update:** Allow administrators to modify existing question records.
- **Delete:** Allow administrators to delete specific questions.

## 6.2.2. Internal Design Flow:

### 1. User Interaction

The frontend (client) sends HTTP requests to the Question Service to perform CRUD operations. These requests contain necessary data (e.g., token, question data, filters) and are routed to the appropriate endpoints.

### 2. Authentication and Authorization

Upon receiving a request, the Question Service passes the attached token to Auth Middleware ([Section 5.2.1](#)) to check for the following:

- **Authentication:** Ensures the user is authenticated and has a valid Peerprep account.
- **Authorization:** Once the user is authenticated, the middleware verifies whether the user has the correct permissions to perform the operation (e.g., ensuring only authorized users can create or delete questions).

### 3. Database Interaction

Once authenticated and authorized, the Question Service interacts with Firebase Firestore to perform the relevant CRUD operation.

### 4. Data Transformation (if required)

If required, the Question Service may also transform the data obtained before sending it back to the frontend. For example, question metadata might be calculated or enriched with additional information (e.g., user ratings, tags) that is not stored directly in the database but needs to be computed dynamically.

### 5. Response to Frontend

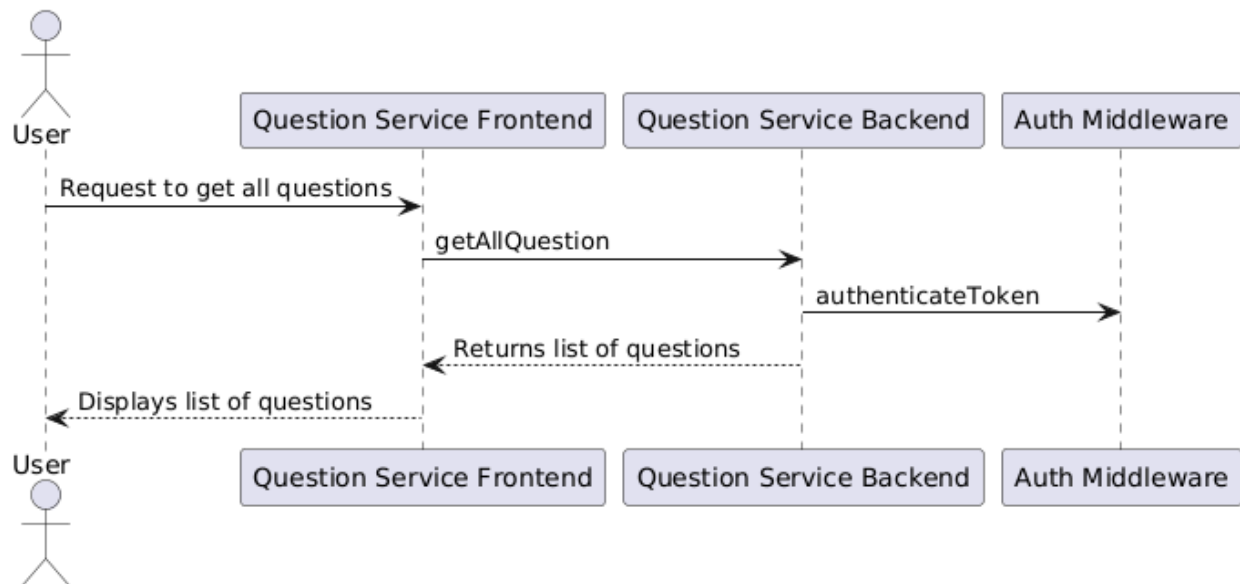
After processing the request, the Question Service sends the appropriate HTTP response to the frontend. This response may include:

- A success or failure status
- The requested question data (e.g., list of questions, a single question, or confirmation of an operation like creation or deletion)
- Any errors or validation messages if the request was invalid or unauthorized.



### 6.2.2.1. Internal Design Flow Example:

Below is an example workflow illustrating how a user would retrieve a list of questions.



**Fig 19: Sequence Diagram of `getAllQuestions` API Call**

1. User sends a GET request to the Question Service to get a list of questions.
2. Question Service authenticates User via Auth Middleware (x.x.x).
3. Once authenticated, Question Service invokes `getAllQuestions` function to retrieve a list of questions stored in the database.
4. The Question Service then returns the list of questions to the Frontend to be displayed to the User.

### 6.2.3. User and Admin Privileges

The question service differentiates between regular users and administrators, granting them distinct privileges and access rights based on their roles. The verification of these permissions is handled in the authentication middleware ([Section 5.2.1](#)) using the user's token.

Below is a summary of the privileges assigned to each user type in the question service.

Type of User	Permissions Granted
Regular User	Ability to read questions
Administrator	Ability to create, read, update and delete questions

**Fig 20: Comparison of User Permission and Administrator Permission**

### 6.2.3.1. Frontend

User View				Administrator View			
Filter		Filter Topics		Filter		Filter Topics	
No. ↑	Questions ↑	Difficulty ↑		No. ↑	Questions ↑	Difficulty ↑	
1	Fibonacci Number	Easy		1	Fibonacci Number	Easy	
2	Reverse a String	Easy		2	Reverse a String	Easy	
3	Repeated DNA Sequences	Medium		3	Repeated DNA Sequences	Medium	
4	Course Schedule	Medium		4	Course Schedule	Medium	
5	Longest Common Subsequence	Medium		5	Longest Common Subsequence	Medium	
6	Rotate Image	Medium		6	Rotate Image	Medium	
7	Validate Binary Search Tree	Medium		7	Validate Binary Search Tree	Medium	
8	Sliding Window Maximum	Medium		8	Sliding Window Maximum	Medium	
9	N-Queen Problem	Hard		9	N-Queen Problem	Hard	
10	Serialize and Deserialize a Binary Tree	Hard		10	Serialize and Deserialize a Binary Tree	Hard	
		Previous Next				Previous Next	

**Fig 21: Comparison of User Interface and Administrator Interface**

When a user logs in, the system fetches their role from the server and adjusts the interface accordingly. Regular users are granted access only to view questions, while administrators are shown an interface with additional options to create, update, and delete questions. This conditional rendering of components improves the user experience by streamlining the interface while also serving as the first line of defense against unauthorized actions.

#### 6.2.3.2. Backend

On the backend, security is implemented through middleware services ([Section 5.2.1](#)) that intercept requests and verify the user's identity and authorization via `isAdminCheck` before processing any administrator-only actions.

### 6.2.4. Design Considerations

#### 6.2.4.1. Pagination Support

Currently, the `getAllQuestions` endpoint retrieves the entire dataset of questions from the database in a single response. Given the current small dataset, this approach provides several practical benefits, and pagination was intentionally omitted for the following reasons:

1. Performance Sufficiency: With a small number of records, response times are fast and the service can comfortably handle retrieving and transmitting all questions in one go without impacting performance or user experience.
2. User Convenience: Users can access all questions in a single query without needing to load additional pages. For a small dataset, this design is more convenient and intuitive.
3. Development Efficiency: Omitting pagination at this stage reduces the complexity of both backend and frontend logic. By avoiding pagination logic, we streamline development and testing, enabling a faster release of the initial service.

However, we recognize that as the database grows, handling all questions in a single response will become inefficient, and pagination will eventually be necessary to maintain performance and usability.

#### 6.2.4.2. Caching

Currently, the `getAllQuestions` endpoint does not implement caching. This decision is based on the fact that, at the current stage of development, the effort required to implement caching outweighs the benefits it would provide. The following reasons outline why caching is not yet necessary:

1. Small Dataset: Currently, retrieving all questions in real time poses no significant performance challenges. As such, caching will not be able to provide a noticeable benefit. Additionally, since the data is unlikely to be updated often, the overhead of caching outweighs the benefits it provides.
2. Development Efficiency: Implementing caching adds another layer of complexity to both backend and system design. At the current stage, time could be better

spent on other development priorities, such as improving the core functionality, UI, and ensuring the reliability of the basic services.

