CS3219 G17 Report: PeerPrep

Chloe Lim Xinying, Justin Lam Seng Onn, Leong See Leng, Sheryl-Lynn Tan, Zheng Jiarui

Table of Contents

A. Introduction

Background

In today's rapidly evolving job market, technical interviews are a critical component in the hiring process for companies in the software industry. The ability to excel in technical interviews is essential, and it is no surprise that candidates who perform well in these interviews are more likely to secure coveted positions at companies. This inspires the creation of our application, PeerPrep. By leveraging the power of collaboration, real-time interaction, and progress tracking, PeerPrep provides an intuitive and accessible way for candidates to prepare for technical interviews.

About PeerPrep

PeerPrep is a desktop web application, designed to provide a platform for individuals to hone their technical interviewing skills. The application offers a suite of features carefully crafted to optimise the interview preparation process.

PeerPrep facilitates meaningful learning experiences by matching users with compatible peers. Once matched, users engage in joint interview sessions, collaboratively solving two interview questions. We chose to default users to attempt two questions to replicate the experience of a structured interview session. This approach allows each user to take on the dual roles of both interviewer and interviewee, fostering a well-rounded and immersive learning experience. To enhance the collaborative experience, PeerPrep includes a real-time chat functionality. This allows users to communicate seamlessly during the interview session, exchange ideas, and seek clarification on concepts. Effective communication is a cornerstone of successful teamwork, and PeerPrep ensures that this vital aspect is seamlessly integrated into the preparation process. To facilitate users' learning, completed interview sessions are automatically saved in the user's profile, helping users to keep track of which types of questions they have more experience in and should work more on.

As a side feature, recognizing that individuals have varying schedules and learning preferences, PeerPrep offers the option to attempt practice questions at one's own convenience. Users can access a rich library of questions, sorted by category and difficulty level, allowing them to customise their practice sessions to align with their specific needs.

B. Requirements

Functional Requirements

Functional requirements	Priority	Sprint
F1 User Service (M1)		
F1.1 User Registration and Authentication		
F1.1.1 Users can create an account with a unique username and password, and a recovery email.	H	2
F1.1.1.1 Password should have certain requirements (e.g. no fewer than 8 characters)	L	2
F.1.1.1.2 Password should be stored securely, by using tools like bcrypt (hash and store passwords)	H	2
F1.1.2 Users can log in securely using their registered credentials.	H	2
F1.1.3 Users can reset their password.	L	2
F1.1.4 Users should still be logged into their session even if they leave the website page.	M	2
F1.2 User Profile Management (M1)		
F1.2.1 Users can update their profile information.	H	2
F1.2.2 Users can view and edit their preferred topic for interview preparation.	H	2
F2 Matching Service (M2)		
F2.1 Users can be matched with another user who has a compatible selected topic and question difficulty level.	H	3
F2.2 If a matching is not found after 30 seconds, the matching will be timed out.	H	3
F2.3 Users can see the time countdown when the matching is being done.	H	3
F2.3 When the matching does not time out yet, users have the option to cancel the matching process manually.	M	3
F2.4 If a match is found, both students receive a notification.	H	3

Functional requirements	Priority	Sprint
F3 Question Service (M3)		
F3.1 Question Management		
F3.1.1 There is a database that contains all available interview questions.	H	1
F3.1.2 The questions are categorised by topics.	H	1
F3.1.3 In each topic, the questions are categorised by difficulty levels.	H	1
F3.1.4 Maintainers can add questions. (non-duplicated)	H	1
F3.1.5 Maintainers can update questions. (non-duplicated)	H	1
F3.1.6 Maintainers can delete questions.	H	1
F3.2 Question Practices (Individual/ Practice session)		
F3.2.1 Users can search the questions by keywords or filter by categories.	L	1
F3.2.2 Users can look at details of a specific question. (e.g. question description).	H	1
F3.2.3 Rich-text display is supported for question descriptions.	L	3
F3.2.4 Users can attempt questions by writing their answers in a code editor that supports syntax highlighting for multiple (popular) languages.	M	3
F4 Collaboration Service (paired session) (M4)		
F4.1 Both students enter a collaborative space where they can view the 1st question. (There will be a default 2 questions in each collaborative session).	H	3
F4.2 Both students can add and edit code in the collaborative code editor.	H	3
F4.3 Both students can see their partner’s input in real-time.	H	3
F4.4 When a student wants to proceed to the next question, a notification will appear on the screen of the other student. Only when both students agree to proceed, the collaborative space will show the next question.	H	3
F4.5 If a user accidentally disconnects, they should be able to rejoin the session.	M	3
F4.6 Users can choose to exit the session, both students will see a notification and the collaborative session will end.	L	3
F4.7 Users can expect a secure collaboration room.	H	4

Functional requirements	Priority	Sprint
F5 Progress Management (N1)		
F5.1 Analytics Service		
F5.1.1 Users should be able to track their activity and progress (eg: which questions were attempted, and when they were attempted).	M	4
F5.1.2 Users should have access to visualisation tools for enhanced data representation.	M	4
F5.1.3 Users should be able to track the number of questions they have attempted over a duration.	M	4
F6 Code Help (N2)		
F6.1 A community thread where users can request help/ hints from other users, or discuss their answers.	L	4
F6.2 Users can comment on existing posts to answer questions or add on to the discussion.	L	4
F6.3 Users can upvote or downvote posts for quality assessment.	L	4
F6.4 Users can search for posts to find if there are any existing discussions that align with their topics of interest, so that they could get relevant information more quickly.	L	4
F6.5 Users can upvote or downvote comments for quality assessment.	L	4
F7 Communication (N3)		
F7.1 Users in the same collaborative session can chat with each other real-time during the collaborative session via an in-session chatroom.	M	4
F7.2 Users should have the ability to view a list of participants in the collaborative session, indicating who is currently active in the chatroom.	M	4
F7.3 Users can view message timestamps.	M	4
F8 Improved Code Editor (N5)		
F8.1 Users can automatically format their code to correct indent and format.	L	3
F8.2 Users can view their code easily with code syntax highlighting (regardless of their language).	L	3

F8.3 Users can make use of a sketchpad/ whiteboard to doodle or jot down rough workings/ do planning while attempting a question.	L	4
F8.4 Users can request a hint for the question they are attempting.	L	4
F9 Code Execution (N6)		
F9.1 Users can execute their code in common programming languages.	L	4
F9.2 Real-time feedback on code execution results should be provided, such as errors messages and output generated.	L	4
F9.3. Users can choose the language for code execution.	L	4
F9.4 Users can execute standard input and output operations, various data types and operations.	L	4

Non-Functional Requirements

Non-Functional requirements
NF1 Reliability: The app should be able to run reliably, minimising faults even during traffic peaks.
NF1.1 The application should be able to service a large number of users (at least 500 users) at any one time without faults.
NF2 Usability: The app should be easy to navigate and understand.
NF2.1 The app should provide clear and concise feedback to the user whenever appropriate. This includes feedback about the status of the system, and error messages.
NF2.2 The app should use language, icons, and follow conventions familiar to the user. The app should avoid the use of jargon that may be peculiar or unfamiliar to the user.
NF2.3 The app should prevent user errors where possible through the use of sufficient safeguards (such as confirmation dialogs on dangerous actions).
NF2.4 The app should only display information relevant to the user at any one time, and avoid cluttering the UI unnecessarily.
NF3 Security: The app should be secure and protect the privacy and confidentiality of its users.
NF3.1 Sensitive data stored in the database should be encrypted and stored securely. This means that the system should avoid storing sensitive data such as passwords in plain text.
NF3.2 Microservices should protect sensitive routes so that only authenticated users can access these routes through secure authentication means.
NF4 Maintainability: The app should be designed with modularity and extensibility in mind, and should be easy to test and debug.
NF4.1 Code written should follow a consistent and good coding style to improve readability, maintainability and code quality.
NF4.2 Developers should aim to write DRY code, reducing code duplication such as by doing consistent refactoring when appropriate.
NF4.3 Developers should strive to write modular code. This means making use of design patterns and keeping in mind principles such as the separation of concerns and the single responsibility principle.
NF5 Compatibility: The app shall be designed so that it supports most common operating systems - Mac, Windows and Linux.
NF6 Portability: The app should be easily portable, even across different operating systems. This means one should be able to install and set up the app without much trouble and in a reasonable time (< 30 minutes).

NF7 Performance
NF7.1 The application should minimise the latency (< 0.1s) for incoming and outgoing chat messages. There should not be any noticeable delays in sending or receiving messages.
NF7.2 The application should minimise the latency (<0.1s) for typing on the real-time code editor. The synchronisation between editors in the same collaboration session should not have noticeable delays.

Functional Requirement Fulfilment

This section examines how each functional requirement is addressed within the development process.

Functional requirements	GitHub PR
F1 User Service (M1)	
F1.1 User Registration and Authentication	
F1.1.1 Users can create an account with a unique username and password, and a recovery email.	[#54] feat: add login / sign up / log out
F1.1.1.1 Password should have certain requirements (e.g. no fewer than 8 characters)	[#54] feat: add login / sign up / log out
F.1.1.1.2 Password should be stored securely, by using tools like bcrypt (**hash and store passwords**)	[#54] feat: add login / sign up / log out
F1.1.2 Users can log in securely using their registered credentials.	[#54] feat: add login / sign up / log out
F1.1.3 Users can reset their password.	[#74] [backend] feat: add reset password [#120] feat: frontend forgot password
F1.1.4 Users should still be logged into their session even if they leave the website page.	[#71] frontend: auth + other misc fixes [#136] URGENT JWT Fixes: Backend Storage, Bugs
F1.2 User Profile Management (M1)	
F1.2.1 Users can update their profile information.	[#67] Edit user profile
F1.2.2 Users can edit the profile to include their preferred coding language.	[#67] Edit user profile
F2 Matching Service (M2)	
F2.1 Users can be matched with another user who has a compatible selected topic and question difficulty level.	[#76] feat: add matching logic
F2.2 If a matching is not found after 30 seconds, the matching will be timed out.	[#76] feat: add matching logic
F2.3 Users can see the time countdown when the matching is being done.	[#90] feat: frontend timer

F2.3 When the matching does not time out yet, users have the option to cancel the matching process manually.	[#90] feat: frontend timer
F2.4 If a match is found, both users receive a notification.	[#76] feat: add matching logic
F3 Question Service (M3)	
F3.1 Question Management	
F3.1.1 There is a database that contains all available interview questions.	[#13] Setup backend [#20] Implement Questions Listing
F3.1.2 The questions are categorised by topics.	[#13] Setup backend [#20] Implement Questions Listing
F3.1.3 In each topic, the questions are categorised by difficulty levels.	[#13] Setup backend [#20] Implement Questions Listing
F3.1.4 Maintainers can add questions. (non-duplicated)	[#14] feat: add api endpoint for creating questions [#19] feat: add frontend form for adding questions
F3.1.5 Maintainers can update questions. (non-duplicated)	[#27] [frontend] [backend] add backend API, frontend form for editing questions
F3.1.6 Maintainers can delete questions.	[#16] [backend] feat: delete question [#33] [frontend] delete question
F3.2 Question Practices (Individual/ Practice session)	
F3.2.1 Users can search the questions by keywords or filter by categories.	[#20] Implement Questions Listing
F3.2.2 Users can look at details of a specific question. (e.g. question description).	[#44] [frontend] Individual question
F3.2.3 Rich-text display is supported for question descriptions.	[#55] Rich text editor
F3.2.4 Users can attempt questions by writing their answers in a code editor that supports syntax highlighting for multiple (popular) languages.	[#51] [frontend] feat: add code editor
F4 Collaboration Service (paired session) (M4)	
F4.1 Both students enter a collaborative space where they can view the 1st question. (There will be a default 2 questions in each collaborative session).	[#69] [frontend] Real time editor
F4.2 Both students can add and edit code in the collaborative code editor.	[#69] [frontend] Real time editor

F4.3 Both students can see their partner's input in real-time.	[#69] [frontend] Real time editor
F4.4 When a student wants to proceed to the next question, a notification will appear on the screen of the other student. Only when both students agree to proceed, the collaborative space will show the next question.	[#101] feat: next question + authorise room + profile page
F4.5 If a user accidentally disconnects, they should be able to rejoin the session.	[#69] [frontend] Real time editor
F4.6 Users can choose to exit the session, both students will see a notification and the collaborative session will end.	[#101] feat: next question + authorise room + profile page
F4.7 Users can expect a secure collaboration room.	[#101] feat: next question + authorise room + profile page
F5 Progress Management (N1)	
F5.1 Analytics Service	
F5.1.1 Users should be able to track their activity and progress (eg: which questions were attempted, and when they were attempted).	[#101] feat: next question + authorise room + profile page
F5.1.2 Users should have access to visualisation tools for enhanced data representation.	[#101] feat: next question + authorise room + profile page
F5.1.3 Users should be able to track the number of questions they have attempted over a duration.	[#101] feat: next question + authorise room + profile page
F6 Code Help (N2)	
F6.1 A community thread where users can request help/ hints from other users, or discuss their answers.	[#108] Forum service
F6.2 Users can comment on existing posts to answer questions or add on to the discussion.	[#108] Forum service
F6.3 Users can upvote or downvote posts for quality assessment.	[#108] Forum service
F6.4 Users can search for posts to find if there are any existing discussions that align with their topics of interest, so that they could get relevant information more quickly.	[#108] Forum service

F6.5 Users can upvote or downvote comments for quality assessment.	[#108] Forum service
F7 Communication (N3)	
F7.1 Users in the same collaborative session can chat with each other real-time during the collaborative session via an in-session chatroom.	[#100] nice to have: chat service
F7.2 Users should have the ability to view a list of participants in the collaborative session, indicating who is currently active in the chatroom.	[#69] [frontend] Real time editor
F7.3 Users can view message timestamps.	[#100] nice to have: chat service
F8 Improved Code Editor (N5)	
F8.1 Users can automatically format their code to correct indent and format.	[#51] [frontend] feat: add code editor
F8.2 Users can view their code easily with code syntax highlighting (regardless of their language).	[#51] [frontend] feat: add code editor
F8.3 Users can make use of a sketchpad/ whiteboard to doodle or jot down rough workings/ do planning while attempting a question.	[#126] feat: add whiteboard
F8.4 Users can request a hint for the question they are attempting.	[#131] feat: init help service
F9 Code Execution (N6)	
F9.1 Users can execute their code in common programming languages.	[#111] Code execution
F9.2 Real-time feedback on code execution results should be provided, such as errors messages and output generated.	[#111] Code execution
F9.3. Users can choose the language for code execution.	[#111] Code execution
F9.4 Users can execute standard input and output operations, various data types and operations.	[#111] Code execution

Non-Functional Requirement Fulfilment

NFR 1: Reliability

NF1 Reliability: The app should be able to run reliably, minimising faults even during traffic peaks.
NF1.1 The application should be able to service a large number of users (**at least 500** users) at any one time without faults.

To check that we satisfied NFR 1 (Reliability), we performed load testing with the Locust framework. Below, you can see that, when we set the load test to **500 users**, there are still **0 failures** as the question and user service both scale with the increasing number of requests. This is possible because our cloud run deployment will start up new instances to respond to the increasing number of requests, as you can see in the figures below.

Request Statistics

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/api/questions/questions	5966	0	1251	289	10273	180348	61.2	0.0
POST	/api/users/login	6094	0	4599	639	10875	735	62.6	0.0
	Aggregated	**12060**	**0**	**2943**	**289**	**10875**	**89588**	**123.8**	**0.0**

Response Time Statistics

Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/api/questions /questions	1000	1100	1300	1600	2100	2600	4300	10000
POST	/api/users/login	5300	6000	6200	6400	6700	7100	8300	11000
	Aggregated	**1800**	**2700**	**4500**	**6000**	**6400**	**6700**	**7800**	**11000**

Fig. Load testing statistics

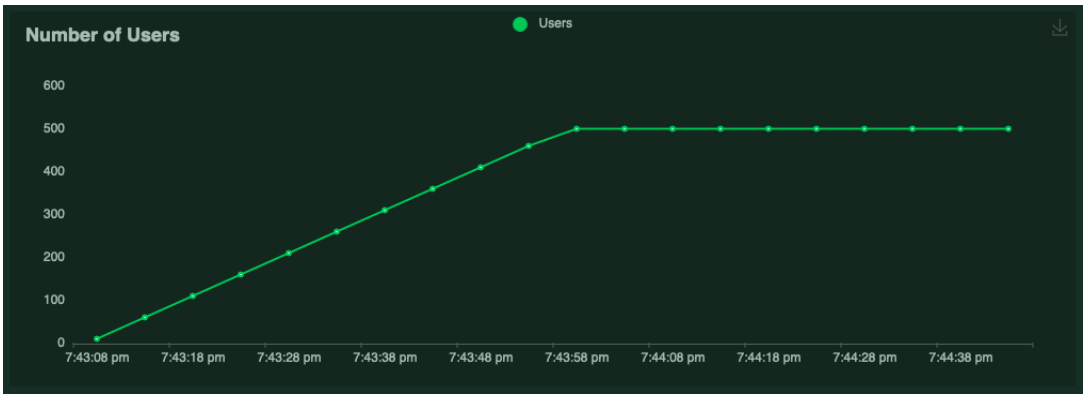


Fig. Number of users

Fig. Total RPS and Response times graph

NFR 2: Usability

NF2 Usability: The app should be easy to navigate and understand.
NF2.1 The app should provide clear and concise feedback to the user whenever appropriate. This includes feedback about the status of the system, and error messages.
NF2.2 The app should use language, icons, and follow conventions familiar to the user. The app should avoid the use of jargon that may be peculiar or unfamiliar to the user.
NF2.3 The app should prevent user errors where possible through the use of sufficient safeguards (such as confirmation dialogs on dangerous actions).
NF2.4 The app should only display information relevant to the user at any one time, and avoid cluttering the UI unnecessarily.

NF2.1 is fulfilled through the use of appropriate feedback to users. This takes the form of using spinners and skeleton loading components when loading content, and using toast messages to show success, error and info messages.

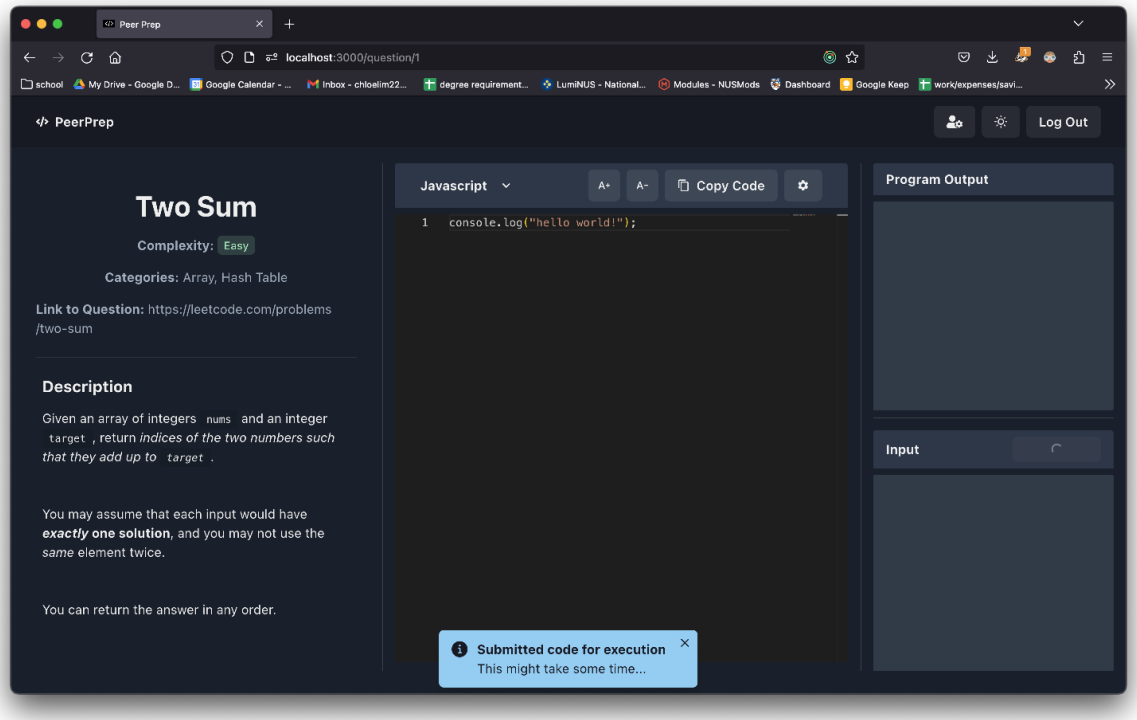


Fig. Example of feedback to users through toast messages and loading skeletons/ spinners

NF2.2 is also fulfilled through the use of words, phrases and language familiar to the user. For instance, the naming of “practice rooms” matches the real-world concept of using rooms to use for practice sessions in real life.

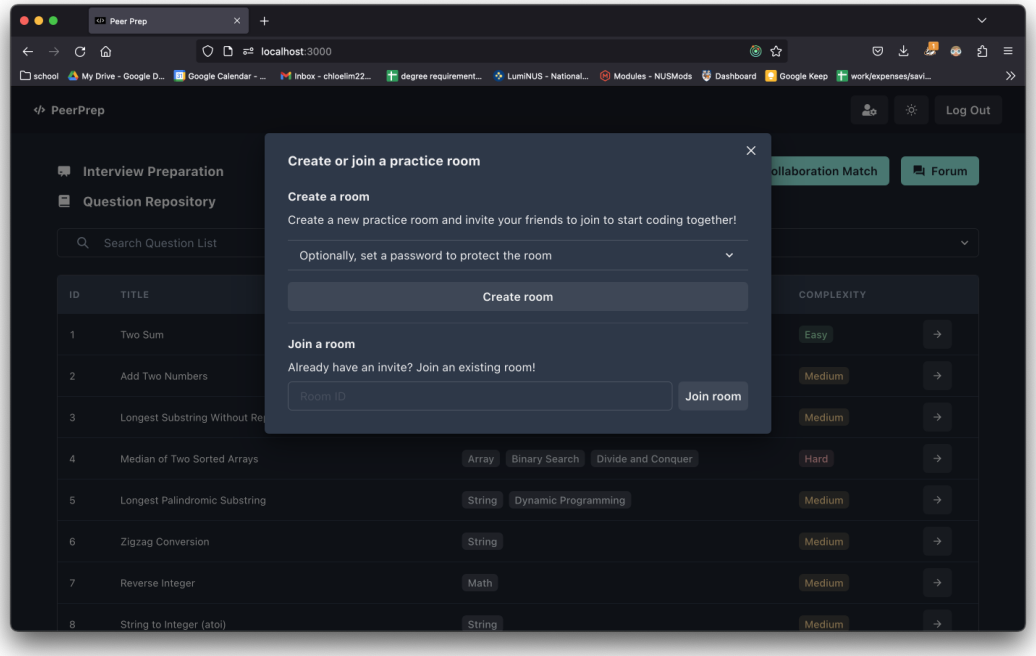


Fig. Matching of real world concepts of rooms with ‘practice rooms’

NF2.3 is fulfilled through the use of confirmation dialogs for dangerous actions, such as deleting a user account.

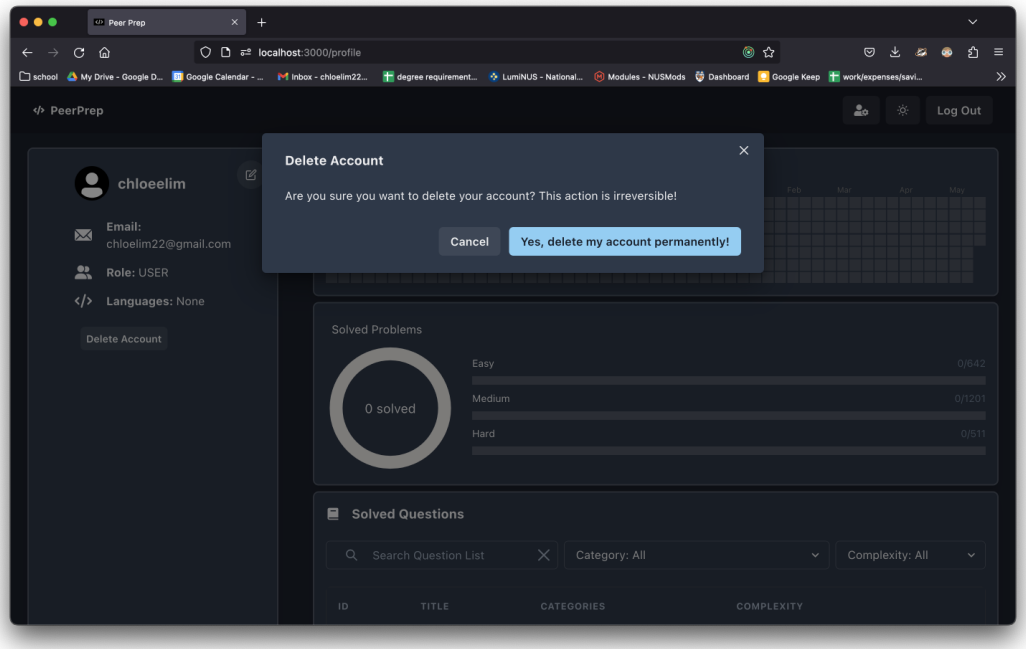


Fig. Use of confirmation dialogs for error prevention

NF2.4 is also fulfilled by reducing the amount of information being shown to the user at any one time so as to maintain a sleek, minimalist interface that does not overload the user with unnecessary and irrelevant information.

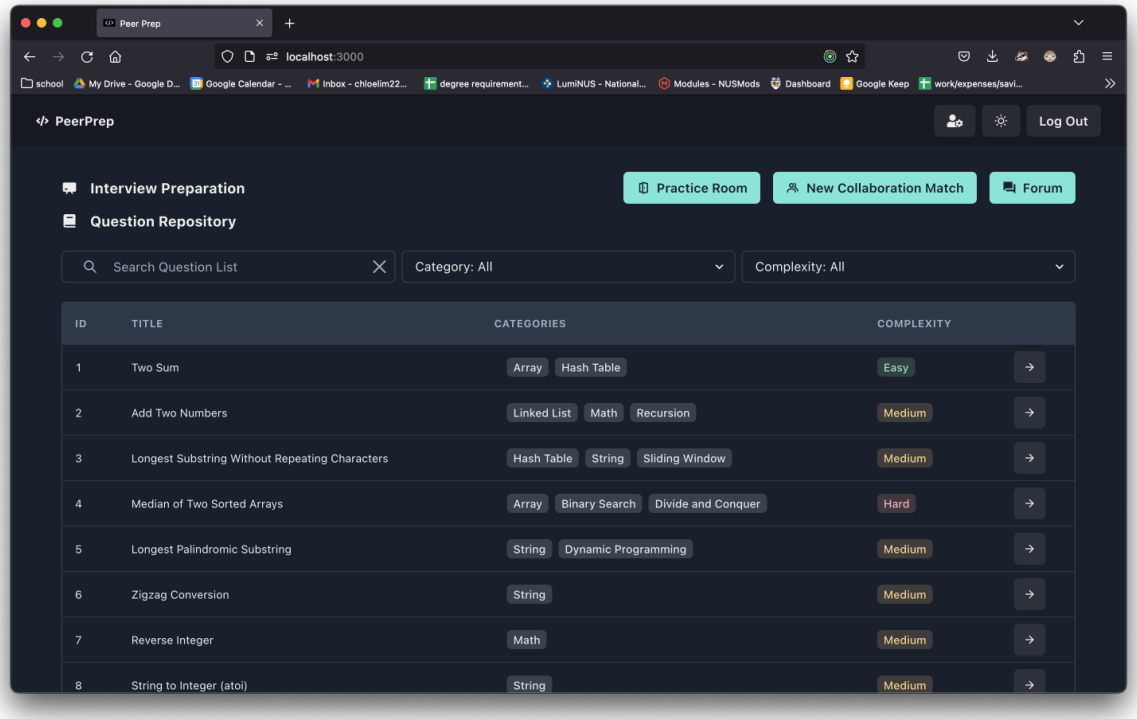


Fig. PeerPrep's sleek and clean interface

NFR 3: Security

NF3 Security: The app should be secure and protect the privacy and confidentiality of its users.
NF3.1 Sensitive data stored in the database should be encrypted and stored securely. This means that the system should avoid storing sensitive data such as passwords in plain text.
NF3.2 Microservices should protect sensitive routes so that only authenticated users can access these routes through secure authentication means.

NF3.1 is fulfilled as passwords in our database hashed with bcrypt, rather than storing passwords plaintext in our database.

```
1   import bcrypt from "bcryptjs";
2   import crypto from "crypto";
3
4   // Password Hashing                    used during sign up
5   export function hashPassword(password: string): string {
6     return bcrypt.hashSync(password, 8);
7   }
8                                         used during log in
9   export function comparePassword(password: string, hash: string): boolean {
10    return bcrypt.compareSync(password, hash);
11  }
```

Fig. user-service (auth.ts)

NF3.2 is also fulfilled as routes in microservices are protected, our backend services will first check that the users are properly authenticated before servicing the request (with the exception of sign up/ sign in routes).

question-service - app.ts

```
// Protected API routes and respective protect middleware
app.use("/api/questions", AuthMiddleWare.verifyAccessToken, questionRoutes);
app.use(
  "/api/questions",
  AuthMiddleWare.verifyAccessToken,
  AuthMiddleWare.protectAdmin,
  adminQuestionRoutes,
);
```

Fig. Authentication middleware in question-service

NFR 4: Maintainability

NF4 Maintainability: The app should be designed with modularity and extensibility in mind, and should be easy to test and debug.
NF4.1 Code written should follow a consistent and good coding style to improve readability, maintainability and code quality.
NF4.2 Developers should aim to write DRY code, reducing code duplication such as by doing consistent refactoring when appropriate.
NF4.3 Developers should strive to write modular code. This means making use of design patterns and keeping in mind principles such as the separation of concerns and the single responsibility principle.

NF4.1 is fulfilled through the use of linters to enforce a consistent and good coding style. NF4.2 and NF4.3 were strived for- we looked out for these standards in our peer reviews and tried to achieve them through the use of design patterns.

NFR 5 and 6: Compatibility and Portability

NF5 Compatibility: The app shall be designed so that it supports most common operating systems - Mac, Windows and Linux.
NF6 Portability: The app should be easily portable, even across different operating systems. This means one should be able to install and set up the app without much trouble and in a reasonable time (< 30 minutes).

NF5 and NF6 are fulfilled as our app can run and is supported by the most common operating systems. In addition, our app is also Dockerised, making it easy to set up and run quickly.

NFR 7: Performance

NF7 Performance
NF7.1 The application should minimise the latency (< 0.1s) for incoming and outgoing chat messages. There should not be any noticeable delays in sending or receiving messages.
NF7.2 The application should minimise the latency (<0.1s) for typing on the real-time code editor. The synchronisation between editors in the same collaboration session should not have noticeable delays.

The performance NFRs are also likely to be fulfilled. There are no noticeable delays when typing in the code editor or sending a chat message. For instance, sending a chat message only results in latencies of a few milliseconds which is likely not noticeable.

