School of Computing

CS3219 Software Engineering Principles and Patterns

AY21/22 Semester 1

Project Report

Team 24

Team Members	Student No.	Email
Joshua Liang Xingya	A0199761L	e0406742@u.nus.edu
Rishabh Paliwal	A0199537J	e0406518@u.nus.edu
Roger Lim Yong Siang	A0199776Y	e0406757@u.nus.edu
Ler Yong Yao, Jazer	A0201866E	e0415675@u.nus.edu

Mentor: Sahil S/O Sanjeev Gathani

Background

As budding software engineers applying for internships, we were faced with an onslaught of brutal technical interviews that challenged our knowledge of data structures and algorithms. Realizing that we were unable to perform to the best of our abilities in showcasing our expertise in data structures and algorithms during these interviews, we came to the conclusion that this was due to our lack of practice in communicating ideas in the high stress environment of a technical interview.

As we searched for a solution to this, we realised that although there were many platforms catered to providing challenging programming questions, they failed to emulate the real world stress of having an experienced engineer on the other side scrutinizing your implementation and design decisions during an interview. They were also not designed in a way that would allow another user to see or modify the programme that was being written in real-time.

Hence, PeerPrep was born as a solution to this problem. PeerPrep is a web application that combines the challenge of technical questions with the communication in designing solutions in real-time, emulating the interview process as a single unified application that can be used by pairs of like-minded students trying to better prepare themselves for their technical interviews.

Purpose

The purpose of PeerPrep is to allow students like us to better prepare for whiteboard style technical interviews in an environment that closely resembles that of real life technical interviews. It contains a question bank of some of the most popular questions that are being asked today to ensure that the questions users will attempt on the platform will be useful for their preparation. PeerPrep would also allow for real time communication between the users through the use of a shared coding environment allowing them to critique and ask questions regarding the implementation in real time.

1 Requirements

1.1 Product Scope

Target user profile

- Is preparing for technical interviews
- Has basic computer literacy

Value proposition

The ability to participate in pair-programming for technical interview questions

1.2 Functional Requirements

1.2.1 User Stories

Priority	User Story
High	As a user, I want to work on interview questions with others at the same time, so that I can learn how to tackle these problems from others
High	As a user, I want to have a shared workspace, so that I can tackle technical interview problems with a partner
High	As a user, I want to be able to choose the difficulty of questions to tackle, so that I can choose an appropriate difficulty for myself
Medium	As a user searching for a match, I want to be notified if I am not able to find a match due to a lack of other users, so that I do not have to waste my time waiting
Low	As a user, I want to be able to see my code from questions that I have attempted in the past, so that I can review my solutions

1.2.2 Use Cases

Use case 1: Logging in

MSS

1. User presses login button (for OAuth service)
2. Frontend redirects to OAuth page
3. User signs in using the OAuth service
4. Backend sends back JWT
5. Frontend stores JWT as a cookie
6. Frontend redirects back to homepage

Use case 2: Finding match

MSS

1. User selects difficulty of question and find match button
2. Frontend displays a loading symbol on find match button, while searching for match
3. User selects difficulty (Easy/Medium/Hard) which is emitted to the server side as an event
4. When another User that is also searching for the same difficulty is found, the server sends the room ID
5. Frontend redirects both Users to a room for pair-programming

Extensions

- 1a. User is unauthenticated
  - 1a1. Find match button is replaced with login button
- 4a. No other user found
  - 4a1. Frontend reenables find match button to search for next match

Use case 3: Pair Programming in Room

MSS

1. User is directed to the Room page after being matched.
2. Frontend displays the question title and description.
3. Frontend provides a code editor.
4. When either user in the room modifies text in the code editor, the same change should also be displayed on the other user's code editor.

Extensions

- 1a. User is unauthenticated
  - 1a1. Find match button is replaced with login button
- 1b. User is not authorised to join the room
  - 1b1. User is unable to see the code editor or the question details of the room.

Use case 4: Submitting answers

MSS

1. One of the users in the room presses submit button
2. Frontend emits to backend to start the end session process
3. Backend emits back to both users to end the session
4. Both users sends the shared answer to the backend, as well as the roomid
5. Backend fetches question details based on the roomid and stores the answer into the database
6. Frontend redirects user back to homepage

Extensions

- 1a. User is unauthenticated
  - 1a1. Frontend redirects user to homepage, prompting them to log in

Use case 5: Viewing profile

MSS

1. User clicks on their name on the header
2. Frontend redirects user to their profile page
3. User's past answers is fetched from the backend
4. Frontend displays answers in an accordion list
5. User clicks on a question title in the list
6. Frontend expands the corresponding accordion to reveal the answers

Extensions

1a. User is unauthenticated

    1a1. Frontend redirects user to unauthorized page, prompting them to log in

1.3 Non-Functional Requirements

1.3.1 Browser Compatibility Requirement

- PeerPrep should be accessible on
    - Internet Explorer 10 and up
    - Microsoft Edge
    - Firefox 21 and up
    - Chrome 23 and up
    - Safari 6 and up
    - Opera 15 and up

1.3.2 Performance Requirement

- The shared coding environment should sync user updates within 1 second
- If available, users should not take more than 30 seconds to match with each other
- Server should be able to handle at least a pair of users on the platform concurrently

1.3.3 Security Requirement

- Users should only be able to remain logged in for at least an hour
- Sensitive information from the account used for OAuth authentication should not be stored

2 Design

2.1 Architecture

For PeerPrep, we are making use of a layered architecture for our backend. We felt that the layered architecture approach would be appropriate, allowing us to easily achieve Separation of Concerns by clearly partitioning out the logic of the API endpoints in the controllers, the business logic in services, and the database logic in repositories. The simplicity in implementing a layered design also played a large part in our decision in choosing this architecture. However there are also certain drawbacks of using a layered architecture, this is that data can only flow in one direction

Since the main problem to solve in this project is to implement a collaborative text editor for practicing for interviews, we felt that the scope of the project is small. Thus, we chose to implement this as a monolithic architecture, as we felt that using something like a microservices architecture would be too complicated for the limited scope of this project. Going with a monolithic architecture also allowed for easier and faster development which was an important consideration given the limited time we had to work on the project.

The following figure displays the architecture of our system. The direction of the arrows indicate the dependencies of each module in the system. From this diagram, it is easy to see how layered architecture helped us in reducing coupling across the system, since each layer only depends on layers below it.