However, as the system grows and the number of questions increases, the demand for performance optimizations will become more critical and caching will eventually become essential to maintain performance.

## 7. Matching Service

### 7.1. Architecture Design

The following sections detail and explain the decisions made regarding the matching criteria, the tech stack for the matching service, and how these choices impact the relevant NFRs being targeted.

#### 7.1.1. Matching Criteria

The criteria for matching two users require both to select the same topic and difficulty level. If there's any mismatch in any of these selections, no match will occur, and the user will time out after 30 seconds if a match isn't found by then. The choice to enforce such a strict matching criteria is due to multiple reasons.

Firstly, **to ensure relevant practice**. Relaxing the matching criteria by allowing **different topics** with the **same difficulty** would not serve users who want to focus on a specific topic where they may feel weaker and wish to practice exclusively. Since no two topics are quite the same, matching users on different topics could lead to mismatched expectations and increase the likelihood of users leaving the session, which is not ideal for both participants.

Similarly, **to align difficulty with user expertise and expectations**. Allowing matches on the **same topic** but with **different difficulty** levels could frustrate users. For instance, a beginner aiming to tackle only easy problems would find it overwhelming to match with someone tackling advanced challenges. Likewise, an expert may feel that practicing fundamental problems is a waste of time, which could lead to dissatisfaction and disengagement.

Secondly, **to optimize matching within a limited question bank**. Given the relatively small size of our question bank (especially when compared to larger platforms like LeetCode), flexible matching criteria could result in extremely far-fetched poor matches that don't align well with users' goals. By enforcing strict criteria, users are channeled toward a smaller, curated set of questions, eliminating the risk of irrelevant matches due to the limited question pool.

Thirdly, **to avoid complexity issues**. Implementing a loose matching criterion would add considerable complexity to the system, including the challenge of defining clear boundaries for how "loose" the criteria should be. Additionally, the technical implementation of a more flexible matching system would require intricate logic, and might require an entirely new tech stack. While a loose matching option could be a

valuable enhancement, it is best suited as a potential future improvement rather than a current priority.

### 7.1.2. Matching Service Tech Stack

**Socket.IO:** The matching service uses Socket.IO to enable real-time, bidirectional communication between the frontend and the backend matching service. This allows for instant updates and responsiveness in the matching process.

**RabbitMQ:** RabbitMQ with the AMQP protocol is employed as the queuing mechanism to match users. We used a routing method (direct exchange) to direct messages to specific queues, with each queue named according to a unique `${difficulty}_{topic}` format (e.g., `Array_Medium`), aligning with our use case of a strict matching criteria.

#### **Why RabbitMQ instead of an in-memory queue?**

Using an in-memory queue limits scalability, as it would only function within a single server instance, making it unsuitable for horizontal scaling (i.e., running multiple backend server instances). For example, if user A is connected to backend server instance 1 and user B is connected to instance 2 (perhaps due to load balancing or geographic routing), user B's matching request would fail to locate user A in instance 2's in-memory queue, as each instance maintains separate memory spaces. By utilizing a global, distributed queue like RabbitMQ, we enable seamless matching across multiple backend server instances.

**Redis:** To further enable horizontal scaling, we need a way for Socket.IO servers to communicate across different instances. For example, if user A connects to backend server instance 1 and waits for a match, and user B later connects to instance 2 and completes the match (via RabbitMQ), backend instance 2 will need access to user A's socket (connected to instance 1). Since user A's connection is not available directly within instance 2, a solution is required to broadcast packets and messages across server instances.

#### **Redis Adapter Solution**

The Redis adapter facilitates this by enabling communication across Socket.IO instances, allowing backend instance 2 to send events to user A on instance 1. This cross-server communication enables us to handle user events and notifications seamlessly, even if the users are connected to different server instances, thereby making the matching process robust and scalable.

### 7.1.3.Relevant NFRs

#### 1. Match Timing and UID Return (5-Second Goal)

- **Socket.IO:** Using Socket.IO for real-time, bidirectional communication supports meeting the strict 5-second requirement to match users and return the UID if a match is found. Socket.IO allows immediate response handling between the frontend and backend, reducing latency, which helps achieve the 5-second response time.
- **RabbitMQ:** RabbitMQ's queueing mechanism facilitates efficient message routing and minimizes wait times, supporting the speed of user matching. By using topic-difficulty specific queues, the system can quickly find and route messages for matches in a structured way, further enabling it to meet the required timing.
- **Strict Matching Criteria:** This design choice of strict matching based on topic and difficulty prevents unnecessary matching attempts, reducing potential delays. Since only users with identical criteria are queued together, the system can return a match (or time out) faster than it would with looser criteria.

#### 2. Usability

- **Automatic Timeout:** The system automatically times out matching requests after 30 seconds if no match is found. This prevents infinite loading, aligning with the usability NFRs for user satisfaction and reliability.
- **User Satisfaction through Relevance:** The strict matching criteria based on both topic and difficulty level ensure that users find matches that closely align with their specific learning goals, improving user satisfaction. Without this strict criteria, users could experience mismatched sessions, potentially leading to dissatisfaction and session abandonment.

#### 3. Scalability (Handling up to 10,000 Users)

- **RabbitMQ:** By implementing RabbitMQ as a global, distributed queue, the system allows horizontal scaling, as multiple backend server instances can share access to the queue. This setup can support increased user load without degraded performance, aligning with the scalability requirement of up to 10,000 users.
- **Redis Adapter:** Redis enables real-time communication across server instances by synchronizing Socket.IO connections across multiple servers. This cross-instance communication ensures that the system can continue to function efficiently at scale, as all users can be matched regardless of which server instance they connect to.

- **Socket.IO for Real-Time Matching:** The use of Socket.IO ensures efficient, real-time communication across a large number of users, supporting the goal of matching within 3 seconds even at higher user loads.

## 7.2. Internal Microservice Design

The matching service provides the functionality for initiating a collaboration between two users based on the selected topic and difficulty.

To implement a real time matchmaking system, websockets via the Socket.IO library are utilized.