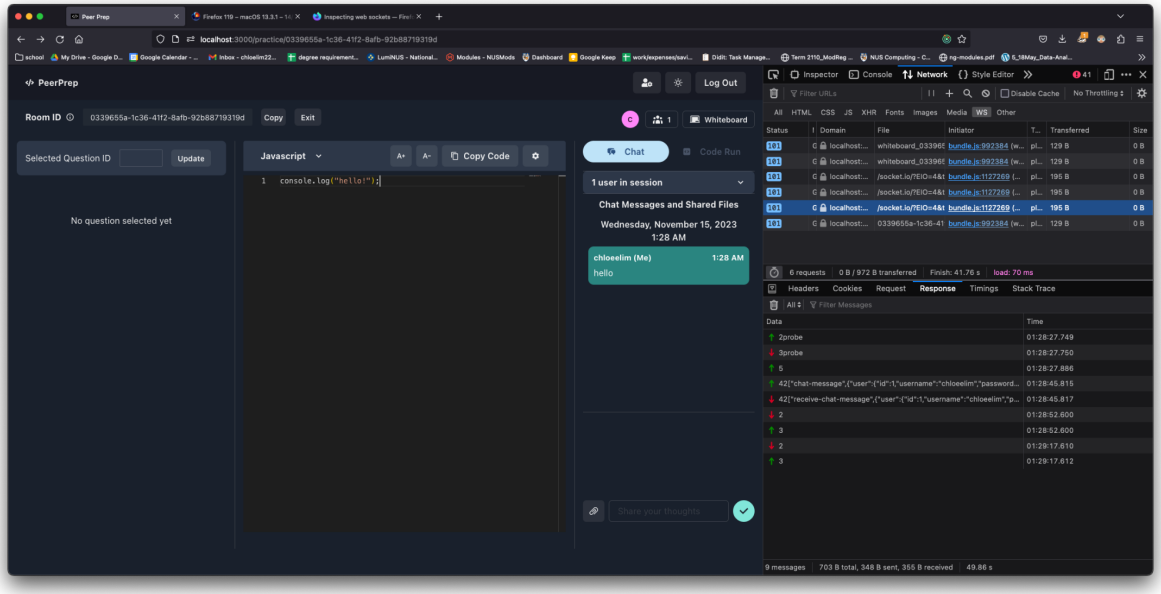


Fig. Websocket communication has low latency

C. Features

While the Must-Have features were done together by all members, the feature allocation is as follows for the Nice-to-Haves.

Subgroup	Members
Subgroup A	Chloe, See Leng
Subgroup B	Justin, Sheryl-Lynn, Jiarui

Must-Haves

Label	Feature
M1	User Service – responsible for user profile management.
M2	Matching Service – responsible for matching users based on some criteria (e.g., difficulty level of questions, topics, proficiency level of the users, etc.) This service can potentially be developed by offering multiple matching criteria.
M3	Question service – responsible for maintaining a question repository indexed by difficulty level (and any other indexing criteria – e.g., specific topics).
M4	Collaboration service – provides the mechanism for real-time collaboration (e.g., concurrent code editing) between the authenticated and matched users in the collaborative space.
M5	Basic UI for user interaction – to access the app developed.
M6	Deploying the application on a local machine (e.g., laptop) using native technology stack. OR Deploy the app on a (local) staging environment (e.g., Docker-based, Docker + Kubernetes).

Nice-to-Haves

Label	Feature	
N1	Progress Management - allows users to easily manage and track their technical preparation progress through analytics.	Subgroup B
N2	Code Help - provides support for users while attempting questions through community help (forum).	Subgroup B
N3	Communication - Implement a chat system to facilitate communication among the participants in the collaborative space. (Chat)	Subgroup B
N4	Deployment - Deployment of the app on the production system (GCP/AWS).	Subgroup A
N5	Improved code editor - Enhance collaboration service by providing an improved code editor with code formatting, syntax highlighting for multiple languages, and tools to generate hints (AI-generated) for a question, should the user request it.	Subgroup A
N6	Code execution: Implement a mechanism to execute attempted solution/code in a sandboxed environment, and retrieve+present the results in the collaborative workspace.	Subgroup A

D. Design

1. Tech Stack

PeerPrep was built with the popular MERN stack, a web popular development framework comprising MongoDB, Express.js, React.js, and Node.js. The project repository can be found here. Some major considerations the team took into account when deciding on the technologies are listed below.

Development Timeline

As our team was working on a tight development timeline, it was important that the technologies we used allowed us to develop quickly and efficiently. This means that the technologies we use should ideally allow for rapid development and prototyping, have extensive support for libraries, are easy to pick up, or are familiar to the team.

Cost Constraints

Being a school project, PeerPrep is working on an extremely tight financial budget. This means that free, open-sourced technologies are preferred over proprietary technologies that require payment.

Integration and Compatibility

With several components making up our application, it is important that these separate components can scale and integrate well together to ensure that they can work cohesively to support key functionalities of PeerPrep. As such, it is important that the technologies we use are compatible with each other.

Industry Standards

It is important that PeerPrep can deliver a performant, and enjoyable experience to its users. As such, it is important that the technologies we choose have good performance, and are of high standards so that they can stack up well against other existing products in the market.

We now discuss the detailed technologies used for the different components of PeerPrep, and our rationale for using them.

Frontend

Technology	Rationale
React.js	React.js is easy to learn, and enables rapid development and prototyping. Our developers are also familiar with the framework. In addition, there are a vast number of libraries that integrate well with React.js. This robust ecosystem can help greatly accelerate our development process. React also has a component-based architecture, which would allow us to build more modular and reusable components. This helps the app scale better. React is also free and open-source.
Typescript	Our team decided to use TypeScript with React, which introduces type safety- this allows us to reduce developer errors arising for type errors, and makes our code more readable and maintainable.
Redux Toolkit	To streamline the use of Redux stores in our React application, we decided to integrate Redux toolkit into PeerPrep. This greatly simplifies the management of Redux stores in our application by eliminating a lot of boilerplate code that would have otherwise have to be written. This also means that Redux Toolkit is highly opinionated, which may be an issue for some custom or complex uses of the Redux store. However, as we do not anticipate any such complicated uses of the Redux store for our application, we felt that any trade offs were well justified.
Chakra UI	Chakra UI is a simple, modular and accessible component library that provides building blocks for building our React application. Chakra UI is easy to learn and use, and streamlines the development process with their well-designed components.
Monaco Editor and yjs-monaco	Monaco Editor is the same code editor that powers VS Code. We decided to pick Monaco Editor over other alternatives because of its rich features, such as syntax highlighting, auto-completion, code folding, error checking, and formatting. To support real-time editing, yjs-monaco is used to bind the editor to a Yjs document.
tldraw	An open-source collaborative digital whiteboard.
Judge0	Judge0 was selected to support code execution in the frontend as it is open-source and offers a free tier. Also, it is easy to integrate and use.
Github OAuth	Github OAuth was integrated to offer users an alternative way of logging into our application. As many of our users would likely have Github accounts, we decided that such means of authentication would be useful for users. In addition, OAuth applications are well supported by Github and are easy to create and integrate into existing applications.
ESLint and Prettier	We decided to integrate linters into our project to enforce a consistent code style and standard to improve maintainability and readability of our code.

Backend

Our backend comprises several separate microservices which make use of different technologies. The architecture can be found in the next section. However, all our backend services are built with Express.js, Node.js, and Typescript.

Technology	Rationale
Node.js and Express.js	Allows us to create fast, lightweight, scalable, and dynamic web applications. Both Node.js and Express.js also have robust community support, and are relatively easy to pick up. Both libraries are also free and open-source.
Typescript	Our team decided to use TypeScript with Express, which introduces type safety- this allows us to reduce developer errors arising for type errors, and makes our code more readable and maintainable.
ESLint	We decided to integrate linters into our project to enforce a consistent code style and standard to improve maintainability and readability of our code.

For our databases, we used either PostgreSQL, a relational database management system, or MongoDB, a NoSQL database program. More details about our database schema and the reason for selecting either database for each microservice can be found in the database schema section of this report.

Technology	Rationale
PostgreSQL	PostgreSQL is a powerful, open source object-relational database system known for its reliability, feature robustness, and performance.
MongoDB	MongoDB is a highly popular NoSQL database that uses a document-oriented model to store data. It scales particularly well due to its flexibility, and can adapt well to evolving data structures. It is also performant and easy to use and integrate.

User Service

Technology	Rationale
PostgreSQL	PostgreSQL excels at handling relational data, making it suitable for scenarios where there are many-to-many relationships, as in the case of users and their answered questions. In addition, considering how future iterations of PeerPrep might grow to include more complex relations between users (such as supporting friends, or followers), or types of users and managing user permissions, we felt that PostgreSQL's relational model would better support the needs of the user service.
Prisma	Prisma provides a type-safe API for interacting with the database, which ensures that data is handled consistently and accurately.
Nodemailer	An open-source module for Node.js to send emails. It is easy to use and integrate. In PeerPrep's case, Nodemailer was used to support password reset emails to users.

Question Service

Technology	Rationale
MongoDB	On top of the aforementioned benefits, MongoDB's document-oriented approach is suitable for storing unstructured or semi-structured data like questions.

Matching Service

Technology	Rationale
MongoDB	On top of the aforementioned benefits, MongoDB's document-oriented approach is suitable for storing unstructured or semi-structured data like matching information.
RabbitMQ	RabbitMQ is a robust and popular message broker or message-oriented middleware used to facilitate communication between various components or services. In our application, RabbitMQ was used to facilitate matching between users.
Socket.IO	An event-driven library that enables real-time, bi-directional communication between web clients and servers. It is easy to use and integrate.

Collaboration Service

Technology	Rationale
MongoDB	On top of the aforementioned benefits, MongoDB's document-oriented approach is suitable for storing unstructured or semi-structured data like user collaboration pair information.
RabbitMQ	RabbitMQ is a robust and popular message broker or message-oriented middleware used to facilitate communication between various components or services.
Socket.IO	An event-driven library that enables real-time, bi-directional communication between web clients and servers. It is easy to use and integrate.
y-websocket	Underlying websocket provider used to support communication between editors of users in the same collaboration session.

Help Service

Technology	Rationale
Google PaLM	Google's PaLM (Pathways Language Model) is a large language model, used to generate hints for questions. It is easy to use and integrate.

Forum Service

Technology	Rationale
PostgreSQL	PostgreSQL excels at handling relational data, making it suitable for scenarios where there are many-to-many relationships, as in the case of posts and their comments.
Prisma	Prisma provides a type-safe API for interacting with the database, which ensures that data is handled consistently and accurately.

Chat Service

Technology	Rationale
Socket.IO	An event-driven library that enables real-time, bi-directional communication between web clients and servers. It is easy to use and integrate.

2. Architecture

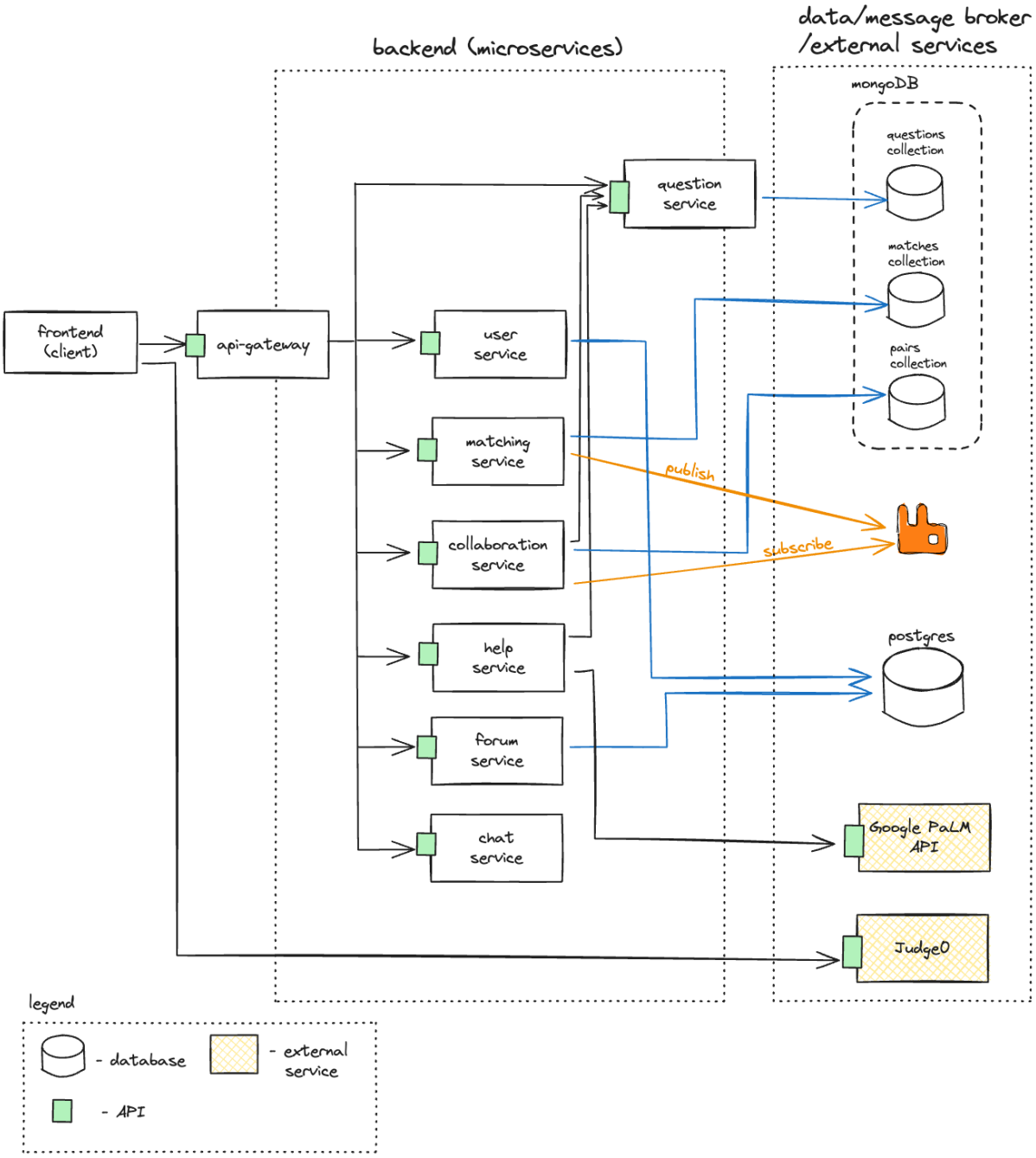


Fig. Microservice Architecture

Our team chose to use a microservice architecture for the following reasons.

Firstly, there is increased scalability. If there was a sudden increase in requests for questions, we could deploy multiple instances of the question service instead of the entire application, which would cause significant overhead.

Secondly, it is more fault tolerant. Consider the case the help service goes down, the rest of the services will still be functioning. The user would be able to complete the entire user flow with the exception of getting an AI generated hint for their question. It will also be simple for us to isolate the issue - we would know which microservice was responsible for that failing feature and hence which small part of our architecture was failing.

However, microservices come with increased complexity. One common issue we faced was coordinating the services. For example, with microservices, we would have to handle communication between them and decide between synchronous and asynchronous communication. With a monolithic backend, we would not have to do that all since the single big service would already have all the data it needed.

In the end, we chose to use microservice architecture because the benefits, such as scalability and fault tolerance, outweigh the increased complexity, which our team can handle.

As for more specific design decisions, details on the various microservices and their respective design choices can be found under Backend section and design choices of our deployment can be found at the Deployment section.

3. User Activity Diagram

The User Activity Diagram below models the workflow of a user from initial authentication to real-time collaboration and finally logging out from PeerPrep.

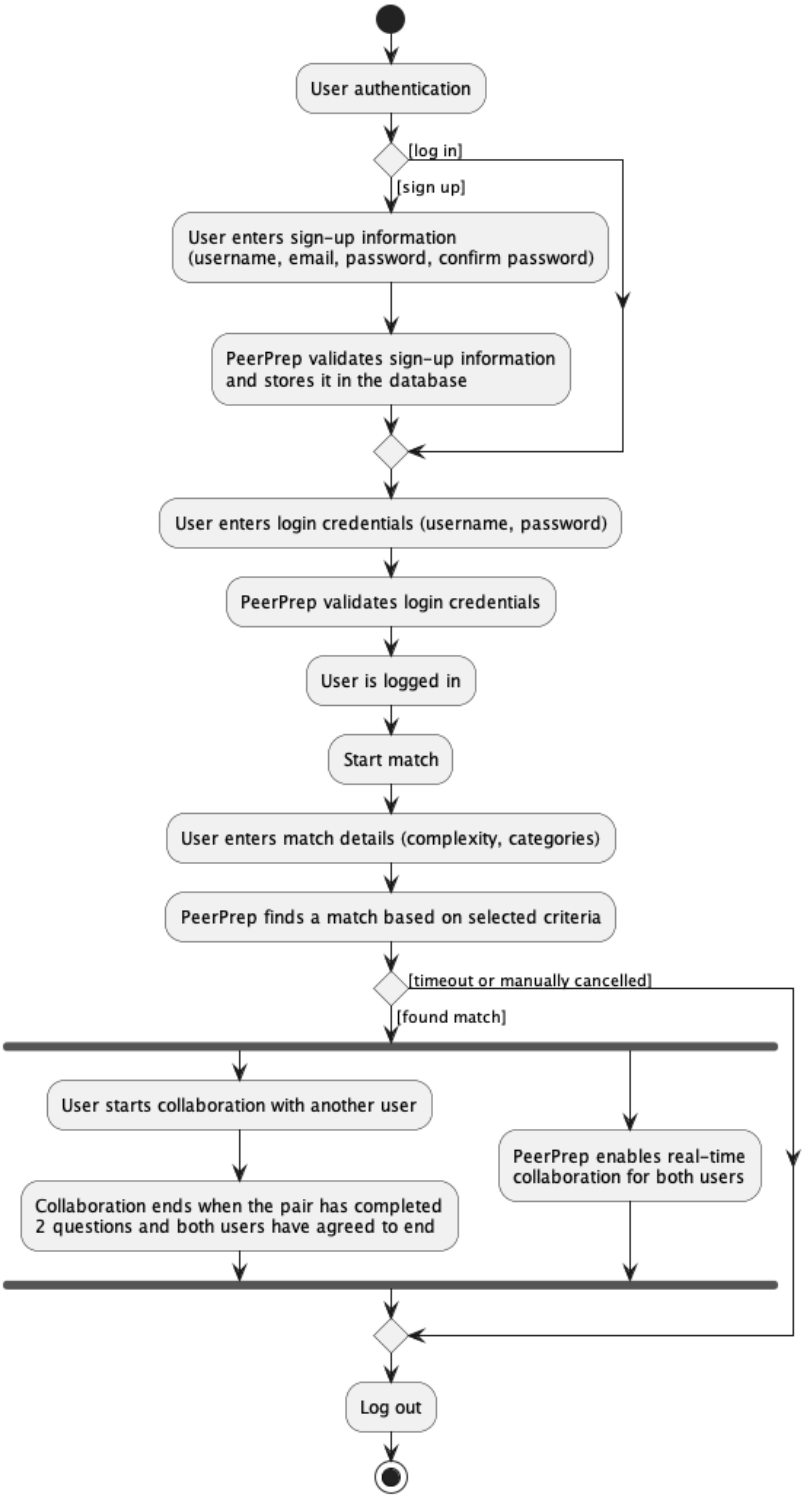


Fig 4. User Activity Diagram

Steps	Name	Description
1	**User Authentication**	The user is required to sign up, or log in if he or she already has an existing account with PeerPrep. If the user chooses to "Sign Up," he or she is prompted to provide essential information, including a username, email, password and password confirmation. Then, PeerPrep validates this information and stores it securely in the database. Thereafter, the user proceeds to log in. Alternatively, an user with an existing account may directly select the "Log In" option, where he or she enters the login credentials (username and password). Then, PeerPrep validates these credentials for authentication. Upon successful authentication, the user is logged in to PeerPrep.
2	**Start Match**	The user proceeds to start a match with another user. The user has the option to enter their match preferences, specifying criteria such as question complexity and categories. Then, PeerPrep employs these preferences to identify a suitable match based on the selected criteria.
2.1	**Found Match**	If a match is successfully found based on the criteria, the user is paired with another user for collaborative coding. They will solve 2 questions, after which the session can be ended with agreements from both users. At the same time, PeerPrep enables this real-time collaboration, ensuring both parties can interact seamlessly and instantaneously during this collaborative session.
2.2	**Timeout or Manually Cancelled**	If a match is not found within 30 seconds or if the user manually cancels the match, the matching is unsuccessful.
3	**Logout**	The user proceeds to log out of PeerPrep.

4. ER Diagram

The following ER diagram captures the relationships and dependencies between different entities in the system.

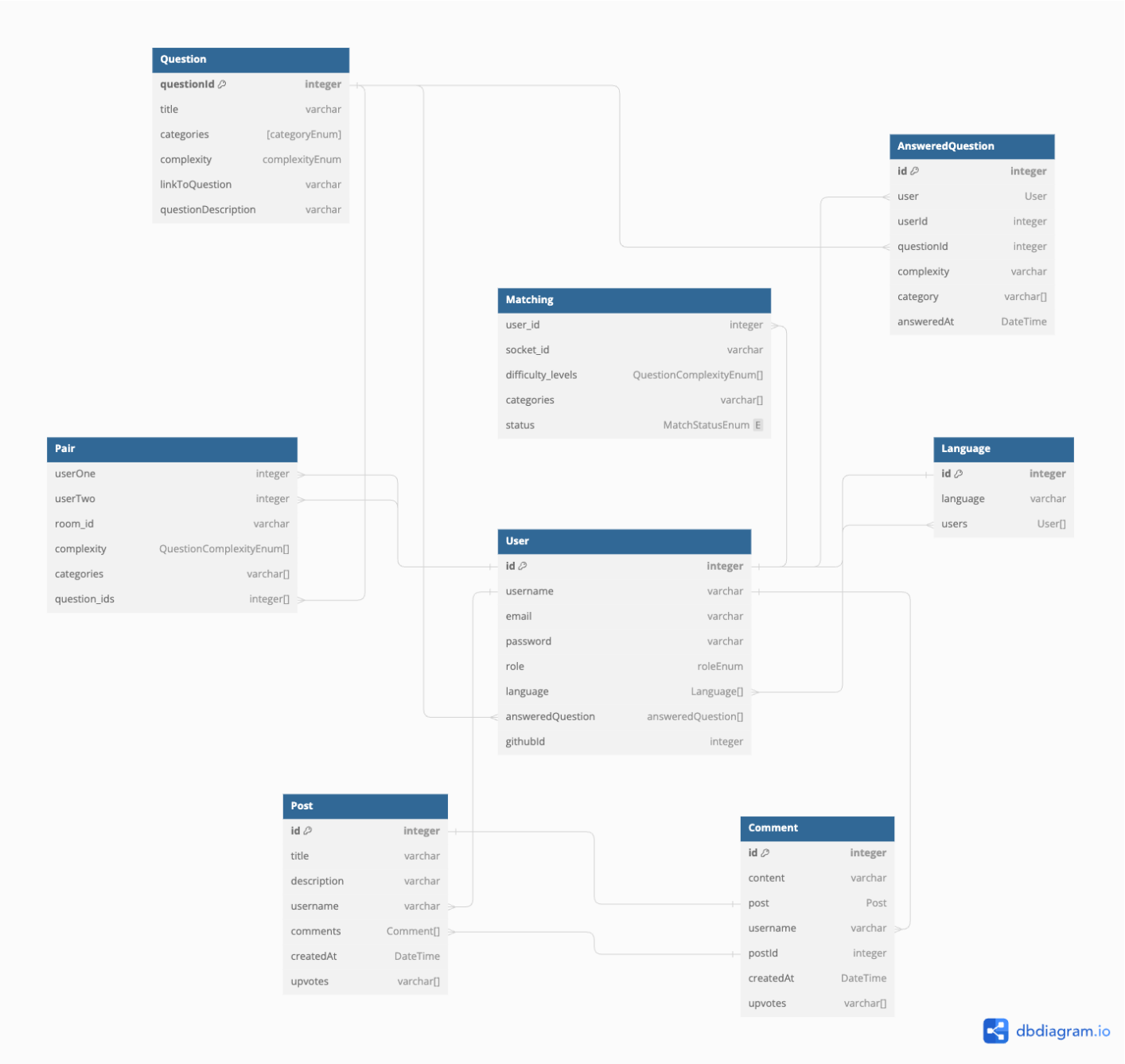


Fig. PeerPrep ER Diagram

5. Database Design

The following section outlines the different databases in our project and the rationale for choosing them.

a. MongoDB

Models

1. Question Bank

The questions are stored in MongoDB. This NoSQL database is well-suited for document-based storage, which aligns with the nature of the questions.

2. Matchings

Information related to user matchings, including user id, socket id, difficulty level, and categories, is also stored in MongoDB. This allows for efficient retrieval of matching information.

Redis was considered due to its strengths in fast data retrieval due to its in-memory storage and efficient caching mechanisms. However, one limitation of Redis is that it primarily stores data in memory, and it was considered that we may need to retrieve the past matching data in the future. Hence, we chose MongoDB, being a disk-based storage solution, so that the data is stored durably and guarantees data persistence.

3. Pairs

User pairs and related information, such as user IDs, room IDs, complexity, and categories, are stored in MongoDB. This facilitates the management of collaborative sessions between matched users.

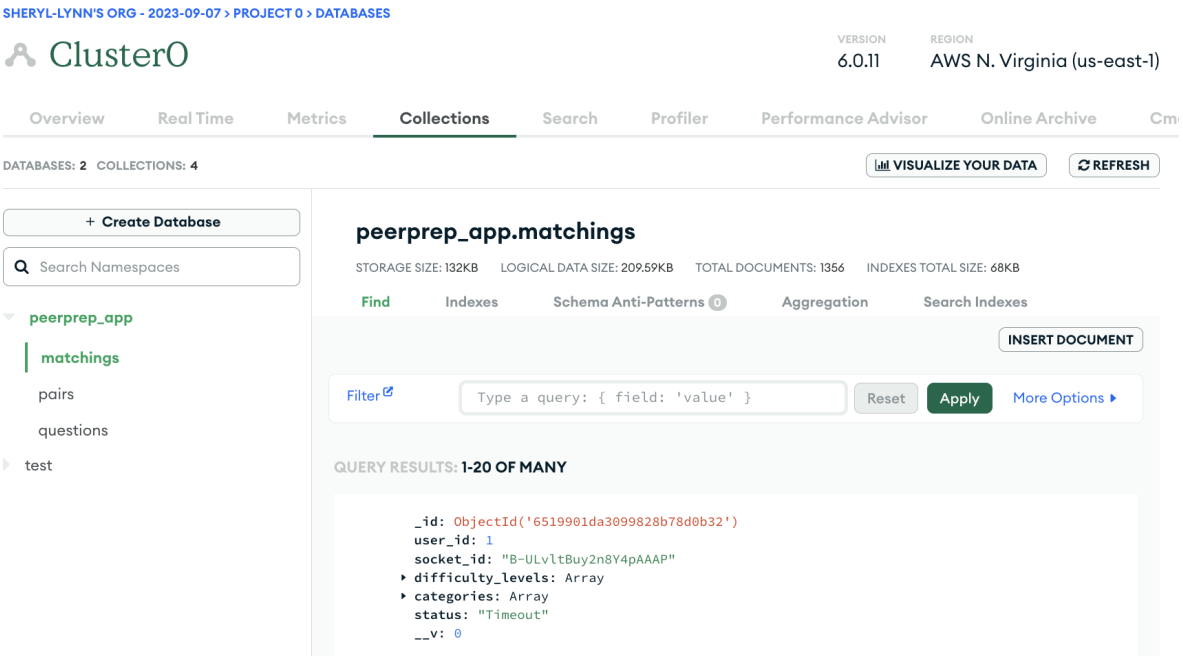


Fig 5. MongoDB hosted on MongoAtlas

Design Choice

We chose to use MongoDB for the following reason:

a. **Non-relational nature:** MongoDB's document-oriented approach is suitable for storing unstructured or semi-structured data like questions and matching information.
b. **Speed and scalability:** MongoDB's sharding capabilities can provide high performance and scalability, which is important for handling large volumes of data.
c. **Low number of relationships:** Since the data structures are relatively simple and don't require complex relationships, MongoDB's schema flexibility is advantageous.
d. **Persistence:** MongoDB offers durability through replication and failover mechanisms, ensuring that data is not lost even in the event of hardware failures.

b. Prisma with PostgreSQL

Models

1. **Users**

Stores essential user information such as user ID, username, email, password (hashed and salted for security), preferred language and their answered questions.

2. **Answered Questions**

Information related to user-saved questions. This includes details such as user ID, question ID, complexity, categories, and the timestamp of when the question was answered.

3. **Forum Posts and Comments**

Forum posts model captures posts created by users in a forum setting. Relevant information such as the post title, description, author and upvotes were stored for each post. To disallow users from upvoting multiple times, we stored the list of users who have upvoted each post. Similarly, this is to ensure a user can only downvote, when they have upvoted before. A similar table was implemented for forum comments.

Design Choice

We chose to use PostgreSQL and Prisma for the following reason:

a. **Relational data model:** PostgreSQL excels at handling relational data, making it suitable for scenarios where there are many-to-many relationships, as in the case of users and their answered questions.
b. **Strong typing and querying capabilities:** Prisma provides a type-safe API for interacting with the database, which ensures that data is handled consistently and accurately.
c. **Data Persistence and Integrity:** PostgreSQL's support for transactions and constraints helps maintain data integrity and ensures that user-saved question records are reliable and consistent.

Frontend

a) Overall

Our frontend is a React-Typescript application. The full tech-stack used for the backend can be found in the tech stack section of our report. To learn more about how the frontend communicates with our various microservices and external APIs, do refer to the Architecture section of this report.

From a high-level perspective, our frontend application is structured by grouping files by file type. In other words, files that share similar functionalities (eg: handling API requests, dealing with Redux, unit components, whole pages) are grouped together in a folder. Each subfolder is then typically divided into various module subfolders so that files belonging to the same module (eg: user module, question module, code module) are further grouped together in the same subfolder. This file structure gives us an easy and clean way to organise our files- which is especially important as the project grows.

```
├── public
├── src
│   ├── api
│   ├── components
│   ├── context
│   ├── pages
│   ├── reducers
│   ├── styles
│   ├── types
│   └── utils
├── .eslintrc.js
├── .prettierrc.js
├── Dockerfile
├── README.md
├── package-lock.json
├── package.json
└── tsconfig.json
```

src folder structure	
api/	Stores the API handlers for the various modules. These API handlers provide an abstraction for axios requests made to API endpoints for the specific module.
components/	The components subfolder stores various reusable components that can be shared between different pages.
context/	The context subfolder stores React contexts for various modules.
pages/	The pages store the actual rendered pages of the application for each module.
reducers/	Stores redux related files.
styles/	Stores configuration files for the application's theming.
types/	Stores the typescript types used in the project for each module.
utils/	Stores utility functions and code for the various modules.

