



# CS3219 Software Engineering Principles and Patterns

PeerPrep Group 53

## **Members:**

Darren Chang Weixiang A0233626H  
Koh Kai Xun A0234276A  
Low Jian Feng A0235089X  
Lu Chen Yu A0231052Y  
Thian Zhang Zhi A0234719Y

## Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>1. Contributions</b>	<b>3</b>
<b>2. Introduction</b>	<b>5</b>
2.1 Purpose	5
2.2 Target users	5
2.3 Project scope	5
<b>3. PeerPrep Features</b>	<b>6</b>
3.1 Must have features	6
3.2 Nice to Have features	6
3.3 Functional Requirements	7
3.3.1 User Profile Management Functional Requirements	7
3.3.2 Matching Session Functional Requirements	8
3.3.3 Collaboration Session Functional Requirements	8
3.3.4 Frontend Functional Requirement	10
<b>4. Non-functional Requirements</b>	<b>10</b>
4.1 Security NFR	12
4.2 Usability NFR	15
4.3 Performance NFR	16
4.4 Reliability NFR	17
<b>5. Architecture</b>	<b>18</b>
5.1 High-level Overview of PeerPrep Architecture	18
5.2 Architectural decisions	19
5.3 Deployment	21
<b>6. Design Patterns</b>	<b>22</b>
6.1 Model-View-Controller	22
6.1.1 View Layer	22
6.1.2 Router	22
6.1.3 Controller	22
6.1.4 Model	23
6.2 Direct Messaging Pattern	23
6.3 Publisher-Subscriber Pattern	23
6.4 Request-Reply Pattern	24
6.5 Data Transfer Object	24
<b>7. Backend</b>	<b>25</b>
7.1 User Profile Service	25
7.1.1 Creation of a new user	25
7.1.2 User Authentication	27
7.2 Question Service	32
7.2.1 Overview	32
7.2.2 Enforcing Authorization	32
7.2.3 Implementation	34
7.2.4 Enhancing Flexibility of Question Service	35
7.3 Matching Service	35

7.4 Collaboration Service	39
7.5 Communication service	41
7.5.1 Implementation of starting call	41
7.5.2 Implementation of ending a call	42
7.6 Question History Service	43
7.7 Testing of Backend Services	45
<b>8. Frontend</b>	<b>46</b>
<b>9. DevOps</b>	<b>48</b>
9.1 Testing	48
9.1.1 Testing Frameworks and Tools	48
9.1.2 Testing Strategy	49
9.1.3 Code Coverage Analysis	49
9.2 Local development	50
9.3 Deployment	52
9.3.1 Task and Task definition	52
9.3.2 Elastic Container Registry (ECR)	52
9.3.3 Overview of deployment process	52
9.4 Monitoring of deployed services	53
9.5 Continuous Integration and Continuous Deployment	53
<b>10. Workflow</b>	<b>55</b>
10.1 Scrum Methodology	55
10.2 Jira	56
10.3 Github	57
<b>11. Proposed Extensions</b>	<b>57</b>
11.1 Image Support	57
11.2 CQRS	58
11.3 Question Scraping	58
11.4 Incorporation of Generative AI	59
<b>12. Reflections</b>	<b>60</b>

## 1. Contributions

Member	Technical Contributions	Non-Technical Contributions
Darren Chang Weixiang	Setup Login and Register Page  Question History database and backend  Implement Question on the fly feature	Final Report  Final Presentation
Koh Kai Xun	Question Page Frontend  Set up authentication middleware with Zhang Zhi  Extend Question Service to include Categories to enable tagging questions by categories	Final Report  Final Presentation
Low Jian Feng	Collaboration Backend  Question History Backend  Assist Frontend	Final Report  Final Presentation
Lu Chen Yu	Set up GCP and create postgresql database  Set up user profile backend and link api to database  Set up functionality of code execution  Create test cases , auto test coverage and set up Continuous Integration	Video Caption  Final Report
Thian Zhang Zhi	Set up backbone for Question Backend  Implement question, matching, communication backend  Dockerise all services  Set up authentication middleware with Kai Xun  Bug fixing for frontend.  Deploy on AWS and set up CD	Organise weekly meetings  Task monitoring  Final Report  Final Presentation

## 2. Introduction

In the wake of the COVID-19 pandemic, companies are progressively shifting to online platforms for interviews. Traditional face-to-face interviews are being replaced with interviews conducted over platforms such as zoom, or MS Teams. This trend has been especially noticeable in technical interviews – a usual requirement of the hiring process within tech companies. In order to provide aspiring programmers with a platform to practise and prepare for such technical interviews, our team has implemented a web application – PeerPrep.

### 2.1 Purpose

PeerPrep is a web application that helps students prepare themselves for an online technical interview. This is done by simulating a technical interview setting to provide them with a rough gauge of what it is like to be in an actual interview. In addition, PeerPrep also helps students improve their technical and communication skills as they work together with others to solve programming problems. These are skills that employers will be looking out for during an actual interview as well.

### 2.2 Target users

Although PeerPrep was designed with students in mind, it can be used by any individual who is preparing for a technical interview.

### 2.3 Project scope

This is the scope within which we identified how PeerPrep should function.

1. Users should be able to have their own profile.
2. Users can collaborate with others to practise questions on PeerPrep.
3. Users should be able to communicate while collaborating on questions.
4. Users can specify a criteria for which they will be matched with others.
5. PeerPrep should allow users to manage their profile details.
6. PeerPrep should keep a user's information away from other users
7. Users should have an interface to interact with PeerPrep.
8. Users can compile and run their code in their preferred language

### 3. PeerPrep Features

#### 3.1 Must have features

Feature	Description
Basic UI	Allows users to easily interact with the application.
User Profile Management	Authenticates and manages multiple users and their associated information.
Matching Service	Matches users together based on predetermined criteria.
Question Service	Maintains a question repository for users to practice.
Collaboration Service	Provides a platform for matched users to collaborate during mock interviews.

These are features that are essential for PeerPrep to function as a platform to match users to perform mock interviews.

#### 3.2 Nice to Have features

Feature	Description
Communication Service	Provides video communication to better mimic real interviews.
History Service	Keeps a record of question attempts made in previous sessions for performance analysis and improvement.
Code Execution	Allows users to test their code in multiple programming languages, mimicking real interview settings.
Enhanced Question Service	Increases versatility in question searching and selection flexibility during collaboration sessions.
Continuous Integration	Automatically tests code to ensure the application remains bug-free.
Deployment on Production System	Deploying on a cloud platform like AWS to ensure high uptime without the need for a physical machine.

The delegation of Nice to Have tasks was given by this table below:

Sub Group	Members	Nice to Haves
Group 1	Darren Chang Weixiang	History Service Code Execution Enhanced Question Service
	Koh Kai Xun	
	Low Jian Feng	
Group 2	Lu Chen Yu	Communication CI/CD Deployment
	Thian Zhang Zhi	

### 3.3 Functional Requirements

This section explores the functional requirements of PeerPrep, separated by the services that PeerPrep provides.

#### 3.3.1 User Profile Service Functional Requirements

ID	Functional Requirement	Priority
US-FR-1	The user profile microservice should allow users to register using an email, username and password	High
US-FR-2	The user profile microservice should allow users to log into the platform with their registered email or username and password	High
US-FR-3	The user profile microservice should allow users to log out of their account	High
US-FR-4	The user profile microservice should support persistent user login across browser sessions	High
US-FR-5	The user profile microservice should allow users to view update their profile information	Medium
US-FR-6	The user profile microservice should allow users to view attempted question history which should contain: <ul style="list-style-type: none"><li>• Question description</li><li>• Attempted code</li><li>• Submission status</li></ul>	Medium
US-FR-7	The user profile microservice should allow admin users to view info of all registered users	Medium
US-FR-8	The user profile microservice should allow admin users to set non-admin users to be admin.	Medium
US-FR-9	The user profile microservice should allow users to deregister from the platform	Low
US-FR-10	The user profile microservice should allow users to add user bio to their profile	Low

### 3.3.2 Matching Session Functional Requirements

ID	Functional Requirement	Priority
MS-FR-1	Matching sessions should allow a user to find a match with someone who specified the same difficulty (Easy, Medium, Hard).	High
MS-FR-2	Matching sessions should provide feedback that it is searching for a user instead of being static while the search is happening.	High
MS-FR-3	Matching sessions should guarantee a match for any two users if they have the same difficulty chosen.	High
MS-FR-4	The matching sessions should not match people with mismatched difficulties	High

### 3.3.3 Collaboration Session Functional Requirements

ID	Functional Requirement	Priority
CSM-FR-1	The Collaboration session should provide a shared text editor for both users to work on the same document.	High
CSM-FR-2	The Collaboration session should allow users to communicate through video during the session.	High
CSM-FR-3	The Collaboration session should display the same question to both users in the same room.	High
CSM-FR-4	The Collaboration session should allow for users to submit their code if they want to verify the working condition of their proposed solution.	Low
CSM-FR-5	The Collaboration session should allow users to save session information for future reference	Medium
CSM-FR-6	The Collaboration session should allow users who might have lost connection or refreshed the page to continue with their session as per normal	High

### 3.3.4 Frontend Functional Requirement

ID	Functional Requirement	Priority
FE-FR-1	There should be interfaces for logging in, signup, viewing questions, looking for a match and managing personal profile details.	High
FE-FR-2	The collaborative Code editor should be updated in real time so that users' changes do not conflict with one another.	High
FE-FR-3	User should be provided feedback when credentials filled do not meet requirements	High
FE-FR-4	Every user should be able to view feedback when matching is initiated, failed or successful.	Medium
FE-FR-5	The Application should provide easy navigation between different pages	Medium
FE-FR-6	The collaborative Code editor component should provide syntax highlighting to simulate a read editor used in an interview.	Low

## 4. Non-functional Requirements

NFR in terms of priority:

- Security
- Performance
- Usability
- Availability
- Reliability
- Compatibility

Attribute	Score	Availability	Performance	Security	Usability	Reliability	Compatibility
Availability	3		^	^	^	<	<
Performance	4			^	<	<	<
Security	5				<	<	<
Usability	4					<	<
Reliability	2						<
Compatibility	0						

Prioritisation of NFRs

Security was prioritised as an NFR to protect sensitive data in PeerPrep and prevent unauthorised access. Implementing robust security measures helps safeguard the system against potential threats, vulnerabilities and breaches. Users should be able to trust the application before they can even use it. As such this was of highest priority.

Performance is also a very important NFR. Performance directly impacts user experience by dictating system speed and responsiveness. It influences user retention and satisfaction, crucial in today's fast-paced digital landscape especially among the young students, ensuring efficient operations and meeting expectations for promptness in delivering results.

Usability was ranked highly as it directly impacts user satisfaction and efficiency. Ease of use, intuitive design, and accessibility significantly affect user adoption and productivity. Prioritising usability ensures a positive user experience, fostering engagement and ultimately leading to the success of PeerPrep.