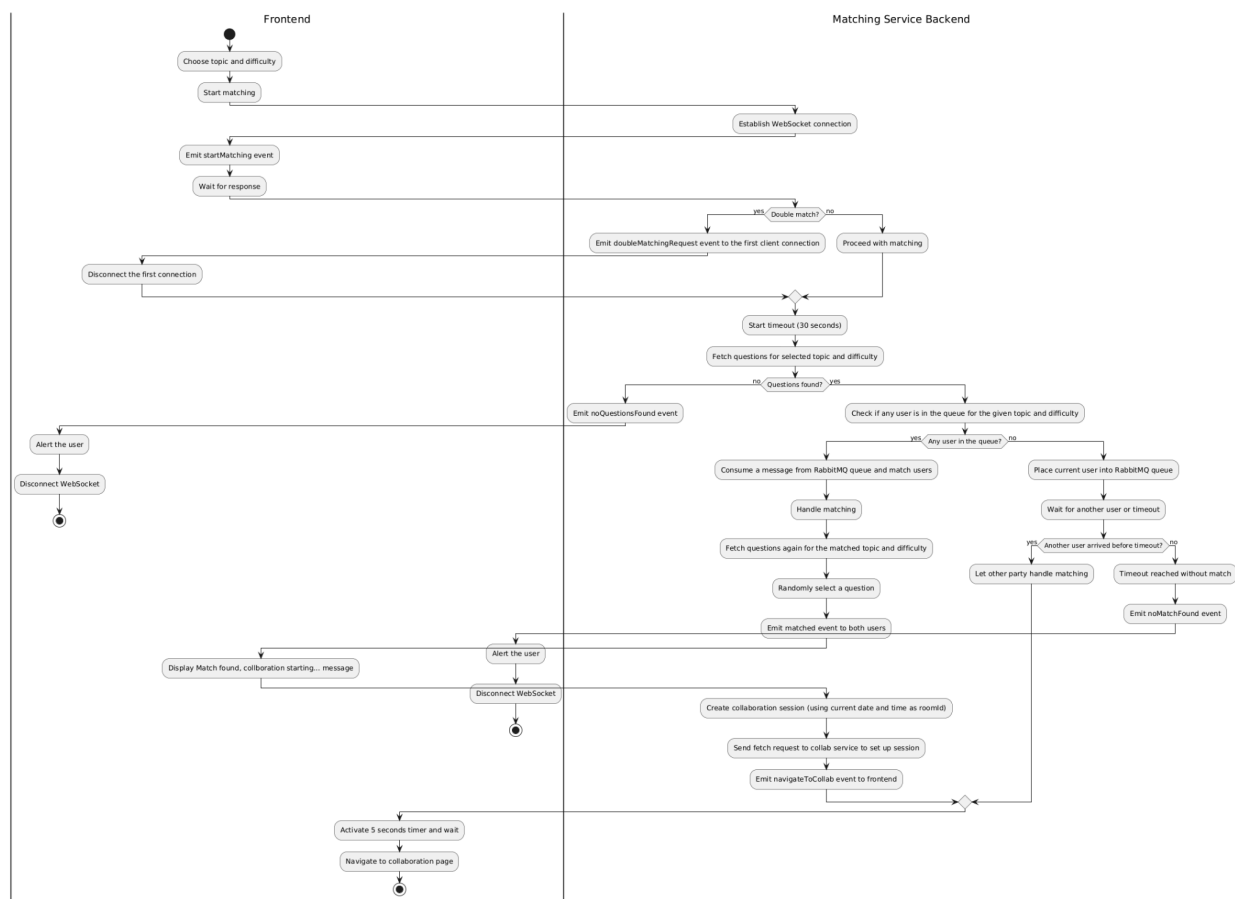


Fig 22: Activity diagram for matching service (for enlarged version, click [here](#))



Event	Purpose	Emitted By
startMatching	Starts matching process for user in the matching service backend	Frontend
cancelMatching	Cancels matching process when user clicks Cancel Matching button	Frontend
DisconnectSocket	Notifies the other party to disconnect the socket, allowing them to complete necessary logic before disconnecting	Backend Matching Service
matchmakingTimedOut	Notifies the user that 30 seconds has elapsed in the matching process and no suitable match has been found yet, and disconnects the sockets	Backend Matching Service
doubleMatchingRequest	If a user opens a second tab and initiates a second matching process, this event cancels the initial matching connection	Backend Matching Service
noQuestionsFound	Alerts that the selected question type is unavailable in the database (i.e., no questions of that type exist) and disconnects the sockets	Backend Matching Service
matched	Notifies the user that he has been matched with another user	Backend Matching Service
navigateToCollab	Navigates users to the collaboration page	Backend Matching Service

cancelMatching	Cancels matching process when user clicks Cancel Matching button	Frontend
error	Encapsulates error message to be displayed in the frontend	Backend Matching Service

**Fig 23: Matching Service Events**

## 7.2.1. Design Considerations

### 7.2.1.1. Creating a session - choice of session id

Towards the end of the matching process, a session ID (or room ID) needs to be generated. This ID will serve as both the collaboration session identifier and the path for the collaboration URL. The session ID is composed of the match date and time, along with the unique user IDs (UIDs) of both participants. This combination is selected to minimize the risk of session ID collisions. Incorporating the current date and time into the session ID is akin to adding a random key, while including the two users' UIDs ensures that other users can be matched around the same time without causing conflicts in session IDs.

## 8. Collaboration Service

### 8.1. Architecture Design

The following sections detail and explain the decisions made regarding the choice and mechanism used for the collaboration service, as well as the seamless integration from matching service to collaboration service.

#### 8.1.1. Socket.io for Real-Time Synchronization

To enable real time collaborative features such as real time communication and concurrent code editing, we choose to use Socket.io as the primary framework for managing WebSocket connections. Socket.io was selected as it provides automatic reconnection, and allows easy handling of events like disconnections and errors.

Additionally, Socket.io facilitates a publisher-subscriber (pub-sub) model by enabling event-based communication. Events like `codeUpdate`, `sendMessage` and `languageUpdate` are emitted by different components such as the collaboration service frontend (collaboration page view) and the collaboration controller. These events are then received by the subscribers interested in those events, ensuring all clients are synchronized without requiring too much server-side logic.

The use of Socket.io allows for efficient real-time data transfer, supporting many simultaneous sessions by isolating traffic per room. This ensures each user experience remains unaffected by other sessions and since data only transmits as needed for each session, resources are optimized, minimizing unnecessary load on the server. This improves scalability of the app, making it easier to scale up and accommodate more users.

#### 8.1.2. Room-Based Collaboration using Session ID

Each collaboration session is uniquely identified by the `sessionID`, which is used to create isolated rooms in the Socket.io server. Users only receive events relevant to the session they are currently in, maintaining a modular and isolated architecture that minimizes interference across sessions. This room-based model ensures that only users within the same session are affected by the code changes and receive the messages sent in the chat, which enhances data integrity and security of the sessions.

By leveraging on session verification and Socket.io's error handling, this maintains data integrity and avoids session rerouting, hence increasing the reliability of the app.

### 8.1.3. Monaco Code Editor

We chose to use Monaco for the code editor as it aligns with our requirements of an in-browser code editor that comes with a package of features such as syntax highlighting, autocompletion, indentation control, code folding, line wrapping, language specific support and many other IDE features.

By integrating Monaco with Socket.io, concurrent code editing is supported, and changes in language used and code typed by one user will also be simultaneously reflected in the code editor of the other user. Hence, changes are seamlessly propagated across users screens in real time.

Monaco improves usability by enhancing the user experience as it is a user friendly code editor that is also used in Visual Studio Code. Hence, this familiar interface lowers the learning curve for new users, making the application more intuitive for those with programming experience, who are the target audience of this application.

In addition, as Monaco has error detection and syntax highlighting, users can work more efficiently, reducing errors and time spent on code writing. This results in a smoother experience for the users, hence boosting the usability of the app.

## 8.2. Internal Microservice Design

The collaboration service enables real time code collaboration, chat functionality, and code execution. Socket.io rooms, identified by `sessionId`, ensure each session is isolated and events are directed to specific users in that session. Socket.io combined with the Monaco Editor component supports synchronized code updates across multiple users, while a chat feature allows direct communication between the users. The code execution controller processes and returns code execution results in real time.