To enforce a consistent coding style throughout our application, we also made use of linters to help format our code. This helped us write more readable, maintainable and consistent code which was important especially because we were working together to develop the application on a short timeline. In addition, here are some good practices that we kept in mind while developing the frontend and looked out during code reviews:

Code Quality		
Developers working on the application should try to aim to write good quality code. Before we can do that, we had to lay out several benchmarks and guidelines that we all agreed on and would keep in mind while developing the application. These guidelines serve to help us write more readable, consistent and maintainable code. This is important in helping us reduce developer errors, code and collaborate more efficiently, and scale more effectively and efficiently.		
1.	**Small components**	In general, we tried to make sure that no component is too big or long. Generally, each file should also only contain one component, and every component should only have one functionality (in accordance to the Single Responsibility Principle). This helps us keep the code we write more readable, understandable, and so, maintainable.
2.	**Naming Conventions**	Our team uses Pascal case for component files, and camel case for local variables and function names.
3.	**DRY Code**	We also try to keep our code DRY (Don't Repeat Yourself). Common components and functionalities should instead be abstracted out into separate files so that they can be reused. This helps us code more efficiently and preserve consistency across our application.
UI/ UX		
While this is not a UI/ UX centred course, an important aspect of software development is still user design. Writing good code is simply not enough. We strive to develop usable applications (in accordance to our non-functional requirement NF3). With this in mind, we tried to use well researched and established guidelines like the 10 Nielson Heuristics. Some heuristics (Flexibility and efficiency of use and help and documentation) were omitted from our guidelines below as they were deemed less important in light of the tight constraints we were working with.		
4.	**Visibility of system status**	Keep users informed about what is going on in the system through appropriate feedback. For instance, use spinners or loading components when loading data and use toast messages upon error or success events.
5.	**Match between system and real world**	Speak the user's language, use terms common and widely understood by users. In addition, icons used should also mirror what they represent in the real-world closely.

6.	**User control and freedom**	Let users feel in control of the system. For instance, when they perform an action by mistake (especially for dangerous actions), provide an ‘easy way out’ (eg: ‘Cancel’ button).
7.	**Consistency and standards**	Follow industry norms and conventions when using words. Actions, components should be consistent with other applications familiar to the user to improve the learnability of the application and reduce the user’s cognitive load.
8.	**Error prevention**	Try to prevent errors even before they happen. For instance, use confirmation dialogs when performing dangerous actions (like deleting a user account).
9.	**Recognition rather than recall**	Reduce cognitive load by minimising the amount of information required for the user to remember where possible.
11.	**Aesthetic and minimalist design**	Avoid showing irrelevant information or overloading the user with irrelevant information.
12.	**Help users recognise, diagnose, and recover from errors**	Use good error messages to allow users to recover from mistakes. They should be descriptive, attention catching, and ideally provide a solution to solving the problem.

At the core of PeerPrep is our collaboration sessions. To see the high-level user activity diagram for the main use case of our application (a user logging in to find a collaboration match), please refer to the above section User Activity Diagram.

Besides this collaboration match, users can also create “Practice Rooms” which are collaboration sessions without any matching done (users can simply share the link to invite their friends to join the collaboration session).

During these collaboration sessions, users can work on the same question, share a synced code editor, communicate via our chat feature, and even collaborate via a shared interactive whiteboard. We use websockets to support this functionality.

b) Web Sockets

Websockets are a bidirectional communication protocol that allows data to be exchanged both from the client to the server, and from the server to the client. The connection is kept alive until terminated by either the client or the server.

As mentioned above, websockets power much of the features in the collaboration rooms, such as synced code editors, real-time chatting, and the real-time interactive whiteboard. These features require a constant stream of information to be exchanged between the client and the server (both ways). The client would have to transmit updates when typing on the code editor, or transmitting a new text message. The server would have to listen to these updates and messages and broadcast messages to other client connections so that they can react appropriately to these changes and reflect them to their user, giving the experience of real-time collaboration. For details on how the backend handles websockets refer here.

From a high-level perspective, two websockets are created each time for each collaboration room (one yjs-websocket and one socket.io socket).

- The yjs-websocket is used to sync the code editors. This websocket is binded to the code editor and listens for updates and changes when the user types or moves about in the code editor. The yjs-monaco library takes care of this for us- and by using this library, we can achieve user 'awareness' (which tracks user positions and status). Jump to the section below for more details on how the code editor was implemented.
- All other communication (eg: user joining/ leaving the room, sending chat messages, or syncing the interactive whiteboard) go through the other socket.io socket.

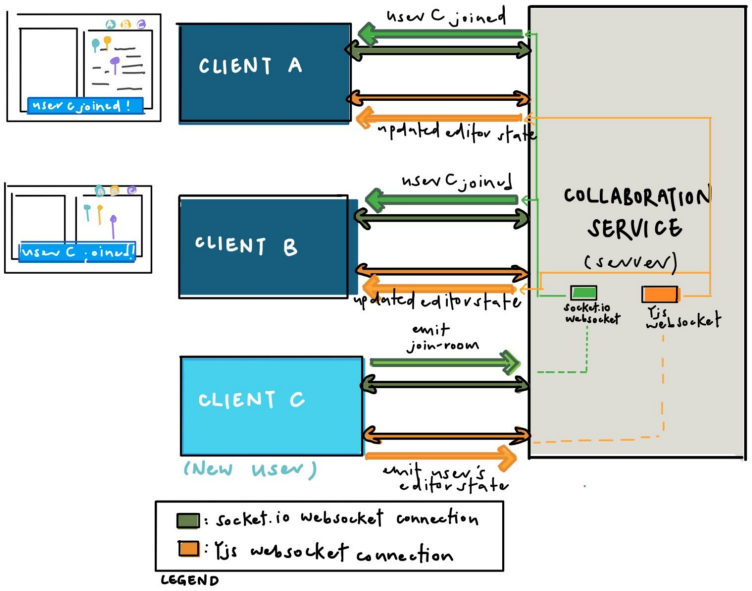


Fig 6. Sample websocket data exchanges on new user connection

Rationale for using websockets

Websockets are highly ideal for the use cases of our collaboration rooms. We quickly realised that traditional HTTP protocols would not suffice for the use cases of our collaboration room for the following reasons.

Reason	Elaboration
HTTP protocols are unidirectional	This means that traditionally, communication goes only ‘one way’. Clients make HTTP requests to servers and servers will then send a response to these requests received. However, data is continuously exchanged from the client to server, and the server to client in our collaboration room. Clients not only have to send messages to servers upon updates to the state of the collaboration room (eg: user changes editor language, user changes question, user types something in the editor); they also have to listen to updates from other clients. If websockets were not used, the client would have to continuously poll the server to check for new updates at very frequent intervals. This results in significant overheads and is clearly not ideal.
Traditional HTTP requests do not reuse the connection	After receiving a response from the server to a request sent by the client, the TCP connection between the server and client is closed. With the large volumes of data being exchanged between the client and server, new connections would have to be established for every new request which adds significant overheads.

On the contrary, websockets help resolve these issues. In addition, websockets are highly scalable, allowing for a large number of concurrent connections. This is useful in the case of our real-time features where support for multiple concurrently connected users is needed.

Reason	Elaboration
Websockets allow for bi-directional communication	Websockets allow client and servers to communicate with each other (bi-directionally), meaning that the client no longer has to poll the server to receive updates.
Websocket connections are kept alive until terminated by the client/ server	This means that the connections can be reused. After the initial connection is established, the client/ server does not need to establish a new connection again when sending a new message (so long as the connection is not terminated).

Choosing Socket.IO

There are quite a number of websocket libraries out there, however, the two most popular libraries for Node.js appear to be Websocket (ws) and Socket.IO. Socket.IO is built on top of Websocket. We decided to go with Socket.IO to handle the websocket exchanges between clients in the same collaboration room (excluding code editor data) for the following reasons.

- **Ease of use:** Socket.IO provides a more simple and intuitive API for emitting and listening to websocket events, which is important for us developers as we were pretty unfamiliar with using websockets and needed something that could be picked up quickly and easily to fit our development timeline.
- **Room-based communication:** Socket.IO provides the concepts of 'rooms' which are arbitrary channels that sockets can join and leave. This is particularly useful as we only want the server to broadcast events to a specific subset of connected clients at one time (to clients in the same collaborative session), rather than broadcasting events to all connected clients.
- **Automatic reconnection:** Socket.IO also helps take care of reconnections in the case of network disruptions by helping the client to attempt to reconnect to the server when it disconnects unexpectedly.

Details on the choice of websocket for the code editor can be found in the section below.

c) Code Editor

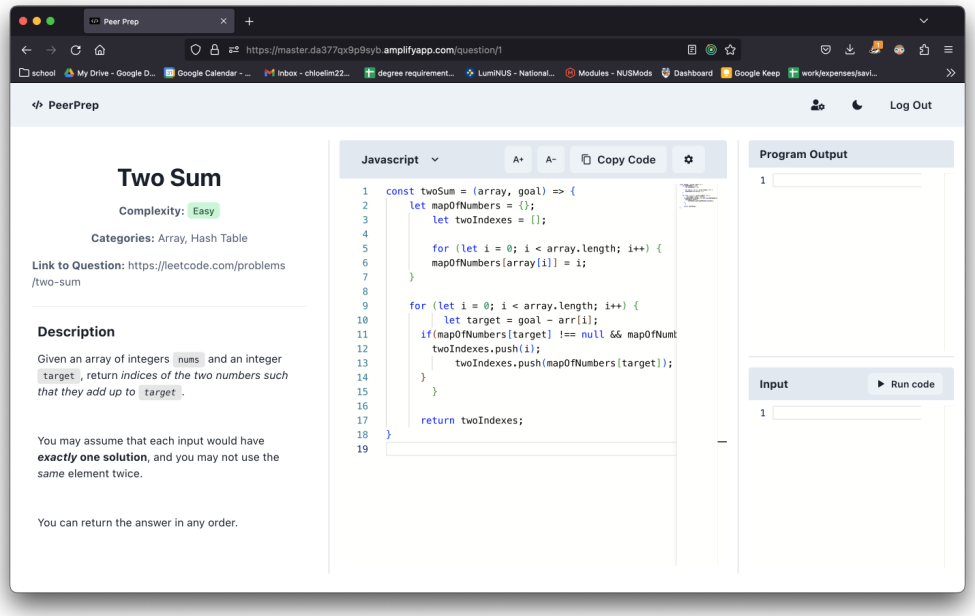


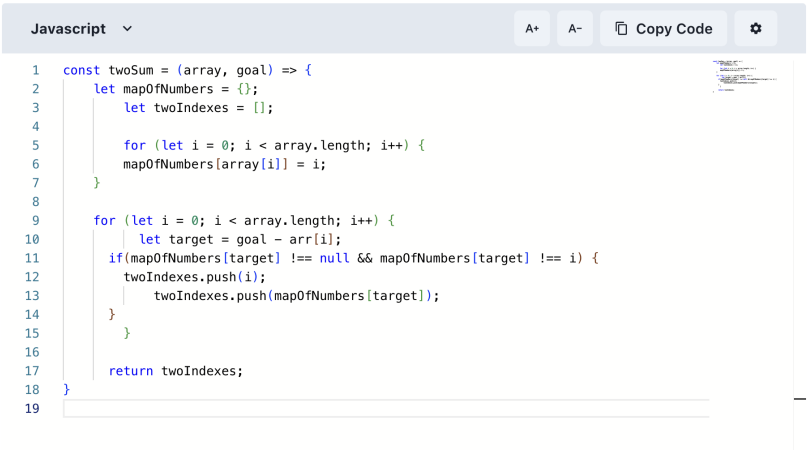
Fig. PeerPrep's Code Editor

As a technical interview preparation site, a good coding interface is essential to helping us craft an immersive and enriching experience for users; which is why PeerPrep's code editor comes with some powerful features.

Core Features

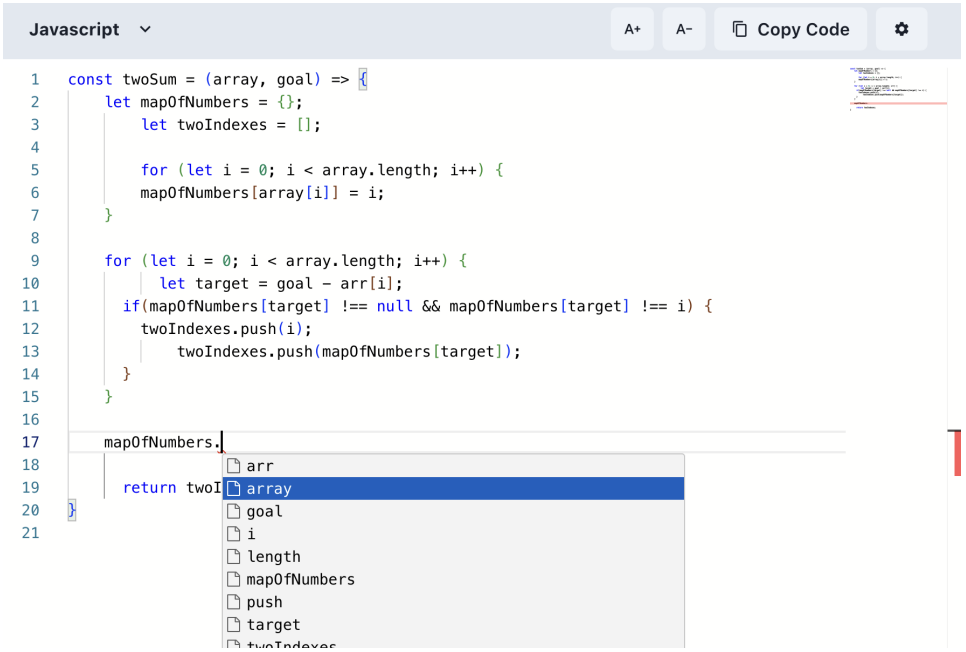
1. **Syntax highlighting, Automatic Indentation and Code Folding**

   PeerPrep's code editor supports syntax highlighting, and provides auto-indentation just like many of the loved code editors. Written code snippets are highlighted with colours to make the code written by users more readable, facilitating writing so that our users can focus on the more important things (the logic and concepts behind the technical questions).



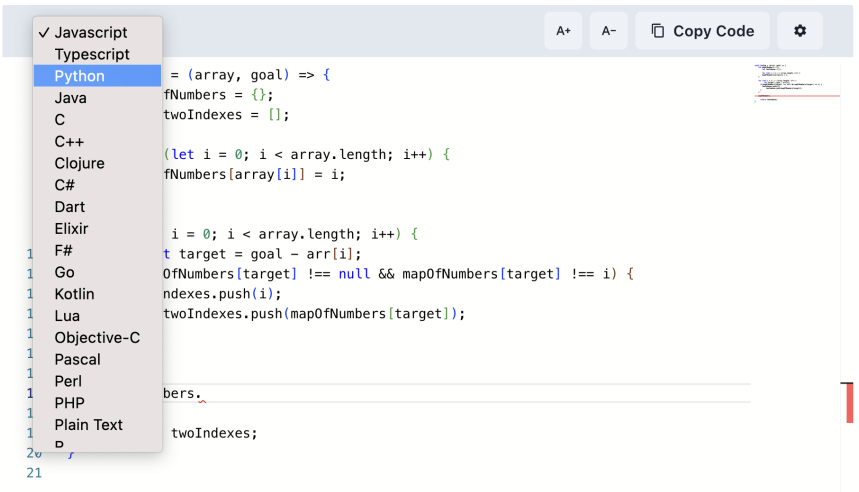
2. **IntelliSense Code Completion**

   For selected languages, the editor also provides IntelliSense code completion which provides users code completion, parameter info, quick info and member lists. This helps users work quicker, and is a much used and loved feature in many of the code editors on the market.



3. **Multi-language Support**

   PeerPrep's editor also comes with support for multiple languages to help support all our user's journeys in preparing for their interviews. Common languages used in technical interviews are supported by PeerPrep's code editor. Users can simply change the language of the code editor to their preferred language and type away.



4. **Real-time editing (in collaboration rooms)**

   In the spirit of peer-collaboration, which is at the core of PeerPrep and our collaboration services, our code editor also supports real time editing in collaboration rooms (ie: during a matching/ practice session). Any code written by a user will be also reflected in the code editors of other users connected to the same collaboration session so that users can code and learn together. User ‘awareness’ (ie: the state/ position of users in the code editor) are also shown so that user’s know who’s typing what. Code language will also be synced across editors in the same collaboration session.

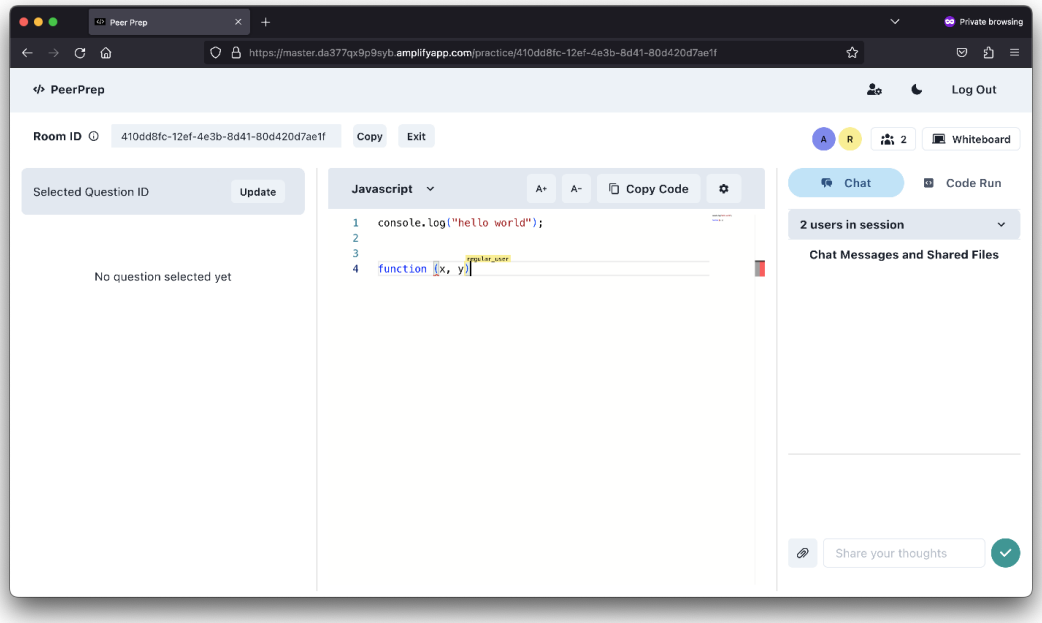


Fig 7. User ‘Awareness’ in action (Real-time code editor)

5. **Editor Controls**

   Users are also able to adjust editor configurations to their personal preferences. For instance, they can adjust the font size of the editor to their liking, and set the editor’s theme to their personal preferences.

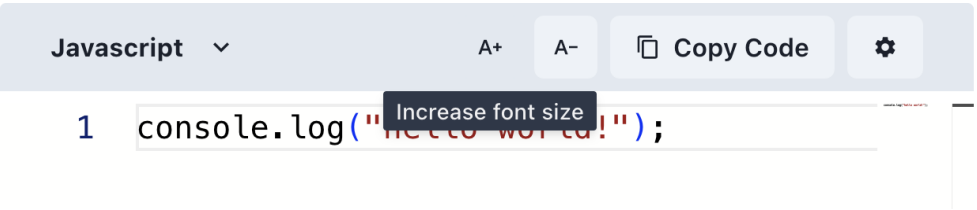


Fig 8. Increasing code editor font size

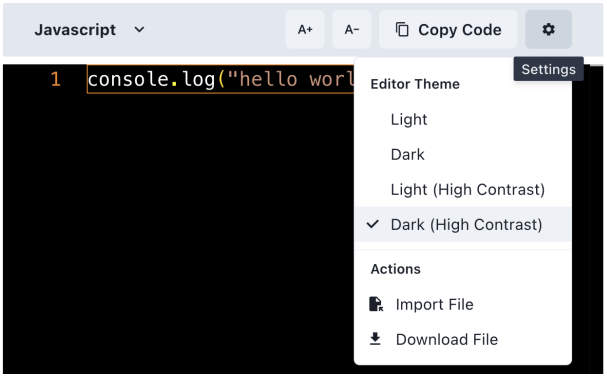


Fig 9. Changing code editor theme

A minor note on changing the code editor theme, toggling between dark and light mode will automatically change the user's code editor theme appearance (to match the dark/ light mode theme of the application). Users can however still choose to customise their code editor's theme to their personal preference.

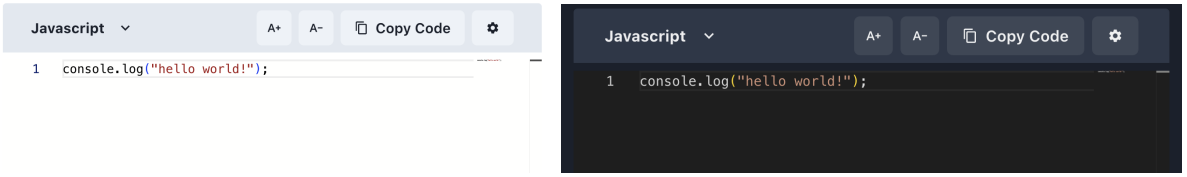


Fig 10. Default Code Editor Themes for Light/ Dark Mode

6. Saving/ Importing Files

Users can also save code written on their current code editor, or import some file from their local computer into the code editor. This helps users who might want to store the progress or programs they've written on PeerPrep to their local devices, or those who want to restore progress on an existing program they've previously written. PeerPrep's code editor will also automatically save the file with the format `<question-title>.<editor-language-file-format>`.

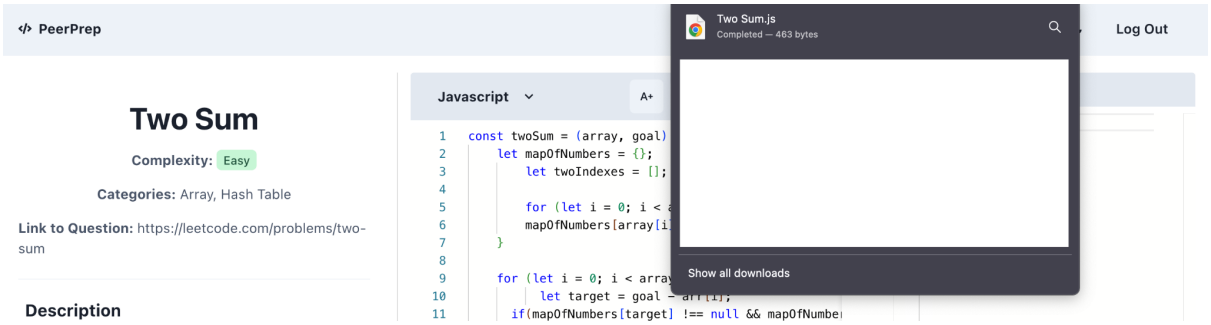


Fig 11. Downloading JavaScript code written for "Two Sum"

Implementation

The code editor uses the react-monaco library, which is a wrapper that allows for seamless integration of the monaco-editor with our React application. Monaco Editor is the same code editor that powers VS Code. We decided to pick Monaco Editor over other alternatives because of its rich features, such as syntax highlighting, auto-completion, code folding, error checking, and formatting. To support real-time editing, yjs-monaco is used to bind the editor to a Yjs document.

Here is a quick diagrammatic overview of the editor, the editor component manages the monaco-editor's states, such as the current font size, language and theme. The component also listens for updates on the editor language, and will then emit changes using the Socket.IO websocket so that editor language is synced between editors in the same collaboration session. The Yjs websocket is used to emit and listen to changes on the real-time code editor.

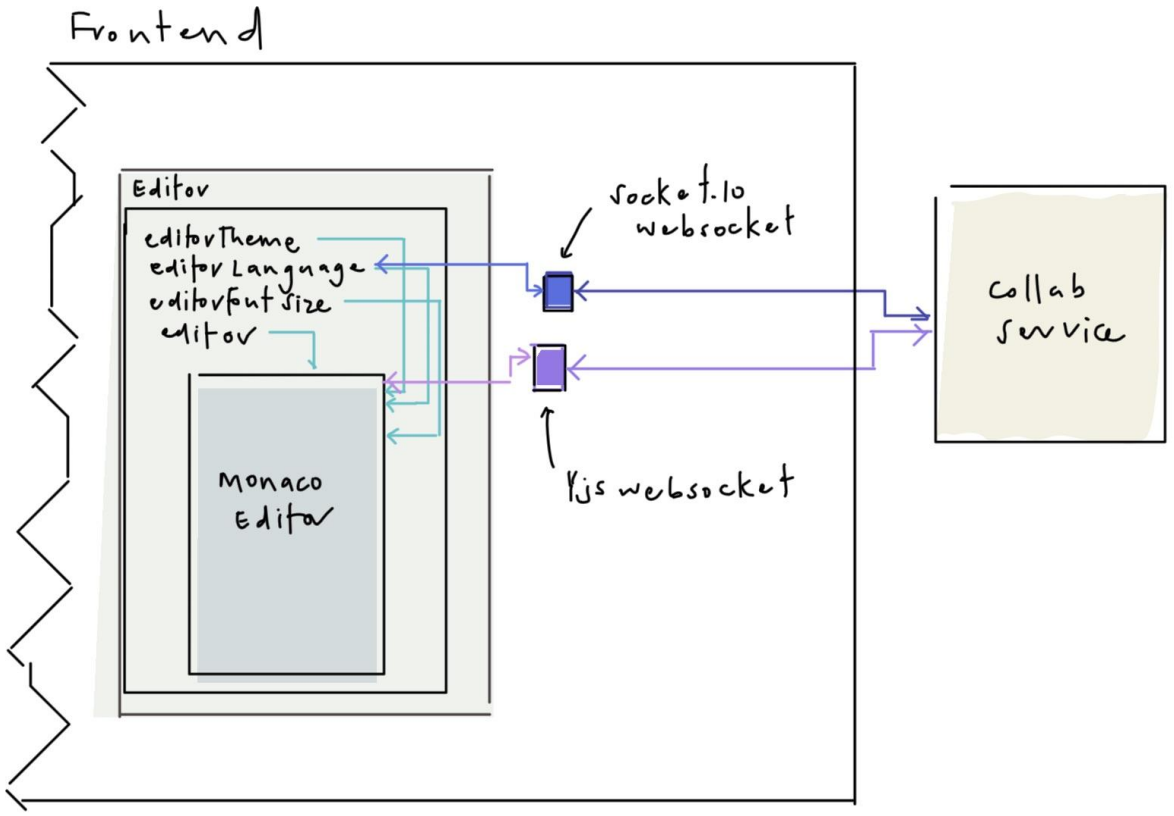


Fig 12. How the Code Editor Works

d) Code Execution

Besides being able to write code, the ability to run code is equally useful. Users would want to know that their solutions can run, and work correctly which is why we integrated a code executor into PeerPrep. Users can simply write some code into the code executor and optionally provide some inputs to their program in the input box before pressing run code to run their program. PeerPrep will then log the results of the program output, as well as the program's resource usage (execution time and memory used) in the program output box.

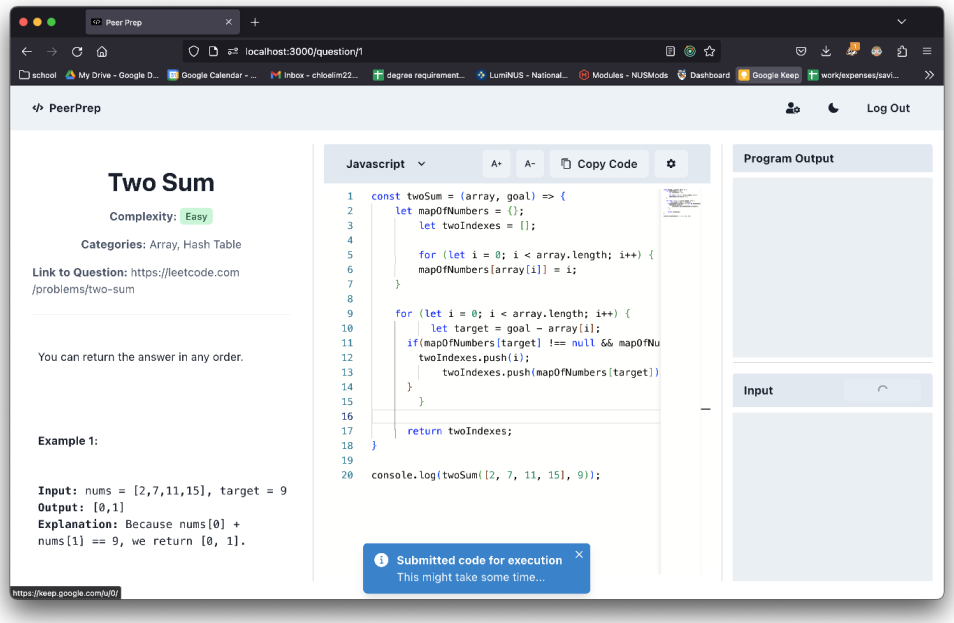


Fig 13. Running a program using PeerPrep's Code Executor

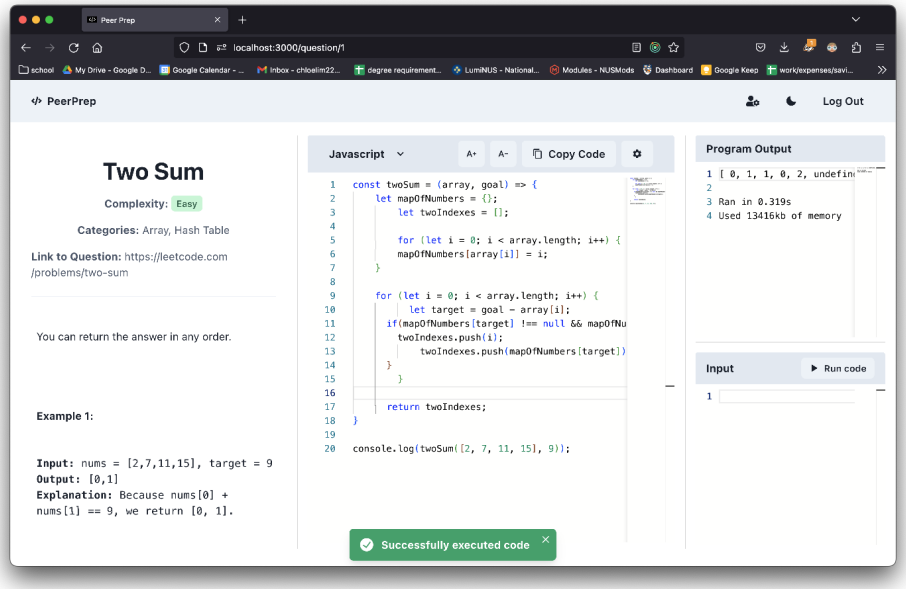


Fig 14. PeerPrep's Code Executor Returning Program Output

Implementation

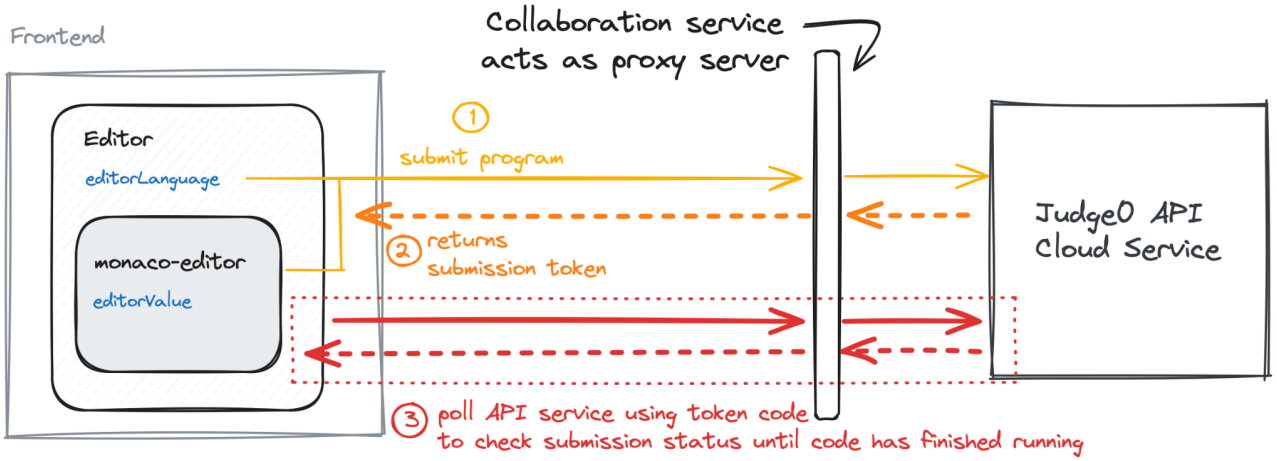
PeerPrep's code executor uses Judge0's code execution API services to execute user programs. Judge0 is an open-source code executor that supports execution of a vast selection of languages.