Availability is an important NFR as it ensures that users are able to access the application anywhere and anytime. This comes with making the application publicly available.

Reliability ensures consistent system performance under varied conditions, minimising downtime and errors.

Lastly, as we believe target clients will mostly use PeerPrep on computers instead of mobile phones, compatibility was ranked as the lowest priority.

#### 4.1 Security NFR

ID	Non-Functional Requirement	Priority
SEC-NFR-1	The application should prevent access from unauthorised users into an interview session	High
SEC-NFR-2	The application should prevent unauthorised users from accessing certain functions	High
SEC-NFR-3	The application should prevent users from viewing/editing any unauthorised user data	Medium
SEC-NFR-4	The database should hide any sensitive information in case it is compromised.	Low

We will discuss some of the implementations to achieve the NFRs listed above.

Issue	Countermeasure	Security achieved
SQL Injections	Use placeholder (parameterized queries) instead of directly accepting raw inputs as shown in figure 4.1.1.	Integrity
Unauthorised querying/ updating of databases	Implement middlewares to verify user status for every request at each endpoint.  Usage of jwt tokens that are signed to verify user information in the middleware.	Confidentiality, Integrity
Unauthorised access to an account of valid user	Password strength enforced both on FE and backend, hash and stored in DB	Authenticity
Hijacking of collaboration room	Use of uuid helps produce a unique and less-predictable id for each collaboration room.  Refer to figure 4.1.2	Confidentiality
Accessing of sensitive information from compromised database	Sensitive information, such as passwords, is securely stored by generating and storing their hashes and salt using the bcrypt library, rather than storing the information in plain text.  Refer to 4.1.3 and 4.1.4	Confidentiality

```
pool.query('SELECT * FROM users WHERE user_id = $1', [user_id], 
  if (error) {
    throw error
  } else if (results.rows.length === 0) {
    return response.status(404).json({ error: 'No users found' })
  }
  return response.status(200).json(results.rows)
})
```

Figure 4.1.1 Sample parameterized SQL query

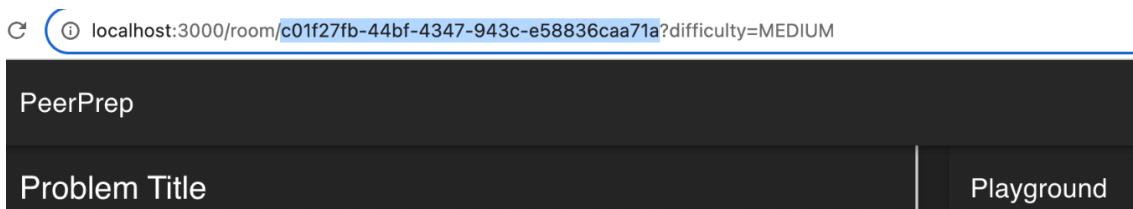


Figure 4.1.2 Sample uuid url for collaboration session

```

const bcrypt = require('bcrypt');

const saltRounds = 10;
const hashedPassword = await bcrypt.hash(password, saltRounds);

```

Figure 4.1.3 Implementation of bcrypt

1	136	jf33	jftest21	\$2b\$10\$/RCy9filX2Bxitf3PnQ0OsWmDKET6U7XTcCTCB0As.Xs0W7D7SSS
2	6	Carmella Boxe	cboxeE	\$2a\$04\$re9R2HP5Qzvz3F08Zwn/5u1dhL3je9ShG6dDH5MSQ/FebtY23/V4W
3	7	Ofelia Gheerhaert	ogheer	\$2a\$04\$12OIB.wizMvlzFqnmju5z.iQaDJEiLrUYT25mGt7RHp4vEkz0d57C
4	55	tzz	zzthiar	\$2b\$10\$Wg3nqBcyfEai4zzhjWeoxutGLohdBa2ksA9nERHePtQf88x8nkBry
5	160	demo1	qqqqqr	\$2b\$10\$Sv1Y8yfmBOE4t4HjZmrNWuuMMV0X.3H4hYNbdYg.FzHKYMqnjzsJe
6	162	evan123	evan@	\$2b\$10\$MrEslz4ee/YQu66dy9ELauvDSWI7u7kdf0C27BTe.aKV/PMOQDLMe
7	56	aaaaaaaaaaaaaaa	a@a.cc	\$2b\$10\$uSxUN4.nAygDgyPBwqHdG.yitaRI7zD/SIfD8Jsd/sHDB356vr6Tu
8	14	Myer Wyper	mwype	\$2a\$04\$CrDQsqlPRVtvd/rkKS07uOUV3UiBogvTLeYplM8W3CdJYB0Cxuby
9	15	Jacintha Bassill	jbassill	\$2a\$04\$W0x/CsZlZAIYs5o8xLMAy.vpD12wR/KOZgJSqcWVyqKQCr7NaSSNe
10	16	Holly Wartonby	hwarto	\$2a\$04\$hDrVhB0hQLrTu4FazGLbCeb38iUkh7D9ILHFipW1gVxWg1zGYUPZC
11	17	Egor Garces	egarce	\$2a\$04\$u1O/XIQYp0ljDkGy.m5ifORapXn4e2oegg2voHd1AH8vr.vUJeSZ6
12	18	Ronni Everix	reverix	\$2a\$04\$OCyrn.w687LD5yE9HwwkwOn0m8tl0ixLfjP8LK5PUtS0jVKMaJa2
13	19	Shae Pearcehouse	spearc	\$2a\$04\$/KBg5Zsv9XDbUH8bOaOjteWpwCpoDghYov1XK1SdzUgrBYzlxJgvG
14	8	Natale Scuffham	nscuffl	\$2a\$04\$YREuiZ5UpTL3xUtmH4rdHexbJ556QDQ6GR6aGRb7qSuDo/.bj2Epu

Figure 4.1.4 Hashed and salted password stored in database

This helps us to achieve SEC-NFR-4 as the passwords of users will not be compromised even if a malicious entity manages to get hold of the user data.

## 4.2 Usability NFR

ID	Non-Functional Requirement	Priority
USA-NFR-1	The application should provide helpful user feedback for every request sent.	High
USA-NFR-2	The collaborative editor should allow both users to edit at the same time. This is explored more in the next section as well.	High
USA-NFR-3	Each user should be allowed to control whether they want their video to be turned on or not.	Medium

The frontend provides helpful error messages or success messages to the user as shown by the images captured below.

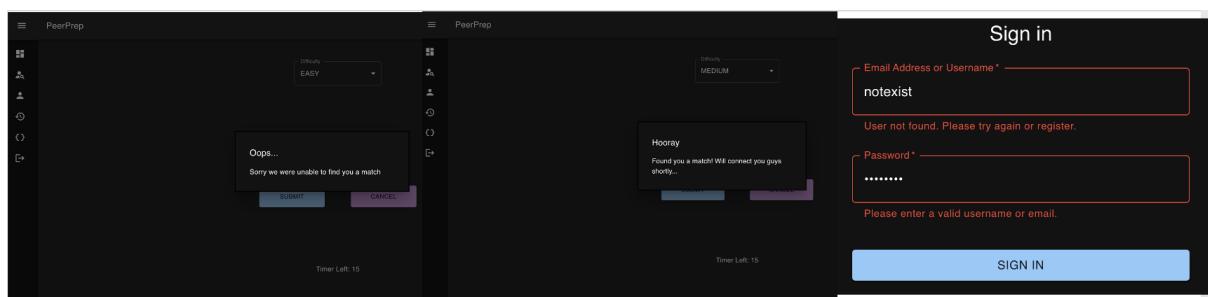


Figure 4.2.1 User feedback reflected in browser for a successful match, failed match, and invalid sign in

The presence of user feedback on the frontend helps make PeerPrep more usable as users are guided as to how the application is meant to be used. This helps us to achieve USA-NFR-1.

## 4.3 Performance NFR

ID	Non-Functional Requirement	Priority
PER-NFR-1	The application should successfully load the interview questions within 5 seconds upon matchmaking	High
PER-NFR-2	The application should match users within 15 seconds of the time they have requested for a match if there is a valid matching.	Medium
PER-NFR-3	During the live coding, each change should be reflected to the other user within 1 second.	High
PER-NFR-4	During the live coding, users can run their code and results should be returned within 5 seconds.	Low
PER-NFR-5	The application should respond to requests within 3 seconds	Medium

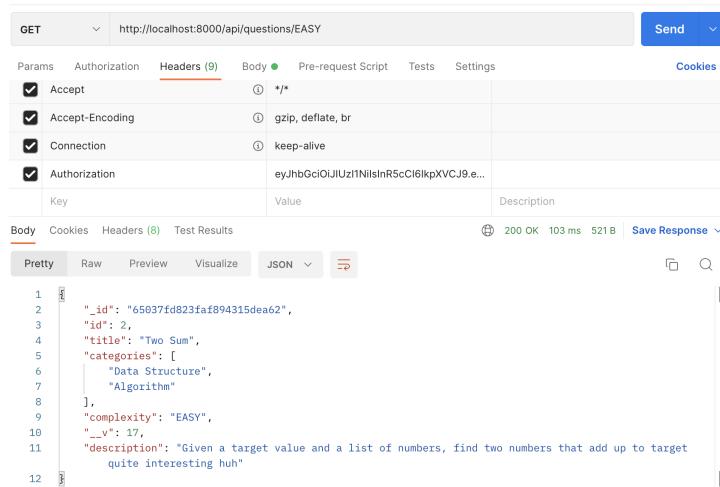


Figure 4.3.1 Request to get question by complexity

The request sent by the collab backend to provide questions to both users in the room during the collaboration is a GET request shown above. As seen from the diagram, we are able to receive a response within the time of 103ms which is far better than the 5 seconds we aimed for, thus achieving PER-NFR-1.

```

const TIME_LIMIT = 15
    const timeout = setTimeout(() => {
        socket.disconnect()
        setIsSubmitting(false)
    }, TIME_LIMIT * 1000)

event.on("disconnect", () => {
    channel.close();
    if (matched) return
    removeFromQueue(userId);
})

```

Figure 4.3.2 Timer to disconnect socket and action on disconnect

As seen from the figure above, we configure the request to only last a duration of 15 seconds for the matching. Once the 15 seconds is up, the user will need to retry matching. Thus a user can be matched within 15 seconds of his request for a match if there is a valid matching. This helps us to achieve PER-NFR-2.