Event	Purpose	Emitted By
sessionJoined	Initializes session, emits session data	Frontend
codeUpdate	Syncs code updates among all users in the same session	Frontend
sendMessage	Sends a chat message to	Frontend

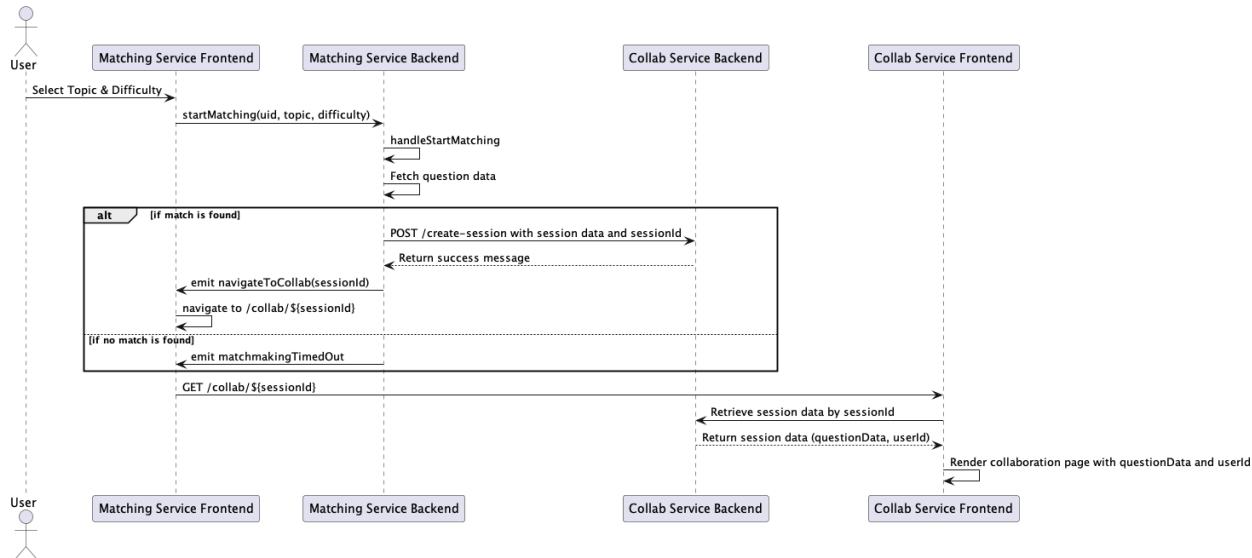
	other users in the session	
languageUpdate	Updates the programming language in the editor for both users	Frontend
terminateSession	Terminates the session and notifies users	Frontend
userLeft	Notifies when a user left the session	Backend Collaboration Service
sessionData	Sends session data such as users' ids and question data.	Backend Collaboration Service
messageReceived	Relays received message to all users in session	Backend Collaboration Service
codeUpdated	Broadcasts code updates to all users in session	Backend Collaboration Service
languageUpdated	Broadcasts language changes to session	Backend Collaboration Service
sessionTerminated	Broadcasts that session has been terminated	Backend Collaboration Service

**Fig 24: Collaboration Service Events**

## 8.2.1. Design Considerations

### 8.2.1.1. Integration with Matching Service

The matching service provides a link containing the `sessionId` when navigating to the collaboration page, enabling the frontend to join the correct session automatically. The session data, including the user ids and question data, is retrieved based on this session id from the “sessions” collection in firebase, and this information is then loaded onto the collaboration page view.



**Fig 25: Sequence Diagram for linking Matching Service and Collab Service**

1. The Matching Service backend receives a **startMatching** request from a user and attempts to match them with another user.
2. Once a match is found, it fetches question data, creates a session via a POST request to the Collaboration Service at <http://localhost:5004/create-session> with a unique session ID.
3. The Collaboration Service returns a success message of the session created with **sessionId**, uniquely identifying the created session.
4. The Matching Service backend emits a **navigateToCollab** event with the **sessionId**, instructing the Matching Service frontend to navigate to the collaboration page with the session ID.
5. The Matching Service frontend listens for the **navigateToCollab** event.
6. Once the **navigateToCollab** event is received, the Matching Service frontend redirects the user to `/collab/${sessionId}`, embedding the session ID in the URL.
7. The frontend requests session data from the Collaboration Service backend using the **sessionId** to identify the correct session.
8. The Collaboration Service backend retrieves the session data including question information and user IDs from the firebase collection and sends it back to the frontend.

9. The retrieved session data (containing question data and user information) is rendered on the Collaboration Service frontend, enabling users to engage in a collaborative session.

### 8.2.1.2. Use of Socket.io in Real-Time Communication and Code Editor Synchronization

Socket.io powers 2 main features in the collaboration service:

1. Real-time chat: Users can send and receive messages instantly within the session using the `sendMessage` and `messageReceived` events. These events facilitate instant communication and allow users to discuss the question and code changes in real time.
2. Concurrent code editor with Monaco Editor: Socket.io keeps the Monaco code editor in sync for both users by broadcasting the `codeUpdate` events whenever a user changes code, hence aiding in code consistency across users.

#### 1. Chat (Messages Sent and Received)



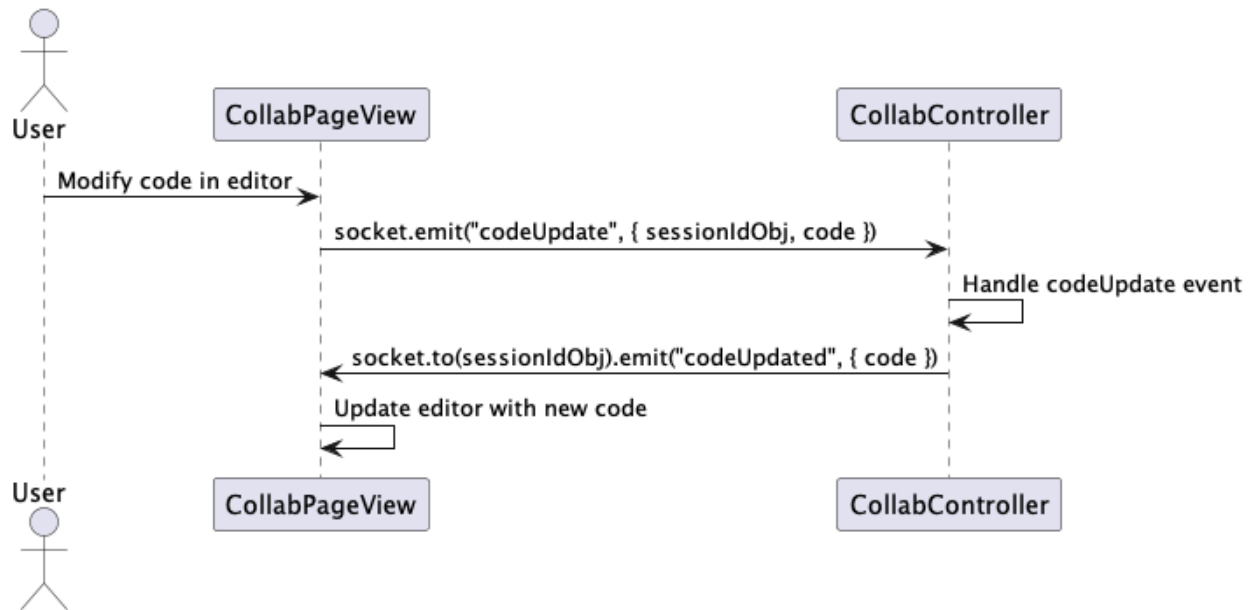
**Fig 26: Sequence Diagram when messages sent and received**

- a. The user types a message and sends it.
- b. `CollabPageView` emits a `sendMessage` event to the `CollabController`.
- c. `CollabController` broadcasts the `messageReceived` event to all users in the session.