While there were other alternatives in consideration for the code executor library choice, such as the code-executor library and onecompiler API, we decided to settle with Judge0 due to its vast language support, easy usage and integration, advanced compiler options, detailed execution results, and good API documentation. In addition, Judge0 also offers a free-tiered cloud API service that allows us to simply execute code using API calls to Judge0's shared cloud service without having to self-host the Judge0 service.

The diagram below gives an (simplified*) overview of how the code executor is implemented.

1. Essentially, the frontend will first make an API call to submit the program (along with the program language and inputs) to the Judge0 API service.
2. The API will then return a submission token (used to identify the submission and the status of the submission).
3. The frontend will then poll the Judge0 API service using this submission token to check on the status of the submission. Once the program has completed execution, the frontend will then return and display the program output to the user.

*Note: The collaboration service actually works as a proxy server to handle the API calls made by the frontend to Judge0's API service. This ensures that our API secret token is not exposed in the frontend.



Fig, Code Execution Implementation Flow

e) Collaborative Whiteboard

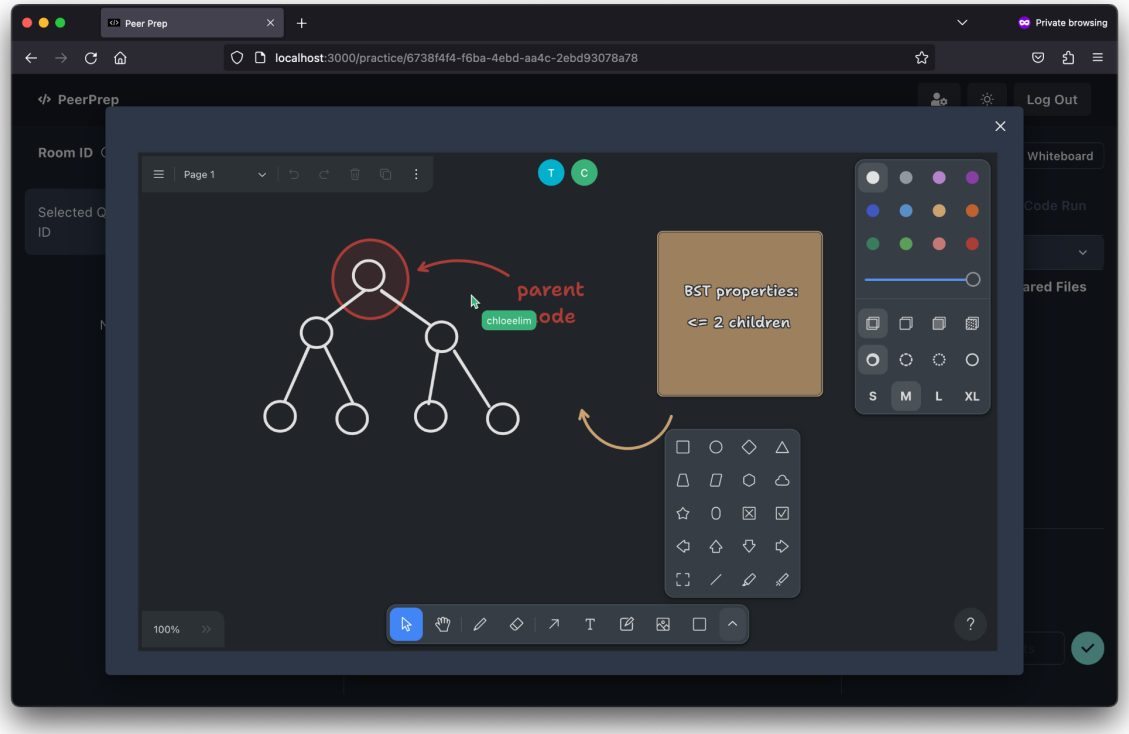


Fig . PeerPrep's Interactive Collaborative Whiteboard

Lastly, to complete our suite of features in the frontend that provide users with a more immersive and comprehensive coding and collaboration experience, we also introduced a real-time interactive whiteboard into PeerPrep.

Often, words are not sufficient to effectively get our ideas across- sketching diagrams are particularly useful in helping us work through solutions and communicate key concepts. This aspect of whiteboard-style interviews on online technical interview preparation platforms are often overlooked- but the team felt that sketching and diagramming would be a valuable asset in helping users better communicate ideas across and working through approaches to take for their questions.

Features

PeerPrep's whiteboard provides users a sleek interface, accompanied by a vast array of diagramming tools and features. Besides basic features like inserting shapes, and text, users can also insert embedded content such as Youtube videos, Github Gists, and more to help our users collaborate more effectively and robustly. Users can also export their whiteboards as PNGs, amongst other features.

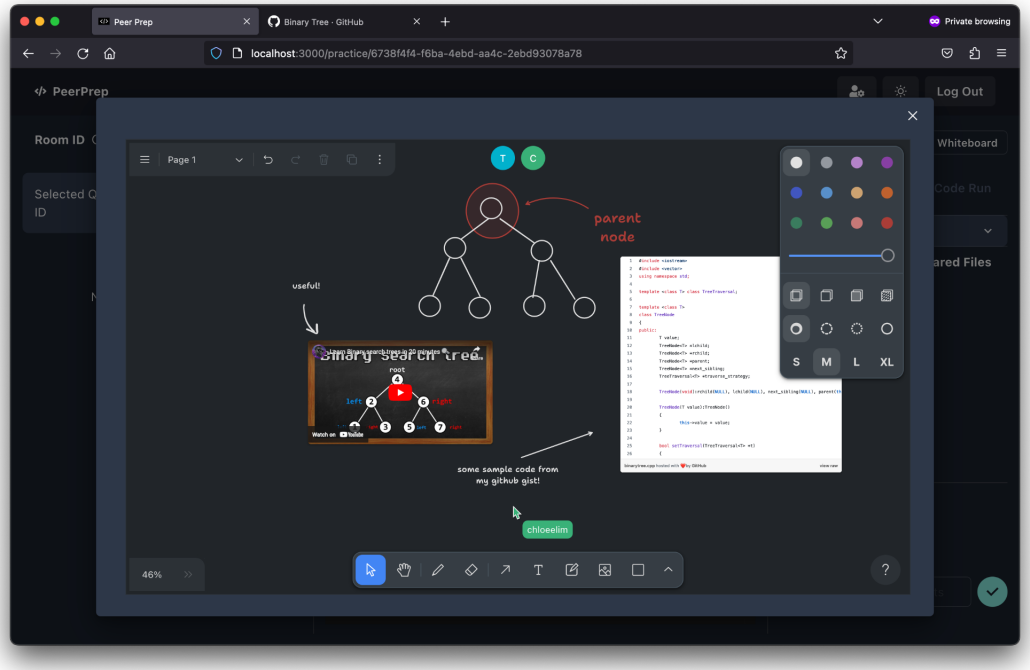


Fig . Adding Embedded Content into the Interactive Whiteboard

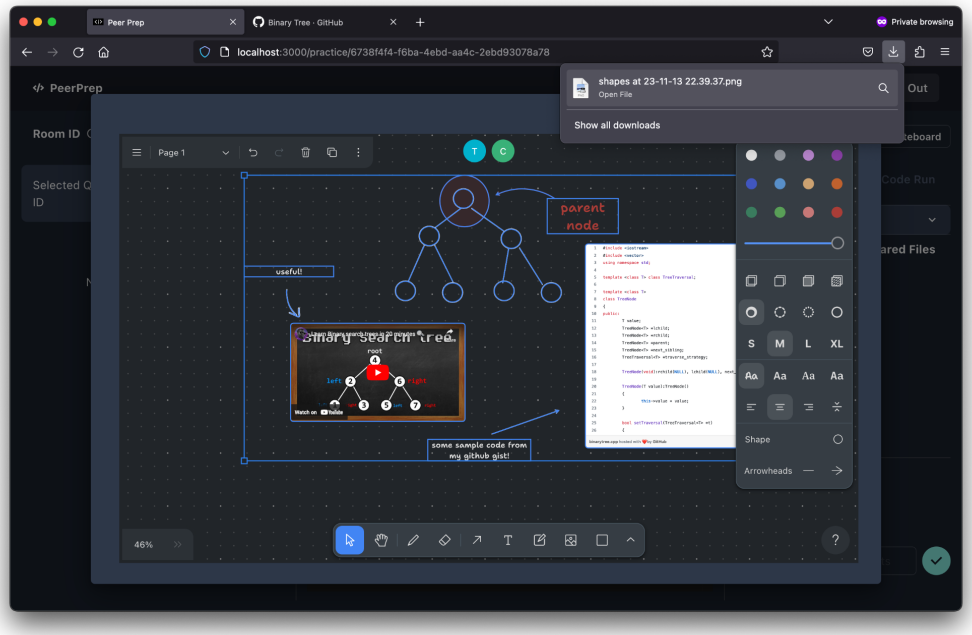


Fig 17. Exporting Whiteboard as PNG

Users can even click on the user icons to ‘follow’ another user around the whiteboard. This is useful in scenarios where one user might be doing the explaining and the other user(s) are simply the observers. By following a user, watches can automatically track a user as it moves across the canvas, making collaboration easy.

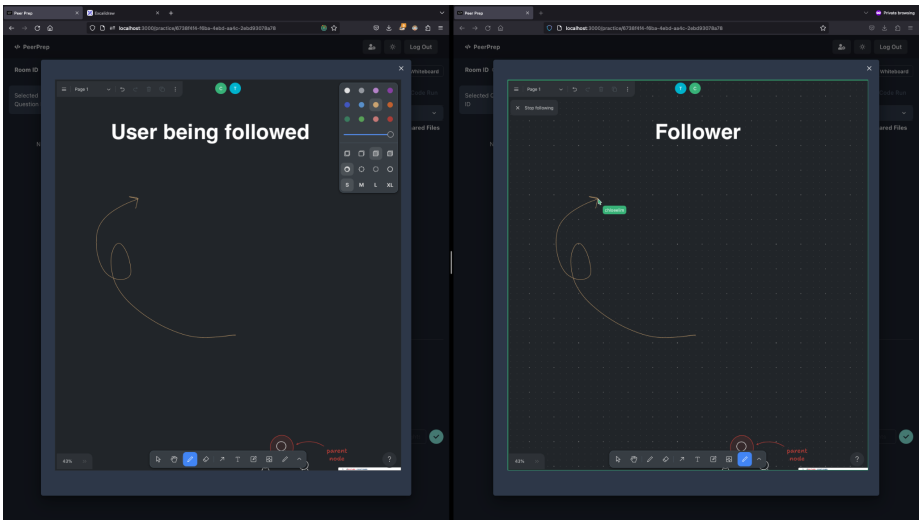


Fig . Demonstration of ‘Follow User’ Feature

Implementation

PeerPrep’s interactive whiteboard uses the powerful library tldraw, which is used to create interactive whiteboards in React applications. We chose the library for its rich set of powerful features and ease of integration and use. It is also open-source and free. To make the whiteboard ‘multiplayer’ (in other words, enable real-time interaction and syncing between whiteboards in the same collaboration sessions), we used Yjs and its y-websocket and binded it to the whiteboard so that it can emit and listen to changes in the shared whiteboard.

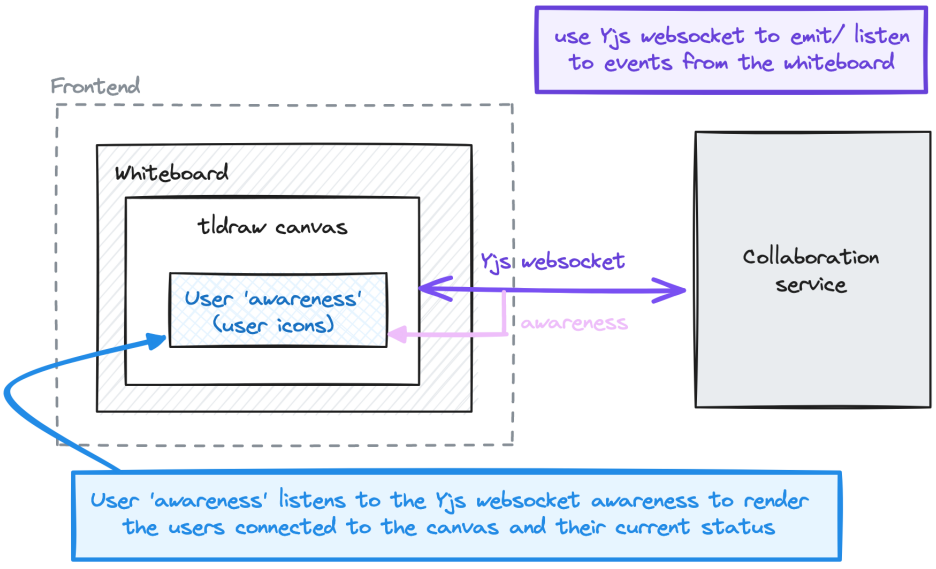


Fig . Overview of Whiteboard Implementation

6. Backend

a) REST API

The REST (Representational State Transfer) API in the PeerPrep project serves as the backbone for managing various components, including user operations and a question repository in the Must-Haves and a community forum in the Nice-to-Haves. It follows the REST architectural style to structure data and operations.

Key RESTful Principles

1. Statelessness: In a RESTful design, each request from a client to the server must contain all the information needed to understand and fulfil the request. This principle aligns with the way the PeerPrep REST API handles user and system state.

2. Resources and Uniform Resource Identifiers (URIs): In REST, resources are identified by URIs. The PeerPrep REST API uses URIs to represent the various components and functionalities. For example, user profiles, questions and forum posts endpoints have unique URIs.

3. CRUD Operations: The PeerPrep REST API employs standard CRUD (Create, Read, Update, Delete) operations to manage resources. This simplifies interactions and allows users and administrators to perform common actions consistently.

Detailed REST API Functionalities

1. User Management:

- Registration: Users can create accounts by providing their details (username, email and password). This information is sent to the server via POST requests to create a new user resource.
- Authentication: After registration, users can log in by sending their credentials (username and password) in a POST request to the authentication endpoint. The server validates the credentials and responds with an authentication token for secure access.
- Profile Management: Registered users can view and update their profiles by sending GET and PUT requests with updated personal information or programming languages.

2. Question Repository:
    - Question CRUD: Administrators can create, read, update and delete questions in the repository using POST, GET, PATCH and DELETE requests respectively. These requests are made to the corresponding question URIs.
    - Question Retrieval: Users can view a diverse range of interview questions by sending GET requests to the question repository URIs.
3. Forum:
    - Post CRUD: Users can create, read, update and delete posts in the community forum using POST, GET, PUT, PATCH and DELETE requests to the respective post URIs.
    - Comment CRUD: Users can also interact with posts by creating, reading, updating and deleting comments using appropriate requests sent to comment URIs.

The RESTful design in the PeerPrep project ensures a consistent and structured approach to managing data and interactions across various components. It facilitates resource management, authentication and data retrieval in a manner that is both scalable and user-friendly. By adhering to REST principles, PeerPrep effectively leverages the power of standard HTTP methods to deliver a seamless and secure user experience.

b) MVC

The MVC (model-view-controller) design pattern was adopted for PeerPrep. The MVC model is a design pattern that separates an application into three main components (the Model, the View, and the Controller). The separation of these components abides by the “separation of concerns” as every component is responsible for a separate concern. This separation helps to make the code we write more maintainable and scalable as it reduces the coupling and increases the cohesion between these concerns.

In the context of PeerPrep, each microservice typically implements a Model which manipulates, validates, and processes data. For microservices using MongoDB, we used mongoose to help model our application data. Microservices using a PostgreSQL database use Prisma as an interface to interact and manipulate our application data. Together, these technologies form the “Model” component of our application.

Each backend microservice also implements a Controller. These controllers act as an intermediary between the Model and the View, bridging them together. They respond to incoming HTTP requests made by the View (our frontend) and use the interface provided by the Model to respond to this request. Essentially, the controller acts as the ‘brains’ and performs the business logic.

Lastly, our frontend implements the View component of the MVC architecture. They provide an interface for users to interact with our application.

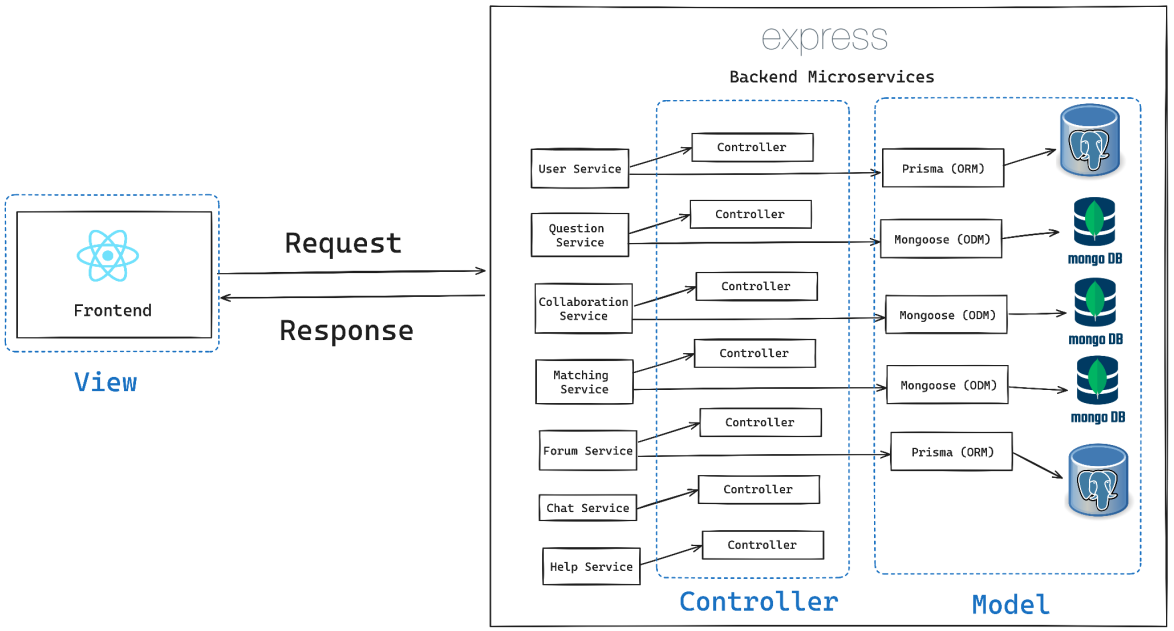


Fig. PeerPrep MVC

c) Publisher-Subscriber pattern

For our Socket.io services, especially the collaboration and chat services, we utilise a publisher-subscriber pattern when enabling the clients and servers to communicate. The publisher-subscriber pattern is said to be similar to a bulletin board, where subscribers are responsible for checking out the board. We follow this pattern when we use socket.io - we subscribe to a "room" in socket.io, and listen to the events published to it, as shown is this figure below.

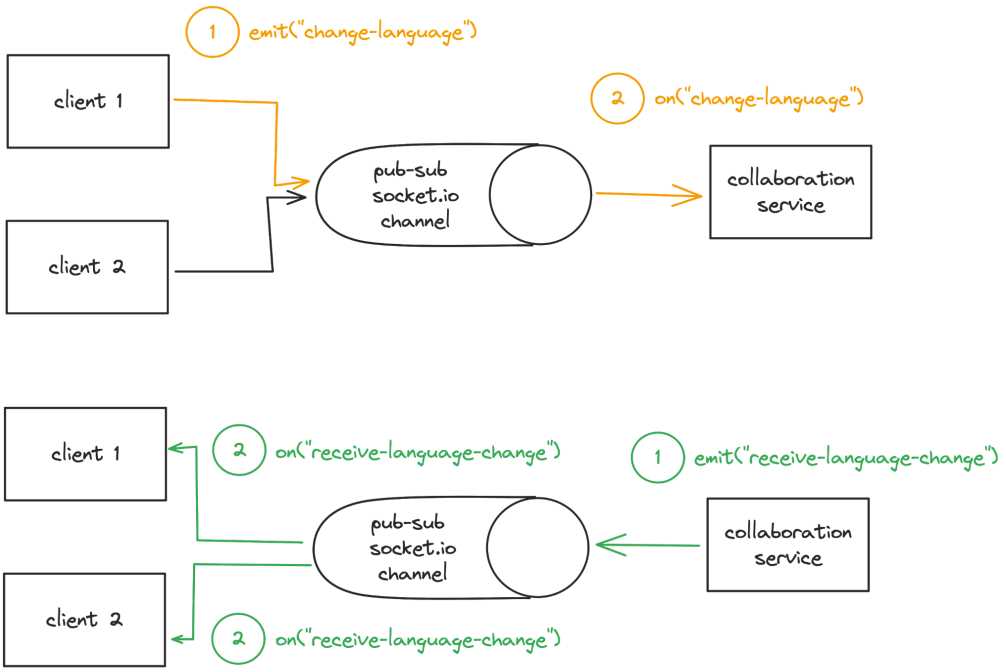


Fig. Example socket.io flow in the collaboration-service

The benefits of using pub-sub via socket.io is as follows:

- Decoupling of communication logic and business logic
    - Our clients just need to define the business logic (their response to socket.io events). As for reliable communication, the message broker (socket.io) is responsible for reliably forwarding published messages to subscribers, freeing us from writing additional code.

d) API Gateway

In between our backend microservices and frontend client, we have an NGINX reverse proxy serving as an API gateway. The main benefits of the gateway are reduced complexity and decoupling. The API Gateway shields client code from the intricacies of the microservices

architecture. Clients don't need to be aware of the specific endpoints, protocols, or data formats of each microservice. The gateway handles these details internally.

The "uniform interface" was particularly useful for us to resolve a deployment issue, where if we deployed it all on separate microservices, we run into problems in passing the JWT cookie and fail authentication checks. This API gateway helped us ensure our microservices are accessible via the same domain.

This is an example of the Facade pattern, where the frontend doesn't need to understand the intricacies of our backend system since we provide a unified interface for the client to speak with.

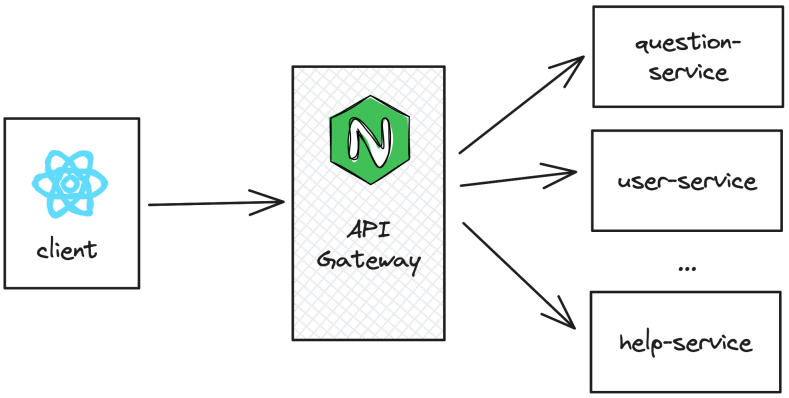


Fig. API Gateway

e) Question Service

API Design

The question service is the microservice that is responsible for providing and maintaining the set of questions and their data in PeerPrep. Users must be authenticated with the user service before accessing the question service.

Here are the API endpoints provided by the question service:

<table>
<tr><th>Route</th><th>Method</th><th>Description</th><th>Permissions</th></tr>
<tr><td rowspan="3">/</td><td rowspan="3">GET</td><td>Gets the set of questions in PeerPrep that are optionally filtered by category or complexity.</td><td rowspan="3">Any user</td></tr>
<tr><td>Optional query parameters:
● “categories”: The categories of questions to filter by.
● “complexities”: The complexities of questions to filter by.
● “limit”: Specifies the maximum amount of questions that should be returned.</td></tr>
<tr><td>Response body format (JSON):
{
data: {
{
questionID: number;
title: string;
categories: string[];
complexity: string;
linkToQuestion: string;
questionDescription: string;
questionImageURLs?: string[] | undefined
} [ ];
}
}</td></tr>
<tr><td rowspan="2">/</td><td rowspan="2">POST</td><td>Creates a new question.</td><td rowspan="2">Only admins</td></tr>
<tr><td>Request body:
● title: string
● categories: string[ ]
● complexity: “Easy” | “Medium” | “Hard”
● linkToQuestion: valid url
● questionDescription: string</td></tr>
<tr><td rowspan="2">/categories</td><td rowspan="2">GET</td><td>Gets the set of question categories in PeerPrep.</td><td rowspan="2">Any user</td></tr>
<tr><td>Response body format (JSON):
{
data: string[];
}</td></tr>
</table>

<table>
<tr><td rowspan="2">/:questionId</td><td rowspan="2">GET</td><td>Gets detailed information of the question specified by the questionId.</td><td rowspan="2">Any user</td></tr>
<tr><td>Response body format (JSON):
{
data: {
{
questionID: number;
title: string;
categories: string[];
complexity: string;
linkToQuestion: string;
questionDescription: string;
questionImageURLs?: string[] | undefined
} ;
}
}</td></tr>
<tr><td>/:questionId</td><td>DELETE</td><td>Deletes the question specified by the questionId.</td><td>Only admins</td></tr>
<tr><td rowspan="2">/:questionId</td><td rowspan="2">PATCH</td><td>Updates the question specified by the questionId.</td><td rowspan="2">Only admins</td></tr>
<tr><td>Request body:
● title: string
● categories: string[ ]
● complexity: “Easy” | “Medium” | “Hard”
● linkToQuestion: valid url
● questionDescription: string</td></tr>
</table>

One key difference between the question service and the other services is the existence of endpoints that are role-protected (only users with the ‘Admin’ role should be able to access these routes). These admin-only routes are protected using an additional middleware that checks for the user's role when they are requesting admin-only resources.

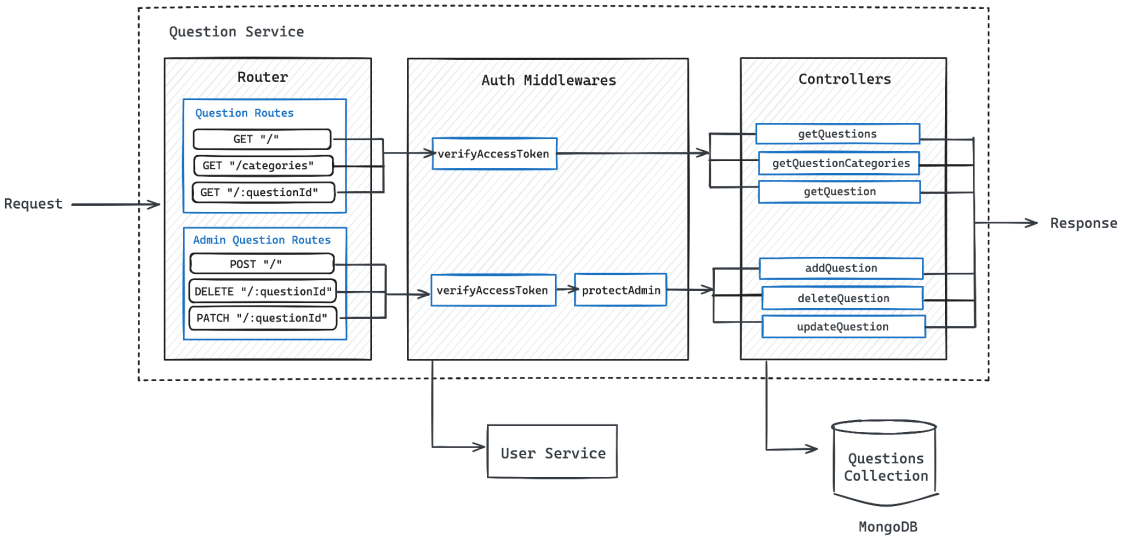


Fig. Question Service Diagram

Key Features

Fetching Questions

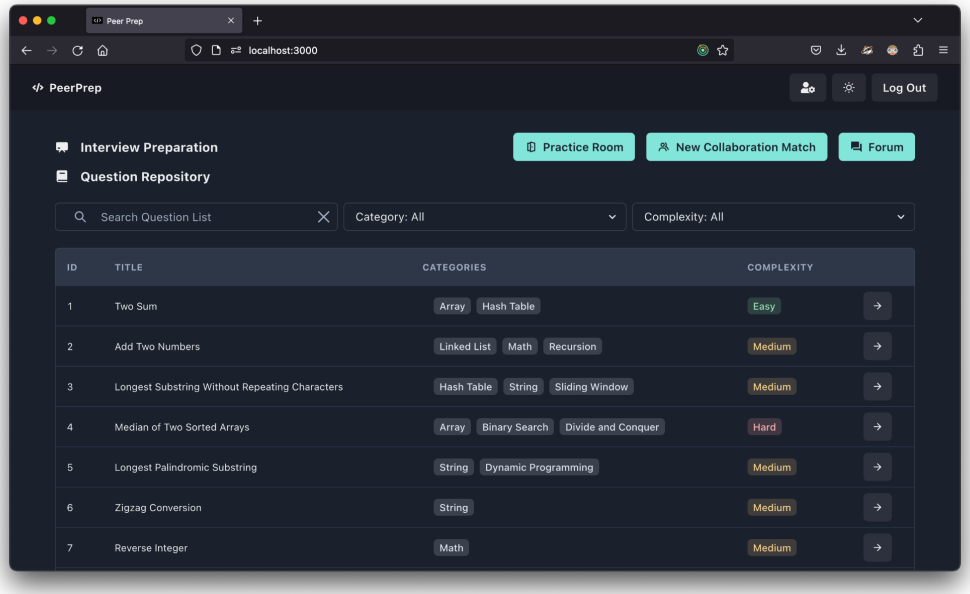


Fig. PeerPrep Questions List

The question service is able to provide the list of questions in PeerPrep's question repository. This set of questions can be maintained by our administrator, and is populated by our Serverless function which scrapes questions from LeetCode. More details on how our Serverless function works can be found in the section on Assignment 6 below.