3624 14.110183 172.20.10.3 104.21.51.50 TLSv1.2	172 Application Data
3631 14.136385 104.21.51.50 172.20.10.3 TLSv1.2	91 Application Data
3632 14.136466 172.20.10.3 104.21.51.50 TCP	66 59057 → 443 [ACK] Seq=1 Ack=26 Win=2047 Len=0 TSval=2690550044 TSecr=2303163698
3633 14.136873 172.20.10.3 104.21.51.50 TLSv1.2	95 Application Data
3634 14.161497 104.21.51.50 172.20.10.3 TCP	66 443 → 59056 [ACK] Seq=1 Ack=213 Win=7 Len=0 TSval=3755486593 TSecr=2578481850
3635 14.164283 104.21.51.50 172.20.10.3 TCP	66 443 → 59057 [ACK] Seq=26 Ack=30 Win=7 Len=0 TSval=2303163726 TSecr=2690550044

Figure 4.3.3 Wireshark packets captured when collab editor used

In order to test the latency of the code editor, we used wireshark to capture packets sent from the code editor onto the deployed site and track how long it takes for the site to acknowledge the change. From packets 3631 and 3632, we can see that a change is acknowledged in 0.00012 seconds which is 0.12 ms and is definitely fast enough to be a real time code editor and achieves PER-NFR-3.

The screenshot shows two separate Thunder Client sessions. The left session is a GET request to `http://localhost:3000/api/users/userById?userId=1`. The response status is 200 OK, size is 316 Bytes, and time is 63 ms. The right session is a PUT request to `http://localhost:3000/api/users/updateUser`. The response status is 200 OK, size is 52 Bytes, and time is 55 ms. Both sessions show JSON responses: the first returns user information and the second returns a message "User information updated successfully."

Figure 4.3.4 Thunder Client Testing gets response within 3 seconds

As shown by the postman querying our API, the backend services are able to respond in 63ms which is well within the 3 seconds we have required, hence achieving PER-NFR-5.

## 4.4 Reliability NFR

ID	Non-Functional-Requirement	Priority
REL-NFR-1	On an update, the application should not fail previous functionalities implemented.	High

Issue	Countermeasure	Effect
Regression bugs	Create a series of tests that are run upon each update of the application's code base. This is done in our github workflow. This is explored in section 9 - the DevOps section, with screenshots to show what happens when tests fail or pass.	Verify that any new changes in the code will not break previous functionalities.

## 5. Architecture

### 5.1 High-level Overview of PeerPrep Architecture

PeerPrep consists of two core components, the frontend and the backend. The frontend component consists of a single cluster of containers serving the static content for the web interface.

The backend component of PeerPrep can be further broken down into its 6 component services

- User Profile / Authentication Service
- Questions Service
- Matching Service
- Communication Service
- Collaboration Service
- Question-History Service

The backend of PeerPrep models a microservices architecture. Each of the 6 services is deployed in a containerised environment supported by its own separate data storage solutions.

The following diagram provides a depiction of the architecture of PeerPrep's backend:

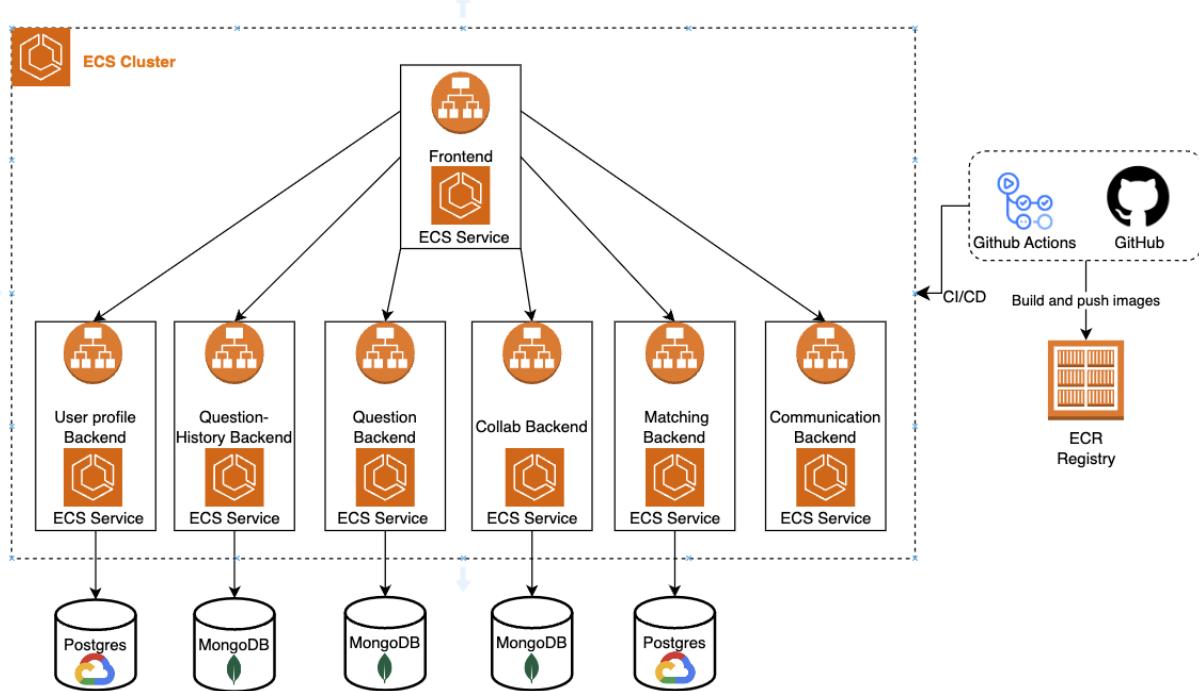


Figure 5.1.1 Architecture diagram of PeerPrep

## 5.2 Architectural decisions

### Microservice vs Monolithic architecture

Architecture type	Microservice	Monolithic
Benefits	<p>Decoupling between the separate service's</p> <p>Each service is focused and can function independently, does not rely on another to function.</p> <p>Separate code bases per service makes it easier for dividing roles and working with others in parallel.</p> <p>Able to scale different services individually.</p>	<p>Easy to deploy only 1 service.</p> <p>Fast development when working as an individual.</p>
Cons	<p>Complicated integration of service interactions when formats of input, naming conventions are not standardised.</p>	<p>Unable to scale a service individually.</p> <p>Any error in a module can cause the entire application to fail and be down.</p>

Decision:

The team opted for a **microservices architecture** to allow each sub-team to take charge of their components and make independent decisions for their services. Such decisions include the choice of data storage service (mainly MongoDB or PostgreSQL) and any relevant libraries. This allows each team to iterate and develop more efficiently due to their intrinsic familiarity with their chosen set of technologies for their services.

In addition, the microservices architecture allows each service in the backend to scale independently to cope with increased loads as required, ensuring greater reliability and scalability in the future.

## Single Database for application vs Database for each service

Architecture type	Single shared database	Database per service
Benefits	<p>Only need to set up once.</p> <p>All information stored in the same place makes it easier to retrieve information.</p>	<p>Each service is granted access to information that it needs, nothing more.</p> <p>Faster lookup with simpler entries since data is spread across multiple databases.</p> <p>Allows for different kinds of databases (eg NoSQL and SQL) for different kinds of data.</p>
Cons	<p>Single point of failure.</p> <p>Can be messy when many different types of data are being stored</p>	<p>Expensive to host multiple databases.</p> <p>Potentially expensive updates if databases are linked in their values.</p>

Decision: We decided that each service requiring a database has its own database. This is because the kinds of data we want to store for each of the services varies vastly. For example, the documents in the Collaboration backend each had variable attributes which NoSQL databases offer flexibility in storing. The user service on the other hand uses postgres. As applications like matching require strict consistency, it would be a suitable choice to use Postgres which enforce data integrity, consistency and reliability through ACID (Atomicity, Consistency, Isolation, Durability) transactions.

Each microservice (except communication that does not need a database) has their own backing data store in a total of 3 MongoDB and 2 Postgres instances. This allows the Separation of Concerns between data models. Should one of the databases fail, only the microservice associated with that database will be brought offline. All other microservices would remain unaffected.

## 5.3 Deployment

Each of the services is deployed in a separate ECS service. This allows individual services to be provisioned and scaled independently, which also reflects the microservices architecture.

We opted to deploy the application onto a cloud platform for several reasons. Firstly, by hosting our services on the cloud, there is a greater pool of computing resources that can be tapped on. Multiple instances of each task can be provisioned which provides scalability to PeerPrep. We also make the application publicly available to users anywhere as long as they have an internet connection. Further information on the deployment process is elaborated in the DevOps section 9.3.

## 6. Design Patterns

### 6.1 Model-View-Controller

The Model-View-Controller architecture pattern is designed to increase modularity and reduce coupling between the user interface and business logic. It consists of three main components - the view which represents the UI, the model which represents business logic, and the controller which links the two.

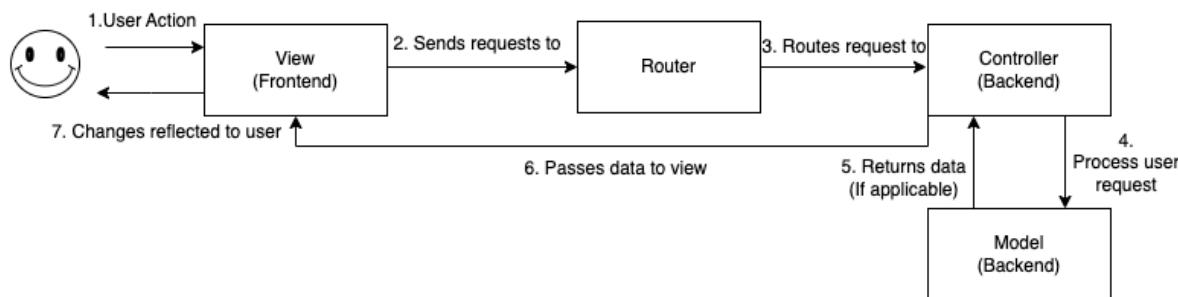


Figure 6.1.1 MVC Pattern in PeerPrep

#### 6.1.1 View Layer

In PeerPrep, the frontend service handles the View layer, determining everything displayed on the user's browser and users interact directly with it. User actions result in HTTP requests being sent to the backend services which implement the Model and Controller layers. Since there are multiple backend services, a router is used to facilitate routing to the appropriate controllers. Any data sent back from the controller then updates the frontend view accordingly.

#### 6.1.2 Router

The router maps controller functions to a URL endpoint which is specified with each request made by the view. Thus, with an incoming request, the router directs it to the corresponding controller action.

#### 6.1.3 Controller

The controller is implemented by each backend service, and it specifies how a request should be translated into actions to be performed on the model by invoking the Model API, specifying how the model should be updated.

#### 6.1.4 Model

The model layer implements the logic to manipulate data in the application. This involves creating, receiving, updating or deleting data in the databases of each of the services. After the model is updated, a response is passed to update the view layer.

### 6.2 Direct Messaging Pattern

As the name implies, this pattern refers to a form of communication where messages are sent directly from the sender to the recipient. This is to support the event driven architecture of the Matching service.

Every match requires two users of similar question complexity preference. Thus, for every match, the first user to initiate a match will be unable to find a match and be deemed as unmatched. A match will only be found when the second user initiates a match and finds an unmatched user with similar question complexity preference, which is the first user.

However, the first user remains oblivious that he has been matched. Hence, the second user will need to inform the first user that he has been matched. Direct Messaging Pattern suits this requirement and is employed when the second user communicates with the first through RabbitMQ in the Matching Service. Implementation of this service will be explored in greater detail in the next section on Backend Services.