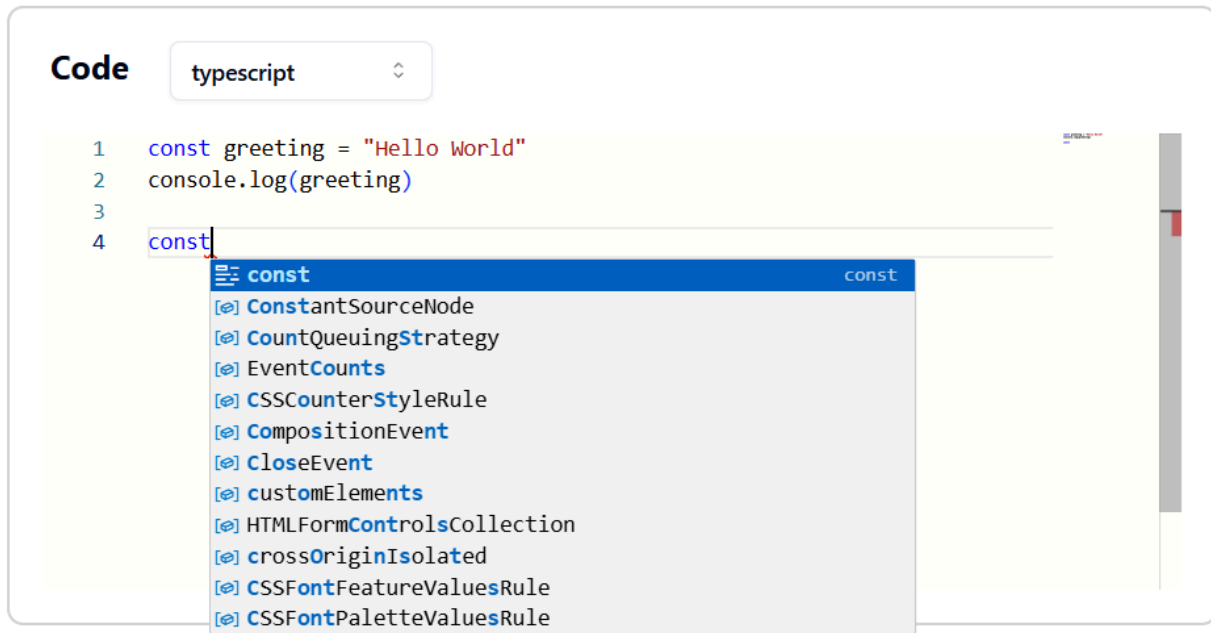
## 2. Code Changes (Code Update)



**Fig 27: Sequence Diagram when code updates**

- The user makes a code change in the editor.
- `CollabPageView` sends a `codeUpdate` event with the new code to the `CollabController`.
- `CollabController` receives the update and broadcasts a `codeUpdated` event.
- `CollabPageView` listens to `codeUpdated` and updates the editor for other users in the session.

### 8.2.1.3. Monaco Code Editor



**Fig 28: Monaco Code Editor**

Integrating Monaco Editor with Socket.io enables concurrent, synchronized editing. Each user's code changes are immediately broadcast to others in the session, supporting real-time collaboration and usability.

While Monaco Editor has built-in support for over 20 programming languages, the wider commonly used coding languages and respective versions implemented are as follows:

Programming Language	Version
Typescript	5.0.3
Javascript	18.15.0
C	10.2.0
C++	10.2.0
Java	15.0.2

C#	6.12.0
Python	3.10.0

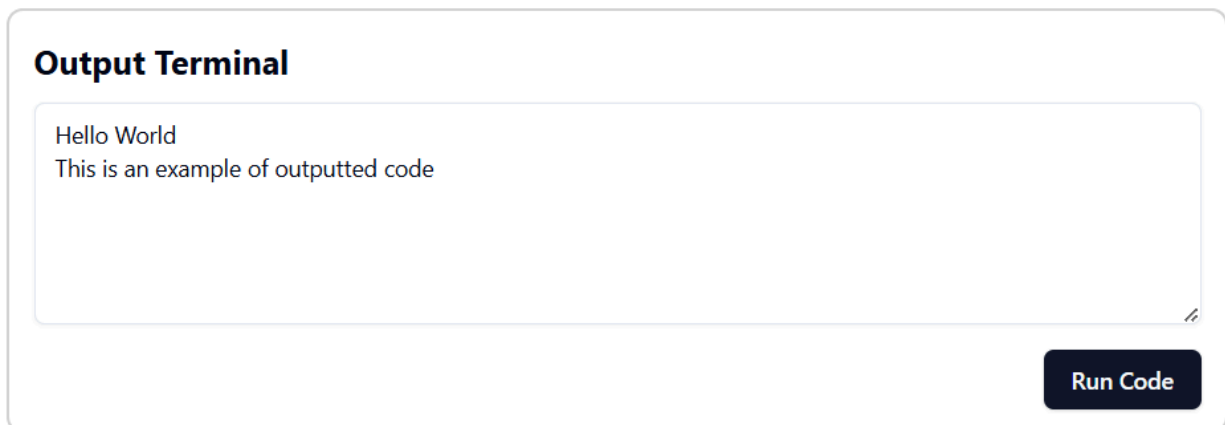
**Fig 29: Supported Programming Language**

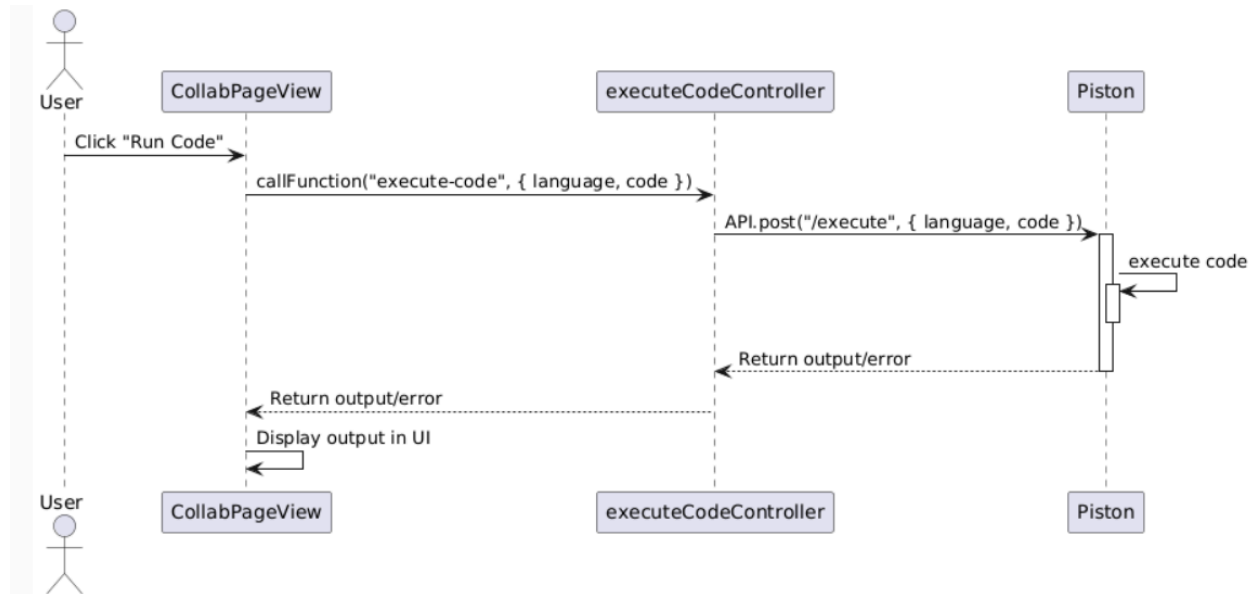
#### 8.2.1.4. Code Execution

The code execution leverages on the Piston API, an external open-source code execution engine designed for running code in multiple programming languages within secure, isolated environments. This versatility allows the Monaco Editor component to offer code execution across different languages seamlessly.

Clicking on the “Run Code” button sends the currently written code by the user, coding language and coding version to the external API endpoint at <https://emkc.org/api/v2/piston/execute> through a post request.

Piston processes the request and executes the code within a secure, sandboxed environment on its external server. The output, error messages, and other relevant details, are returned and thereafter displayed to the user on the Collaboration Page.