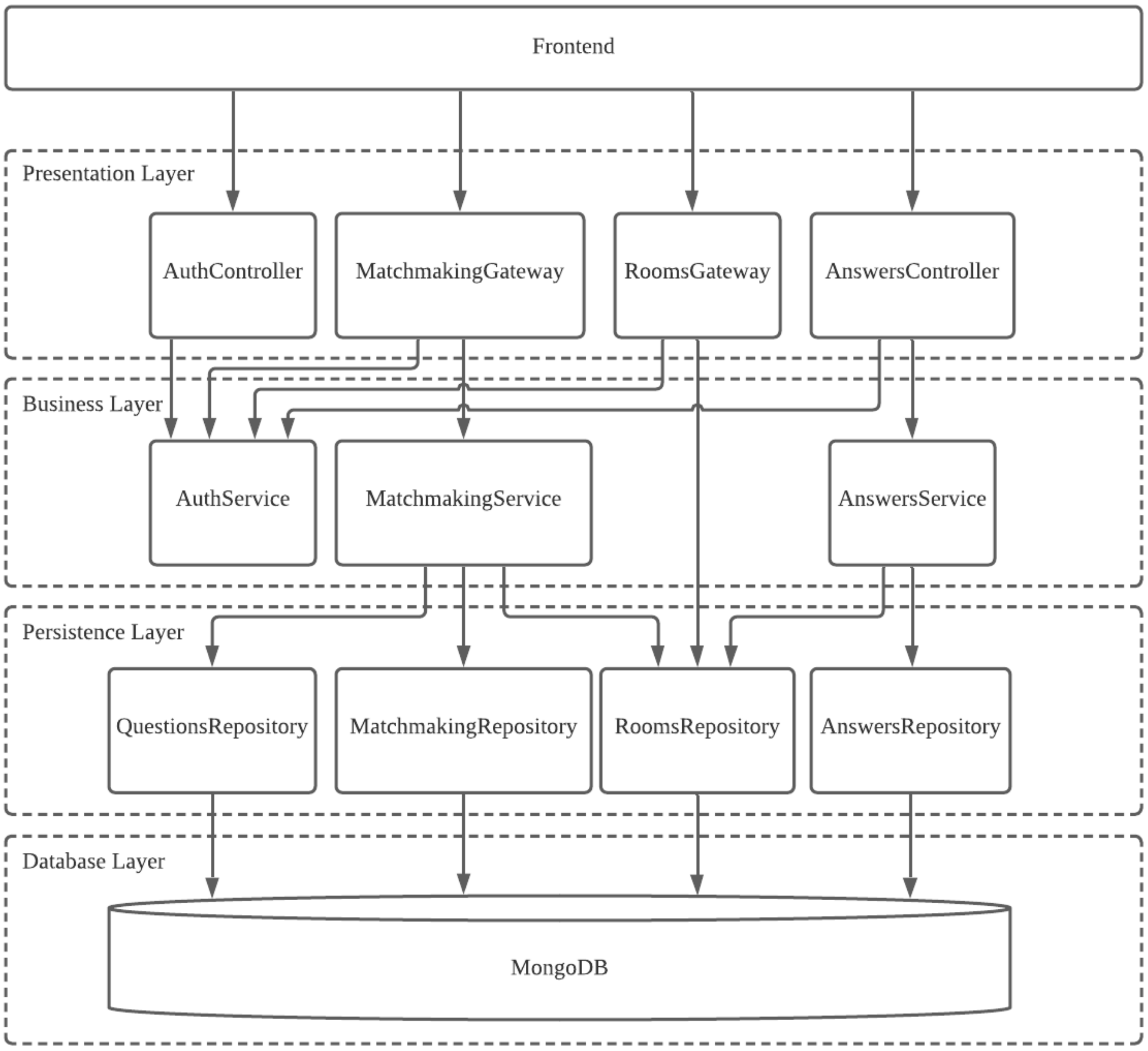


Figure 1: Architecture diagram of PeerPrep

2.2 Tech Stack

	Technology Used
Frontend	React
Backend	NestJS
Database	MongoDB
Pub-Sub Messaging	Socket.io
CI	GitHub Actions
Deployment	Heroku
Project Management	GitHub Issues

2.3 Implementation

2.3.1 Authentication

For authentication, we had to make a decision whether to implement authentication.

Not implementing authentication would mean that we will not be able to save any information about the user, such as the questions they've attempted, or the answer they came up with their partner while attempting these questions.

If we do implement authentication, we would then need to choose between implementing our own authentication, or making use of OAuth.

Implementing our own authentication

Pros:

- We have more control over what information about the user we want to store, since we would be implement the registration service for our authentication

Cons:

- Requires more effort to implement
- If implemented badly, we are putting the information of our users at risk

Using OAuth

Pros:

- Easy to implement
- No risk of bad implementation

Cons:

- Little control of user information coming from the OAuth service

Our team ultimately decided to make use of OAuth, since we valued the ease of implementation. We also felt that we did not need much user information for PeerPrep, so having more control of user information via implementing a registration service was not needed.

In addition to that, since most users have accounts on popular OAuth services, allowing users to skip the registration step translates to higher usability for users. In addition to that, since we are making use of JWT rather than storing the users in the database, this means that scalability is not an issue, no matter how many users want to make use of our product.

For our implementation, we will make use of a library called passport. This allows us to add a middleware to redirect our requests to Google's OAuth service.

1. The incoming request first goes through the passport middleware, which redirects to Google's OAuth service, which will then make use of our callback to the validate method in GoogleOAuthStrategy, as we show below in Fig 2.
2. The validate method will return us a user object, which will be injected into the request as a field.
3. The request is then passed on to the login method in AuthController
4. Next, we use JWTService to sign the user object to generate the JWT
5. Finally, we send the resulting JWT and user object back

Relevant API:

Endpoint	`GET /login`
Headers	–
Body	–
Description	Redirects to the OAuth Service