### 6.3 Publisher-Subscriber Pattern

The Publisher-Subscriber (Pub-Sub) pattern is a communication pattern where publishers broadcast messages to subscribers. This pattern was chosen for Collaboration and Communication Services not only because of the event-driven nature of their architecture, but also due to the extensions we hope to explore in future iterations as well where more than two users are allowed into a single room.

In the Communication and Collaboration backend services, socket connections to each backend are made and users in the same room subscribe to a single room specified by a room Id. Thus events happening within the room are published by the backend and all users are able to receive the event. Further details are provided in the section on the corresponding microservices.

## 6.4 Request-Reply Pattern

The Request-Reply communication pattern is common in microservice architectures where services need to exchange information. A service sends a request to another service and receives a reply.

The services communicate to each other with HTTP requests, which uses this pattern. It also supports both synchronous and asynchronous communication. Pictured below is a sample of code that demonstrates the request reply pattern in PeerPrep.

```
const response = await axios.get(QUESTION_HOST, {
  headers: {
    'Authorization': getAuthCookie()
  }
})
const data = response.data
const questions = []
for (var i in data) {
  questions.push(data[i])
}
setQuestions(questions)
```

Figure 6.4.1 Code of HTTP call request in Questions Page

## 6.5 Data Transfer Object

The Data Transfer Object data pattern encapsulates multiple related data in an object to reduce function calls. In PeerPrep, Users and Questions use this pattern.

For example, a question object encapsulates many fields - question id, title, description, categories and complexity. So multiple queries for each field can be combined into one.

```
const questionSchema = new mongoose.Schema({
  id: Number,
  title: String,
  description: String,
  categories: [String],
  complexity: String,
});
```

Figure 6.5.1: Code of Question Database Schema

## 7. Backend

As we have stated above, PeerPrep is built upon the microservices architecture. Hence, our backend consists of a total of 6 different services – User Profile, Question, Matching, Collaboration, Communication and Question History.

### 7.1 User Profile Service

The User Profile service is in charge of handling user credentials related operations. This includes creating, storing, reading, updating and deleting user data. In order to facilitate the storage of user data, it utilises a Postgres database. This is where we keep track of all the users that have created an account with PeerPrep and make all user related queries.

For every user account, we decided to store the password in its hashed form instead of plaintext. This would prevent leaking actual passwords in the event a malicious entity gains access to the database.

#### 7.1.1 Creation of a new user

When a new user registers for PeerPrep, they are required to enter a chosen username and a password. There is a restriction on the username and email fields in which it has to be unique for each user in the User data database.

```
CREATE UNIQUE INDEX unique_username ON "users" USING btree ("username");
CREATE UNIQUE INDEX unique_email ON "users" USING btree ("email");
```

Figure 7.1.1.1 Requirements for username and email

When a user tries to register with a username or email that already exists in the database, the User Profile Service will reject the request and not allow the user to register.

The screenshot shows the 'Sign up' form on the PeerPrep website. The user has entered 'demo1' for both First Name and Last Name, 'example@a.com' for Email Address, and 'demo1' for Username. A red error message 'Username is already taken' is displayed below the Username field. The Password field contains several dots. At the bottom is a blue 'SIGN UP' button, and below it is a link 'Already have an account? Sign in'.

Figure 7.1.1.2 Unsuccessful registration when existing username used during registration

Otherwise, the user successfully registers and their information is stored in the database. Below is the implementation of creating a new user:

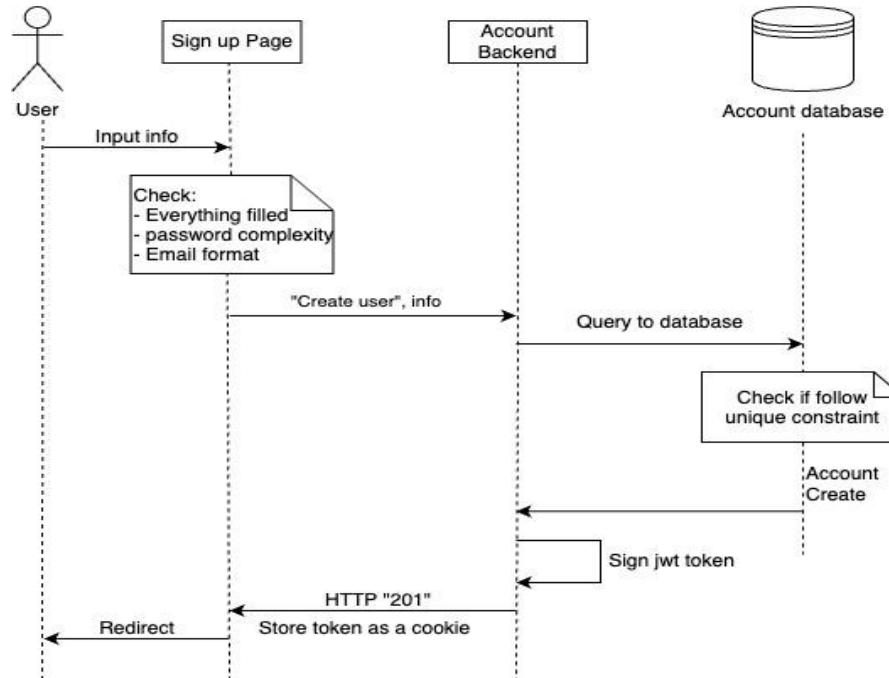


Figure 7.1.1.3 Sequence diagram for successful user registration

### 7.1.2 User Authentication

A user can choose to login by either username or email as their identifier.

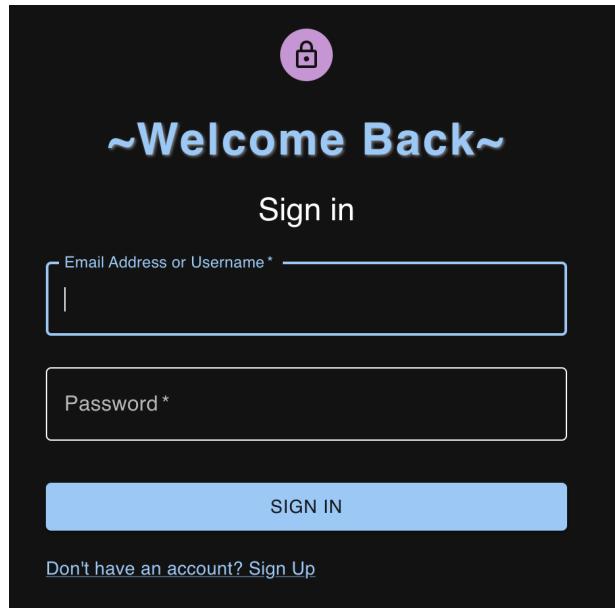


Figure 7.1.2.1 Sign in Page

After a user submits their identifier and password, the user profile backend checks the database for a user possessing the username or email, and then computes the hash of the provided password to compare with the stored hash for that user's entry in the database. Only if the user credentials exist and the hash matches, then will the backend allow the user to log in.

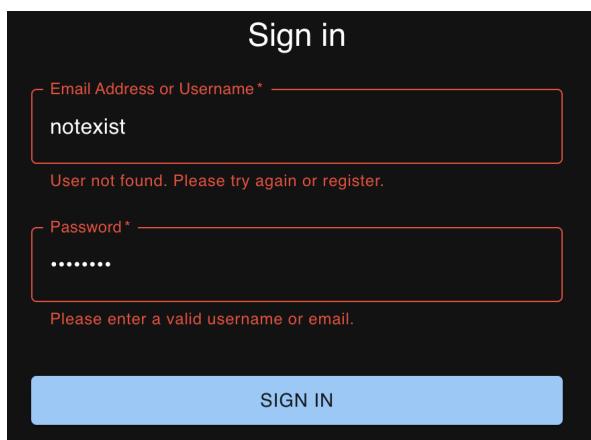


Figure 7.1.2.2 Non-existent user identified

Figure 7.1.2.3 Password incorrect

On successful login, the user profile backend sends a signed JWT token, containing the user's ID and admin status, back to the frontend, which is then saved as a cookie in the user's browser containing information pertaining to the user's credentials. The user's session will persist until the token expires or the user logs out.

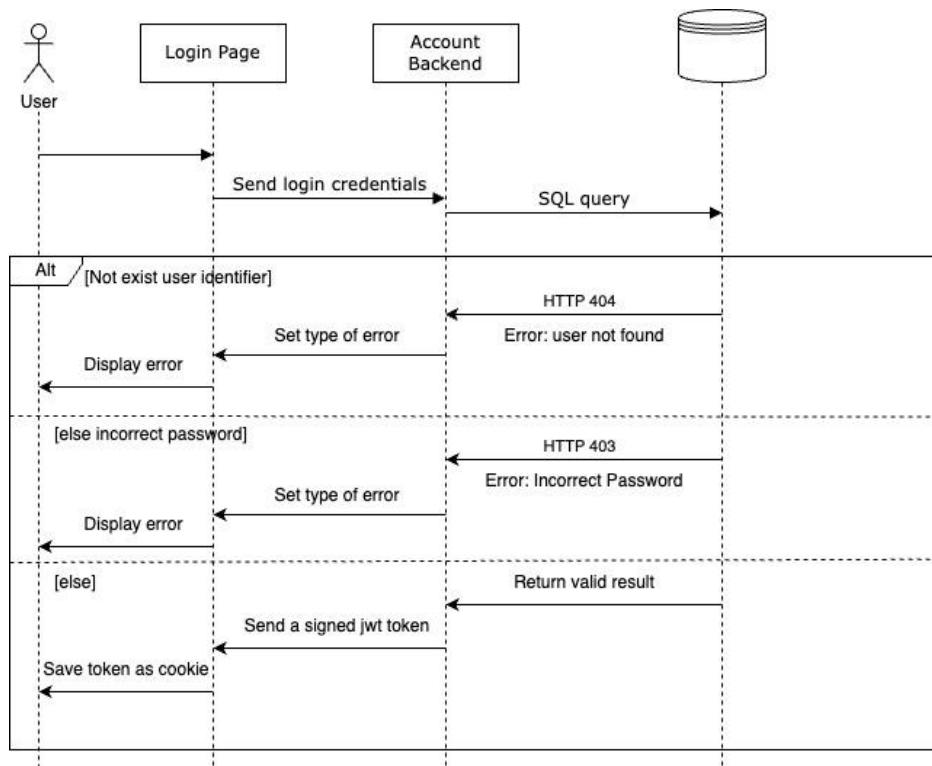


Figure 7.1.2.4 Sequence Diagram for user login

These tokens are used to verify the authenticity of a user as it is granted by the frontend only on a valid username with the corresponding password being entered. The token is stored in the cookies of the web browser. As shown in Figure 7.1.2.5, all requests that are made from the user will pass through middlewares to ensure that the user possesses a valid token.