**Fig 30: Example code execution**



**Fig 31: Sequence Diagram when code is executed**

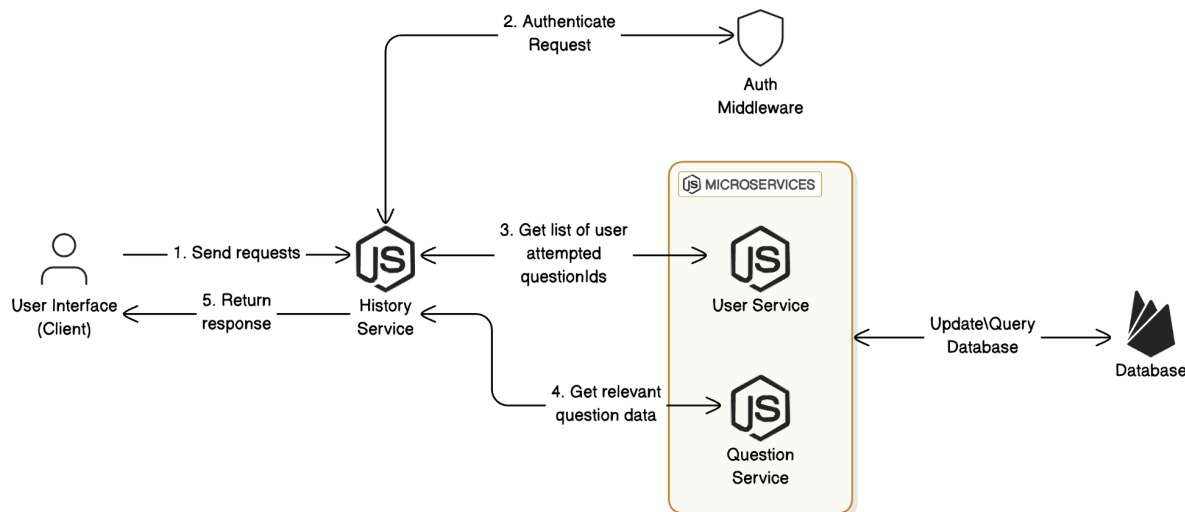
- The user clicks "Run Code".
- `CollabPageView` calls the backend's `execute-code` endpoint.
- `executeCodeController` sends the code to the external API execute endpoint `Piston`.
- `Piston` returns the output or error, forwarded by `executeCodeController` which `CollabPageView` displays in the UI.

## 9. History Service

This section provides an overview of the History Service, detailing its role within the overall system architecture and the specific internal design decisions that govern its behavior and operations. It covers both the high-level architecture of how the service interacts with other components in the system, as well as the internal mechanics that define its core functionality.

### 9.1. Architecture Design

The History Service is a key component in the microservices architecture, providing functionality for managing the question histories of individual users within the application. This service is designed as a RESTful microservice that exposes a set of HTTP endpoints, allowing users to retrieve their question history. While it operates independently as a microservice, it performs its functionality by acting as an orchestrator between the User and Question services, retrieving user data and question details, and transforming this information into a cohesive history response.



**Fig 32: Architecture Design of History Service**

### 9.1.1. Key Architectural Components:

#### User interface (Client)

The frontend (client) communicates with the History Service via a RESTful API. Through this interface, users can perform various operations on question history, including retrieving, creating and updating question history records.

#### Auth Middleware

Before any operation is performed on question history data, the History Service first authenticates incoming requests using the Auth Middleware ([Section 5.2.1](#)). By intercepting requests early in the process, the middleware acts as a security layer that prevents unauthorized access and ensures the integrity of the data.

#### Database

Once the request is authenticated, the History Service interacts with other microservices such as the User Service and Question Service to update the database (Firebase Firestore) accordingly.

### 9.1.2. Why Orchestrator

By centralizing data retrieval and aggregation in one service, this design decision improves system performance by reducing the number of network calls and backend interactions required from the client. Additionally, It also simplifies frontend logic, as the client does not need to manage multiple service dependencies, leading to a more streamlined and efficient user experience.

## 9.2. Internal Microservice Design

The History Service is responsible for managing all history-related operations within the application. It is built to handle Create, Read and Update operations.

### 9.2.1. Core Responsibilities:

- **Create:** Allow users to record new attempts for each question they have attempted.
- **Read:** Enable users to retrieve their list of attempted questions.
- **Update:** Allow users to update existing question attempt records with new data during ongoing collaboration sessions with the latest code executed.
- **Delete:** Automatically deletes history documents that are 180 days old or later.

## 9.2.2. Internal Design Flow:

### 1. User Interaction

The frontend (client) sends HTTP requests to the History Service to perform operations. These requests contain necessary data (e.g., token, userId, dateAttempted) and are routed to the appropriate endpoints.

### 2. Authentication and Authorization

Upon receiving a request, the History Service passes the attached token to Auth Middleware ([Section 5.2.1](#)) to check for the following:

- **Authentication:** Ensures the user is authenticated and has a valid Peerprep account.

### 3. Microservice Interaction

Once authenticated, the History Service interacts with:

1. **User Service:** To retrieve or update the list of attempted questions for the user, including creating or modifying a question attempt record.
2. **Question Service:** To fetch detailed information about the questions (e.g., titles, descriptions) based on the question IDs, if needed.

### 4. Data Transformation (if required)

If required, the History Service transforms the retrieved data from both User Service and Question Service before returning it to the frontend. For example, it would populate the data obtained from the User Service with additional data from the Question Service, such as question titles, descriptions, or other related details before being pushed to the frontend.

### 5. Response to Frontend

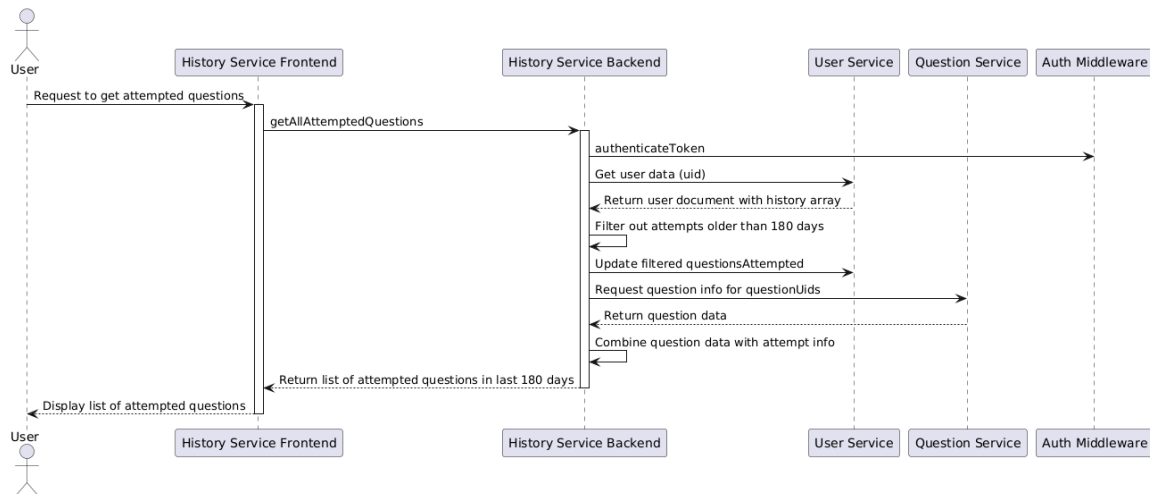
After processing the request, the History Service sends the appropriate HTTP response to the frontend. This response may include:

- A success or failure status
- The requested attempted question data (e.g., list of attempted question)

- Any errors or validation messages if the request was invalid or unauthorized

### 9.2.2.1. Internal Design Flow Example:

Below is an example workflow illustrating how a user retrieves a list of attempted questions from the database.



**Fig 33: Sequence Diagram of `getAllAttemptedQuestions` API Call**

1. Client sends a POST request to the History Service to retrieve the list of attempted questions.
2. The History Service authenticates the user through the Auth Middleware ([Section 5.2.1](#)).
3. Once authenticated, the History Service queries the User Service to obtain the user's document, which includes the list of attempted question UIDs.
4. The History Service filters the list of attempted questions, removing any attempts older than 180 days.
5. It then updates the User Document with the filtered list.
6. The History Service calls the Question Service to fetch detailed data for each of the filtered question UIDs.
7. The History Service aggregates data from both the User Service and Question Service, performing any necessary transformations (e.g., adding question data, formatting data) to align with the frontend's requirements.
5. The History Service returns the aggregated and transformed data back to the frontend, which displays it to the user.