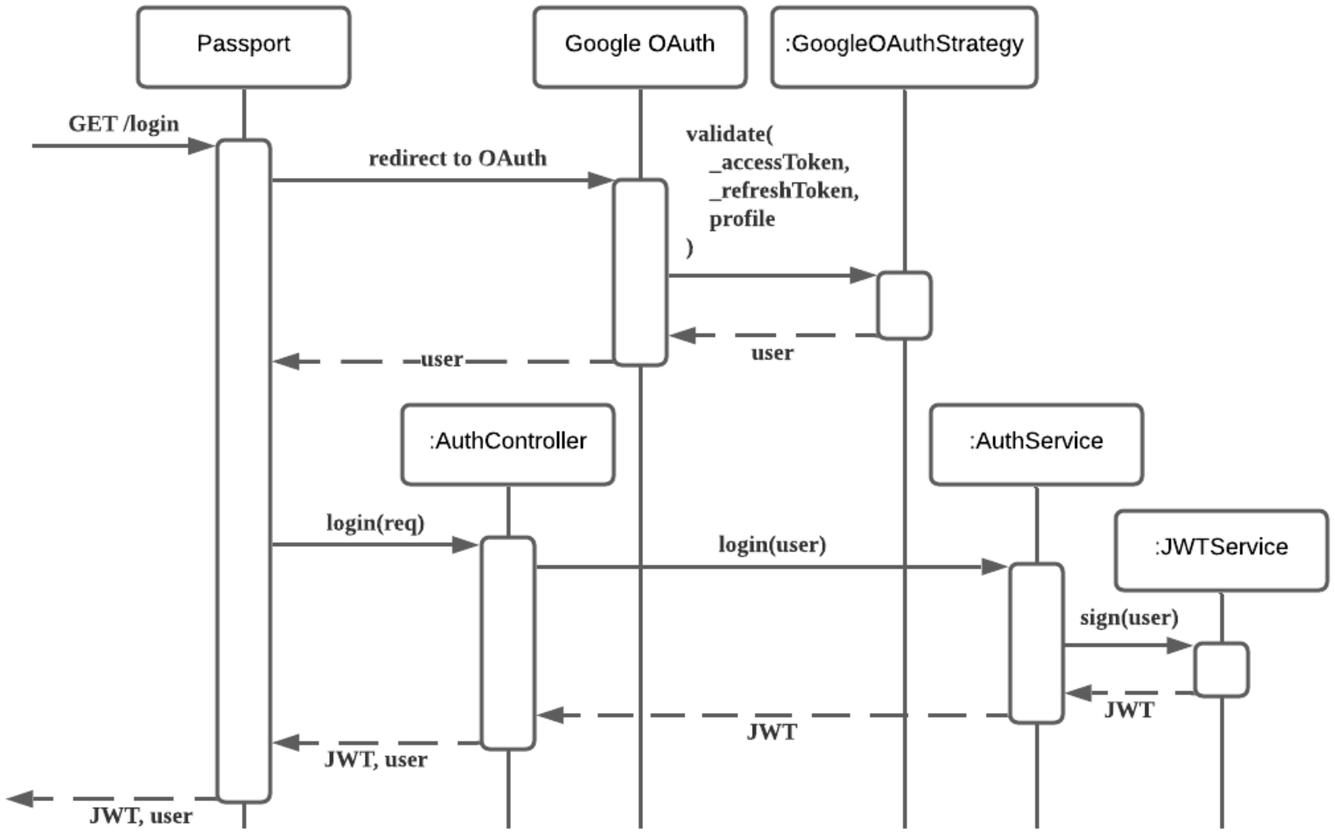


Figure 2: Sequence diagram for login in backend

2.3.2 Matchmaking

With matchmaking, we decided to go with MongoDB, implementing matchmaking via a simple check on the database. When the user tries to find a match, the user connects to a matchmaking gateway under the namespace `matchmaking`, emitting an event via the socket to signal to the server to attempt to find a match. This searches a MongoDB backend for a Match document that contains another user's ID, difficulty chosen, and Socket ID, and removes that from the collection. If no such document can be found, then the current user will have their details placed into another such document for the next user to match with. This also triggers an `noMatch` event to be emitted to the user, where the user's client will wait for the timeout period before it prompts the user if they want to requeue. On the backend, the match documents automatically expire after 1 minute, which prevents users from being matched with users who have already given up on the queue. Furthermore, match documents are also deleted if the associated user disconnects from the gateway.

Once a match has been found (triggered by either user's request), the gateway will first create a room by fetching a question from the Questions Repository, writing into the Rooms repository, using the document ID as a room ID. This room ID is then forwarded to both users via the `assignRoom` event, which is emitted over the Socket.io connection to be handled by the clients. This process is illustrated in Figure 3.

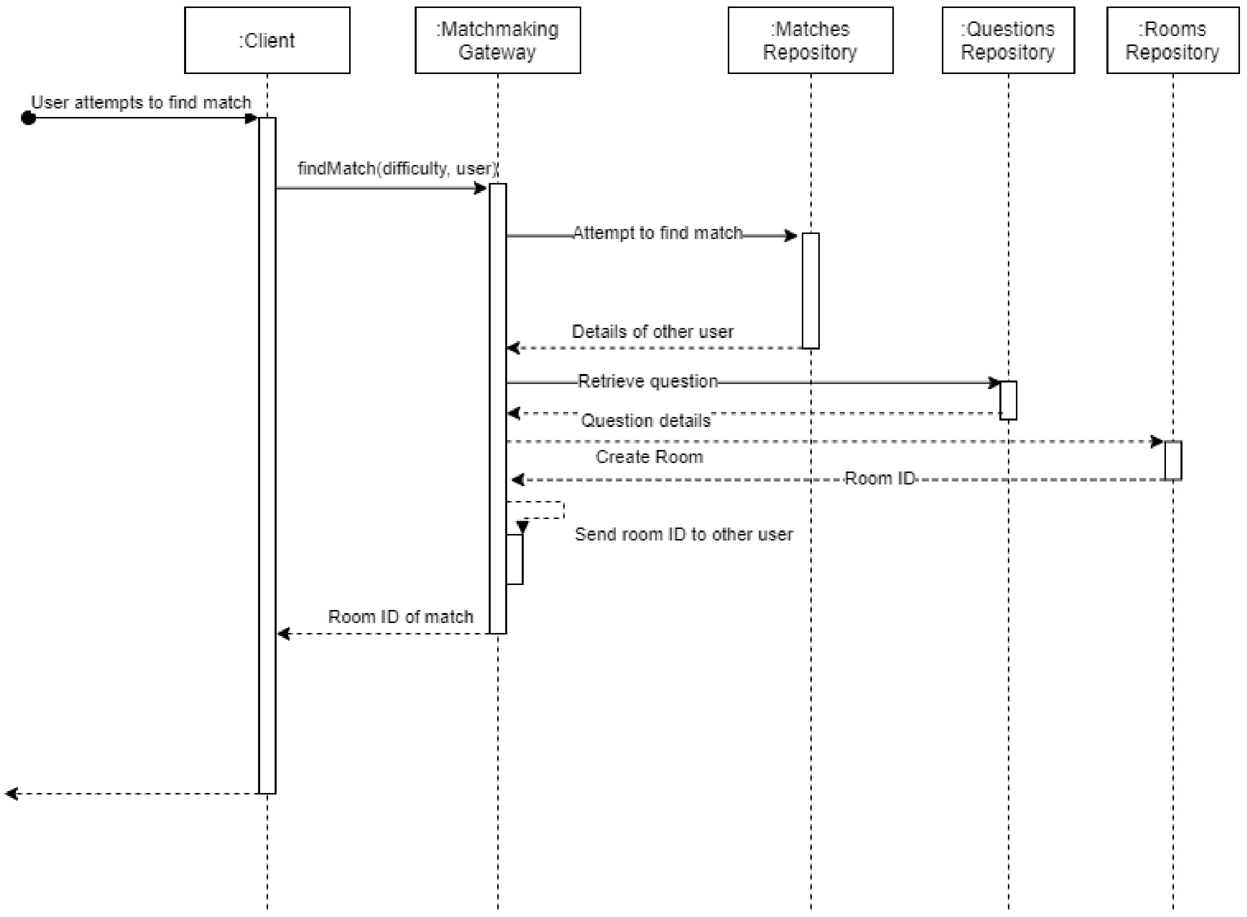


Figure 3: Matchmaking process

Events that are handled on the server side:

Event	Payload	Description
findMatch	difficulty: string auth: JWT	Difficulty must be ‘easy’, ‘medium’, ‘hard’. Depending on whether a match is found, the server will emit different events, which are shown in the table of client events handled.

Events handled on the client side:

Event	Payload	Description
assignRoom	roomId: string	Room ID of the room assigned to the current user.
noMatch	-	Emitted if no match could be found. When this event is sent, the client's information has been placed into the queue. If a match is found after this is received, another assignRoom event will be emitted to the client.