Name	Value	Domain	Path	Expires / ...	Size
user_data	%7B%22user_id%22%3A160%2C%22is_admin%22%3Afal...	localhost	/	Session	59
_auth	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJ1c2VyX2RhGEi...	localhost	/	Session	193

Cookie Value  Show URL-decoded  
%7B%22user\_id%22%3A160%2C%22is\_admin%22%3Afal...%7D

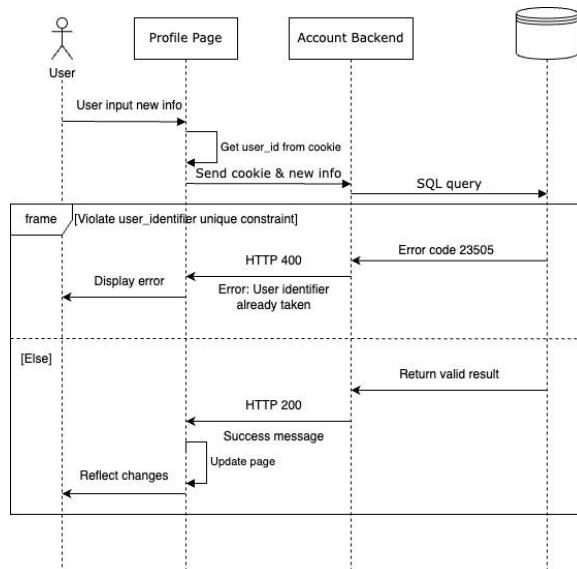
Figure 7.1.2.5 Tokens signed with hash and stored in cookies

Tokens are signed with a secret key to ensure its integrity and prevent attackers from easily forging one. Furthermore, introducing a one hour expiry for the token reduces the chances of unauthorised access.

### 7.1.3 Updating user information

In the usage of PeerPrep, there are two ways in which a user's information can be changed. The user themselves can change their information and an admin can grant administrator role to a user of their choice.

When a user decides to update their own information, a request is sent to the User Profile Backend, containing the changes proposed by the user. The User Profile Backend then verifies the token of the requesting party to be a genuine user via the middleware, and then makes the update if the verification is successful.



7.1.3.1 Sequence Diagram of Update user

When an admin gives the user administrator role, the middleware checks the admin's token to verify that he is an admin before allowing the request to proceed on successful verification.

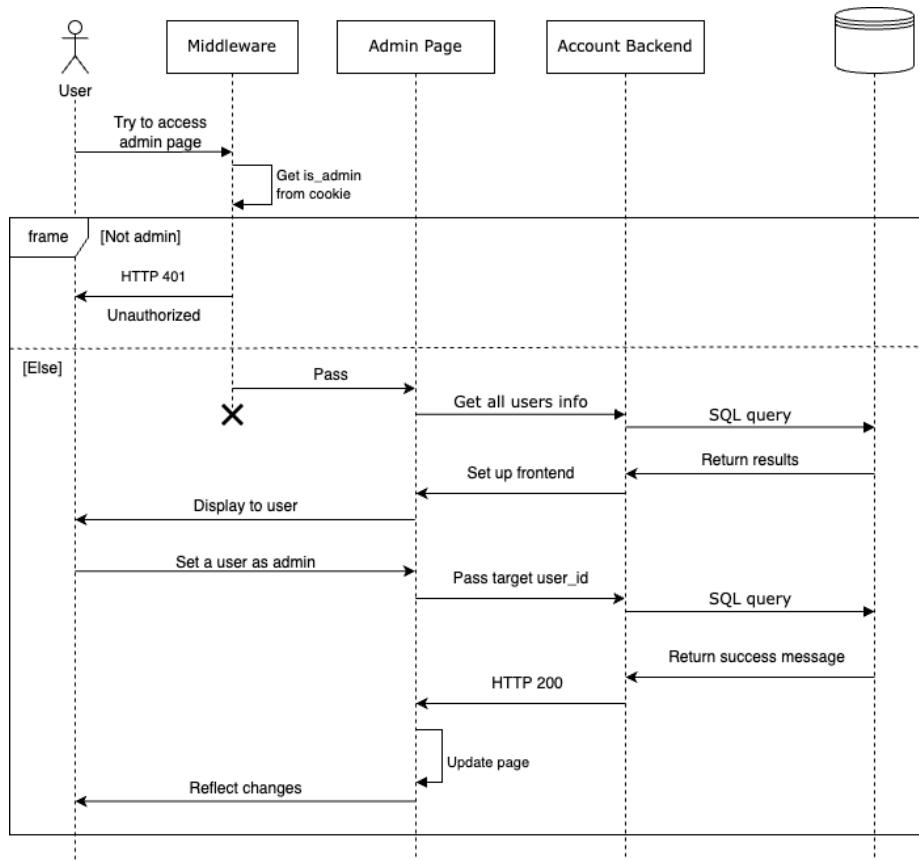


Figure 7.1.3.2 Sequence diagram for updating a user account

User Profile utilises RESTful API architecture for communication with other services. Any interaction with its database is done through the 9 routes that have been implemented as shown in figure 7.1.3.3.

Profile Management		
Authentication		
POST	/users/login	GET /users/users Get all users information, only admin can call
POST	/users/createUser	GET /users/userById?user_id = {user_id} Get user by id
PUT	/users/setUserAdmin?username = {username}	GET /users/userByName?username = {username} Get user by name
GET	/users/checkUserAdmin?username = {username}	PUT /users/userupdateUser
		PUT /users/deleteUser

Figure 7.1.3.3 User Profile Service routes

The two routes accessed through POST requests are used when a user first logs in or registers an account. Hence, they do not require the user to already have a token and are publicly accessible. All other endpoints are guarded by middlewares that check that the user is logged in or that the user is an admin (if required) by validating their token.

## 7.2 Question Service

The Question microservice communicates with the database containing PeerPrep's question repository. There is also a sub-collection within the Mongo Database that stores all existing categories. As we allow users to filter questions by their categories, we have limited the categories of questions to those in the Categories sub-collection that have been created by admins. The question service is in charge of supplying PeerPrep with questions and to provide questions to the collaboration backend during the collaborative session as well.

### 7.2.1 Overview

This service offers the following:

CRUD Operations on Questions and Categories, as well as retrieving questions by a specified complexity. Each of these are done via http request sent to the Question microservice, which utilises a router to route requests. There are 9 possible requests as shown in the figure 7.2.1.1.

Question		Category	
POST	/questions/addQuestion	POST	/questions/addCategory
PUT	/questions/updateQuestion	PUT	/questions/updateCategory
GET	/questions/getQuestions	GET	/questions/getCategory
GET	/questions/getQuestionByComplexity		
DELETE	/questions/deleteQuestion	DELETE	/questions/deleteCategory

Figure 7.2.1.1 Question Service routes

### 7.2.2 Enforcing Authorization

All routes that make changes (ie POST, PUT, DELETE) to the database require the user to have admin privileges. Requests to these routes will need the authorization token of an admin to be included in the header of the request. This is to prevent any malicious entities from tampering with our questions. Requests made to those routes will return a status code of 401 Unauthorised for users that do not possess the authorization token of an admin user.

The 3 other routes serving GET requests and do not make changes to the databases have been left publicly accessible to increase our outreach and benefit similar projects that require such questions without requiring them to make an account. The enforcement of admin details is demonstrated in the middleware as shown below.

```

const router = express.Router();

router.get("/", getQuestions);
router.get("/:complexity", getQuestionsByComplexity);
router.post("/", [checkLogin, checkAdmin], addQuestion);
router.put("/", [checkLogin, checkAdmin], updateQuestion);
router.delete("/:id", [checkLogin, checkAdmin], deleteQuestion);

export default router;

```

Figure 7.2.2.1 Middleware guarding questions in Question Service

```

const router = express.Router();

router.get("/", getCategorys);
router.post("/", [checkLogin, checkAdmin], addCategory);
router.put("/:oldName", [checkLogin, checkAdmin], updateCategory);
router.delete("/:name", [checkLogin, checkAdmin], deleteCategory);

export default router;

```

Figure 7.2.2.2 Middleware guarding categories in Question Service

Below is a picture of the authorization path and how requests are handled by the service:

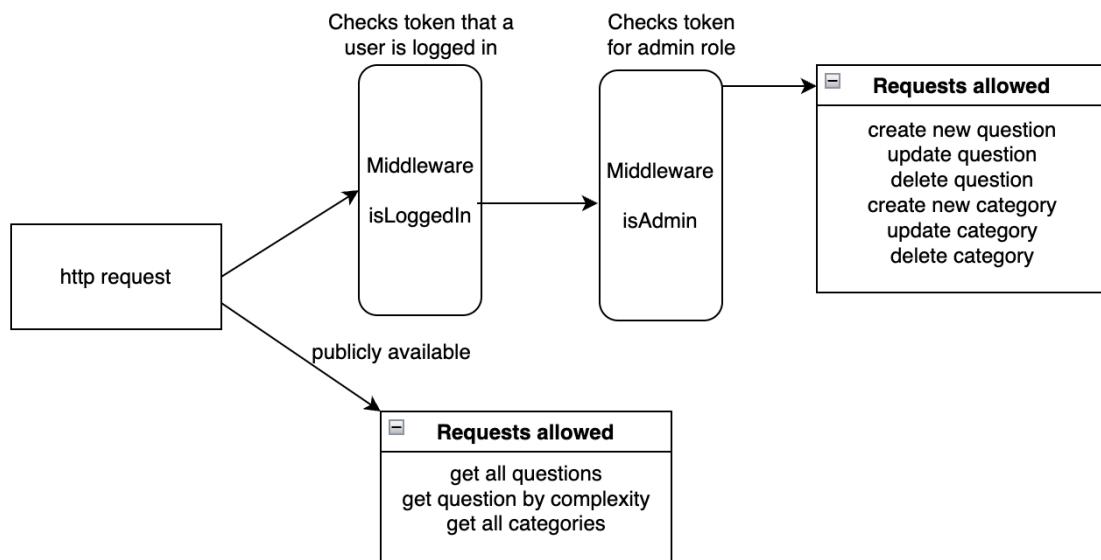


Figure 7.2.2.3 Authorisation path of HTTP requests

The middleware `checkAdmin` will check the authorization token for the value `isAdmin`. If it is set to true, then the user is an admin and requests to all paths are allowed and will be forwarded to the corresponding controllers.