## 9.2.3. Design Considerations

### 9.2.3.1. Storing of questions attempted

A key design consideration was whether to record attempted questions within each user's document in Firebase Firestore in an array field called `questionsAttempted`, or to create a separate `history` collection where each document represents an attempted question.

Ultimately, the first approach was chosen, as it provides direct access to a user's full question history by simply navigating to their document. However, this approach introduces limitations. Due to Firebase's [1,048,487-byte limit per field](#), it would be infeasible to store the entire question content within each `questionsAttempted` entry. Instead, we store only the `questionUid`, along with `dateAttempted` and `codeWritten` (the user's own code).

Assuming each `questionsAttempted` entry averages 1,000 bytes (as it includes the user's code), the array will reach its maximum size after approximately 1,000 entries, preventing further additions.

To address this, we implemented a 180-day TTL (Time to Live) for each entry in the `questionsAttempted` array. After 180 days, entries are automatically deleted. With a capacity of 1,000 questions in 180 days, this setup allows users to attempt about 5 to 6 questions daily—well above typical usage. Additionally, 1,000 bytes per entry is a conservative estimate, ensuring that this design can meet user needs effectively.

### 9.2.3.2. Handling deleted questions in users history

Another important design consideration was how to manage cases where a question is deleted after a user has attempted it. Since we store only the `questionUid` in the `questionsAttempted` array, deleting a question would result in the loss of information such as the question's title and description.

To address this, we provide a default placeholder for deleted questions with a message like "This question has been deleted from the database." This placeholder appears when users view their question history, while the details of their previous attempts, including `dateAttempted` and `codeWritten`, remain accessible.

A possible future improvement would be to implement soft deletion rather than hard deletion of questions. This approach would involve adding a "deleted" status field to

question documents, allowing us to retain access to the question's title and other fields, even if the question is marked as deleted.

#### *9.2.3.3. Criteria for marking a question as attempted*

A key design consideration was determining the criteria for marking a question as "attempted" in the history service. Should a question be considered attempted when the user navigates to the collaboration page and views the question, or only when he clicks "run code"?

We chose the latter approach, as it provides a more meaningful record. Logging an attempt only when the user has actively run their own code ensures that the history reflects genuine attempts and past code submissions—information the user is more likely to find useful and relevant.

## 10. API Documentation

This section provides an overview of all the API endpoints in the backend. It outlines the routes, request types, and required parameters or request bodies for each operation. These endpoints cover a variety of actions such as retrieving, creating, editing, and deleting resources, along with other specific functionalities. This documentation serves as a comprehensive guide for developers to understand how to interact with the backend API to perform various tasks and manage data across the entire system.

### 10.1. User Service

HTTP Method	API Route	Purpose	Payload (JSON)	Require Admin?
POST	/verify-token	Verifies the authentication token and returns user data	-	No
POST	/user/addToUserCollection	Adds a new user to the Firestore collection	uid, email, username	No
POST	/user/removeFromUserCollection	Removes a user from the Firestore collection	uid	No
GET	/admin/checkAdminStatus	Checks if the user is an admin based on the token	-	No

GET	/check-username	Checks if a username exists in the Firestore collection	username	No
GET	/user/username/:uid	Retrieves the username associated with a UID	-	No

**Fig 34: User Service API Endpoint**

## 10.2. Question Service

The following are the API routes for question service:

HTTP Method	API Route	Purpose	Payload (JSON)	Require Admin?
GET	/get-all-questions	Get all questions	-	No
POST	/get-questions-of-topic-and-difficulty	Get all questions of a certain topic and difficulty	topic difficulty	No
POST	/create-question	Create a new question	title, description, difficulty, topics, example, constraints	Yes

PUT	/edit-question	Edits an existing question	questionId, title, description, topics, difficulty	Yes
DELETE	/delete-question	Delete an existing question	questionId	Yes
POST	/get-questions-by-ids	Get questions by specified ids	Ids (array)	No

**Fig 35: Question Service API Endpoints**

### 10.3. Matching Service

The matching service does not use any API routes, as all matching functionalities are implemented via WebSocket events.

### 10.4. Collaboration Service

The following are the API routes for collab service:

HTTP Method	API Route	Purpose	Payload (JSON)	Require Admin?
POST	/create-session	Create new collab session	sessionId, sessionData, questionData	No

POST	/verify-session	Verify if a collab session exists	sessionId	No
POST	/execute-code	Execute code in the specified language	language, code, langVer	No

**Fig 36: Collab Service API Endpoints**

## 10.5. History Service

The following are the API routes for history service:

HTTP Method	API Route	Purpose	Payload (JSON)	Require Admin?
POST	/get-all-attempted-questions	Get a list of questions attempted by the user within the last 180 days	userId	No
POST	/create-question-attempted	Create a new history document when a session is created	userId, questionId, dateAttempted	No
POST	/store-user-executed-code	Update the history document with the code executed by the user during the session	userId, questionId, dateAttempted, codeWritten	No

**Fig 37: History Service API Endpoint**

## 11. Project Management

### 11.1. Product Backlog

We leveraged a streamlined task-tracking system that centered around Excel for maintaining a comprehensive backlog of products and tasks. This Excel-based backlog served as a central repository where each item was categorized, prioritized, and updated regularly. Each row represented an individual task or product feature, accompanied by columns detailing priority, expected completion date, and allocated team members. This approach provided the team with a quick view of project priorities and statuses and also facilitated transparent and real-time updates, enabling every member to have a clear understanding of what needed attention, helping minimize any potential overlap or miscommunication.