With matchmaking, we decided to use MongoDB to store our matches as the implementation was a lot more straightforward within a monolithic architecture. Although we could have made use of libraries like Redis or other forms of caching to improve the performance of the matchmaking, the linear structure of a Redis cache means that removing a user from the queue (as some list of users) on disconnection would be non-trivial. With MongoDB, we can make use of operations supported by the database to query for any existing match in the database, as well as to delete the match document from the database if the user disconnected or was put into a match.

Handling the disconnection was a key factor in deciding to implement matchmaking via sockets, as the typical REST APIs would not be able to provide this information to the server. While some lifecycle components from the frontend could provide some of this information, it would not be a strong guarantee that the user was still available and ready to match with another user. For example, in the event of the internet connection disconnecting (power outage, weak signal, etc.), the server would be completely unaware of this without the use of sockets maintaining a persistent connection. In such a scenario, it would then be possible to match with a user that cannot participate in the room, which would be a major issue in terms of usability.

2.3.3 Synchronisation

Once users have received the room ID of their match, the client will connect to the Rooms Gateway, where they will be joined into a room of the given room ID. On the server side, checks are carried out to ensure that the users belong in the room they are trying to access based on the JWT stored by the client.

Once users are connected to each other in a room, code edits on the client side are emitted to the server and forwarded by the server to the other user. CodeMirror provides us with details about each update, which we encapsulate in a Changes object that contains the positions and text modified by the user. The schema of this Changes object is as follows:

```
Changes {
    from: Position;
    to: Position;
    text: string[];
    removed: string[];
    origin: string;
}
Position {
    ch: number;
    line: number;
}
```

In this schema, the `Position` contains both the line number and the character position in this line. This is used to store the positions of the range of text changed, which are categorised by type in the `origin` field.

This object is immediately sent over the socket connection to the server, which is forwarded to the other user to be applied to their editor.

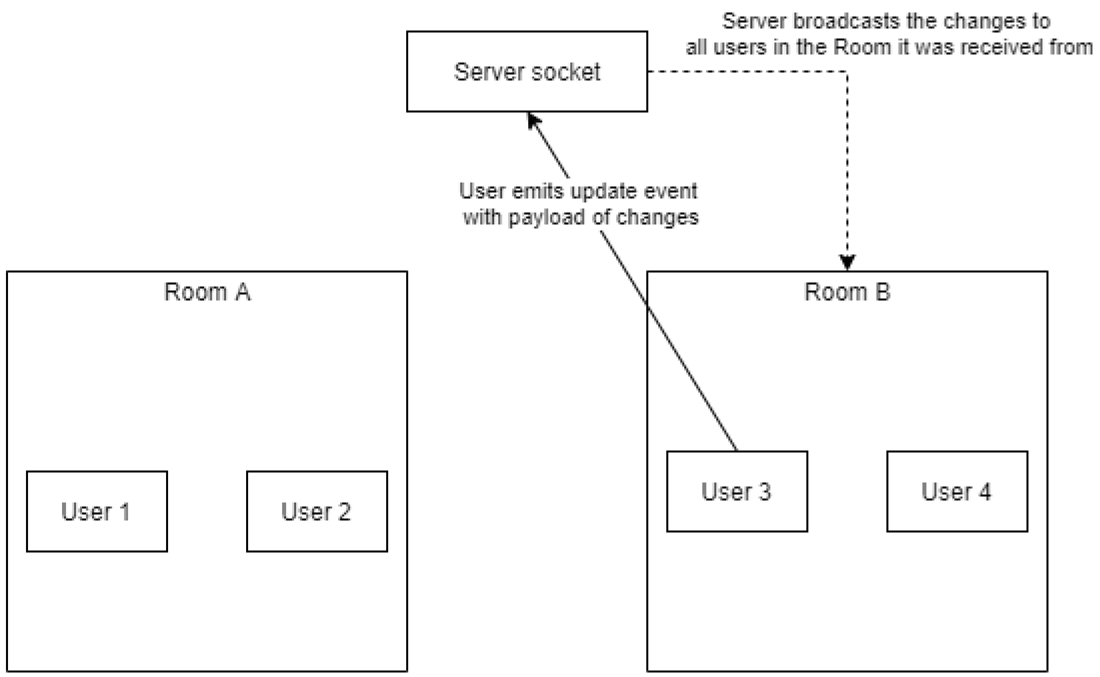

Server socket
Server broadcasts the changes to all users in the Room it was received from
User emits update event with payload of changes
Room A
User 1
User 2
Room B
User 3
User 4

Figure 4: Update propagation over sockets

The event types received and handled by the sockets are shown in the following tables.

Events handled on Server Side:

Event	Payload	Description
joinRoom	room: string, auth: JWT	Joins the user's socket to the room specified by the Room ID if the user is authorised for this room. Otherwise, returns a failure message back to the client. The details of the room are returned for this event, which include the difficulty, users' names and UIDs, as well as the question title and description.
update	auth: JWT, room: string, update: Changes	Forwards the given update message to the other user in the same room.

Events handled on Client Side:

Event	Payload	Description
docUpdate	update: Changes	The Changes object consists of from and to positions, text added and removed, and the type of operation. This object will then be applied through CodeMirror on the client's end.

When implementing the synchronisation of the code editor, a major concern was the latency between users. For a real-time collaboration application, updates made should appear for the other user as quickly as possible. Furthermore, this application would also need to support data transfer in both directions, from user to server and from server to user. In designing this application, we had considered using a REST API to send and receive these changes, but ultimately decided on using websockets via Socket.io, mainly due to these challenges of low latency and bidirectional communication. With REST, the communication is largely uni-directional in nature, optimised mainly for large numbers of GET requests (receiving from server), which would be more suited to occasional communication between user to server. On all levels of our requirements, this was vastly inferior to websockets, which offered better performance for frequent communication in both directions, which was ultimately the reason for choosing to use Socket.io over REST.

In terms of extensibility, we also felt that Socket.io would offer more functionality in user-to-user communication in future extensions. For example, the ability to monitor connections could be used to count the number of users online at any one time. The low latency capabilities could also be leveraged to offer more information about the other user's status, such as cursor position.

Another decision we had to consider was the update messages to be sent over the connection. We had 3 main options to choose from:

1. Send the entire text in the code editor as a string.
2. Store the text as a Conflict-free Replicated Data Type (CRDT) and send only the operation messages generated by the data type.
3. Send the position, range, and characters removed or added.

For the first option, although the implementation is extremely simple, there is a high possibility of user text being overwritten by incoming changes.This is illustrated in the table below:

Current Code State	Incoming Update	Final Code State
`// Example`	`// Excellent`	`// Excellent`

In this example, the incoming update overwrites the user's existing text, despite the current user already having changes written. In this approach, the text would constantly be overwritten as the user writes it, and would require both users to take turns writing to prevent conflicts from occurring. On the other hand, there is no risk of complete desynchronisation, as the full text can be retrieved from the update message.