### 7.2.3 Implementation

This shows the implementation of handling those the requests that create, update and delete questions and categories:

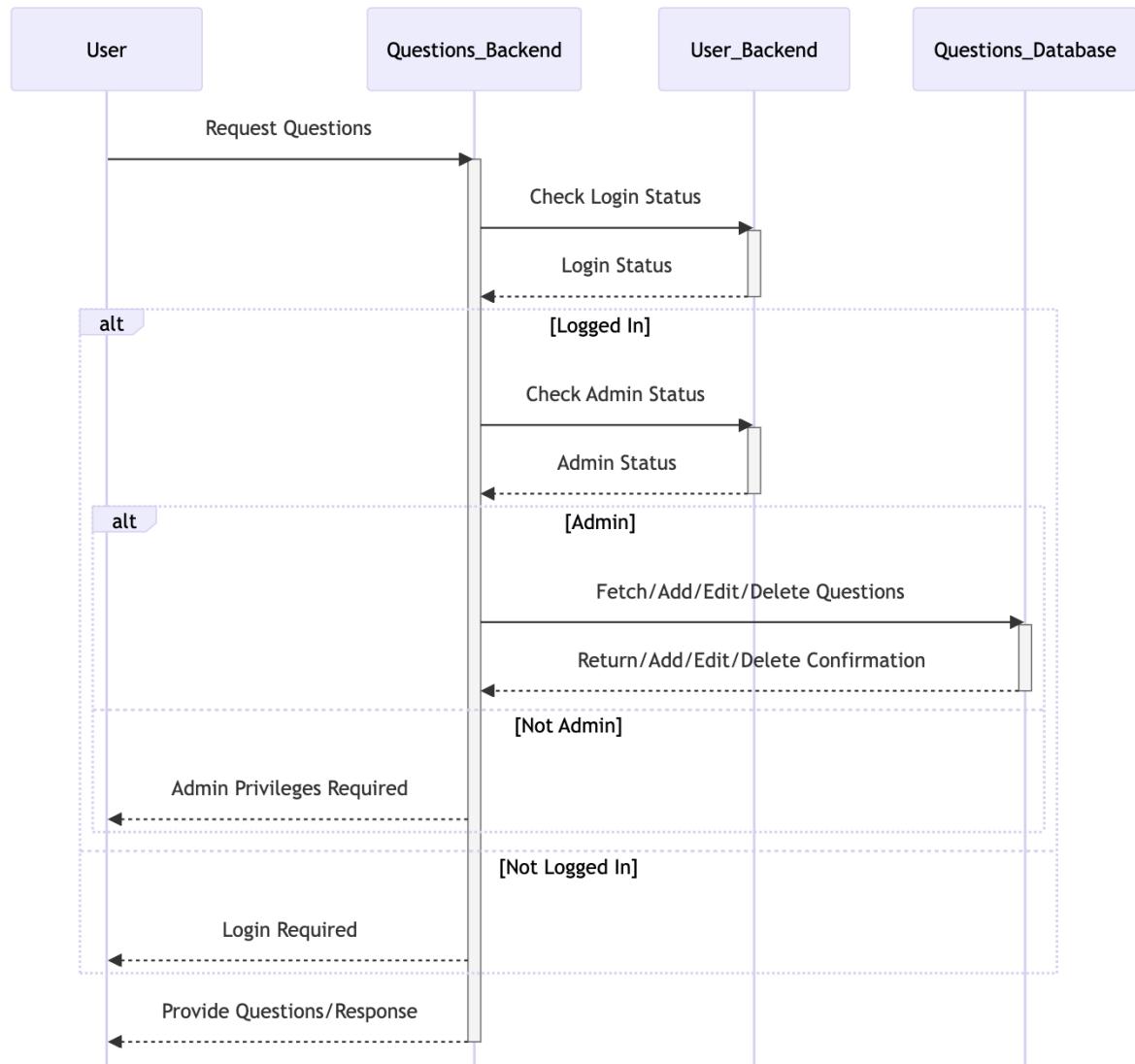


Figure 7.2.3.1 Sequence diagram for Question Service

For the implementation of requests that only read from the database, it is the same as the above, without the middlewares, so requests go directly from user to the controller in the backend which then retrieve information from the database and return it to the user.

## Questions Model

Each question has an ID, title, description, categories and complexity. When creating a question, all fields other than ID must be filled up by the admin user. As stated above, we have limited the categories of a question to those in the Categories sub-collection. We also defined duplicate questions as questions that have similar titles, hence we do not allow the creation of a new question that shares the same title with an existing question.

### 7.2.4 Enhancing Flexibility of Question Service

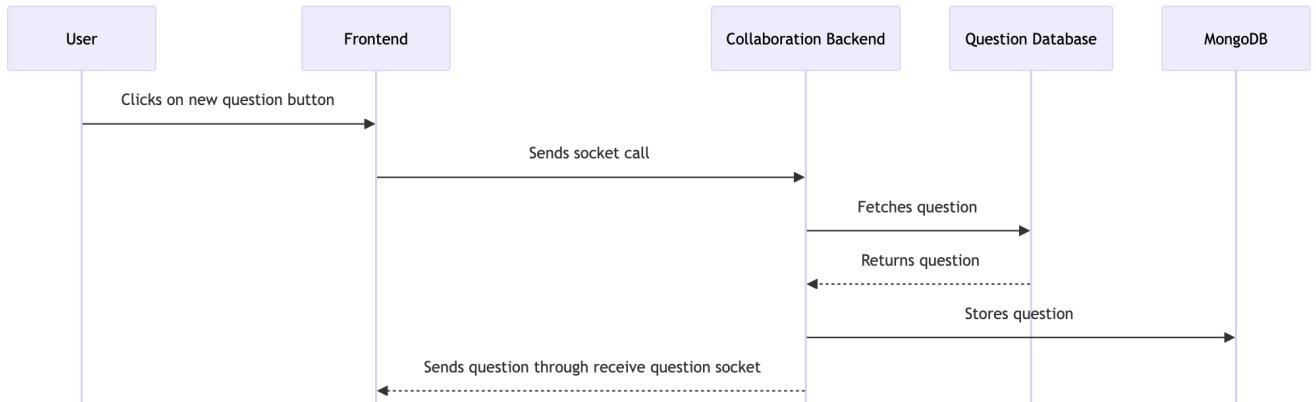


Figure 7.2.4.1 Sequence diagram for question on fly

To enable flexibility, we chose to allow users to request for a new question of the same difficulty. When a user initiates a new query by clicking the 'new question' button on the front end, a socket call is sent to the collaboration backend. This backend then retrieves the question from the question database. Once the question is obtained, it is stored in the MongoDB database. Additionally, the question is transmitted back to the front end through a 'receive question' signal.

We chose to store the question in a MongoDB database instead of local storage so that the questions on both users' interfaces are always in sync and the questions are persistent.

## 7.3 Matching Service

Matching Service is responsible for matching two users who have selected the same question complexity preference together. A Postgres database is used to keep track of currently unmatched users and acts as a queue. When a user trying to match has the same question complexity preference as an unmatched user in the queue, RabbitMQ is used for communication between these two users.

When a user tries to match, they connect to the matching service using Socket.io and pass their token to the matching service. The matching service then verifies that

the user is indeed a valid user. If this is not the case then the matching request is rejected as shown below.

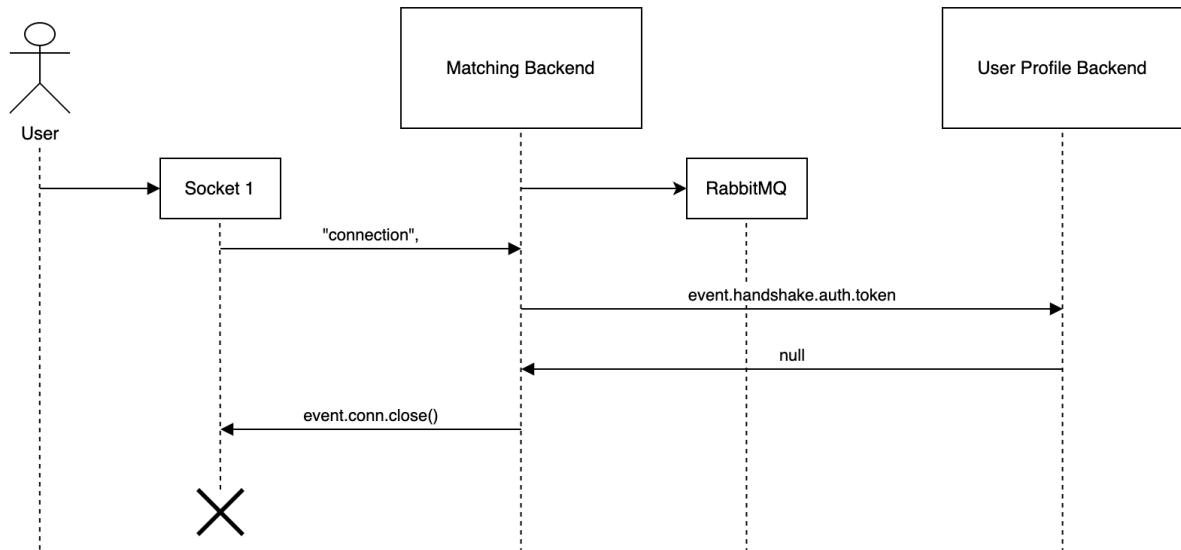


Figure 7.3.1 Sequence Diagram of Matching Service failure due to invalid user

After verifying a valid user, it will check the database which acts as a Matching Queue for any unmatched user with the same complexity preference. Should there be such an unmatched user, these two users are matched and Matching Service will query Communication Service for a room Id. A randomly generated room Id is returned and the Matching service passes the room Id to the user A that retrieved the user B from the database, and then user A will then pass a message via the message Queue of user B.

Since the query to Communication Service is triggered by the user who removed the previously unmatched user from the queue, the room Id is immediately available to him. RabbitMQ is required to communicate the room Id to the user who was removed from the queue.