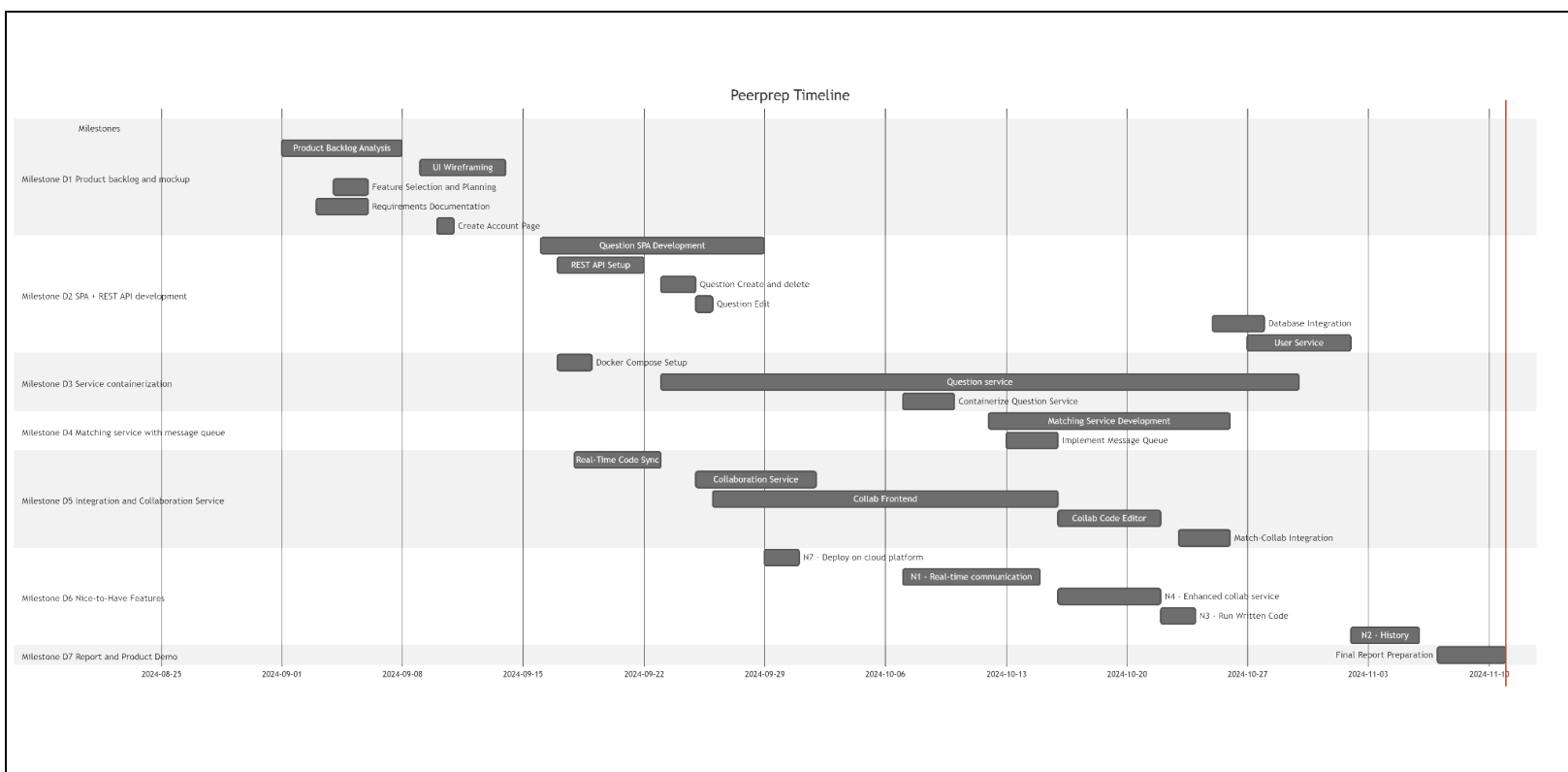
### 11.2. GitHub

To track and measure our progress, we also utilized GitHub, which allowed us to break down our backlog into smaller, manageable goals aligned with our project's phases. By associating each GitHub milestone with specific deliverables, we could better visualize the completion of critical project stages and prioritize tasks that directly contributed to these objectives. Each milestone included a timeline and clearly defined objectives, which ensured that team members understood the importance of their contributions and adhered to project timelines. The approach provided an efficient way to document our iterative progress while maintaining a consistent workflow.

### 11.3. Agile Framework

Our team adopted the Agile framework, where we structured our development into weekly sprints, with Monday being designated for sprint planning and reviews. Every Monday, we discussed the previous week's progress, assessed any roadblocks, and set goals for the upcoming week. This cycle of short, achievable sprints fostered an adaptive workflow, enabling the team to respond flexibly to new insights and priorities while ensuring steady, incremental progress. By setting focused objectives at the start of each week, we maintained clear alignment across team members, staying focused on our immediate goals while being able to adjust our approach based on ongoing feedback and the evolving project requirements.

## 11.4. Gantt Chart



**Fig 38: Gantt chart (For an enlarged version, click [here](#))**

The above Gantt chart outlines the project's timeline, organized by milestones and their associated tasks. Each milestone represents a specific project phase, with tasks grouped according to their start times within the milestone. Building on the previous milestone to maintain continuity and allow flexibility in timelines. Notably, some tasks required extended durations due to an incremental approach adopted in the development process, especially for core functionalities like the SPA, REST API setup, and real-time features.

This approach allowed for flexibility in adapting to changing requirements and integrating feedback progressively. However, it also meant that certain tasks evolved over multiple iterations, leading to adjustments in our initial timelines. The structured milestone grouping within the chart provides a clear view of task dependencies and project flow, reflecting our commitment to continuous refinement and quality improvement throughout the project timeline.



## 12. Conclusion

### 12.1. Future Enhancements

#### 12.1.1. Test Cases

For future enhancements, we envision incorporating features that allow users to access sample answers and test cases for a more enriched coding experience. In a collaborative setting, these additions could provide real-time feedback loops, enabling users to compare their solutions with sample answers and refine their approach. The introduction of shared test cases would also foster collaborative debugging, allowing users to validate their solutions collectively. This functionality enhances learning outcomes by offering a deeper understanding of problem-solving approaches.

#### 12.1.2. Loose Matching Criteria

A relaxed matching criterion could be beneficial for improving matching times, especially in scenarios with a low user count. By broadening the parameters for potential matches, the system can increase the likelihood of finding compatible users faster, minimizing wait times. This approach is particularly useful in early stages of the platform or during off-peak hours, as it allows more flexibility in matching based on less stringent criteria. In turn, this enhances the user experience by reducing idle time and promoting more frequent interactions, even when the pool of available users is small.

#### 12.1.3. Soft Deletion of Questions

A potential future enhancement could involve implementing soft deletion instead of hard deletion for questions. This would entail adding a "deleted" status field to question records, enabling the application to retain access to the question's title and other details, even if marked as deleted. With this approach, users could still view previously deleted questions through the history service, preserving access to essential information from past questions without permanently removing them.

#### 12.1.4. Collaborative Whiteboard and Diagramming Tools

Another possible future enhancement for our application is the addition of a Collaborative Whiteboard and Diagramming Tools feature. This would allow users to visually brainstorm and outline complex algorithms, data structures, and problem-solving strategies in real time. By enabling users to sketch flowcharts, pseudocode, or annotate algorithmic steps, this feature would provide a shared space for conceptual planning, enhancing communication and understanding among participants. In a collaborative context, the whiteboard would foster a more dynamic and interactive experience, where users could analyze and refine their approaches collaboratively. This addition would create an integrated environment for both coding and strategic discussion, ultimately strengthening the teamwork and problem-solving skills of all participants.

### 12.2. Reflections

#### 12.2.1. Technical Growth

One major takeaway from PeerPrep was the opportunity to deepen technical expertise, especially in areas like backend development, database integration, and real-time communication for pairing users. From setting up servers to managing data flows, the project reinforced the importance of a well-architected system and solidified the team's understanding of concepts like scalability, performance optimization, and system resilience. Testing and debugging in this environment was invaluable and added to our problem-solving skills.

#### 12.2.2. Collaboration and Communication

PeerPrep was a team-based effort, and success hinged on strong communication and collaboration. We learned the value of consistent updates, documentation, and clear role delineation. Effective communication helped us tackle challenges, whether by jointly debugging issues or brainstorming features. This project also highlighted the significance of code reviews and version control, particularly in a collaborative environment.

### 12.2.3. Adaptability and Problem-Solving

Working on PeerPrep underscored the need for adaptability. The project threw some curveballs, whether technical setbacks or feature adjustments. Being agile and receptive to changes proved crucial. Each challenge refined our problem-solving approach and strengthened our ability to pivot as needed.

## 12.3. Acknowledgement

We would like to express our heartfelt appreciation to our Teaching Assistant, Lee Zi Yang, for his invaluable guidance and unwavering support throughout the development of our PeerPrep project. Zi Yang's insightful feedback and constructive suggestions have greatly contributed to the improvement and refinement of our work. His prompt responsiveness and proactive assistance have been instrumental in helping us navigate challenges and ensuring the successful progress of the project. We are deeply grateful for his dedication and support.

Additionally, we would also like to extend our sincere appreciation to our professors, Dr. Akshay Narayan and Dr. Bimlesh Wadhwa, for their engaging and thought-provoking lectures throughout this course. Their teaching not only provided us with key concepts in software engineering but also encouraged us to think critically about their real-world applications. Their enthusiasm and commitment to our learning made even the most complex topics easier to understand and enjoyable, greatly enhancing our understanding and application of software development principles throughout the PeerPrep project.

Thank you, Dr. Narayan and Dr. Wadhwa, for creating such a supportive and inspiring learning environment. Your guidance and encouragement have been invaluable to us.