In the second option, we use the operations applied to the CRDT to resolve conflicts between users. In an operation-based CRDT, each change to the text can be stored as some operation, like inserting some text to the right of a specific point in the code. This storing of operations would provide us with an interface we can use to communicate changes between users, which provide conflict resolution based on the implementation of the library used. For synchronising users, this approach would be ideal, as it reduces the amount of data that needs to be sent, while also allowing for conflicts to be resolved on a more reliable level. However, the implementation for this approach was the hardest of the 3, with poor reference material in regards to integrating with the rest of our tech stack.

The last option, which was what we ultimately chose, was the middle ground between the first 2 options. Implementation-wise, it was easier than using CRDTs, but significantly more difficult than sending the entire string. The Changes objects we used also helped to solve most cases of conflicts between users, but was not perfect, and could still result in conflicts in some edge cases. Due to the approach of inserting or removing text at a given position, it could still be possible for these update operations to be affected by changes that occurred before the update is applied, and would likely run into further problems in poor connectivity, which further emphasises the need for low latency communication through websockets.

2.3.4 Answer submission and closing of room

For submission of answers, we would need to synchronise both users together, since we need to ensure both users' answers are saved. For this, we are using both websockets as well as RESTful APIs to coordinate closing of the rooms as well as submission of answers.

When one user starts the submission process, the user emits an event to the server to initiate the clean up process. On receiving the event, the server emits back to both users in the room to finalize the end event.

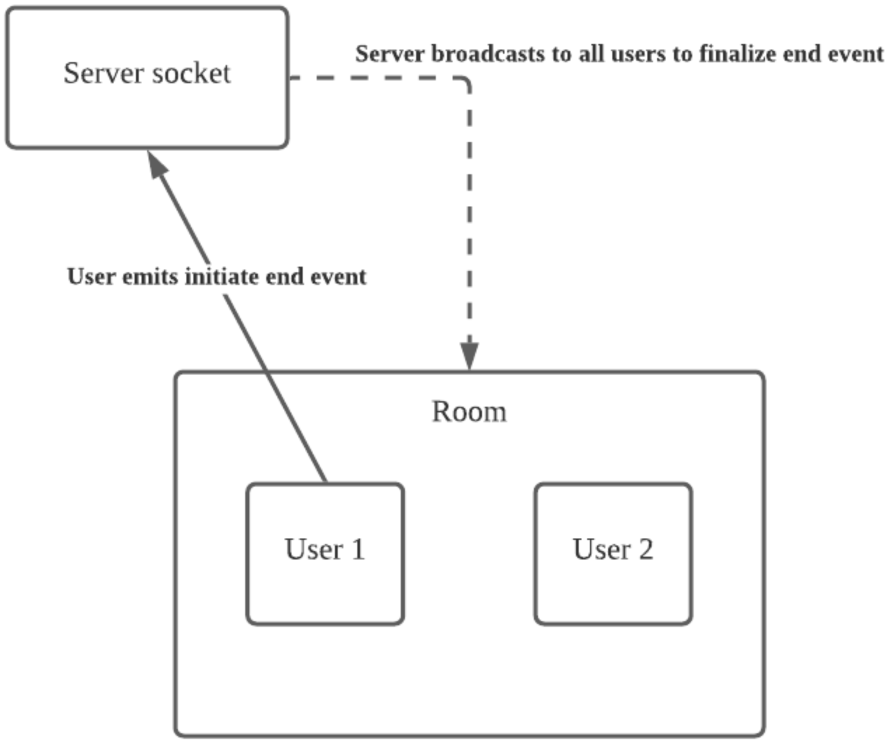

Server socket
Server broadcasts to all users to finalize end event
User emits initiate end event
Room
User 1
User 2

Figure 5: Initiation of end event

Events handled on Server Side:

Event	Payload	Description
initiateEnd	auth: JWT, room: string	Starts the ending process for the match.

Events handled on Client Side:

Event	Payload	Description
finalizeEnd	-	Triggered when a user initiates ending the session. This signals to the client to submit the answers into the database via the `/answers` endpoint.

After the users receive the finalize end event, the clients initiate a `POST` request to the `/answers` endpoint, sending the answer and the roomid the question is from, to the server. We will then fetch the question from the room, using the roomid to fetch the room details from the database to use with the submitted answers to construct an Answers object, which will be stored into the database.

Relevant API:

Endpoint	`POST /answers`
Headers	`Authorization: Bearer {token}`

Body	`{`
	`    answer: string;`
	`    roomid: string;`
	`}`

Description	Stores both question and answer to the database

Schema for Answer in MongoDB:

```
Answer {
    uid: string;
    title: string;
    description: string;
    difficulty: 'easy' | 'medium' | 'hard';
    answer: string;
    createdAt: Date;
}
```

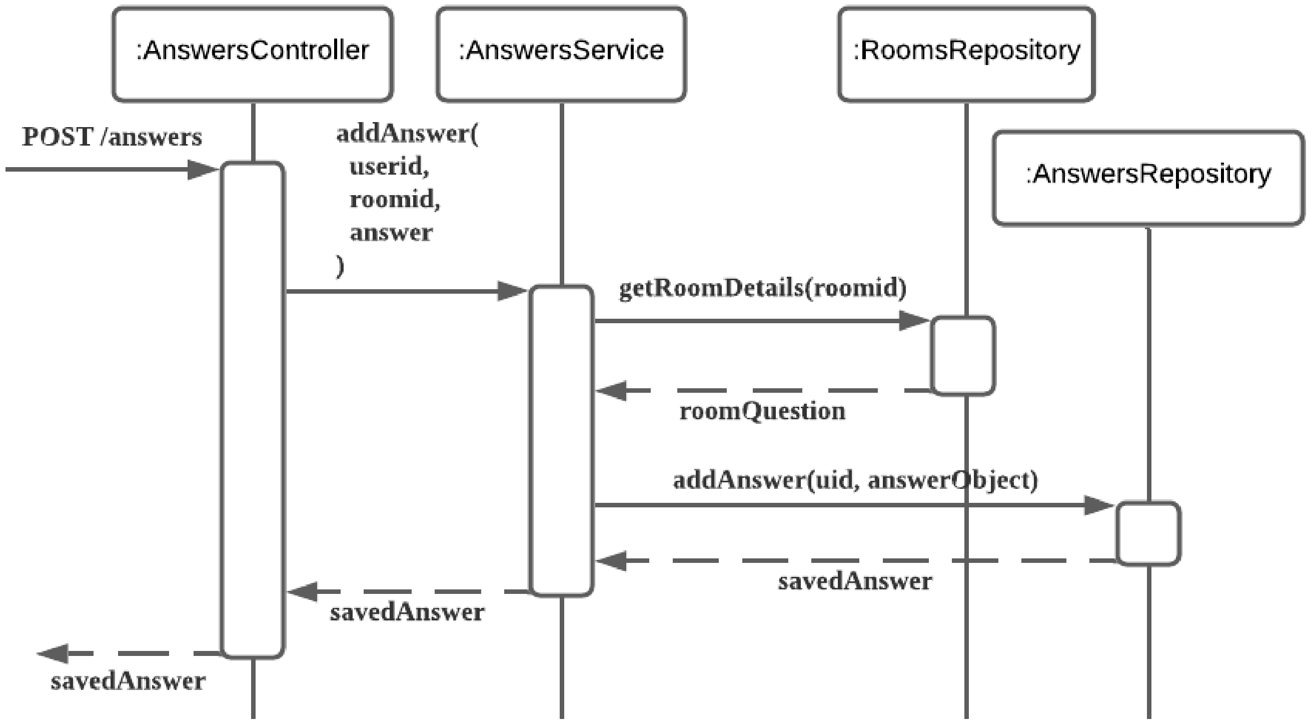
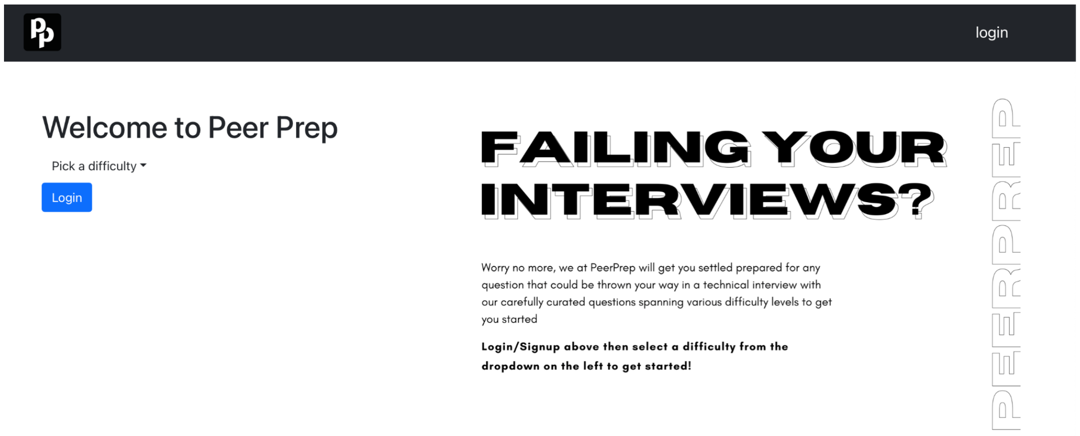