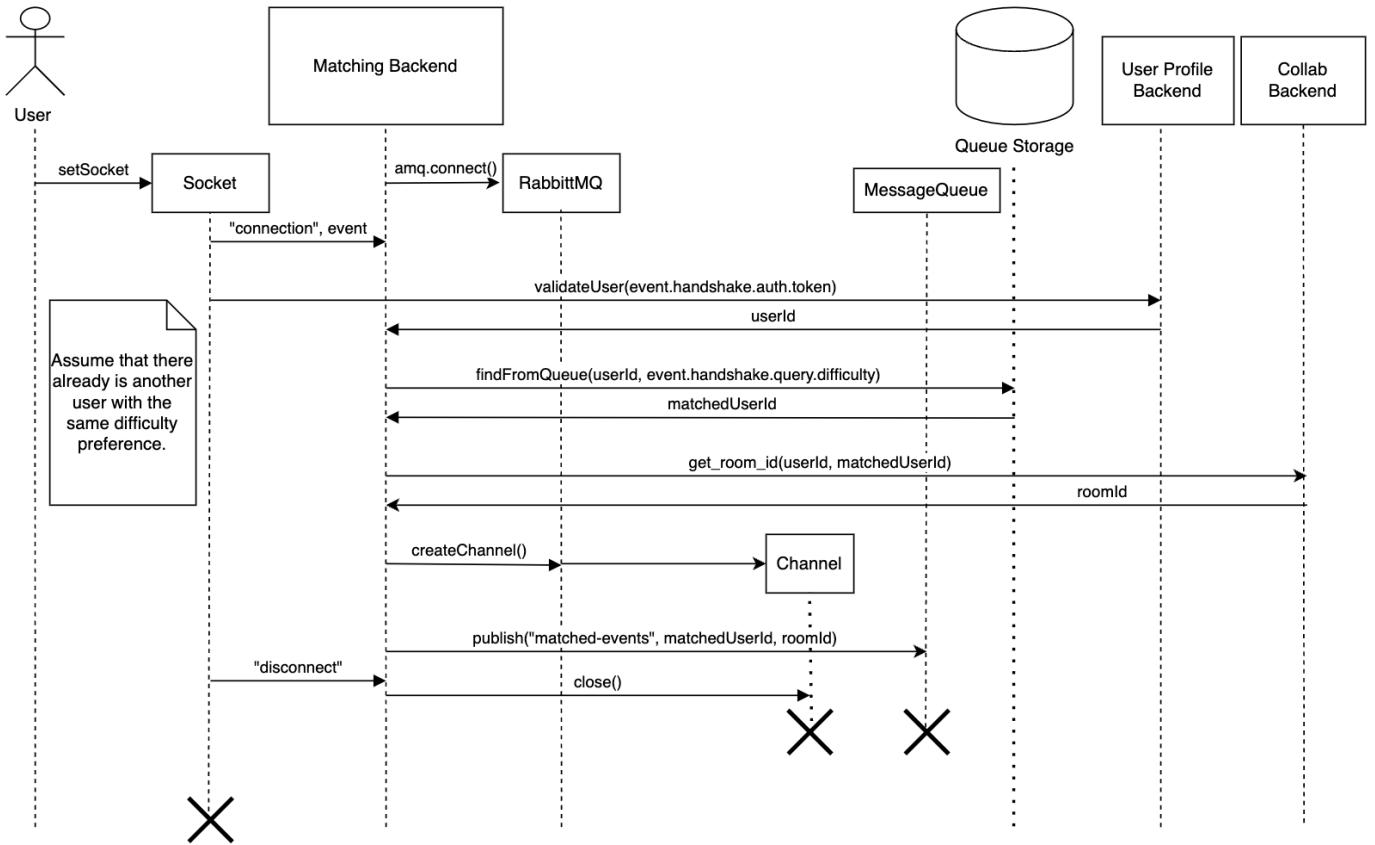


Figure 7.3.2 Sequence Diagram of Matching Service successful match

However, if no such unmatched user exists, Matching Service puts him in the database queue. Where he will wait for another user to match with him and notify him of the matching.

To fulfil PER-NFR-2 and prevent users from waiting too long for a match, we included a timeout of 15s. If a user remains unmatched in the database queue for 15s, Matching Service will remove this user from the database queue and inform them of the unsuccessful matching.

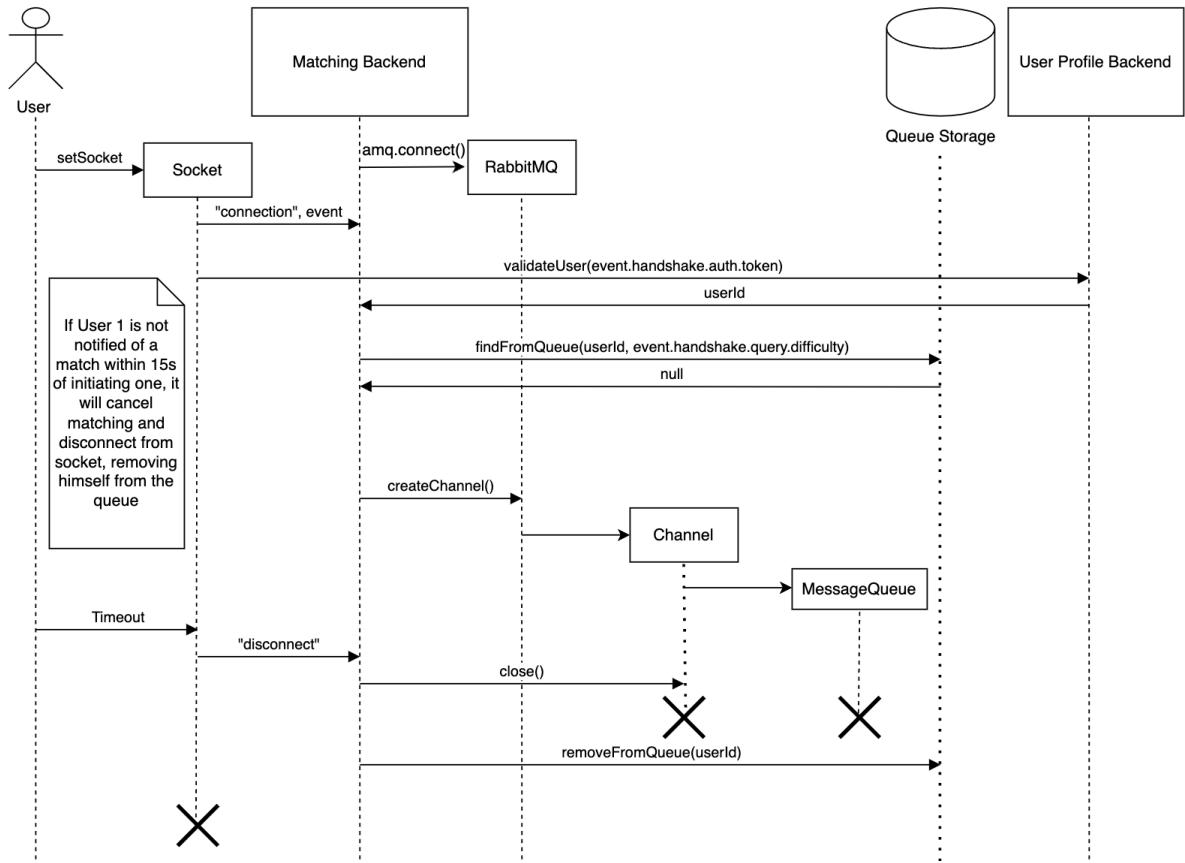


Figure 7.3.3 Sequence Diagram of Matching Service failure due to no match

Our decision to use a Postgres database was to prevent race conditions where an unmatched user is matched to two other users who happened to submit their complexity preferences at the same time due to the atomic nature of Postgres databases' operations. If a user trying to match checks the database and sees another unmatched user of the same complexity preference, he will then remove the unmatched user from the database, get a room Id from Collaboration Service and use RabbitMQ to inform that user of the room Id so that both users can proceed to this shared room. The ACID property of Postgres ensures that if two users query the database at the same time, only one of them will successfully retrieve the user from the database.

## 7.4 Collaboration Service

The collaboration service is in charge of allowing two users to share a collaborative code editor that reflects changes from one user to both users.

Upon a successful matched pair, a new, random room Id is generated by the collaboration backend and returned to the matching service to pass to both of the matched users. The room Id is generated using UUID4 and has  $2^{128}$  possible combinations, making it resistant to brute force and more secure.

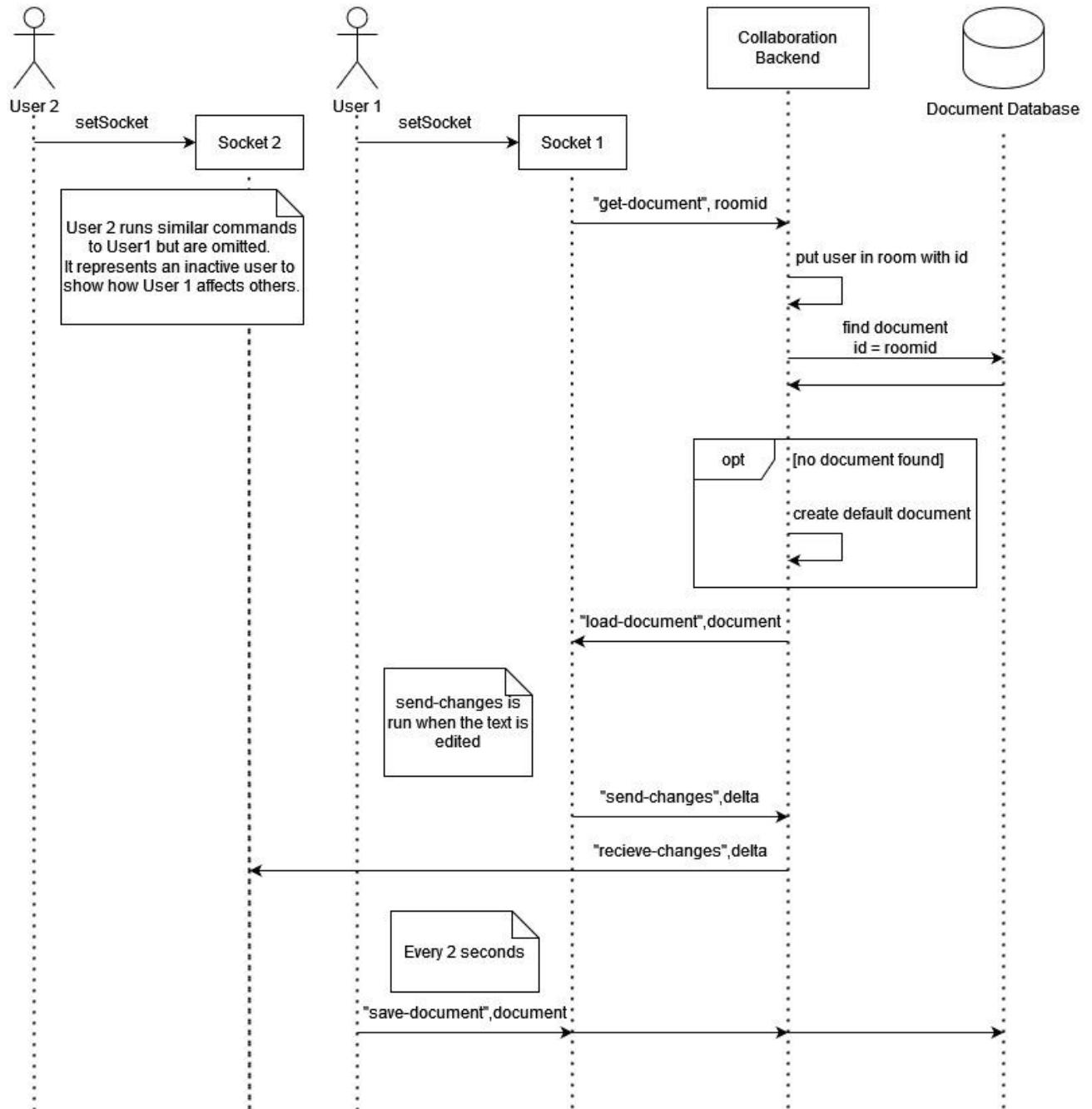


Figure 7.4.1 Editor interaction with Collaboration Backend

In the Room.js page, users open a socket connection to the collaboration backend. A seamless collaborative experience is a priority as per our NFR, so we use sockets, which have a persistent connection for the duration of the collaboration, emit events faster instead of the HTTP requests used in other services.