Viewing a Question

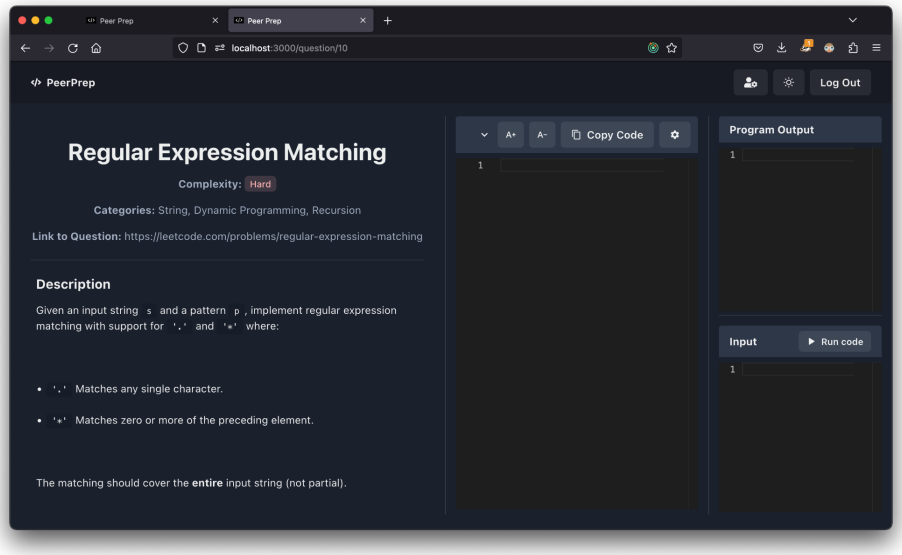


Fig. Viewing a Question

The question service can also provide details on a specific question (identified by the unique question ID). The question service returns all the details of the question, such as its title, complexity, categories, links, and description. The question description is returned as HTML to support the rich-text display of question descriptions (allowing us to render code snippets and images).

Maintaining Questions (Create, Update, Delete)

Administrators can create, update and delete questions in PeerPrep's repository.

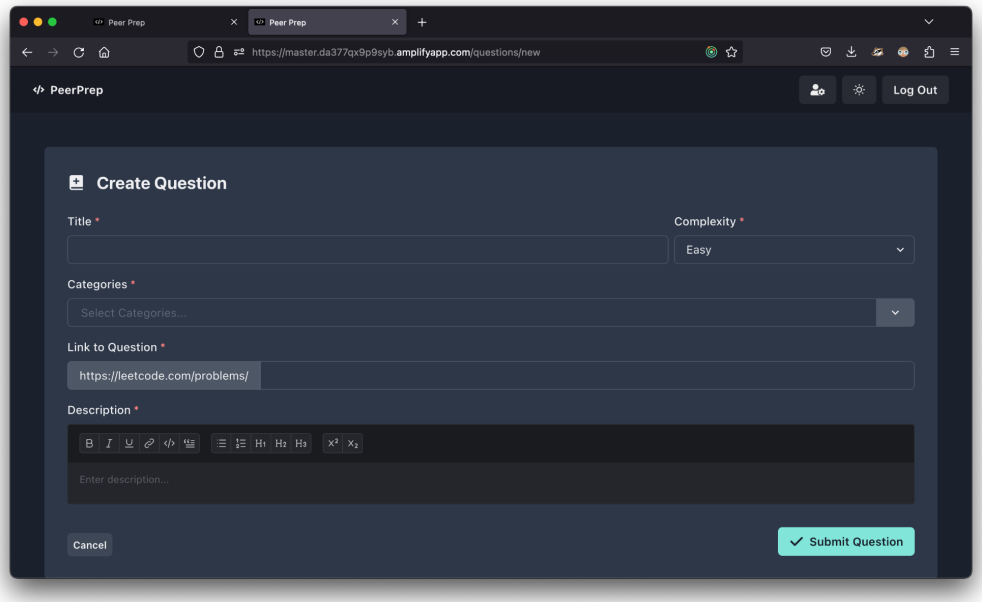


Fig. Creating a question with an admin account

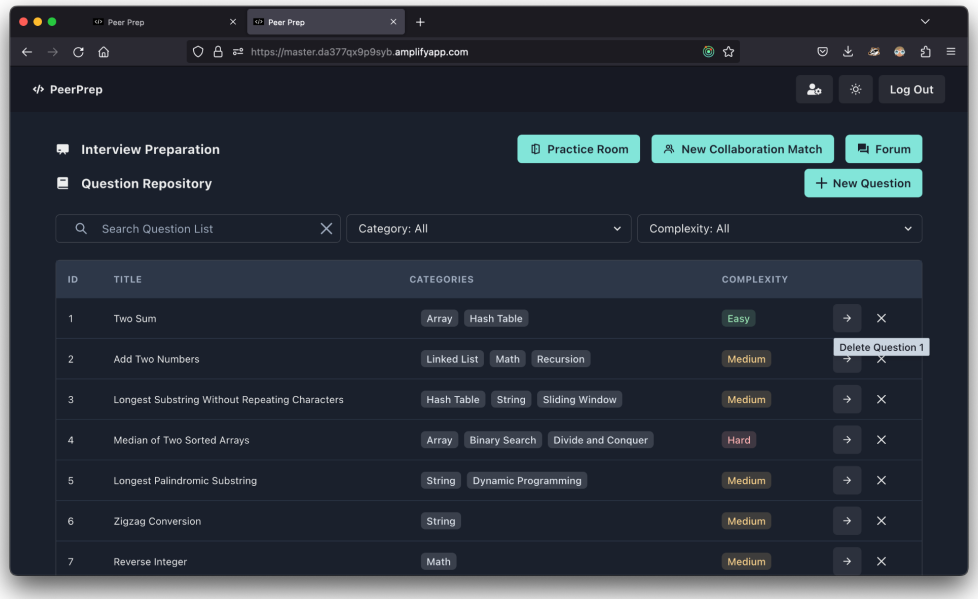


Fig. Deleting a question with an admin account

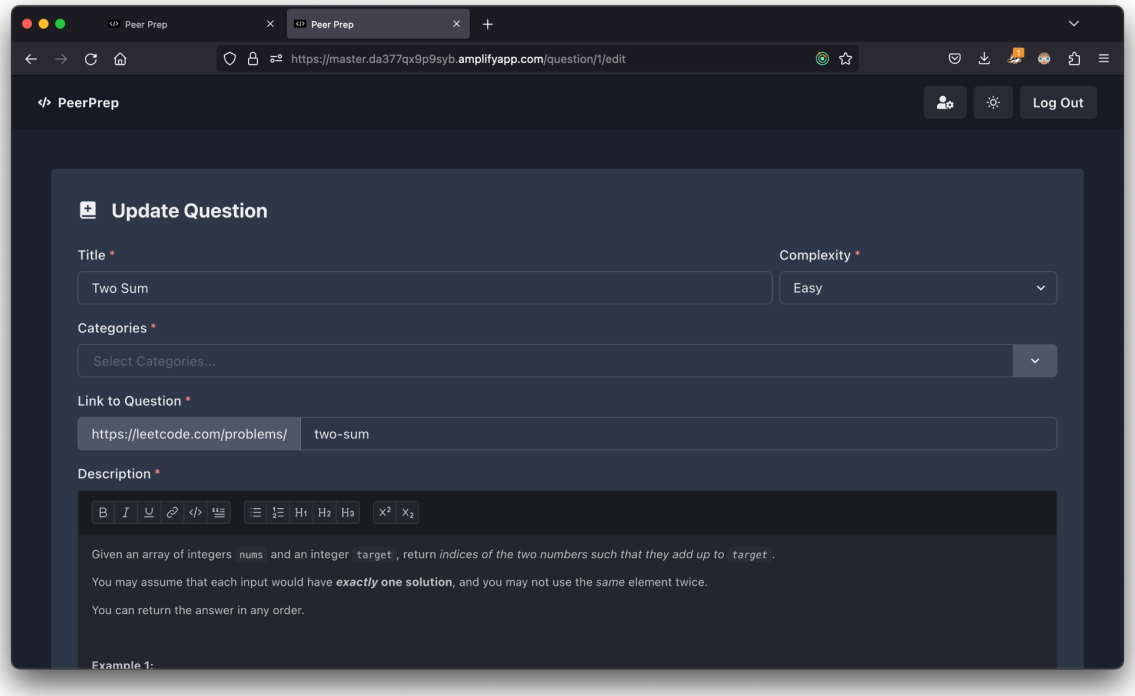


Fig. Updating a question with an admin account

Design Choice- Database

As mentioned in the above section on Database Design, the questions are stored in MongoDB which is a NoSQL database. We decided to use a NoSQL database as it is well-suited for document-based storage.

The nature of the question data is unstructured and loosely related. As such, we did not see a benefit to using a relational database such as SQL for the question database. Using a non-relational model also allows us to scale better (horizontally, ie: allowing us to store different types of data easily) and is more flexible and adaptable to fast-changing requirements.

In the case where we wanted to build quickly and saw that there might be database schema changes required along the way as we developed our other microservices, we hence decided to go with a NoSQL MongoDB for the question service.

f) User Service

Authentication

For authentication, we used JSON Web Tokens (JWT) to implement a token-based authentication system. This feature is crucial in providing users with protection over their credentials and account, which gives them a sense of security when visiting, and encourages them to stay engaged on this platform and return to this application.

Features of JWT

We opted to use JWTs as they offer security, compatibility, statelessness, and performance:

- They are digitally signed and encrypted, providing a strong level of security. The signature ensures the integrity and authenticity of the token, making it difficult for attackers to tamper with or forge tokens.
- They use a widely accepted and standardised format (RFC 7519) for representing claims between two parties, allowing compatibility across libraries and platforms, making our application's security expandable and long-supported.
- They are stateless, meaning the server doesn't need to store session information or maintain session state for individual users. This simplifies our application's architecture, making it more scalable and easier to manage.
- They are also self-contained and don't rely on additional databases for session validation, which improves performance and reduces latency.

Statelessness and performance are especially important factors in our decision-making process as our application is expected to have relatively acceptable performance with a high number of concurrent users.

Implementation of Keys

The access and refresh tokens both use respective symmetric keys, where the key for the access token has been shared with the rest of the services -- user service is the issuer -- in the application. The key is used to verify the signatures of the access token -- each service has authentication middleware which uses the key in checking the integrity and authenticity of the access token.

For future considerations, the access token key can each be easily changed to a public-private key pair, in the context of asymmetric cryptography. In this setup, the issuer server (user service) holds the private key for signing JWTs, and the other services hold the corresponding public key to verify the JWT signatures, which has been shared by the issuer

server (user service). The issuer server can create an access token and sign it using its private key, while the other services can use the public key to verify the access token's signature.This enhances security, as the private key is never shared, reducing the risk of compromise.

JWT Architectural Model

We opted for the access and refresh token model, a robust authentication and authorisation mechanism widely adopted in modern applications and APIs.

1. **Access Token**

- The access token is a JWT issued by the user service upon successful user authentication. These tokens are used by clients to access protected resources on behalf of the authenticated user.
- The access token is stored by the client in the cookies, which is made secure by setting it as httpOnly to protect against XSS attacks, and ensuring that the cookie is only transmitted over secure HTTPS connections (secure: true). The cookie is allowed to be sent in cross-origin requests by setting sameSite: "none", which is necessary for our microservices architecture.
- Its payload contains all immutable user information about the user's ID number and role (i.e., designated maintainers as "ADMIN", regular registered users as "USER", and unauthenticated and unauthorised users). This is so the jwt tokens are unaffected by changes to the mutable user information, e.g. the email address associated with each account..
- Here, they have a medium lifespan (i.e. 1 day) as a compromise between minimising the risk of being used for unauthorised access if they are compromised, while also allowing for the current JWT architecture, explained under "**Generation of new Access Token**".

2. **Microservices' Token Verification**

- Authentication middleware exists on each service, enabling each service to, without consulting the user service, verify the authenticity and integrity of the access token using the provided jwt library verify method, which verifies the signature of the access token using the secret access token key.
- For regular requests, authentication is handled for protected routes before the request is processed further in the service APIs.
- For services that use sockets, which facilitate single, long-lived connections between the client (frontend) and the server (service API), authentication is handled when the socket connects, where the access token is retrieved from the cookie and is verified

using the provided jwt library verify method. If unsuccessful, it disconnects the socket from the server, and provides an error message; otherwise, it allows it to proceed with the connection.

- By verifying the signature, the receiving service ensures that the token has not been tampered with (ensuring integrity) and that it was indeed issued by a trusted authentication server, in this case the user service server (ensuring authenticity).

3. Refresh Token

- A refresh token is another JWT issued alongside the access token during the initial authentication process. They are used to obtain new access tokens without requiring the user to re-enter their credentials, helping to create a seamless user experience.
- Here, they have a much longer lifespan (i.e. 1 month) and are stored by the client in the cookies, which is made secure by setting it as httpOnly to protect against XSS attacks, and ensuring that the cookie is only transmitted over secure HTTPS connections (secure: true). The cookie is allowed to be sent in cross-origin requests by setting sameSite: "none", which is necessary for our microservices architecture. They are also stored on the server-side upon creation, to keep records of which refresh tokens have been issued by the user service.
- The refresh tokens being also stored server-side is for additional verification, where only one refresh token, the latest one, is stored -- when the user service is presented with a refresh token, aside from verifying the signature of the refresh token ensuring its authenticity and integrity, it further checks that it is indeed the latest refresh token issued by it, by decoding the refresh token to obtain the user id, and then checking the corresponding user's records to see if that token is the user's latest generated refresh token.
- The reasoning behind only storing the refresh token from the latest login:
    - Ensure 'Hygiene' of Backend Storage: in some scenarios, users may have their refresh tokens go missing, which could make deleting disused missing refresh tokens while keeping those still being used a complicated affair. More than likely, the backend storage could accumulate with these 'stale' refresh tokens, clustering the storage. Having only one for each user thus ensures that even if the user loses their refresh token, the next login will overwrite the previously stored token on the backend storage, thus ensuring the backend storage does not accumulate with 'stale' information, while staying a predictable constant size per user.
    - Security: Prevent the potential stealing of 'stale' refresh tokens that may go unnoticed due to there being many refresh tokens per user. Additionally, if the

user suspects their refresh token has been stolen, they can log out and log in which overwrites the old refresh token with a new one, thus rendering the potentially stolen one invalid.

Generation of new Access Token

Initially, we opted to have new access tokens be generated reactively: the access tokens were to have an even shorter lifespan, and the responses from the services' APIs would be intercepted and checked for 401 error statuses, along with checking if it is not possible to get the current user during the application's rendering through the application router. This would have covered places when an invalid or missing access token would affect user experience, such as the initial rendering and subsequent re-renderings of the application, as well as data requests through the services' APIs.

In the case of intercepting services' APIs, upon getting a 401 error response, the application would call the user service API to generate a new access token if refresh token verification is successful, following which the original request is then tried again; otherwise, if the verification yields an error, the error is thrown for the frontend to handle, like through displaying error notifications.

Similarly, in the application router handling the rendering and re-rendering of the application, upon a render, it would attempt to get the current user if the log-in state appears to be false, and set the state of the user accordingly. If unsuccessful, it would then call the user service API to generate a new access token, before attempting again to set the current user.

However, in practice, the intercepting of service APIs affected the responses passed to the frontend from the service APIs, or even responses from one service to another, like through a message broker like rabbitmq. Thus, it unfortunately was decided to drop the intercepting of the service APIs -- to compensate for this loss of functionality, it was decided to extend the access tokens' lifespan to 1 day -- the user can perform a page refresh on their browser if their access token becomes invalid, which generates a new one.

This approach largely preserves the seamless user experience -- the user just has to perform an additional refresh daily, which hardly affects their experience on the application, while still having security in mind -- with an access token that expires in a relatively short time. Most importantly this approach removes possible interference with the responses from service APIs, which could limit their functionalities or produce unwanted errors.

Reset Password

Password reset functionality is a critical aspect of user account management, and implementing it securely is of utmost importance. By employing Nodemailer, web developers can send password reset emails containing secure tokens or links, allowing users to regain access to their accounts in a straightforward and secure manner.

Upon receiving a password reset request, the system generates a unique and cryptographically secure token. This token is essential for verifying the identity of the user initiating the reset. A unique URL, usually containing the token, is constructed. This URL serves as the password reset link and directs users to the password reset page on the application. Nodemailer is employed to send an email to the user's registered email address. The email contains instructions and the password reset link.

Users receive the email with the password reset instructions and link. It informs them of the necessary steps to reset their password. Users click on the reset link within the email, directing them to the password reset page. The token is included as a query parameter in the URL. If the token is valid, users are granted permission to reset their password. They can enter a new password, which is securely hashed and stored in the database.

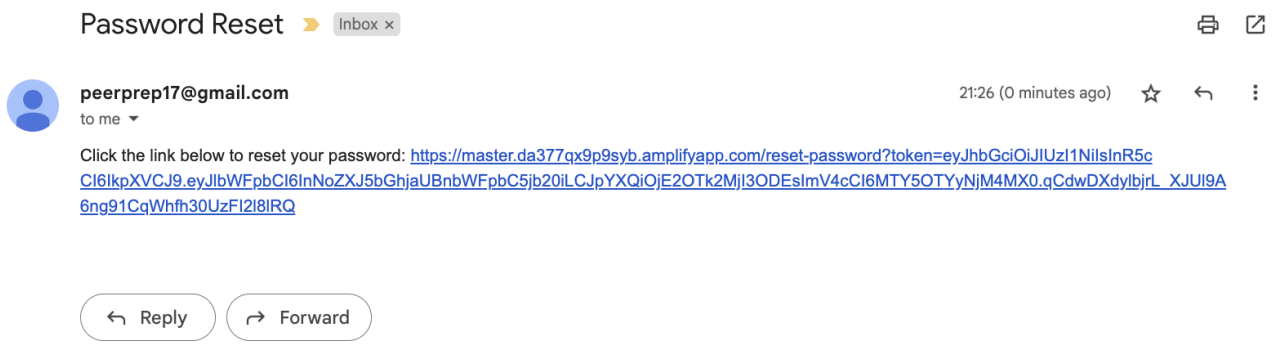


Fig . Reset password email sent by nodemailer

Progress Management

For our Nice-to-Haves, we have implemented a feature to track and save users' attempted questions. This feature is valuable in providing users with a clear view of their progress and encourages them to stay engaged with the platform. Seeing their accomplishments visually can be highly motivating and satisfying, which increases the likelihood of users returning to the application.

While a history service separate backend could have been created, we chose to use the existing user service due to the following reasons:

a. **Resource Efficiency**: Creating and maintaining a separate backend service requires additional development time and resources. By reusing the existing user service, it saves time and reduces the complexity of the system.

b. **Code Reusability**: The existing user service already has components or functionalities that can be repurposed for the history service, reusing that code can be more efficient than creating from scratch.

Users' attempted questions are saved in a PostgreSQL database using Prisma. We chose to use a relational database as there are relationships between users, attempted questions, and questions themselves, using a relational database like PostgreSQL along with Prisma makes it easier to model and manage these connections. Each time a user attempts a question, relevant information (e.g., user ID, question ID, timestamp, etc.) is saved in the database. This allows for detailed tracking of user activity and progress over time.

With the data stored in the database, we performed various analyses, such as calculating metrics like the number of questions attempted in a time period. Inspired by GitHub's heatmap-style visualisation, we used a similar grid where each day is represented by a square, with colour intensity indicating the level of activity (e.g., darker squares for more activity). By mapping the user's attempted questions to a similar heatmap format, users can visually assess their progress.

The GitHub heatmap-style visualisation adds a familiar and intuitive element to the application. Users who are already familiar with GitHub will find it easy to interpret and engaging to use for tracking their progress in the application.

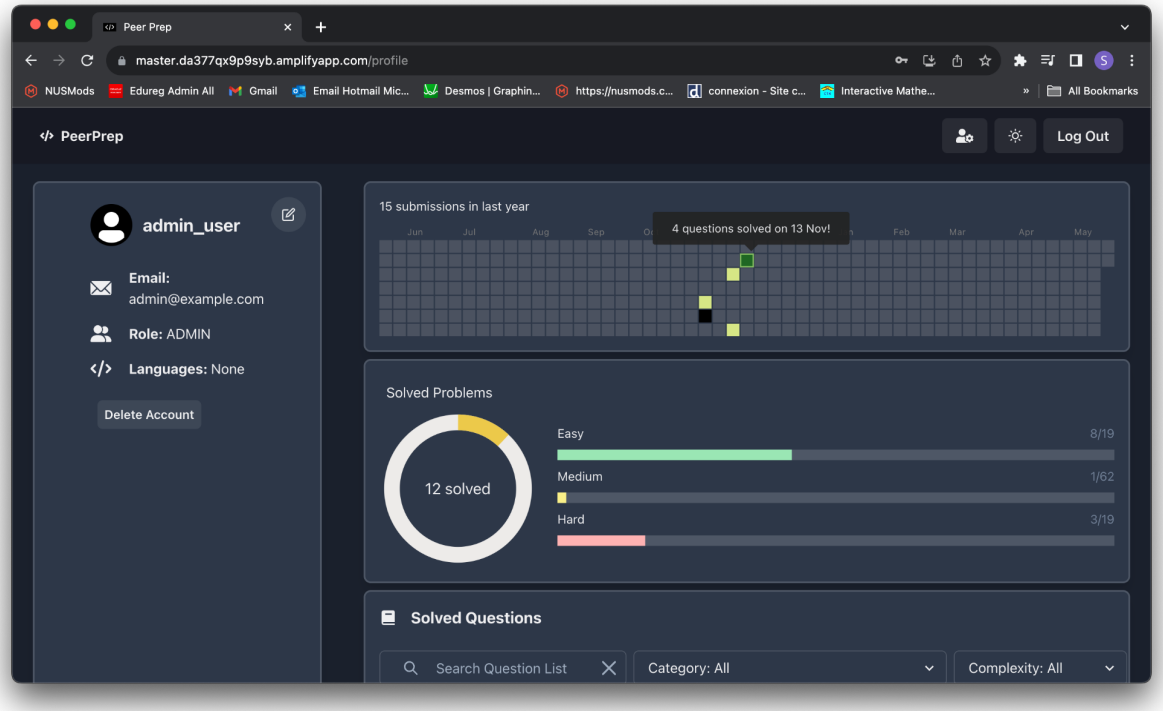


Fig . GitHub-style heatmap representing user progress

GitHub OAuth

Password and account management can be hard with so many apps. So, to provide a more seamless experience with PeerPrep, the team decided to integrate GitHub OAuth into PeerPrep so that users can simply login to PeerPrep with their GitHub accounts at the click of a button without the need to remember any login credentials for PeerPrep. GitHub was selected as the OAuth Provider considering future features which might see further integrations with GitHub in PeerPrep, such as synchronisation with GitHub repositories.

Authentication flow with GitHub OAuth (User Activity Diagram)

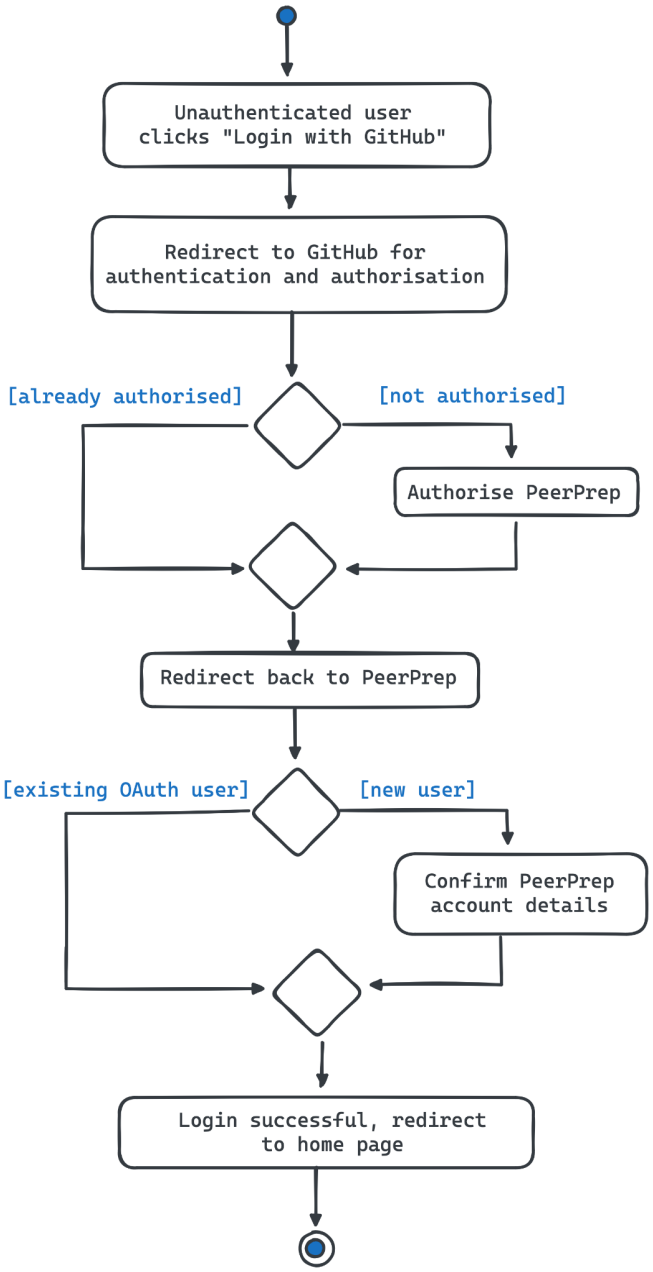


Fig. User Activity Diagram for Login with GitHub

To log in to PeerPrep with GitHub OAuth, users should select the “Login with GitHub” button in the login form instead of performing regular login.

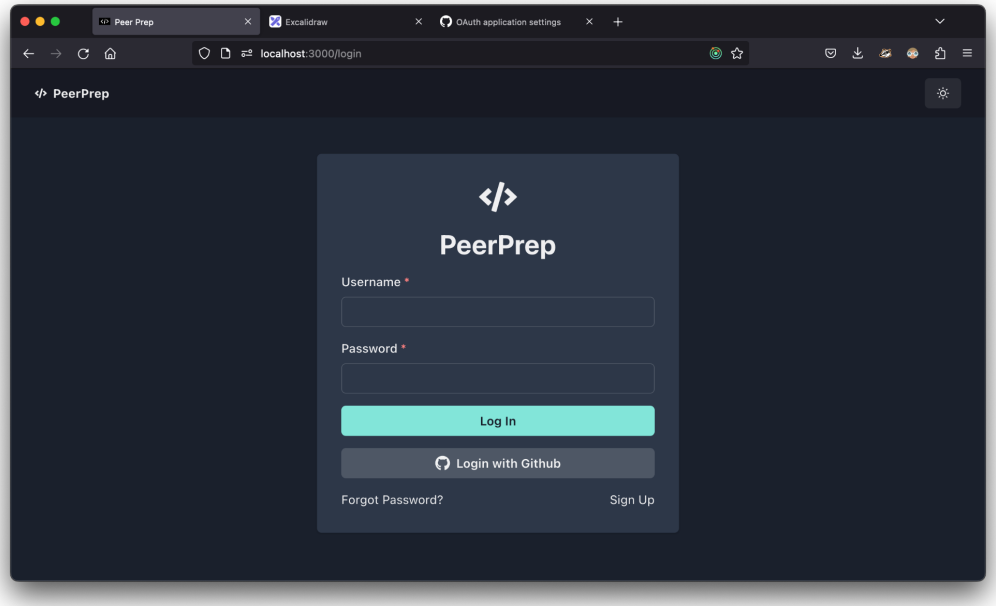


Fig. Login Page with Option to Log In with GitHub

Users who have not authorised PeerPrep to access their GitHub accounts will then be directed to the GitHub authorisation page to authorise PeerPrep for GitHub OAuth. These can be new users or users who have revoked access for PeerPrep on their GitHub accounts.

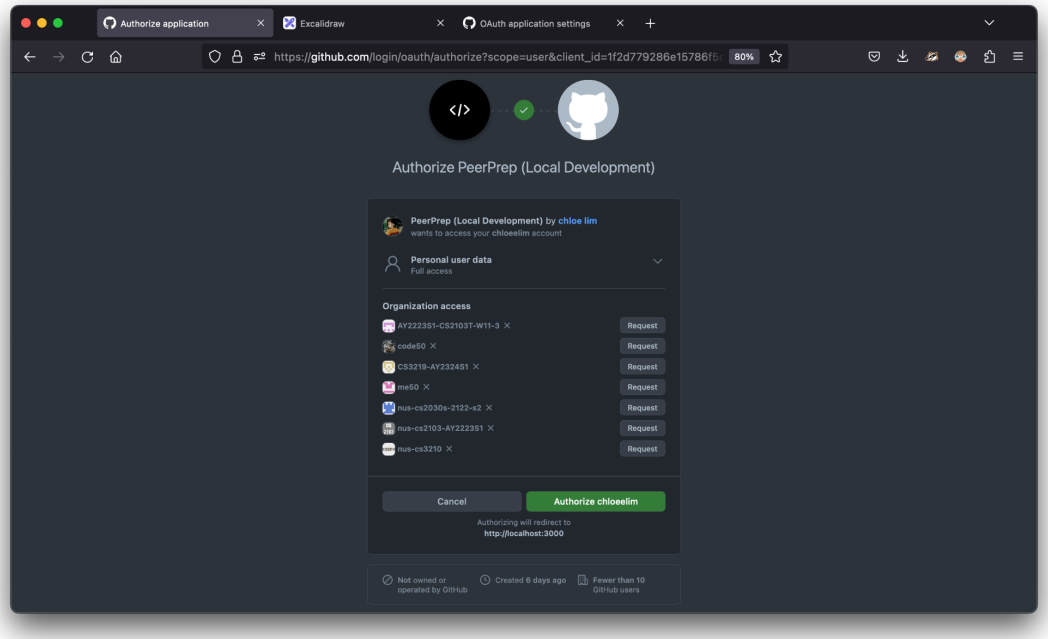


Fig. Redirected Authorisation Page on GitHub

Upon successful authorisation, new users will be prompted to confirm their user details (username and email are populated from user's GitHub data obtained using GitHub's API, authorised by OAuth). These user details are still needed from the user despite using GitHub to login; usernames serve as human-friendly unique identifiers for our users and emails are needed for account backup (see section on password reset).