Figure 6: Sequence diagram of answer submission

2.3.5 Landing Page

With regards to the design of our user interface, we wanted to avoid alienating the user by making things appear complicated, so we went with a minimalistic design, using the home page to provide new users with a basic description of the application as well as instructions on how to start using the application.



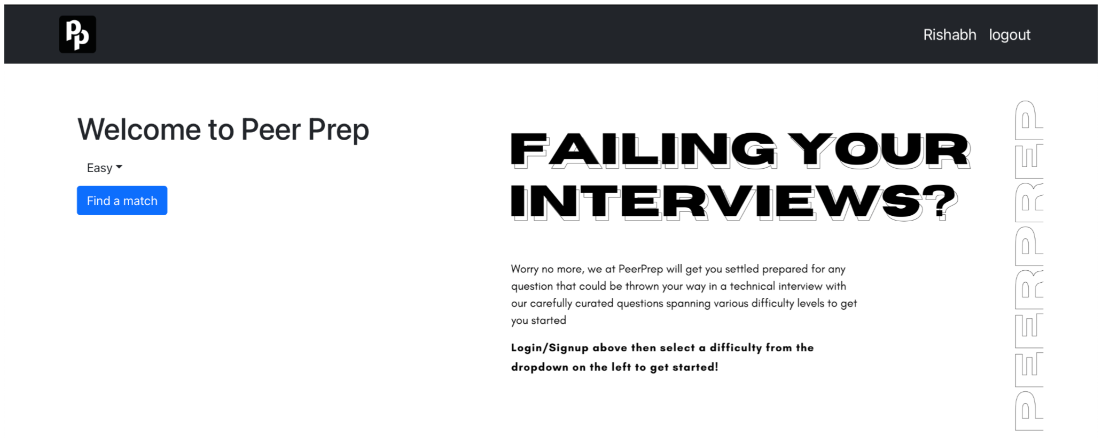
Welcome to Peer Prep

FAILING YOUR INTERVIEWS?

Worry no more, we at PeerPrep will get you settled prepared for any question that could be thrown your way in a technical interview with our carefully curated questions spanning various difficulty levels to get you started

Login/Signup above then select a difficulty from the dropdown on the left to get started!

Figure 7: Main application landing page



Welcome to Peer Prep

FAILING YOUR INTERVIEWS?

Worry no more, we at PeerPrep will get you settled prepared for any question that could be thrown your way in a technical interview with our carefully curated questions spanning various difficulty levels to get you started

Login/Signup above then select a difficulty from the dropdown on the left to get started!

Figure 8: Page displayed after user is logged in

To prevent users from finding matches before being logged in, we designed the blue button below the difficulty selection in Figure 5 to act as an additional Login button, which would otherwise show the “Find a match” button as shown in Figure 6. This prevents users from attempting to find a match before they are logged in.

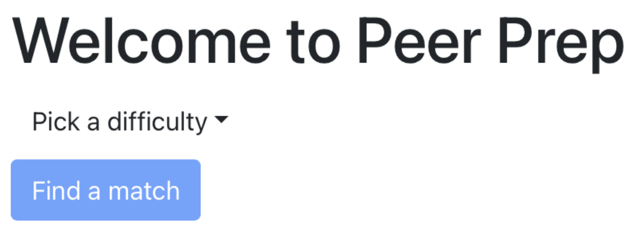


Figure 9: Find a match button before difficulty is selected

Additionally, we greyed out the button to provide a visual indication to the user that they need to select a difficulty before they will be able to find a match.

Once the user selects a difficulty and searches for a match, the “Find a match” button as shown in Figure 5 has its text replaced with a spinning loading icon that provides feedback to the user that a match is currently being found for them. If the backend server is unable to find a match within 30 seconds, the button reverts back to its state in Figure 5 to indicate that the matchmaking request has timed out.

2.3.6 Room Page

After a match has been found, the user is routed to a game room with the other player they are playing with. At this stage, a ‘Join Room’ event is sent through the socket to the backend that returns the room details consisting of the following items:

- Question Title
- Question Description
- Question Difficulty
- User information (Token & Username)

Using the React framework, the Room component acts as a parent and supplies its subcomponents such as PartnerInfo, Question and CodeEnv with the required information. The benefit of using Flux in this way is that it promotes loose coupling since the state of the application is contained in the store of each of the components.

2.3.7 Profile Page

After the user submits their answers, they would be able to view their answers by going to their profile page. As explained in section 2.3.4, when the user submits their answer, it would be saved into the database in the backend. This would imply that to get the answers, we would only have to call a simple GET request. We implemented this feature to allow our users to review their past submissions and find ways to improve on their solution.