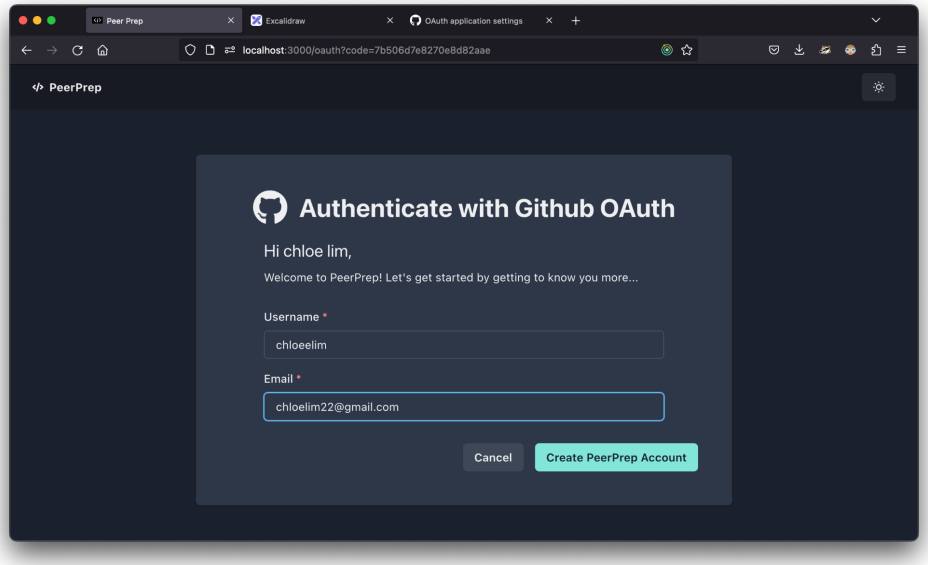


Fig. Setting up PeerPrep account for new users after successful authorisation

Upon successful account creation, users will be automatically redirected to PeerPrep's home page. Existing users will simply be logged in and redirected to PeerPrep's home page upon successful authentication/ authorisation with GitHub.

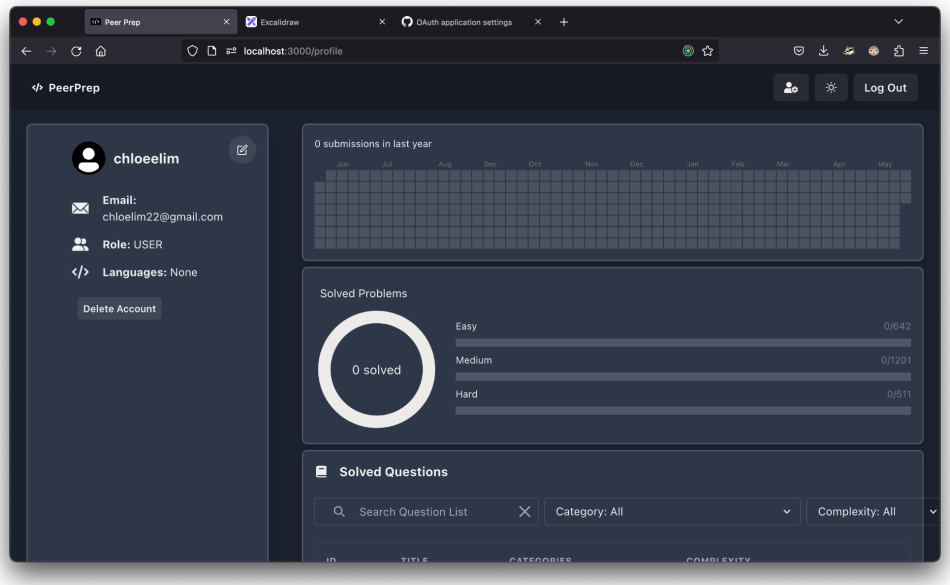


Fig. New OAuth Authenticated User Profile

Implementation (Sequence Diagram)

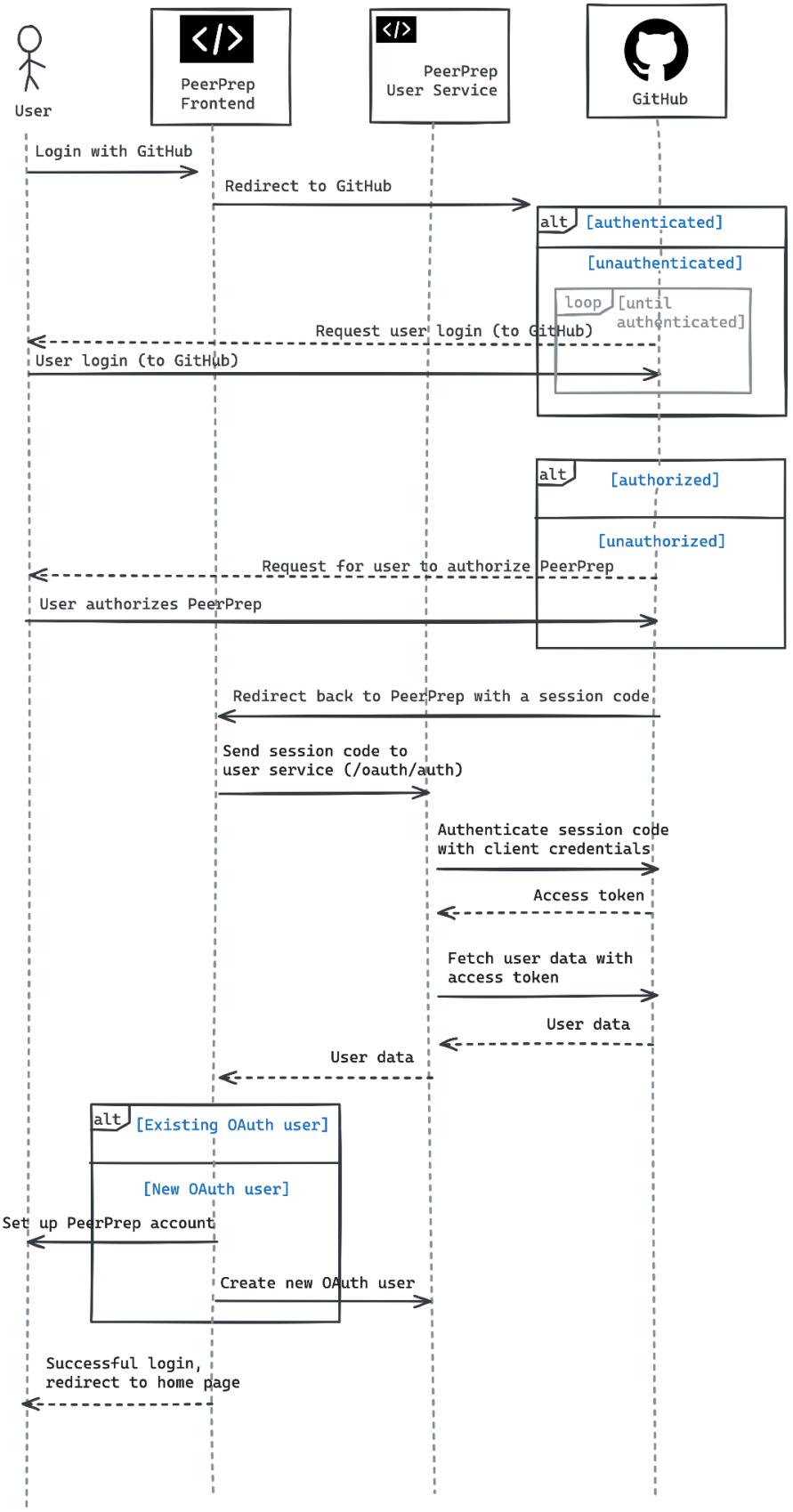


Fig. OAuth Login Sequence Diagram

PeerPrep's GitHub OAuth user authentication was implemented following the authentication flow outline in GitHub's OAuth Authentication guide. It follows the basic flow shown in the sequence diagram above.

Essentially, the frontend client will redirect the user to GitHub who will authenticate the user, and check if the user has authorised PeerPrep to access their account. If PeerPrep is not authorised, GitHub will ask for the user to authorise PeerPrep before the user can proceed.

Upon successful authentication and authorisation on GitHub's end, GitHub will call the redirect URL provided to it, and redirect the user back to PeerPrep's frontend with a session code that can be used to obtain the user's access code. The frontend then sends this session code to the user service so that the user service can authenticate the user.

This session code can only be used once, and is needed for PeerPrep to obtain the actual access code from GitHub. The access code is used by PeerPrep to make authenticated requests to GitHub's APIs to access data the user has authorised it to access. This may seem like a round-about way of getting the access code, but is required for security reasons. This is because PeerPrep needs to be sure that the access code comes from GitHub and so, that the user is really the user (rather than malicious third party applications that may, for some reason, have valid access codes for some user as the user had previously granted access for the application to access their data with GitHub OAuth).

To integrate OAuth as a valid authentication method with the existing login method in PeerPrep, a new column 'github_id' was added to the User's column. This is the unique identifier for all GitHub OAuth logins. It was selected over fields like usernames and emails as it is sure not to be changed (conversely, usernames and emails may be updated).

With the user's data obtained from GitHub with the access code now, the user service will then try to find a user in the database with the matching GitHub ID and then log the user in (since the user was already authenticated with GitHub).

If no such user is found, that means that the user is new- and it prompts the frontend to ask the user to set up their PeerPrep account before creating a new user account using these credentials and data.

g) Matching Service

Matching via DB

The matching service's primary responsibility is to match users' who requested a collaboration match based on their selected complexity and category. To maximise flexibility for the user, we allow them to select any number of options, including zero, for both complexity and category.

Um

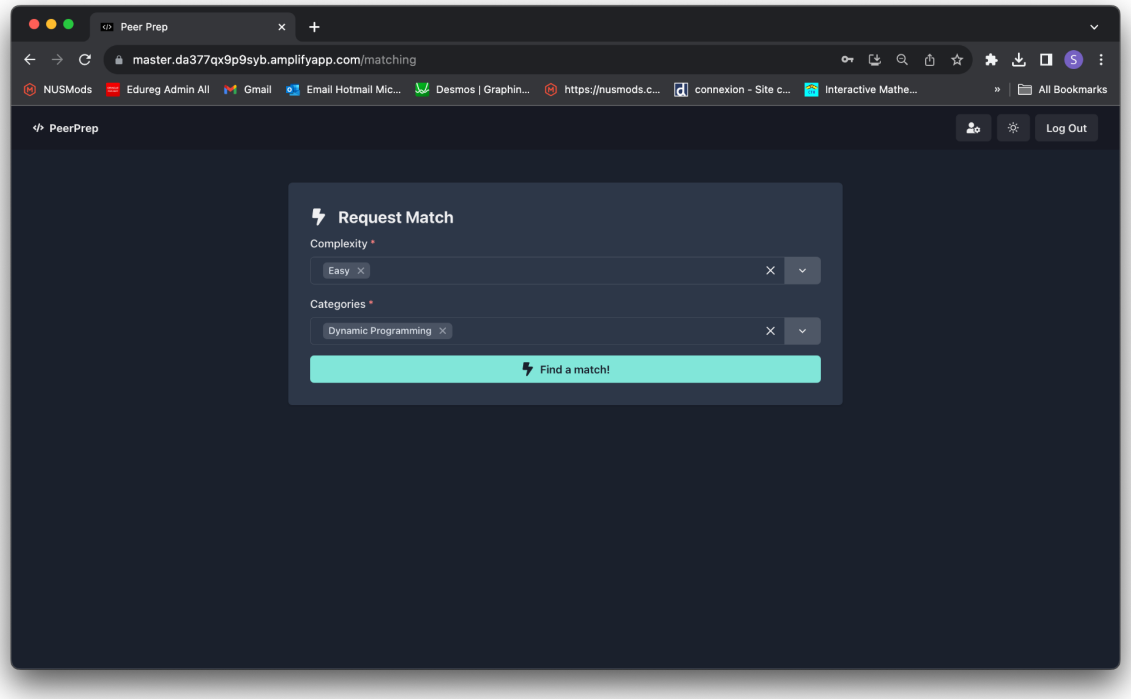


Fig . Match Request Form

We have a MongoDB database that stores information about matching requests. They contain information such as the userId, status(pending/timeout/matched) and their requested category/complexity.

Our algorithm will then match queries as follows:

- If the match request has specified categories, filter existing pending match requests for either no category requested or containing at least one shared question category.
- Do the same for question complexity.

We chose this method to minimise the wait time and increase the chance of a successful match for matching requests, since the alternative (for example, exact matches for question category) is quite unlikely, since we allowed multiple choices during selection.

If there are multiple qualifying matching requests after filtering, the earliest is chosen. If there are no qualifying matches, it is inserted into the database as a pending matched request.

Finally, when a match is found, there will be two kinds of asynchronous communication that must follow.

- Both users will be notified of their successful match via socket.io, and will be sent to the next stage, the collaboration room.
- Details on the successful match, such as users' ids and combined question categories/complexities will be published to the RabbitMQ queue. More details about this are found below.

Design choice - MongoDB

We chose to use MongoDB to store and query pending matches due to its in-built flexible query language. This is in comparison to key-value stores such as Redis. This is because our matching involves more than one column, and we are doing more than just exact matches, so it is better for our use case.

Messaging protocol (Sending message to the queue)

Once a suitable match is found, the pair is sent to the RabbitMQ queue. RabbitMQ acts as the messaging protocol, facilitating the communication between different components of the system. It handles the distribution of messages (in this case, matched user pairs) to the appropriate consumers. Once the connection is established, a channel is created. A queue named `match_results` is defined. A queue is a place where messages are temporarily stored until they are consumed by a recipient. This is further elaborated in the next section.

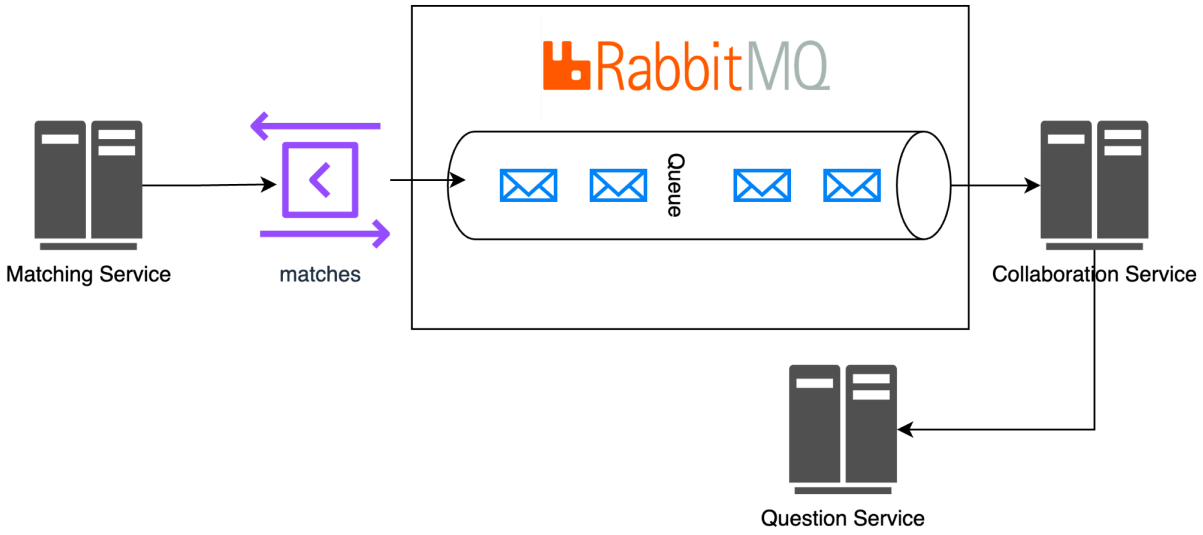


Fig . Message queue protocol using RabbitMQ

As shown from the logs below, messages containing the matching information would be sent to the RabbitMQ queue via the matching service, as seen from the log statement `[x] Sent ...`

```
➜  matching-service git:(master) npm start

> matching-service@1.0.0 start
> nodemon --exec ts-node ./src/app.ts

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node ./src/app.ts`
Connected to Mongoose
Server is running on http://localhost:9000
Found a match!
{
  userOne: 2,
  userTwo: 1,
  categories: [],
  difficulty_level: [ 'Easy' ],
  roomId: 'dc403479-e05b-49e8-b73d-43a36c3ae469'
}
[x] Sent {"userOne":2,"userTwo":1,"categories":[],"difficulty_level":["Easy"],"roomId":"dc403479-e05b-49e8-b73d-43a36c3ae469"}
socket NJIib957qXjpijnGAAAC disconnected
socket QweMXn6GiczBOpGZAAAF disconnected
```

Fig. Logs from the message queue

h) Collaboration Service

Messaging protocol (Consuming message from the queue)

There is a consumer for the `match_results` queue. This means it is actively listening for messages in that queue. When a message is received, it is processed in an asynchronous callback function. The message content is parsed and based on the matched result, it makes a request to fetch the relevant questions.

As shown from the logs, the consumer receives and consumes the message in the queue as seen from `[x] Received ...`, thereby creating the pair.

```
➜  collaboration-service git:(master) npm start

> collaboration-service@1.0.0 start
> HOST=localhost PORT=8081 npx y-websocket & nodemon --exec ts-node ./app.ts

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node ./app.ts`
running at 'localhost' on port 8081
Socket.io server is listening on http://localhost:8082
[*] Waiting for messages in match_results. To exit press CTRL+C
New connection: p28JrjenU7GYo9dYAAAB
New connection: KJeSbbtevvxUjnwQAAAD
New connection: NForWIcNtW99fAh3AAAF
New connection: 9olHrWwyg8sWlxaOAAAH
[x] Received {"userOne":2,"userTwo":1,"categories":[],"difficulty_level":["Easy"],"roomId":"dc403479-e05b-49e8-b73d-43a36c3ae469"}
New connection: urjYvC171WoEEqZWAAAM
New connection: 13GJGRlGiw5gVQaXAAAN
New connection: eEdB3P00OMpYyX5OAAAO
```

Fig . Logs showing consumer of message queue

The retrieved question IDs are stored in the pairInfo object. A new Pair object is created with this information and saved to the database. On the frontend side, it creates a collaboration room where users can work together in a live code editor environment.

```
Created pair: {
  userOne: 2,
  userTwo: 1,
  room_id: 'dc403479-e05b-49e8-b73d-43a36c3ae469',
  complexity: [ 'Easy' ],
  categories: [],
  question_ids: [ 69, 70 ],
  _id: new ObjectId("6551cf79384751b70af4fa3c"),
  __v: 0
}
```

Fig . Pair object created

Ensuring Secure Access to Collaboration Rooms

It is essential to implement a security measure to ensure that only authorised users can access the collaboration room. This is achieved by implementing a validation check on the room ID, which is stored in the pairInfo object. When a user attempts to access the collaboration room, the application first verifies if the user's credentials match the room ID stored in the database. If there is a match, the user is granted access to the live code editor environment.

However, if the credentials do not align, the system denies access, preventing unauthorised users from entering the collaboration space. This security measure helps maintain the integrity and privacy of the collaborative work environment, safeguarding it from any potential interference from outsiders.

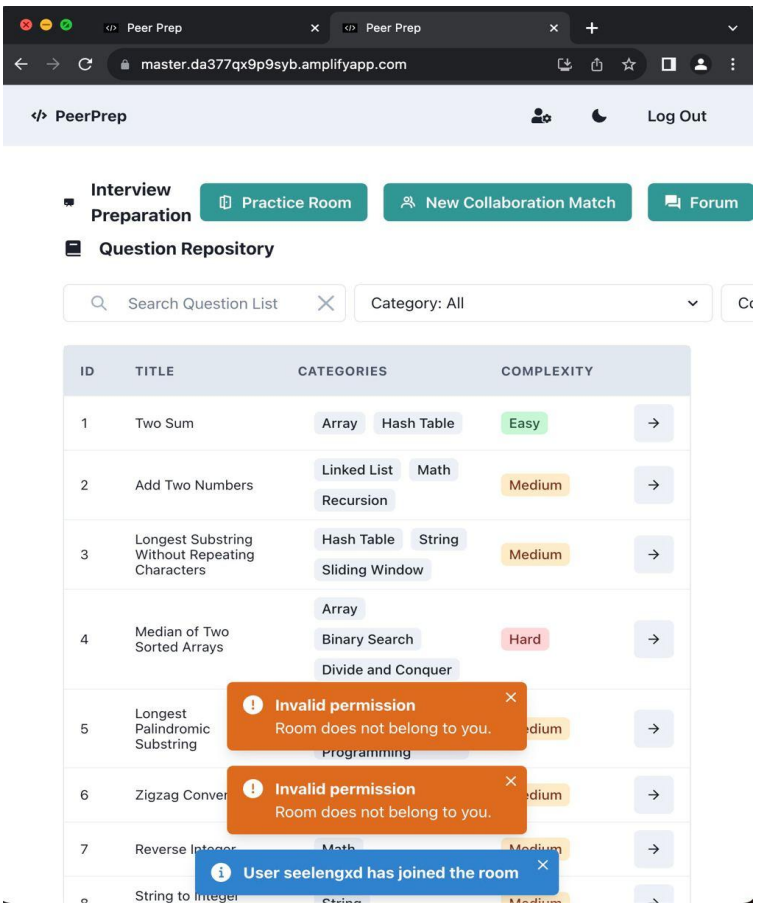


Fig . Toast message when an invalid user attempts to join a room

Web socket

Websockets are a bidirectional communication protocol that allows data to be exchanged both from the client to the server, and from the server to the client. As mentioned previously, websockets are behind much of the features in the collaboration rooms, such as synced code editors, real-time chatting, and the real-time interactive whiteboard. These features require a constant stream of information to be exchanged between the client and the server (both ways). The client would have to transmit updates when typing on the code editor, or when changing the editor language. The server would have to listen to these updates and broadcast messages to other client connections.

For more detailed details on the implementation of websockets on the frontend, and the rationale behind decisions for using websockets and for using the specific websocket libraries, refer to the section above. This section will focus more on how the websockets are handled from the server side, in the context set from the earlier websocket section.

Handling new connections

When a new client joins the collaboration room, it will emit a “join-room” event and also transmit its username to the server. Upon receiving this “join-room” event, the server will respond by emitting the question-id and initial-language of the room to sync the client with existing clients. Lastly, the server broadcasts a “user-join” event with the new client’s username to all connected clients so that they can notify their users.

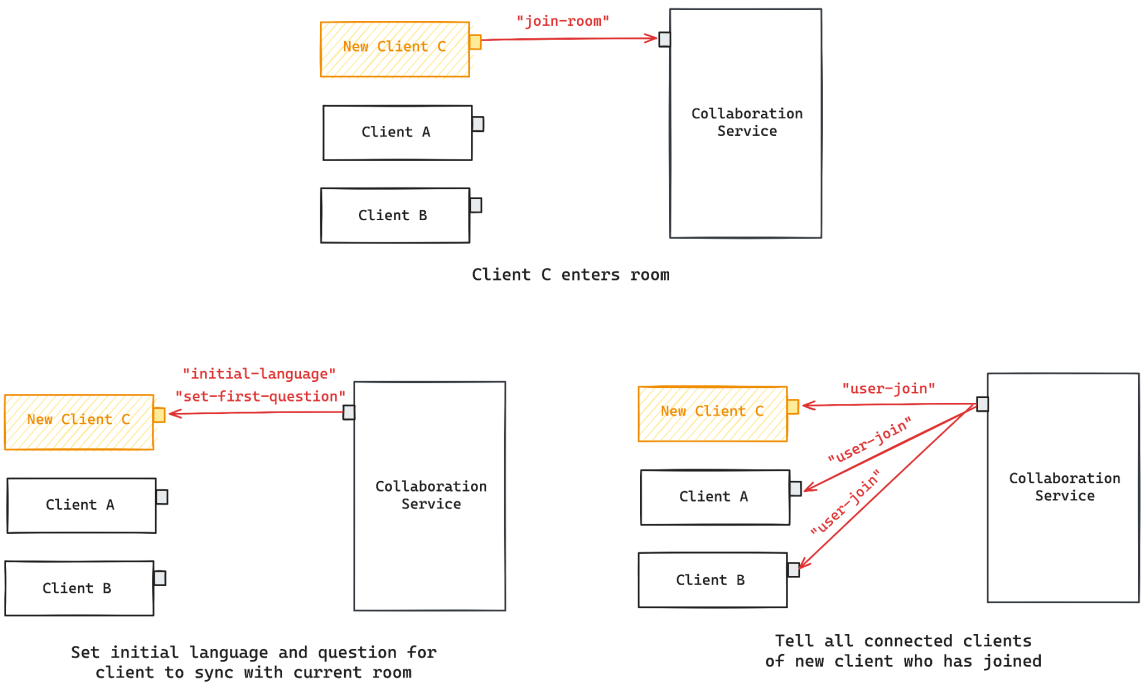


Fig. New client join websocket events

Handling Question Change

In practice rooms (ie: collaboration rooms formed without matching), users can set the question being attempted by the room. The question being attempted by all connected users in the same room should be synchronised. This synchronisation is handled by Socket.IO. When a user changes the current question, they will emit a “change-question” event, transmitting the new question ID with this event. The server will then respond by emitting the new question ID to all connected clients in the room so that they can also update their set question IDs.

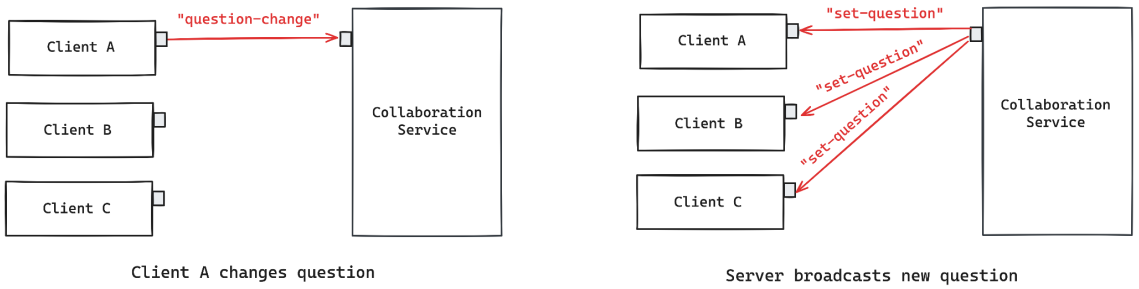


Fig. Client changes question in practice room

In matching rooms, questions cannot be manually set. Both users in the matching room must first agree to change the question before they can move onto the next question. As such, the question change event must be handled differently in the matching room. If a client tries to move to the next question, the server checks if both users have requested to move to the next question. If both users have agreed to move to the next question, the server will emit a “both-users-agreed-next” event to let the connected users know that they can proceed to the next question. Else the server emits a “waiting-for-other-user” event to let the connected clients know that both users must agree to move to the next question before proceeding.

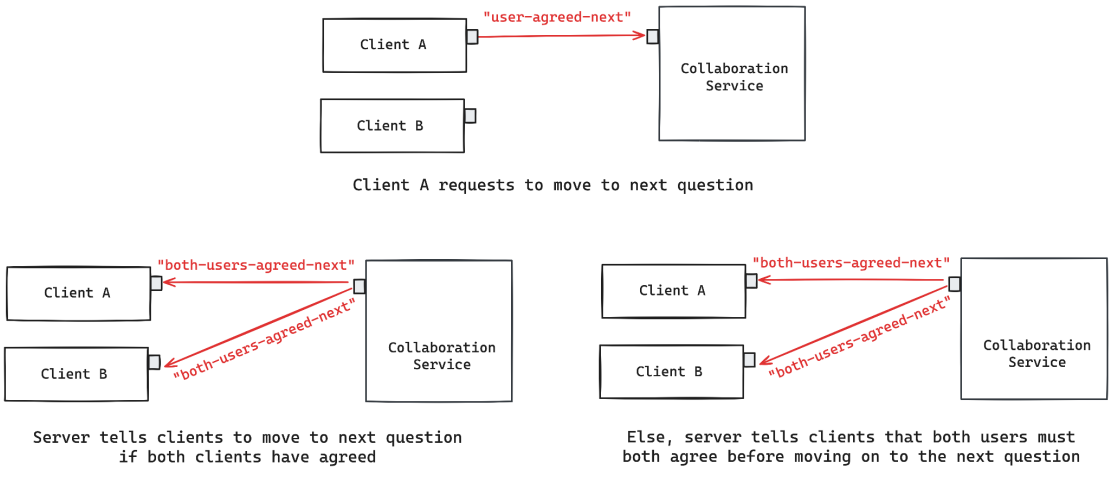


Fig. Client changing question in matching room

Handling Language Change

Languages of code editors in the same collaboration room must be synced as they are shared editors. We use Scoket.IO to handle language changes within the collaboration room. Whenever a client requests to change the room's coding language, the client will emit a "language-change" event along with the new language. The server will then update the saved room's language in the server (so that new users who join will also use this new language). Following which, the server broadcasts a "receive-language-change" event to all connected users in the room with the changed language so that the clients can react appropriately and change their code editor language to match.