Relevant API:

Endpoint	`GET /answers`
Headers	`Authorization: Bearer {token}`
Body	-
Description	Retrieves all user answers from the database

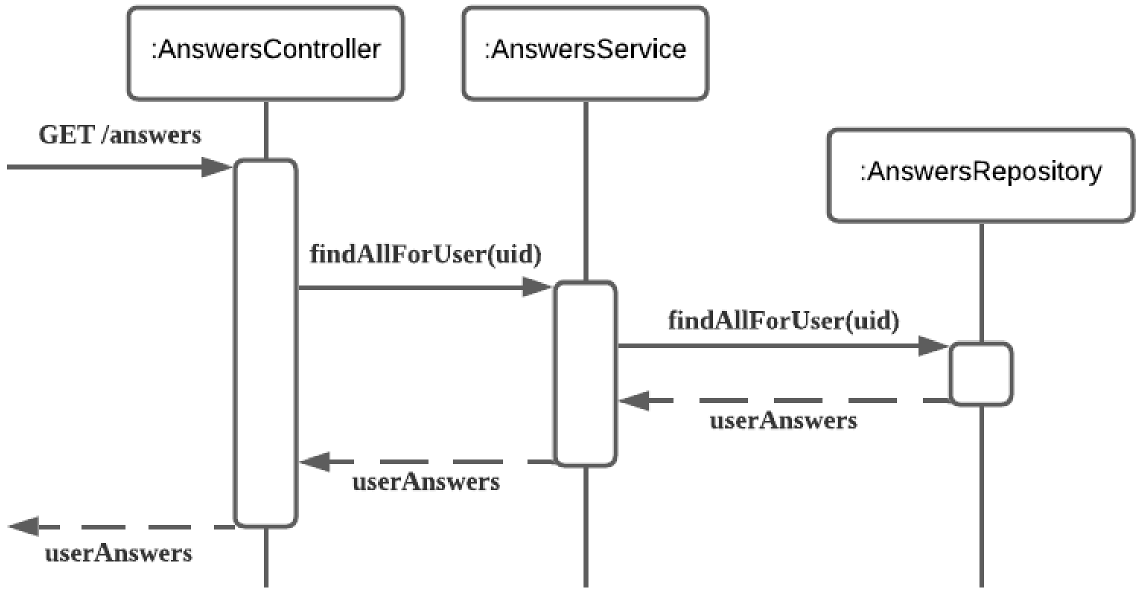
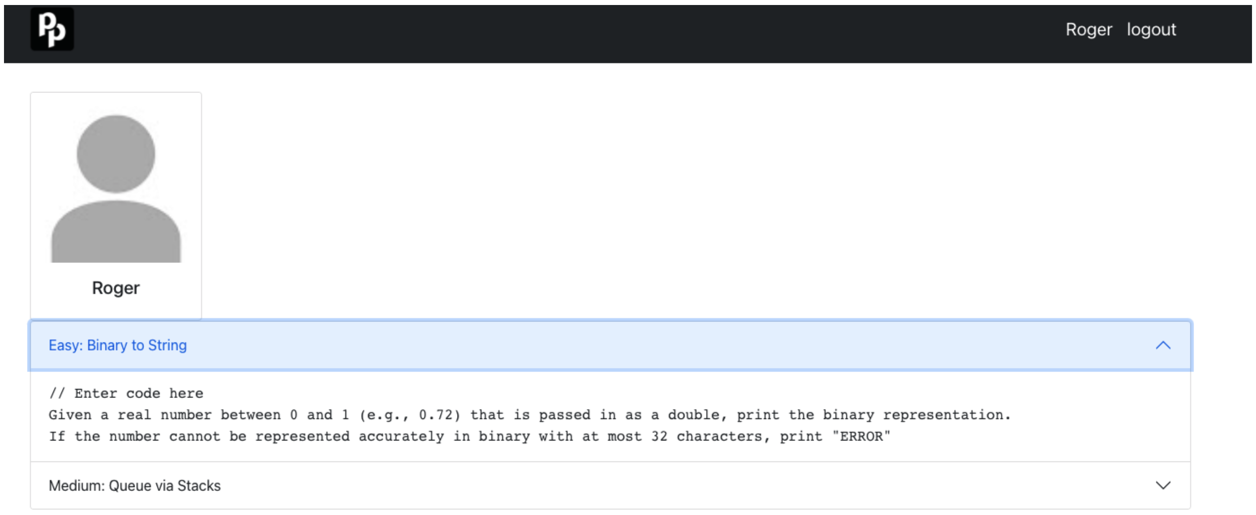


Figure 10: Sequence diagram for getting all user answers



```
// Enter code here
Given a real number between 0 and 1 (e.g., 0.72) that is passed in as a double, print the binary representation.
If the number cannot be represented accurately in binary with at most 32 characters, print "ERROR"
```

Figure11: Showing the profile page with the users answers

3 Project Management

3.1 Development Process

For this project, after careful consideration we decided to use the Waterfall development process model. We made this decision as for this project we had generated a fixed set of requirements at the beginning which were not going to change throughout the duration of the project. As a result, choosing the Waterfall process allowed each of us to know exactly what we would be working on throughout the duration of the project and allowed us to effectively plan out our progress. The risk that we undertook by choosing this approach over other popular processes such as Agile was that we would not have a working product until we were approaching the deadline. Another related problem would be that if one process got delayed, it would cause a delay in all other processes. We felt that these issues, while present, could be mitigated with good time management as well as by helping each other in each of the tasks which would not only allow us to adhere to our internal deadlines but would also allow us to learn how each of the components in the application operate.

3.1 Individual Contributions

Team Members	**Technical Contribution**	**Non-Technical Contribution**
Joshua Liang Xingya	● Authentication (Users) ● Storage of answers ● Storage of questions ● API Endpoints ● Profile Page	● Documentation ● Backend design
Rishabh Paliwal	● Front-End (Room & CodeEnv) ● DevOps	● Project Management ● Documentation
Roger Lim Yong Siang	● Front-End (UI) ● Front-End (Landing page, routing and authentication)	● Documentation ● Front-End design
Ler Yong Yao, Jazer	● Synchronisation ● Rooms ● Matchmaking	● Documentation

3.2 Project Timeline

Front End						
Task	**Week 1**	**Week 2**	**Week 3**	**Week 4**	**Week 5**	**Week 6**
Planning & Designing	🟦🟩				🟦	
Landing Page		🟦			🟦	
Authorisation			🟦	🟦		
Room Page		🟩	🟩		🟩	
Coding Environment			🟩	🟩		
Routing		🟦		🟦		🟦
Socket IO integration					🟦🟩	🟩
Profile Page						🟥
Bug Fixing					🟦🟩	🟦🟩

Backend						
Task	**Week 1**	**Week 2**	**Week 3**	**Week 4**	**Week 5**	**Week 6**
MongoDB Setup	🟥					
Backend framework	🟥	🟥				
Questions Module		🟥	🟥			
Matchmaking Module		🟥	🟧	🟧	🟧	🟧
Rooms Module			🟧	🟧	🟧	🟧
Answers Module			🟥	🟥	🟥	🟧
Auth Module	🟥	🟥	🟥			
Code Editor Synchronisation				🟧	🟧	🟥

Legend: 🟥 Joshua 🟧 Jazer 🟦 Roger 🟩 Rishabh

4 Future Enhancements

4.1 Online user count

Allowing users to see the number of users online would allow them to know the likelihood of them being able to find a partner to match with. We feel this is an important metric to display especially in the early stages to allow the users to know if there are enough active users on the platform for them to be able to find a match. The implementation of this feature would be fairly simple as we can use a method provided by Socket.io that would return us the number of current users. However, this method can be complicated to implement if we need to rely on multiple socket servers in horizontal scaling, requiring us to either aggregate the user counts for each server, or maintain some shared memory between the servers that can be updated as users connect and disconnect.