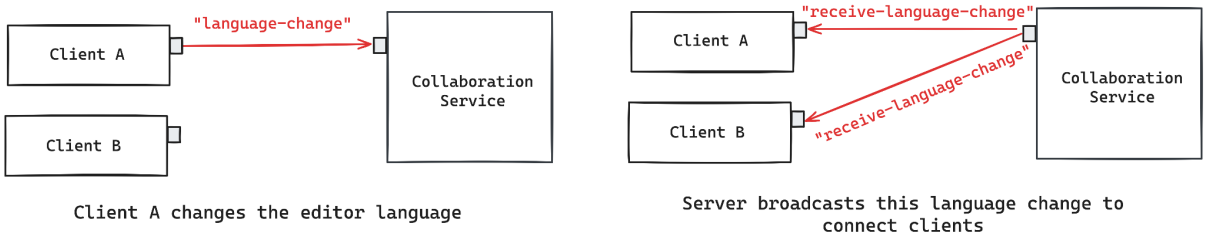


Fig. Language change in collaboration room

Demonstration

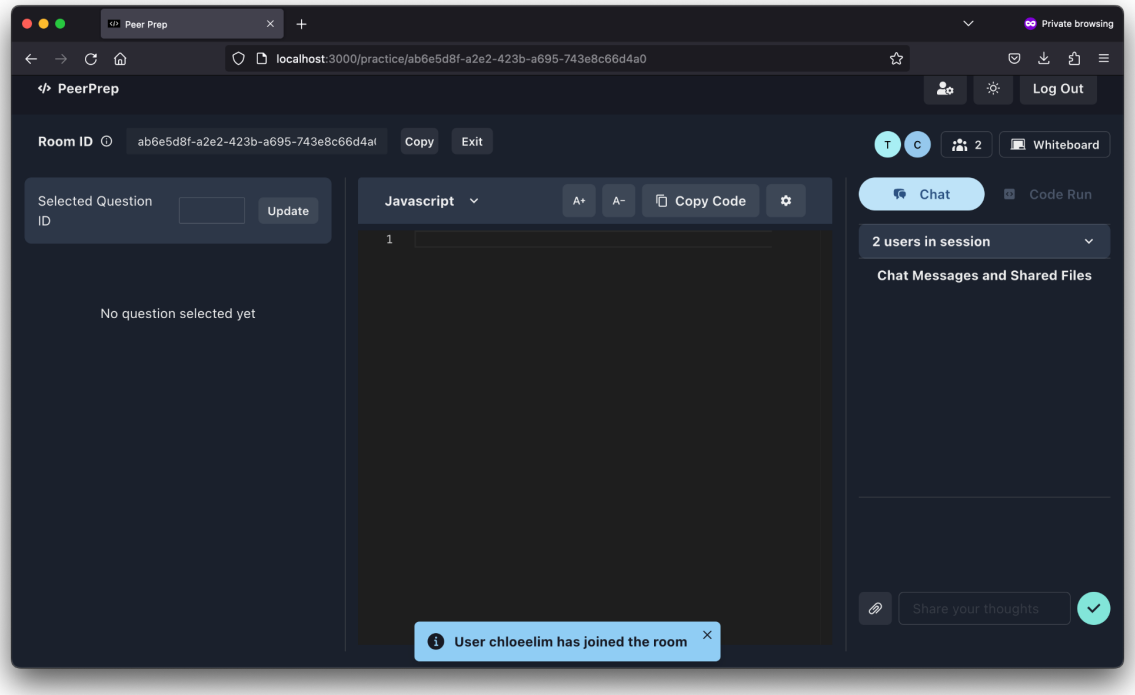


Fig. New user join room (new connection)

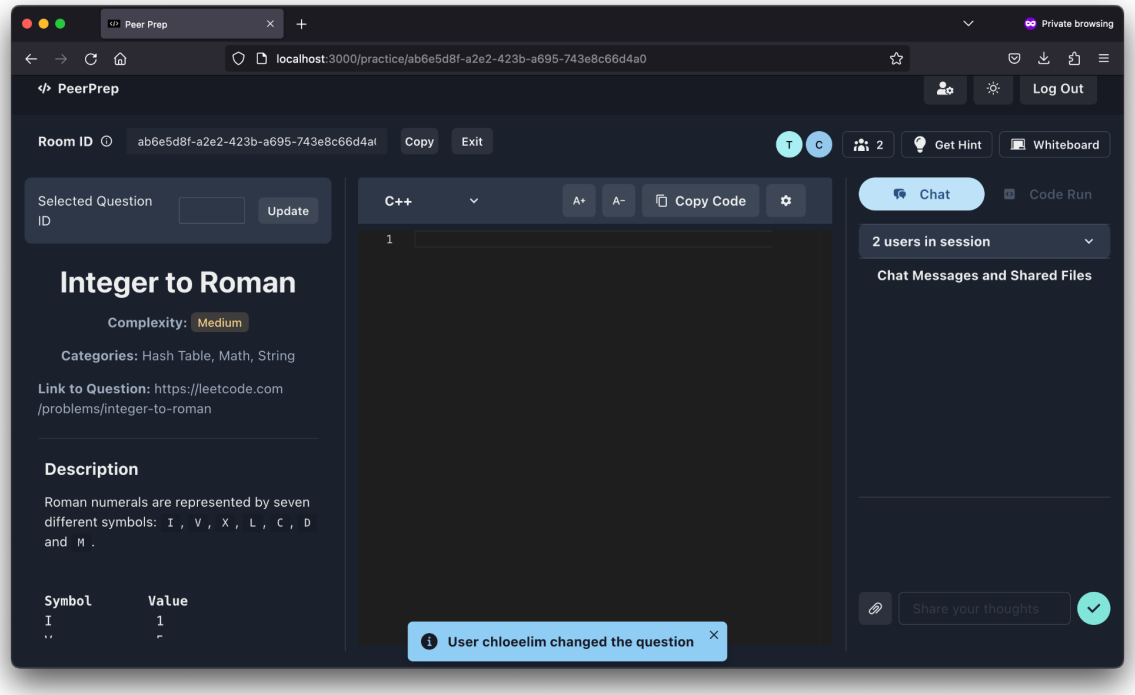


Fig. User changing question

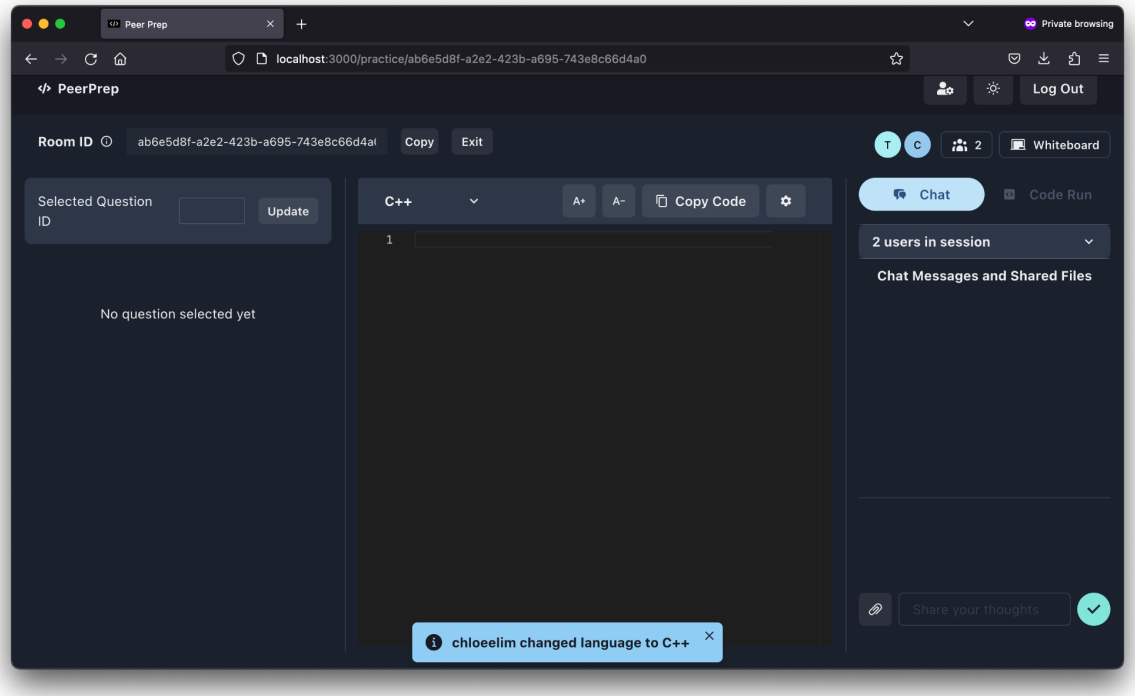


Fig. User changing language

i) Chat Service

The Chat Service is designed to further enhance the collaborative nature of the collaboration service, by providing another means of communication with the other users in the shared room other than the code editor. It facilitates real-time text messaging and file sharing with the other users in the shared room.

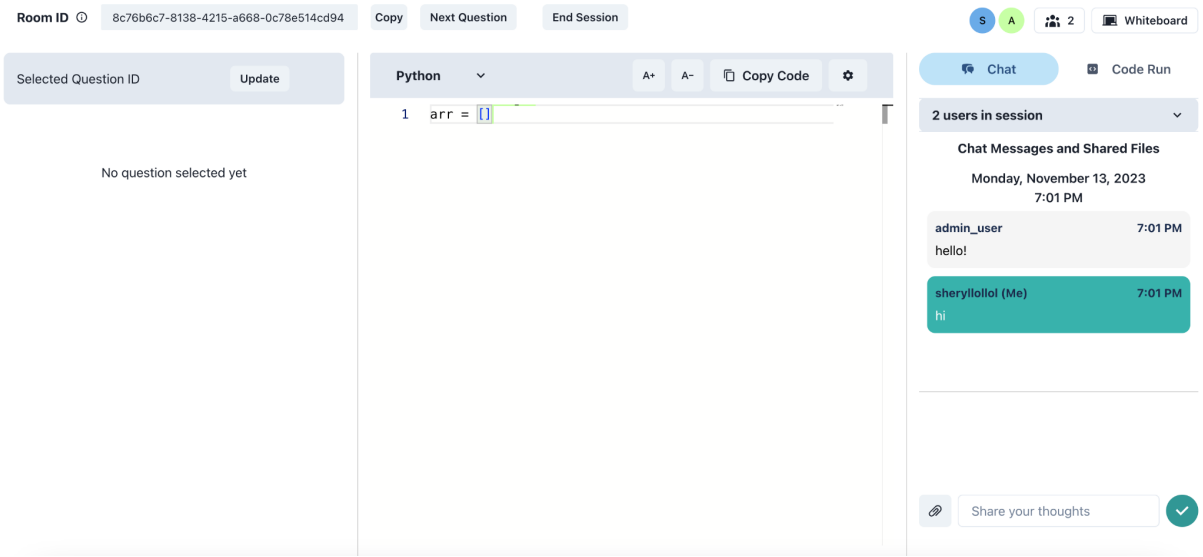


Fig . Real-Time Chat

Key features:

1. **Text Communication**

Text is sent to other users in the room, with received messages (which contain information like the message's user and its sent timestamp) being stored in the frontend, using state management. The application generates a visual representation for the messages, with each messaging displaying the information of the user, timestamp, and text. As the messages are stored in the frontend state, the messages only persist as long as the user does not leave or refresh the page.

There are also user experience considerations, such as auto-scrolling, providing visual separators between messages if they are sent on different days, or the time difference between consecutive messages is more than 15 minutes, providing a comfortable navigation experience when using the chat service's text messaging.

2. **File Communication**

Files with sizes below 10MB are allowed to be sent to other users in the room, with received files being stored in the frontend, using state management. The 10MB upload limit is such as it was the decided maximum buffer limit for the chat service server, as most useful files like PDFs and pictures to facilitate collaboration are largely below 10MB.

The application generates a visual representation for the list of files, with each file being displayed with the information: its filename, the user who uploaded it, its uploaded timestamp. As opposed to downloading the files immediately upon receival, a download button is provided for each file, so the files can be downloaded at the other users' discretion. As the files are stored in the frontend state, the files only persist as long as the user does not leave or refresh the page.

Like the messages, there are also user experience considerations for the files, such as auto-scrolling, providing visual separators between messages if they are sent on different days, or the time difference between consecutive messages is more than 15 minutes, providing a comfortable navigation experience when using the chat service's file sharing.

Technology and Architecture:

1. **Web sockets**

Websockets are a communication protocol that provides full-duplex -- both client and server can send and receive data independently at the same time -- communication channels over a single, long-lived connection. They were chosen mainly for this service due to their real-time communication enabled by its low-latency, broadcasting capability, and full-duplex capability. Additionally, its support for large data allows for the sharing of files.

Enables Real-Time Communication

- **Low-latency**:
    - Once a connection is established, data can be sent and received with lower latency compared to traditional HTTP polling or long-polling methods; this is because the connection is persistent, and thus there is no need to repeatedly open and close connections for each communication.
    - Additionally, this long-lived connection reduces the amount of unnecessary HTTP header data transmitted with each request, lowering network usage especially for continuous communication in this chat service. This efficiency also translates into lower latency.

- **Full-duplex**: Websockets allow both the server and the client to send messages independently, allowing users to send messages and receive updates without waiting for a response from the server.
- **Broadcasting capability**: Websockets support broadcasting messages to multiple clients simultaneously, which allows for the delivery of messages to all connected users in real-time.

Support for Large Data

- Websockets are capable of handling large amounts of data, making them suitable for supporting file sharing. Unlike traditional HTTP requests, which may have size limitations per request, and thus may require numerous requests for a file transfer, websockets can efficiently transmit large files without needing so many requests.

2. **Client-side storage, no server-side storage**

The messages and files received are all stored in state management in the frontend, with there being no persistent storage on the backend for messages or files. This decision was pursued to facilitate a large volume of concurrent users, by not needing to worry about handling persistent storage of messages or files on the backend, which allows for scalability of the chat service.

j) Forum Service

The Forum Service is designed to foster an engaged and collaborative community of users within PeerPrep. It serves as a hub to facilitate discussions, knowledge sharing and interaction among users through the creation, browsing and management of forum posts and comments.

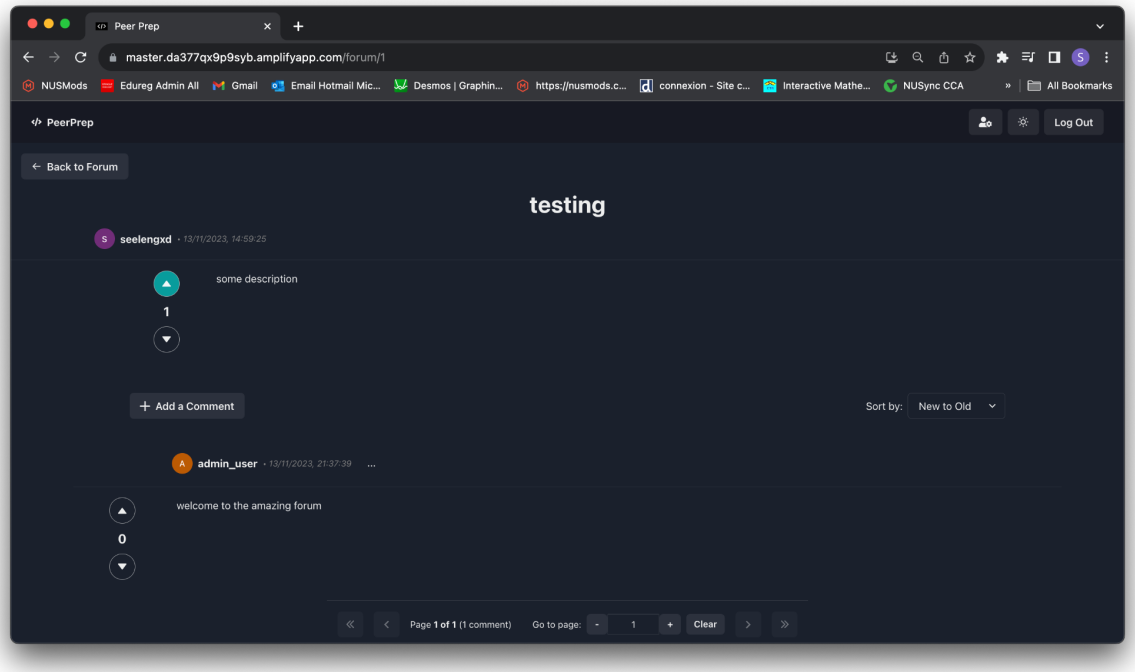


Fig . Forum UI

Key Features

1. **Post Creation and Management**: Users can initiate discussions on interview questions, tips, advice or any relevant discussions with CRUD operations (create, read, update, delete). This allows them to express their queries and thoughts as well as maintain control over their posts and comments.

2. **Comment System**: Users can engage in conversations by commenting on existing posts with CRUD operations (create, read, update, delete) to contribute to discussions and share insights. This collaborative feature fosters a dynamic and information-rich environment.

3. **User Interaction**: The service incorporates user interaction mechanisms through the use of upvoting and the "un-upvote" function. Unlike traditional downvoting, the "un-upvote" function is designed to encourage active user participation and foster constructive discussions. When users choose to "un-upvote" a post or comment, they negate their previous upvote.

By refraining from traditional downvoting, the service prioritises the importance of maintaining a respectful and non-confrontational environment, where disagreements are encouraged to be expressed through constructive comments rather than downvotes which may discourage post and comment owners from future creations. This promotes positive user engagement where quality contributions are acknowledged to maintain a respectful and constructive community.

4. **Search and Sort Functionality**: The Forum Service prioritises user-friendly navigation by providing an array of tools for efficient content exploration. Users can utilise the search feature to pinpoint specific posts and comments aligned with their interests, employing keywords for swift access to relevant discussions.

Furthermore, the service empowers users with versatile sorting options, allowing them to arrange both posts and comments based on creation dates or the number of upvotes. This flexibility ensures that users can seamlessly discover the latest content and access highly valuable posts and comments. Additionally, pagination functionality has been integrated for both posts and comments, ensuring that users can navigate through these curated results effortlessly, optimising their browsing experience by presenting a manageable subset of content per page.

Relational Database Integration (PostgreSQL)

A PostgreSQL relational database is used to ensure efficient data management, storage and retrieval.

- **Data Schema**: The PostgreSQL database is designed with a well-defined schema to store information about forum posts and comments related to each post. It ensures data consistency and integrity by enforcing relationships and constraints between tables.
- **Data Persistence**: All forum posts and comments are stored persistently in the database, allowing historical data to be retained for future reference.
- **Data Security**: PostgreSQL provides robust security features, such as access control and encryption, to safeguard the integrity and confidentiality of forum data.

k) Help Service

The Help Service microservice, embedded within our software architecture as an Express API, plays a pivotal role in enhancing user experience by delivering tailored hints to assist users in solving specific questions.

The sheer volume of questions within our database, exceeding 2.3k, posed a significant challenge for our 5-person team to manually curate hints. To address this hurdle, we leveraged the power of generative AI to automate the hint generation process. In particular, we chose Google PaLM API for its ease of use and affordable pricing (about 0.05cents per query).

The flow for how it is currently implemented, along with the UI, is depicted below.

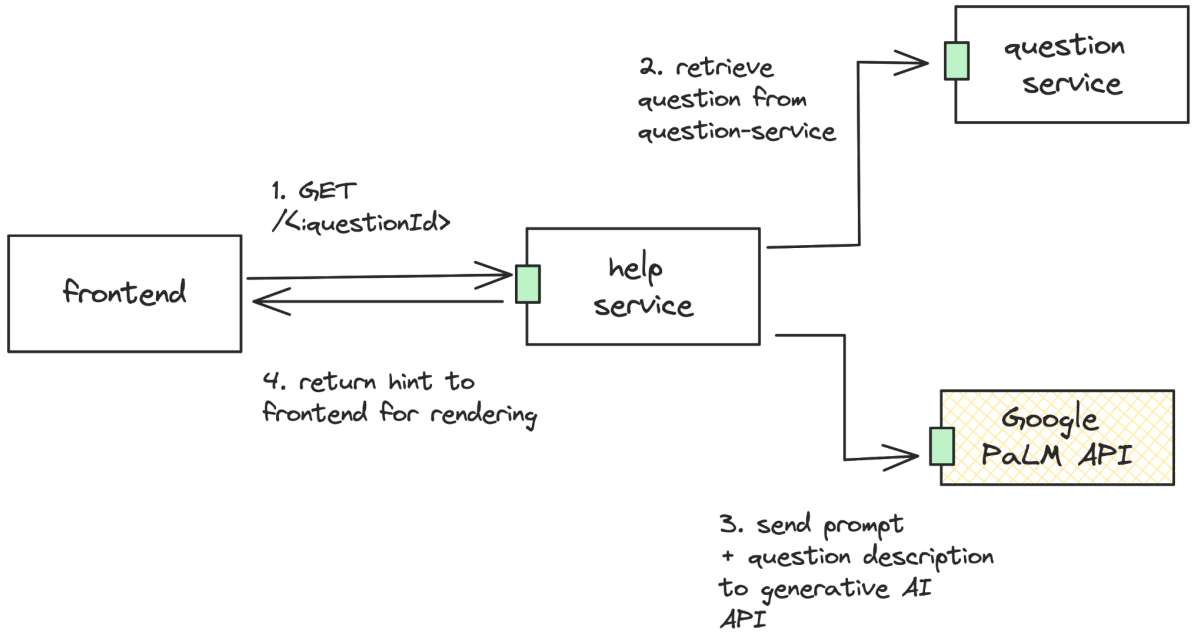


Fig. Flow for generating hints via the help service

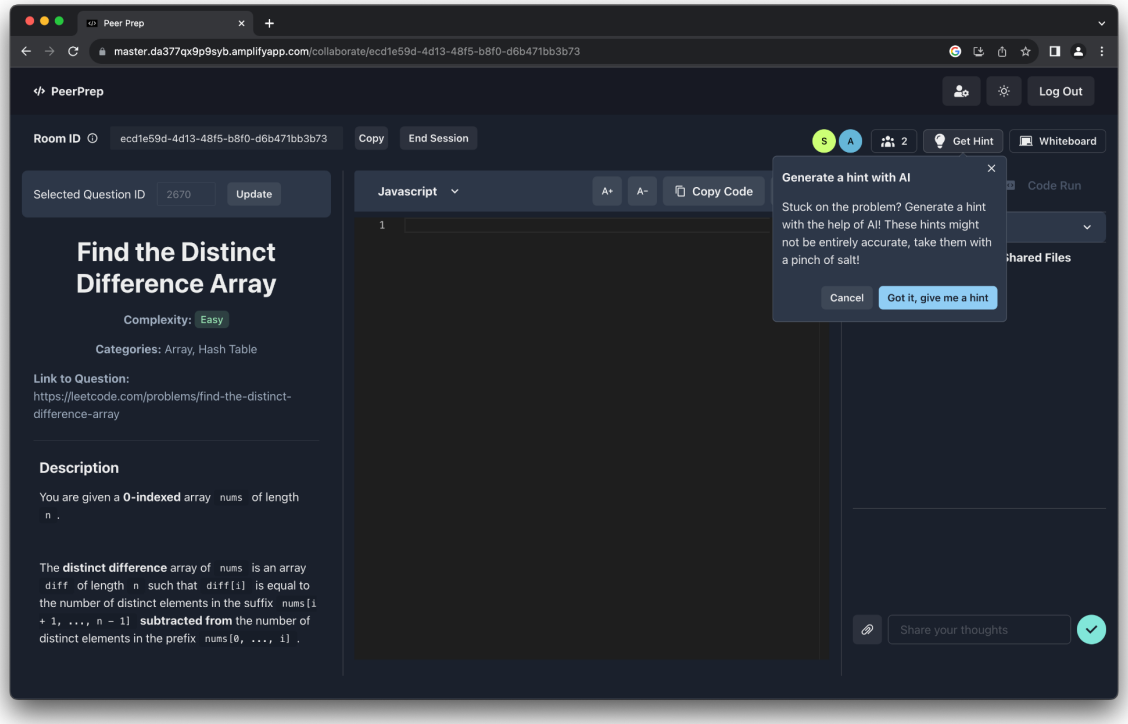


Fig. AI Hint prompt button

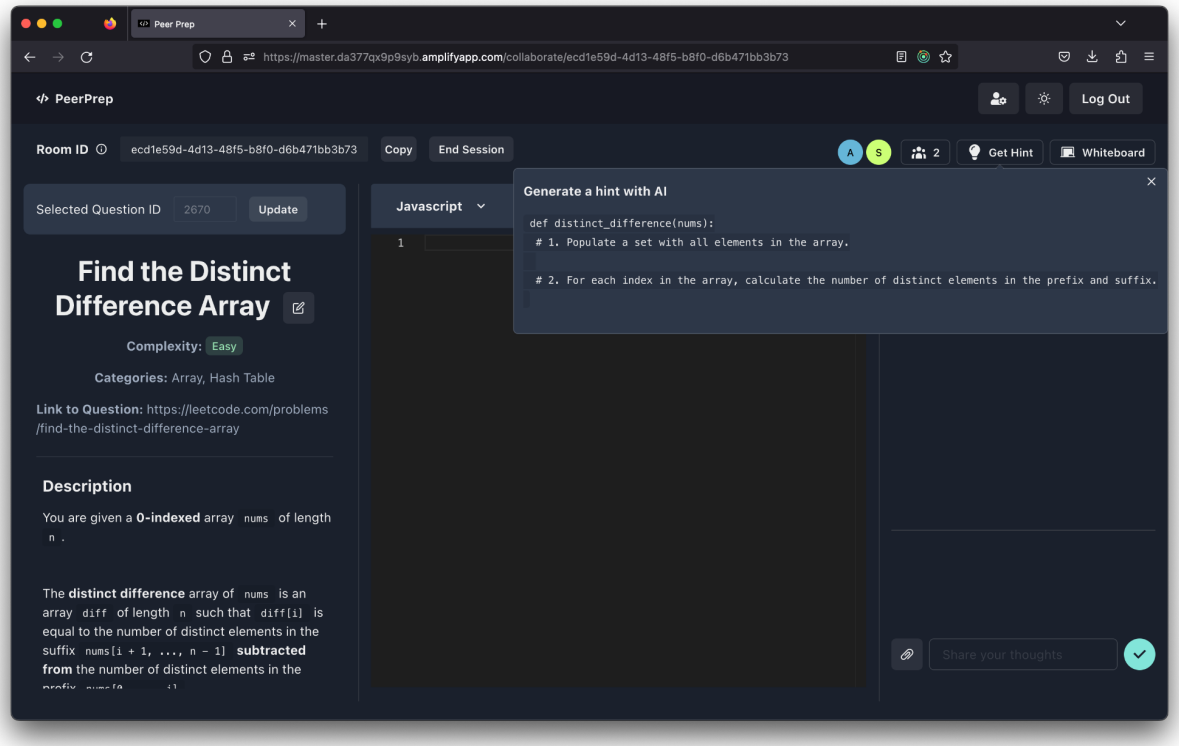


Fig. AI generated hint

7. Deployment

a) Overview

Our deployed application can be found here. We have used both AWS and GCP to do deployment. Our main considerations for what services to use are cost and ease of use. For example, AWS RDS and AWS Amplify were both chosen because they had a free offer while in the 12 month term. This table below summarises our deployment tools.

Frontend	AWS Amplify
Backend Microservices	Cloud Run
MongoDB	MongoDb Atlas
Postgres	AWS RDS
RabbitMQ	AWS EC2 t2.micro instance
Serverless Functions (scraping leetcode questions)	GCP Cloud Functions, Cloud Scheduler

b) Design choice - Cloud Run

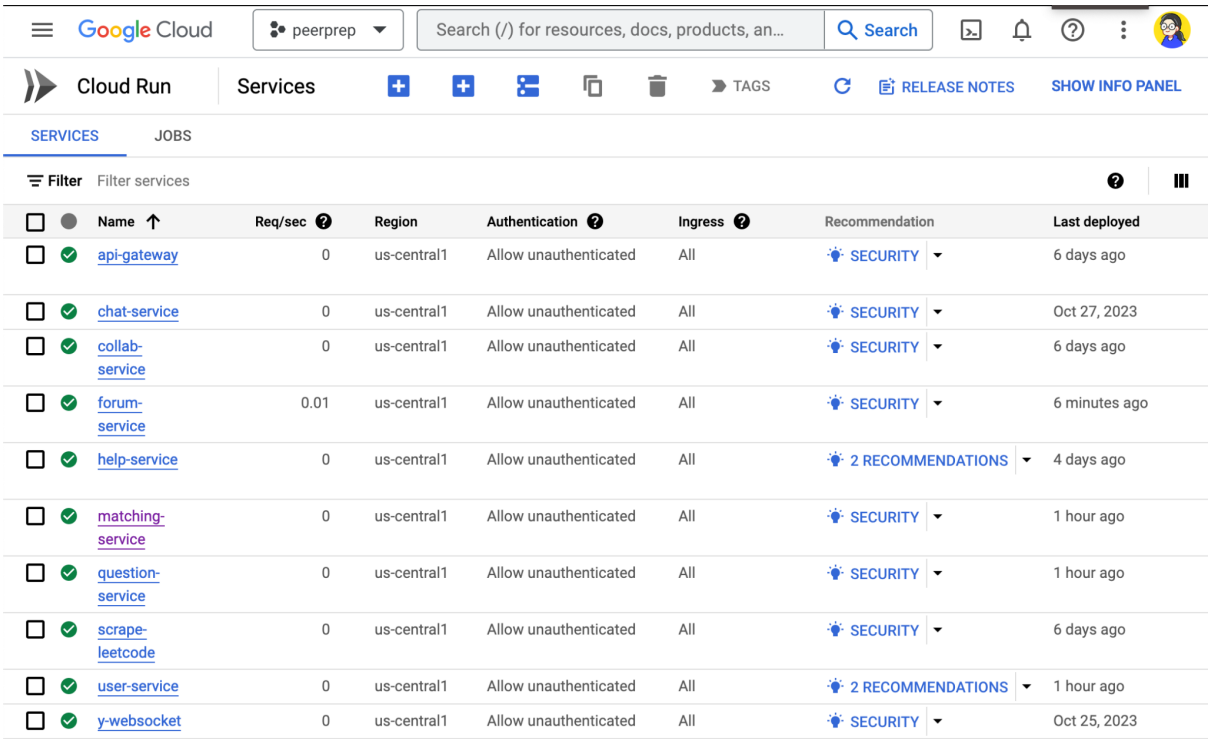


Fig. Backend Microservices (Cloud Run)

We chose to use Google Cloud Run to deploy the majority of our backend microservices for the following reasons.

1. **Cost effectiveness**
    a. Cloud Run offers "scale to zero", allowing us to have 0 instances running when there are no requests.
2. **Unique HTTPS endpoint**
    a. Cloud Run manages TLS for us, providing us a HTTPS endpoint that supports both WebSockets and HTTPS. This was particularly useful for our socket microservices such as y-websocket for the code editor, since we needed to use wss (web socket secure) or browsers would block necessary communication from happening.
3. **Simplicity of use**
    a. As long as we have a Dockerfile, we just needed to specify its file path and environment variables and deploying that microservice was basically done.

c) Automated deployment

Both backend and frontend microservices pull the latest code from the master branch for automated deployment. For example, our backend microservices each have an associated Cloud Build trigger to do so. AWS Amplify also has an auto-build function that allows us to link our GitHub repository and trigger redeployment of the frontend automatically. This automation helps developer productivity - we can spend it doing other developer work.

E. Future Enhancements

1. Integration with GitHub

Rationale: GitHub integration enhances Progress Management by allowing users to synchronise their code with personal repositories, promoting code sharing, management and storage on a widely used platform.

Implementation: GitHub API integration could be explored to develop such a feature that allows users to select repositories and enable automatic code uploading or syncing.

2. Track Correctness of Attempts

Rationale: Adding a correctness tracking system enriches the feedback provided to users, enabling them to gauge their coding accuracy and identify areas of improvement. Users can receive immediate feedback on the accuracy of their solutions, leading to more targeted learning and improvement.

Implementation: Develop an automated system that runs user-submitted code against predefined test cases based on some sample outputs required by the question to assess the correctness of users' solutions.

3. AI-Powered Analytics

Rationale: Implement AI and machine learning to deliver personalised analytics, helping users identify specific areas for improvement and offering tailored practice question suggestions.

Implementation: Create algorithms that analyse user performance and preferences to make intelligent recommendations. Continuously gather user feedback to refine the AI-powered analytics feature and enhance user satisfaction.

4. Web Responsiveness

Rationale: Ensure that PeerPrep is accessible and user-friendly on various devices, including desktops, laptops, tablets and mobile phones. Web responsiveness is crucial to provide a consistent and adaptable user experience. Users can access and use PeerPrep on their preferred device without compromising the quality of their preparation experience. This ensures convenience and flexibility in how they engage with the platform.

Implementation: Implement responsive web design techniques to adapt the layout and interface based on the screen size and orientation. Test the application thoroughly on different devices to ensure a seamless experience.

5. Gamification

Rationale: Add gamification elements to enhance user motivation, participation and overall experience using PeerPrep.

Implementation: Design a gamification system with features like badges to reward users for progress and achievements made in their interview preparation process on PeerPrep.

6. Enhanced Community Support

Rationale: Improve the forum by introducing tagging and thread organisation to enhance community support and make it easier for users to find relevant discussions.

Implementation: Implement tags to categorise the types of posts (general or based on specific topics).

F. Reflection & Learning Points

1. Communication and collaboration in a team

Effective communication and collaboration with group members were essential in shaping the project's direction. Regular updates, constructive feedback sessions and collaborative discussions played a crucial role in refining features and adhering to project deadlines. This experience highlighted the significance of clear communication in aligning project goals and fostering a cohesive and collaborative development environment within a team.