4.2 User Question Submission

Allowing users to submit questions would allow users to feel more involved with Peer Prep while giving the team a simple and renewable source of questions. Submitted questions would have to be manually reviewed by the team to ensure that questions are of certain quality and to remove duplicated or plagiarized content. The implementation would be relatively simple as we would only have to add an additional page with one API call.

Endpoint	`POST /submit-question`
Headers	`Authorization: Bearer {token}`

Body	`{`
	`    difficulty: string;`
	`    question: string;`
	`}`

Description	Submits the question to the backend for review

However, a key concern that would need to be addressed is the manpower required to audit and review each submission. For a question to be accepted, the question should be original and unique to our questions database, and should also be solvable. At a low number of questions, performing this check manually would not be an issue, but as the number of questions increases, it would no longer be practical to manually verify each question's integrity. Therefore, we would also need to have some form of categorisation or other kinds of organisation to allow us to look for similar-looking questions more easily when vetting submissions.

4.3 Adding an online compiler to allow the users to run and test code

A compiler would be a very useful tool for users to ensure that their solution can run as expected. However, test cases would not be provided as we feel that creating test cases is an important skill that would be tested during a technical interview. The easiest way to implement the compiler would be to send the code to the server, compile and run it. After which the server would return the output back to the client to be displayed. While this is probably the most straightforward solution to the problem, it can introduce certain security vulnerabilities as malicious users could run code that would do harm to the server. One possible solution to this is containerisation where we would run the user's code in a container and extract the results. This would prevent malicious code from doing harm to the server as when the code is run in a container a lot of the security issues would be solved automatically.

4.4 Allow users to view model answers for selected question:

When reviewing past questions, it is common for users to want to view the optimal solution to problems. One easy solution to this is to link the users to a page with a model answer that is prepared by the team. However, this does introduce a similar issue to 4.2, where manpower must be allocated to solve or verify solutions that can be published as a model answer.

Relevant API:

Endpoint	`POST /model-answer`
Headers	`Authorization: Bearer {token}`

Body	`{`
	`    question: string;`
	`}`

Description	Returns a model answer that is prepared by the team

5 Reflections

5.1 Scalability

As mentioned in Section 2.1, we had chosen to use a monolithic design due to the limited scope of the project. However, if we were to scale this project up to serve large numbers of people, perhaps even in the scale of hundreds of thousands, we would likely have to convert our implementation into a microservices-based design, due to the large number of socket connections that would have to be managed by a single server.

Based on benchmarks[1] for socket.io, it appears that the program begins to malfunction past 10,000 messages per second. At an average of 200 characters per minute (possibly considerably higher for programmers), this works out to be more than 3 characters per second for an average person, which would put our limit at around 3000 concurrent typing users.

This could be partially alleviated by batching update messages sent from the code editor, as the current implementation sends every operation as a single message to the server. However, this would still be limited to sending small chunks of text, as larger update granularity could make for a poor user experience.

Therefore, in order for user-to-user synchronisation to scale properly, socket connections will need to be split across multiple servers, using technologies like reverse proxies, load balancers, and Redis Master-Slaves to forward messages to the correct server.

With matchmaking, the current approach of MongoDB may be usable, making use of MongoDB's ability to horizontally scale via sharding. However, the current method of reading and deleting a match document may be difficult to scale, and other approaches may need to be adapted. For example, a message queue broker, or a AWS lambda would also provide the scalability needed to handle large numbers of users in the queue.

[1] http://drewww.github.io/socket.io-benchmarking/

5.2 Challenges

We faced numerous challenges implementing the synchronisation between users. As the frontend was using CodeMirror to provide syntax highlighting and other text editor features common to programming text editors, we needed to find solutions that would integrate well with CodeMirror's functionality, while also being able to send changes over socket.io. Originally, we had planned to use Conflict-free Replicated Data Types (CRDT) implementations like Automerge or Yjs to synchronise code changes and avoid race conditions between users, but the scarcity and low quality of relevant documentation with regards to its integration with our chosen libraries proved to be too complicated for the limited time we had to implement the program.
Furthermore, CodeMirror's documentation was incomplete in many places, and required us to look through the source code to find relevant functions that we ultimately used to implement the synchronisation, namely; the information that could be retrieved from changes in the editor state which contained positions and text modified. This was encapsulated in a Changes object that would be sent and applied through the socket.

Another problem that arose from CodeMirror's lacking documentation was the problems related to maintaining the cursor position as updates are applied. Naively, setting the code in the editor to have the new changes applied would reset the state of the editor, which would in turn reset the cursor to the end of the text. This meant that users could be in the middle of typing something in the middle of the line, only to be sent to the end of the text file when the other user updated the code. From a usability perspective, this would be a terrible experience for the user, since this would consistently interrupt the flow of the user's inputs and by extension, the user's train of thought, which was why this was a top priority for us to fix. However, documentation was severely lacking in regards to how this information could be retrieved and used, and much trial and error was used to figure out the details of the format used in the changes.

Lastly, the introduction of new technologies caused some delays in the initial stage of development as the team needed time to familiarize ourselves with the tech stack. As we wanted to use the opportunity to pick up skills in new frameworks and tech stacks, this applied to both the front and back end teams. This did allow the backend team to gain experience with Nest.JS and socket.io and the frontend team to learn with React and the frontend implementation of socket.io, this came as a tradeoff with time that we could spend actively implementing the project.

5.3 Learning Points

When we started planning this project, we had many possible libraries and approaches to choose from. Libraries like Automerge, CodeMirror, and other libraries that we had tried to use in implementing the project were all very suitable for our requirements on paper, but had limited compatibility with each other. As mentioned in the synchronisation section, CRDTs like Yjs or Automerge were effectively an ideal solution to the problem of contention between user inputs, and was a top choice for us when implementing the synchronisation. CodeMirror was also incredibly helpful in providing us with syntax highlighting out of the box, which would have otherwise taken a lot of time and effort to implement. However, we did not look deeply enough into the possible issues in integrating the different libraries together during the planning phase, which ended up taking a lot of time to rectify towards the end, as shown in the above section on challenges.

One of the most important things we have learned from this project is the importance of planning for unforeseen circumstances, and better anticipating issues that may arise, especially from attempting to use new technologies or frameworks. With projects that make use of many different libraries, it is imperative that the compatibility of each moving part is verified during the planning stage, in order to smoothly implement functionality later on. Much of our challenges came from assuming that things would work as we expected them to, when we had only but a surface-level understanding of the implementation during the planning phase, and in future, we will bear in mind the challenges that each library can introduce into a project.