2. Technical problem-solving

Overcoming the challenge of integrating real-time matching, coding and chat service into collaborative sessions involved a deep dive into Socket technology and meticulous testing. This experience underscored the importance of problem-solving, technical exploration and manual testing in addressing complex challenges encountered during the development process.

3. Independent Self-Learning and Exploration

The project demanded extensive independent self-learning and exploration, especially in navigating a myriad of frontend and backend technologies. The project fostered a culture of continuous self-learning of different technologies like SQL and NoSQL databases, JWT tokens, Docker and RabbitMQ. This experience not only enhanced technical expertise but also reinforced the importance of proactive exploration in the ever-evolving landscape of software development.

4. Adaptive Decision-Making

Key decisions, such as defaulting users to attempt two questions in each collaborative session and creating flexible practice rooms for users who want to practise with friends, demonstrated adaptability. This experience honed our decision-making skills by tailoring solutions to accommodate different user preferences.

G. Development Methodology

1. Software Development Process

A. Agile Methodology

In this project, the software development process aligned closely with the Agile framework, specifically following the Scrum methodology. Agile emphasises iterative development, adaptability to change, and close collaboration between cross-functional teams.

Our development process consists of

- Sprints: Every Wednesday where specific tasks are undertaken by the development team
- Sprint Review: Evaluation of completed work at the end of each sprint
- Sprint Retrospective: Reflection on the sprint to identify the improvements

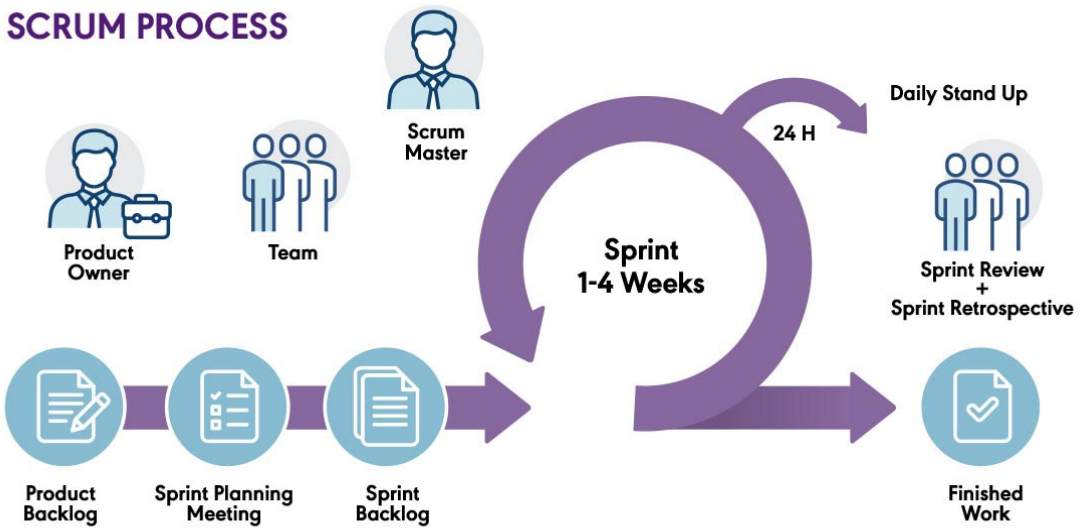


Fig . Scrum Framework

B. Iterative Development Cycles

This involves breaking down the development process into smaller and manageable iterations. The cycles are namely:

Step	Iteration Process
1	**Planning**: Identifying the functional requirements, features and tasks in a given iteration

2	**Development**: Execution and implementation of tasks
3	**Testing**: Basic form of testing was done to verify that the features were working as intended, though due to time constraints, it could have been more comprehensive
4	**Review**: Peer assessment of pull requests and completed work for feedback and improvements
5	**Deployment**: Integration of completed features into the application

This helped us establish a focus on clear project goals and user needs, early on in the project which was crucial in adhering to the tight deadlines.

C. Task Allocation and Sprint Planning

During each sprint, we broke down the project into manageable tasks and assigned them to team members based on our expertise and availability.

During sprint planning, when deciding on which tasks to work on, we considered:

1. **Backlog Grooming**: Review backlog and existing pending tasks
2. **Task Estimation**: Estimate the relative effort and workload for each task
3. **Prioritisation**: Decide which tasks are of higher importance
4. **Capacity Planning**: Consider the team constraints (ie. time, expertise) when allocating tasks

D. Continuous Integration

By leveraging GitHub Actions, we have created a structured workflow that automatically triggers on specific events, such as code pushes or pull requests. This automation framework has been configured to streamline our development process, ensuring that our React frontend can be built and deployed seamlessly. This has also been implemented for the backend question service.

GitHub Actions facilitates the deployment process, handling tasks such as environment setup, artefact deployment, and configuration management.

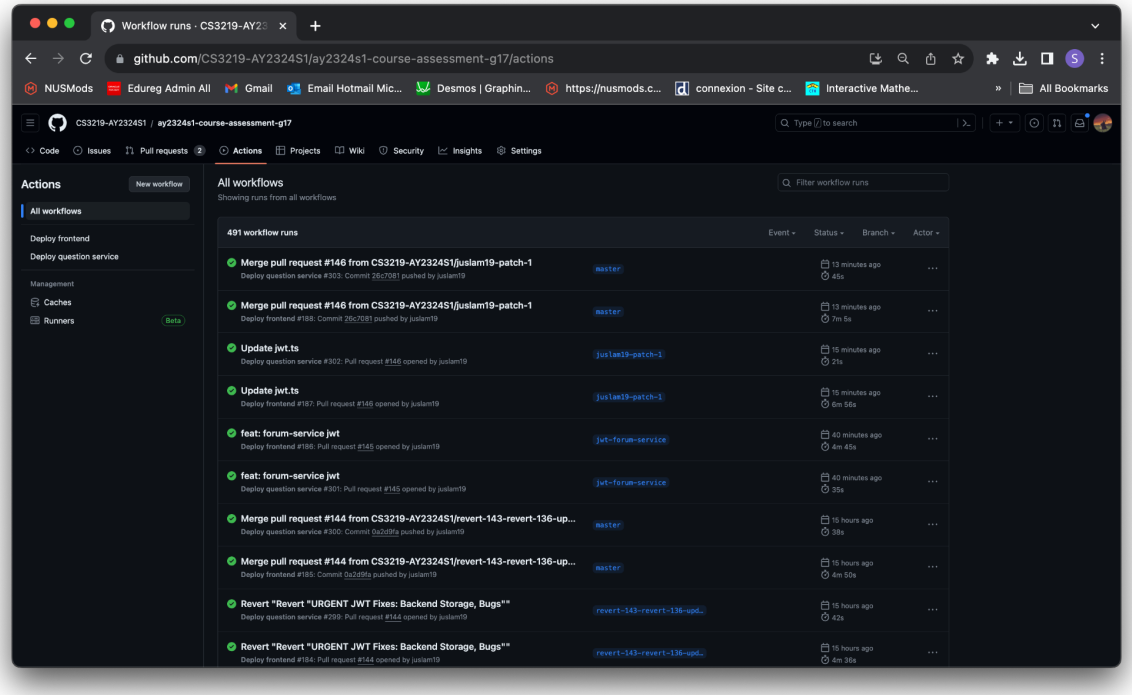


Fig . GitHub Actions workflow

2. Project Management

a. Communication and collaboration

We used Telegram for quick and concise discussions and messaging, promoting prompt communication among team members. Zoom was used for weekly meetings, utilising features like screen sharing for collaborative analysis of project and assignment requirements. During meetings, notes were taken and mind maps were drawn to facilitate in-depth discussions. Comprehensive meeting notes were documented for future reference.

b. Task management

During weekly meetings, tasks were collaboratively discussed and specific assignments were made to each team member to ensure everyone had a focused goal for the week. Task tracking and assignment details were recorded systematically in our GitHub repository under Issues, with clear assignments to respective team members.

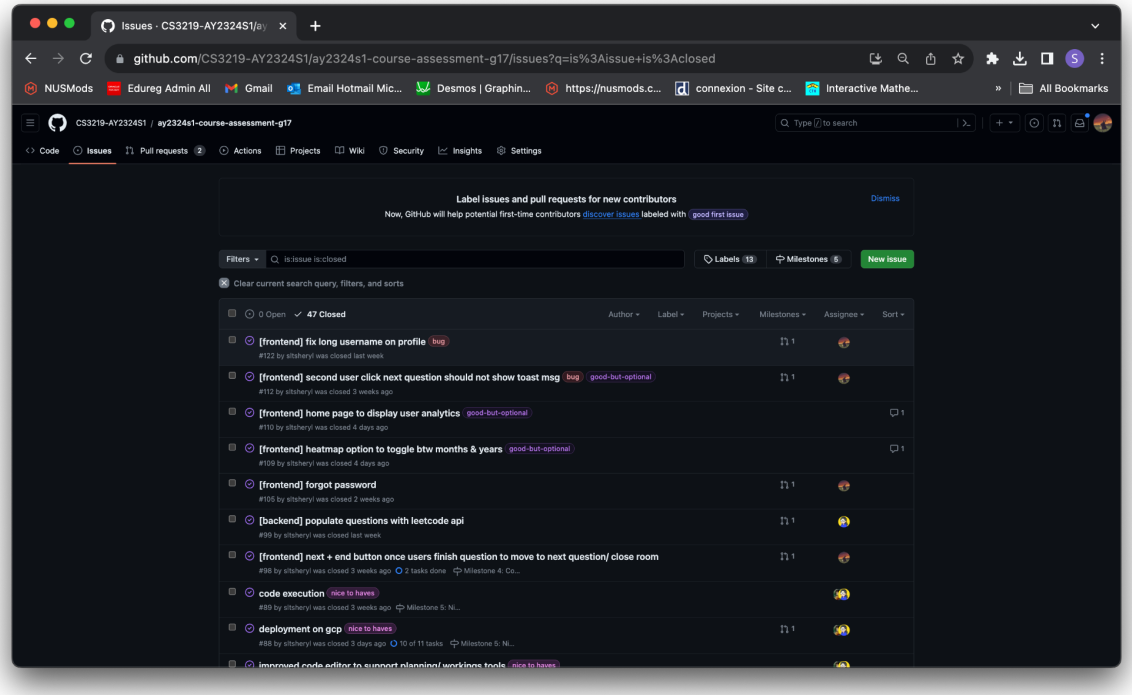


Fig . Delegation of Issues via GitHub

c. Documentation

Centralised project-related documents are stored within Notion, providing a collaborative and organised space for documentation. This served as a convenient way for members to update and share environment variables, which should not be exposed on public platforms like GitHub. Google Docs was employed for report writing, leveraging its collaborative features for seamless content creation and editing.

H. Assignment Checkpoint

1. Assignment 1

The application offers a comprehensive set of features for managing a question repository. The landing page features a well-organised data table presenting basic and essential information, including question ID, title, categories and complexity. New questions can be added through a user-friendly form, which incorporates validation checks to prevent duplicate entries or incomplete fields. Deletion of questions is also supported.

Furthermore, the application facilitates an in-depth exploration of each question, allowing users to view detailed information such as the question description on top of basic information. Users can click on the respective question and be redirected to the individual question page to view all the question details. In addition to these functionalities, the use of inline styles and the Chakra UI library enhances the user experience by presenting information in an organised and aesthetically pleasing manner. For the storage of questions, we chose to implement a NoSQL backend database at this stage using MongoDB.

2. Assignment 2

a) CRUD for question service (with MongoDB)

CRUD features were implemented for the question service. Note that the create and edit questions can only be accessed with admin permissions.

b) CRUD for user service (with PostgreSQL)

Similarly, CRUD features were implemented for the user service, namely, registering (C), viewing profiles (R), updating profiles (U), and deregistering from the platform (D).

c) RESTful API

The API endpoints are structured using RESTful principles. For instance:

- GET /questions retrieves a list of questions.
- GET /questions/:questionId retrieves a specific question by ID.
- POST /questions adds a new question.
- PUT /questions/:questionId updates an existing question.
- DELETE /questions/:questionId deletes a specific question.

The API follows a resource-based approach, where the URL endpoints correspond to the resources (questions) being manipulated. The use of HTTP methods (GET, POST, PUT,

DELETE) aligns with the intended CRUD operations. The responses are structured to provide meaningful information or error messages, following REST conventions.

3. Assignment 3

Authentication and authorisation functionality -- user authentication for accessing questions, session management (e.g., on page refresh), and authorisation management using required roles (i.e., designated maintainers, registered users, and unauthenticated and unauthorised users) -- was implemented using authentication state management with JWT.

Considerations for the use of JWT for authorisation and authentication management include its features of security, interoperability and compatibility, statelessness, and performance. Further elaboration can be found under Design - Backend - User Service - Authentication. In this application, the JWTs are signed and encrypted, ensuring the integrity and authenticity of the token. Moreover, the JWTs are implemented using a Refresh and Access Token model, which balances security and user experience. The access token is short-lived and used to authenticate the user through authentication middleware in each service, whereas the refresh token is long-lived and is used to authenticate the user to generate another access token -- it first checks the authenticity and integrity of the refresh token using the provided jwt library verify method, then it decodes the refresh token to obtain the user id, and checks the corresponding user's records to see if that token is the user's latest generated refresh token.

The generation of a new access token is done reactively. On the frontend exist several axios clients for various services APIs.

The responses from the APIs for services other than user service are intercepted, and checked to see if they have an error, an error configuration and error response. Then, they are checked to see if the error response status is 401, indicating that they are unauthorised, which might be due to an expired access token. In such a case, the user service API is then called for to generate a new access token if refresh token verification is successful. Then, the original request is tried again if refreshing the token is successful; otherwise, if the verification yields an error, the error is thrown for the rest of the frontend to handle, like using notifications.

Similarly, the responses from the API for user service is also intercepted, though in addition to checking that the error response status is 401, the request url is also checked to not be involved in token handling to avoid potential “endless loops” of unauthorised responses.

4. Assignment 4

Assignment 4 involves containerising our applications using Docker. The code for this assignment can be found in this release.

Docker is a platform for developing, shipping, and running applications in lightweight, portable containers. These containers encapsulate an application and its dependencies, ensuring consistency and reproducibility across different environments. Docker also facilitates efficient deployment by isolating applications from the underlying infrastructure, streamlining development workflows, and promoting scalability. For example, Docker was instrumental to our deployment process, since we could specify the Dockerfile path to Cloud Run and if it worked locally, there is a high chance it would already work on deployment as well.

In addition, we also used docker compose to help with local development. This was convenient as it helps us start services with one command, `docker-compose up`, instead of individually starting each of our microservices.

5. Assignment 5

Matching Algorithm

Very briefly, matching was implemented using both question difficulty and category as criteria and matching requests are stored using MongoDB. More details can be found under the matching service section. The writeup for assignment 5 can be found here.

Message Queuing

The queuing technology, RabbitMQ, was implemented to facilitate the matching of users. By using a queue, the matching process is decoupled from the user interaction. This means that the user does not have to wait for the matching to occur before continuing to use the application. The matching process can happen in the background.

An added advantage is RabbitMQ supports multiple consumers, which allows for scalability. This means that one can have multiple instances of the matching service running concurrently, each consuming from the same queue. This can help distribute the load and handle a larger number of matching requests.

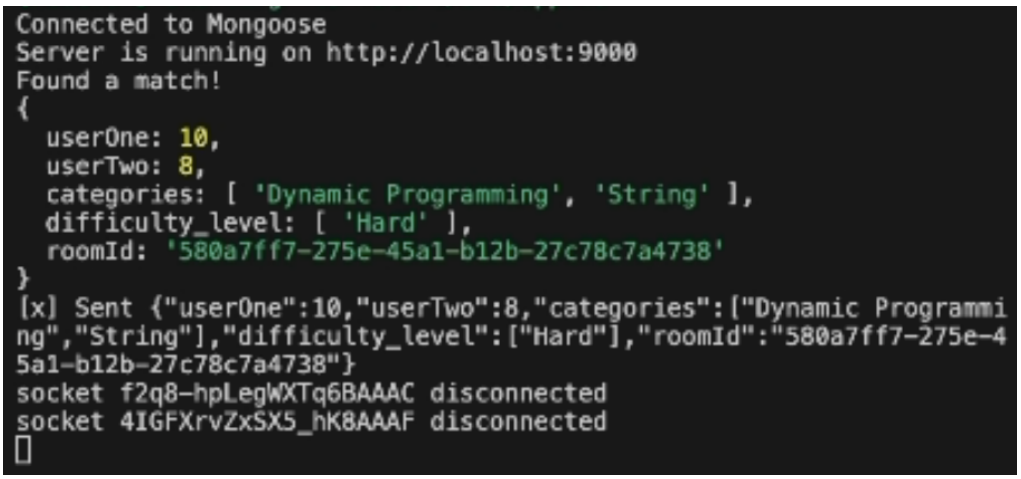


Fig . A match is sent to the queue in the matching service

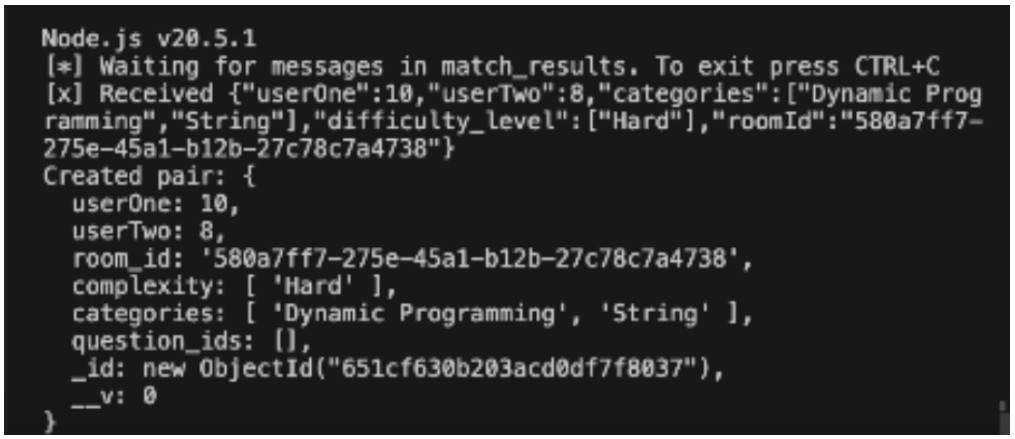


Fig . A match is received and consumed in the collaboration service

RabbitMQ dashboard was used to analyse the network requests and data for debugging purposes.

Fig . RabbitMQ dashboard interface

Frontend Timer and Matching

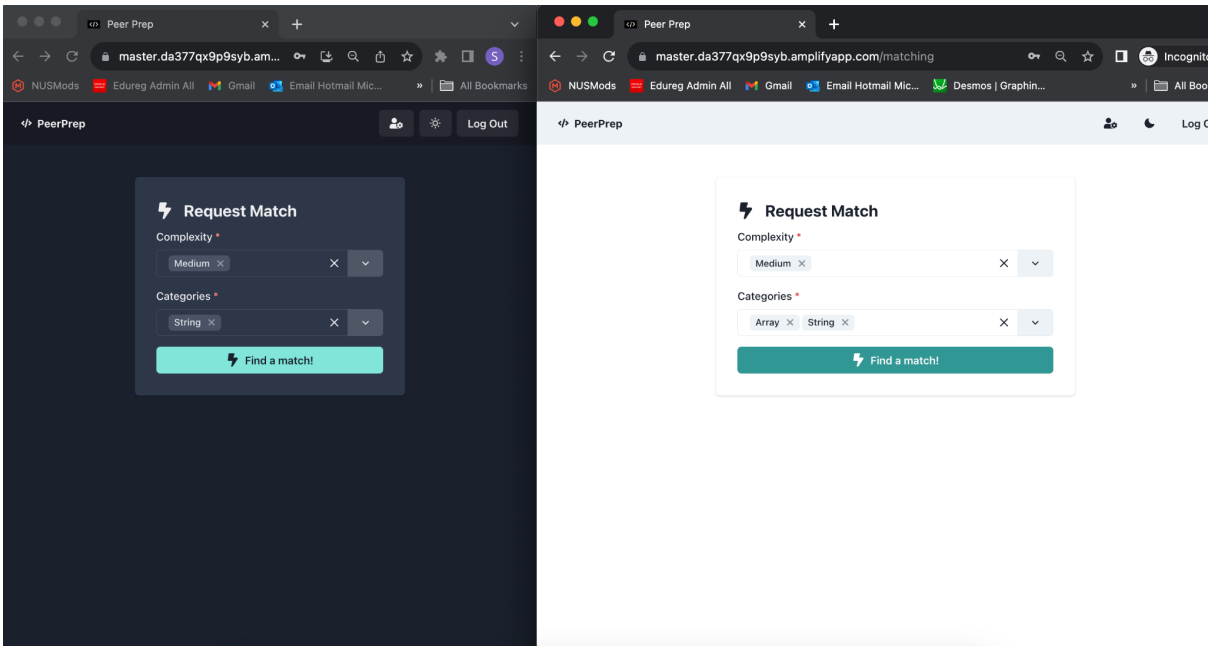


Fig . “Matching form” page: Form for user to specify the fields for their matching request

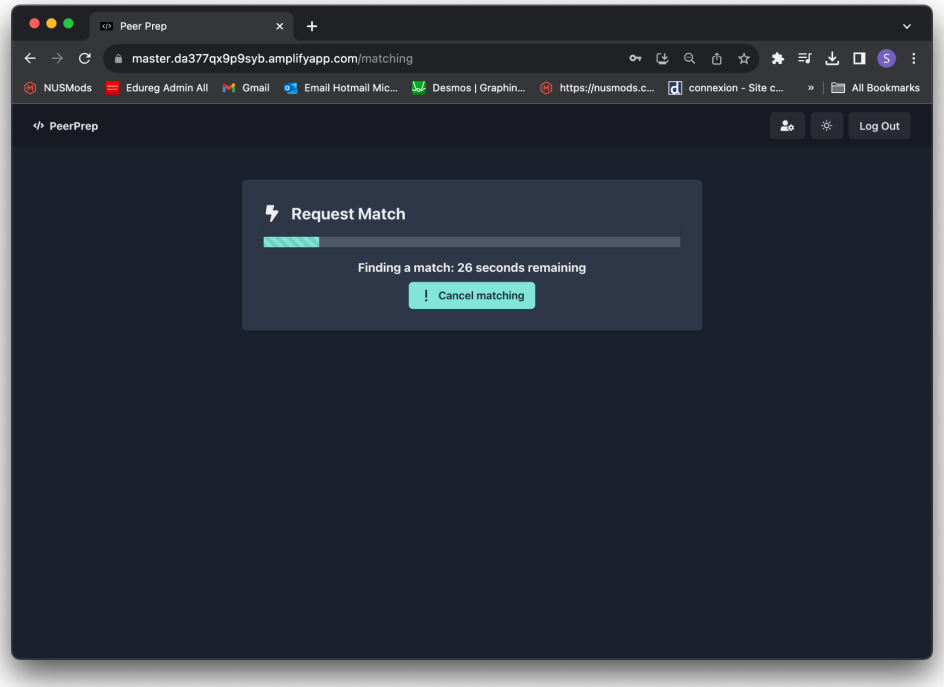


Fig . “Timer” page: timer displaying remaining time before user’s matching request timeout, with cancel matching option - which removes user’s request from matching service

The state management of the pages, “matching form” and “timer”, is handled on the frontend. Initially, the frontend will display the “matching form” for the user to input the fields used in their matching request. Upon submission of the form, the frontend will change the page to the “timer” page. The “timer” page handles the countdown on the frontend, with a progress bar which increments every second for the remaining time.

If no match is found, the state management is informed of the timeout from the backend -- which means the user is disconnected from the matching service, and their request is marked as timed-out and removed -- which changes the frontend state management of the page back to the “matching form”, along with an notification that the matching was unsuccessful, ensuring that the frontend is reflective of the backend matching process, though the timer countdown is handled entirely on the frontend.

The cancel matching button provides the additional option for the user to remove their matching request immediately, through the same process as a timeout happening on their matching request -- by disconnecting from the matching service and having their request marked as timed-out and removed.

6. Assignment 6

A serverless function, also referred to as function-as-a-service (FaaS), allows us to execute code in a cloud environment without the hassle of managing server infrastructure. In Assignment 6, we write one to scrape leetcode to populate our questions database.

We have a python function (found here) that scrapes leetcode for the questions found. Leetcode uses a GraphQL API and by using developer tools, we can mimic the queries they use to populate our database. (Interestingly, we can bypass their CSRF by using GET requests instead of POST requests.)

We have hosted our serverless function on Google Cloud Functions, and we use the Cloud Scheduler to run it daily at 12:00AM.

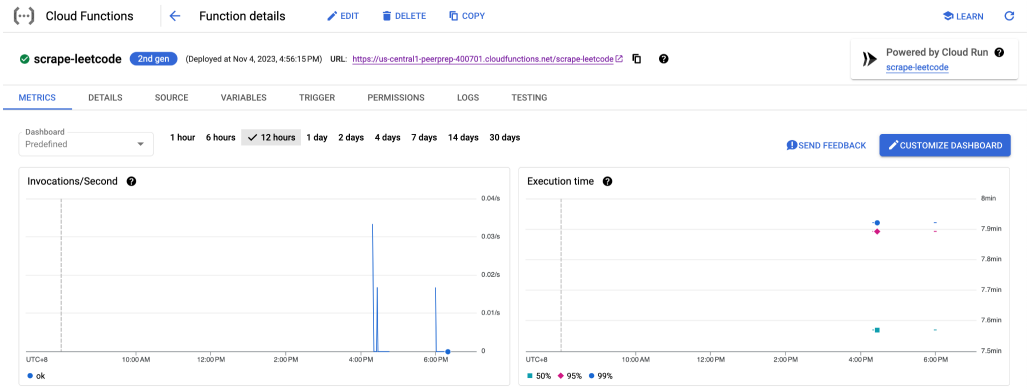


Fig. Serverless function - Google Cloud Functions

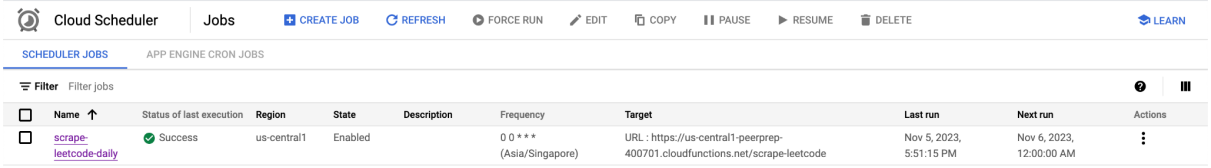


Fig. Cloud Scheduler scheduling function to run daily

I. Individual Contributions

Member	Contributions
Chloe Lim Xinying	**Implementation:** - Set up frontend project, project structure and linters - Question service > question index (get all questions): frontend [question table with filtering and sorting] and API - Question service > individual question API endpoint - Question Service > Add code editor (with syntax highlighting, auto-indent etc + custom features such as copying code, export/ import) (nice-to-have) - Collaboration Service > Create practice rooms (collaboration rooms without matching) with support for real-time code editor and user awareness (synced editors) [frontend + backend] - Collaboration Service > Add code executor (nice-to-have) - Collaboration Service > Add interactive whiteboard + show user awareness for whiteboard (nice-to-have) - Collaboration Service > Refactor collaboration room code (practice room + matching room) - User Service > Add GitHub OAuth login method (frontend + backend) **Report:** - Requirements > Non-Functional Requirements & Non-Functional Requirement Fulfilment > NFR 2-7 - Design > Tech Stack - Frontend > Overall - Frontend > Websockets - Frontend > Code Editor - Frontend > Code Execution - Frontend > Collaborative Whiteboard - Backend > MVC - Backend > Question Service - Backend > User Service > GitHub OAuth - Backend > Collaboration Service > Websocket

Justin Lam Seng Onn	**Implementation:**
	- Question service > edit question: frontend and APIs; prefilled with initial question data; allows updates for question fields (one has dropdown and autocomplete).
	- User service, and other services > authentication and authorisation functionality with different roles using JWT.
	  - Utilises refresh and access token architecture, which includes backend storage for refresh tokens
	  - Utilises authentication middleware in other services
	- Modified backend model for user service (PostgreSQL and Prisma) to accommodate JWT refresh token
	- Matching Service > frontend
	- Chat Service > text and file messaging: frontend and APIs
	- Added docker to chat service
	- Forum service > post comments details: frontend CRUD for comments, upvote/downvote, search, sort, pagination

	Report: - Design > Backend > User Service > Authentication - Design > Backend > Chat Service - Assignment Checkpoint > Assignment 3 - Assignment Checkpoint > Assignment 5
Leong See Leng	**Implementation:** - Question service > create question: frontend and APIs. - Question service > initial seed data - User service > implement login, signup, logout - Matching Service > initialise and set up socket.io - Matching Service > implement matching logic - Help Service > set up backend - Collaboration Service > fetch first question - Added docker to matching, collaboration, chat, help services - Deployment (GCP/AWS) - Implemented serverless function to scrape leetcode and populate questions (Assignment 6) **Report:** - Architecture - Requirements > Non-Functional Requirement Fulfilment > NFR1 - Design > Backend > API Gateway - Design > Backend > Matching Service (matching via DB) - Design > Backend > Help Service - Design > Backend > Publisher-Subscriber pattern - Design > Deployment - Assignment Checkpoint > Assignment 4 - Assignment Checkpoint > Assignment 6
Sheryl-Lynn Tan	**Implementation:** - Set up backend models for question service (MongoDB) - Question service > delete question: backend method - Set up backend models for user service (PostgreSQL and Prisma) - Frontend > Rich text editor - Reset password using nodemailer (frontend forms and backend) - Frontend > Refactor send match details form - Set up RabbitMQ and implement matching queue and consumer - Add GitHub Actions (frontend and question service) - Add docker to question service - Progress management (Frontend: heatmap and user analytics, Backend: restructure user models and add answered question model) - Set up backend forum service + styling to forum frontend - Implement next question and finish session for an interview session (while saving attempted question) - Implement authentication for room (so outsiders can’t enter via roomid) **Report:** - Background, About PeerPrep

	- Outline of Requirements (Group) - Outline of Features (Group) - Design > ER Diagram - Design > Database Design - Design > Backend > User Service (progress management) - Design > Backend > User Service (reset password) - Design > Backend > Collaboration Service (queue messaging protocol) - Development Methodology > Software Development Process - Assignment Checkpoint > Assignment 2 - Assignment Checkpoint > Assignment 5 > Queue system
Zheng Jiarui	**Implementation:** - Question service > delete question: frontend and APIs. - Question service > individual question details on separate pages. - User service > view profile - User service > update profile: prefilled with initial user data; allows updates for username, email, languages (with dropdown and autocomplete). - Collaboration service > set up socket.io for real-time updates. - Forum service > Forum page + individual post details: CRUD for posts, upvote/downvote, search, sort, pagination **Report:** - Outline of Requirements (Group) - Outline of Features (Group) - Design > User Activity Diagram - Design > Backend > REST API - Design > Backend > Forum Service - Future Enhancements - Reflection & Learning Points - Development Methodology > Project Management - Assignment Checkpoint > Assignment 1