For the editor, the user starts by asking the socket for a document - an object where we store the code. The backend will check the database for a document associated with the room Id. If it does not exist, an empty document will be sent instead.

When the user edits the code, the changes - referred to as deltas - are sent to the socket and broadcast to all other users in the same room, which will be listening for the changes.

Periodically, the editor contents will be saved to the database to allow users to come back to the code after a disconnect. However, the document will be deleted after a room has been abandoned for 90s to save storage space.

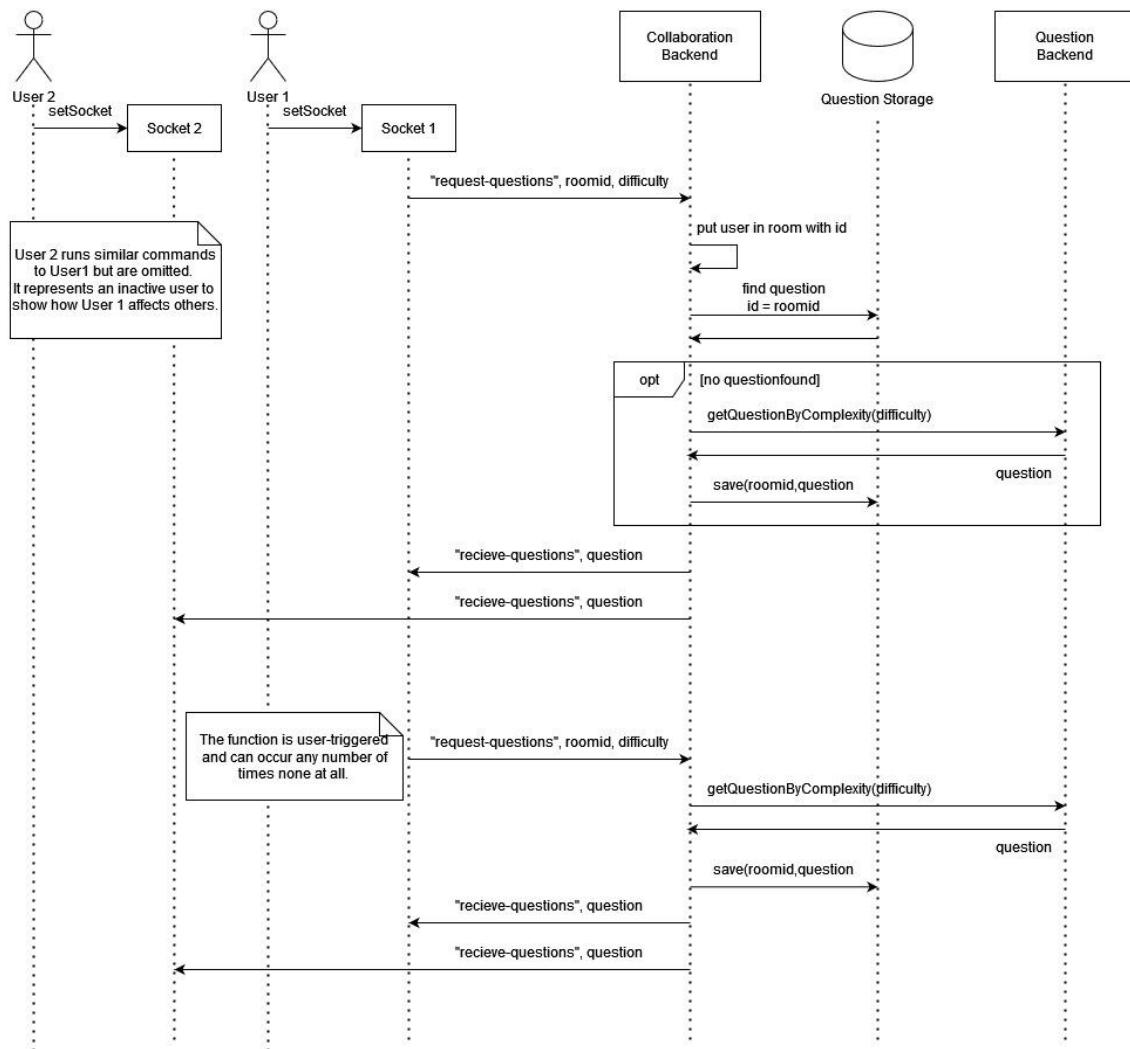


Figure 7.4.2 Sequence diagram for users joining a room

The question supplied to the users are managed in a similar manner. The collaboration service will check its database for a question, but instead of giving a default question, it queries the Question Service instead.

At any time, the user may request a new question, and the backend will fetch another question.

The database of choice was MongoDB. Being a NoSQL database, it can store complex data types. It allows the questions and documents we store to be more flexible, and the database to adapt easily.

One unintended feature of using sockets, is that the collaboration service can support more than 2 users. This feature could be expanded upon in future versions, but it is not within the current scope.

## 7.5 Communication service

The communication service is in charge of helping two users to communicate via a video call whilst in the room page. This communication is set up by using sockets between the user and the communication backend. Below shows the implementation of calling and ending a call. The arrows from each socket show the signals emitted by that socket, and the signal name is given in quotes.

### 7.5.1 Implementation of starting call

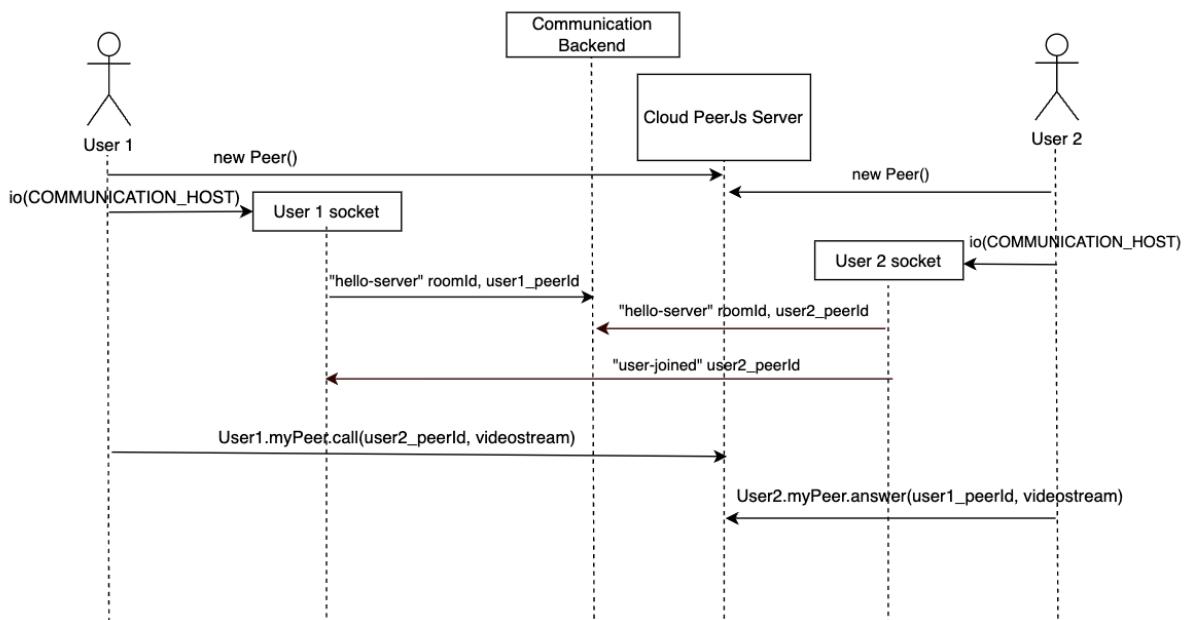


Figure 7.5.1 Sequence diagram of starting a call

When a user initiates a call, they create a socket connection to the communication backend. The communication backend lets the socket join a room based on the room Id provided which should be unique to both users in the room. When another user starts the call as well, their socket will join the same room given by the same room Id. Next, upon joining the room they will broadcast their peerID to the socket of the user already in the room. The first user then calls the second user via the peerJS server to send their video stream, and the second user answers the call by sending its own stream back as well. Each user then renders both videos on their screen.

### 7.5.2 Implementation of ending a call

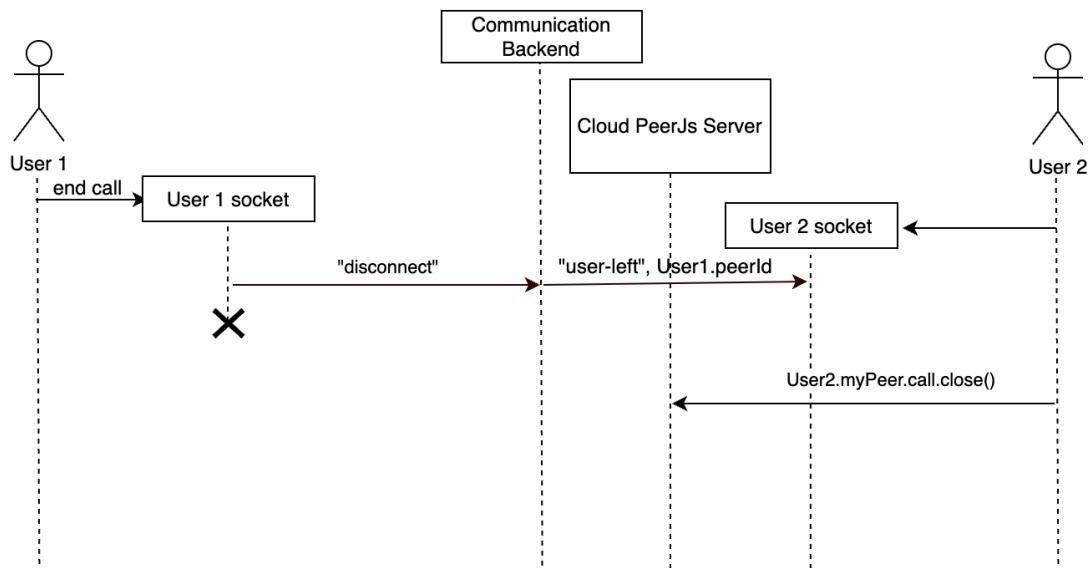


Figure 7.5.2 Sequence diagram of Ending a call

When user 1 decides to end a call, user 1 disconnects their socket from the communication backend and the user's socket is no longer active. The communication backend also removes user 1's socket from the room. This communication backend also then broadcasts to user 2 with a "user-left" signal and this is received by user 2's socket, which will then end the call which was transferring the media stream between both users. For user 1, both videos are then removed from the screen while for user 2, user 1's video will disappear.

We have decided that the communication service should be separate from the collaboration service as the collaboration service was meant to facilitate the collaborative editor communication between both users. This is to reduce the load handled by the collaboration service, and gives the communication a responsibility of setting communication between two parties in the same room.

## 7.6 Question History Service

The Question History Service manages users' previous question attempts in the collaboration sessions, for future review.

When a user makes an attempt during the collaborative session, they can choose to submit their attempt. Submission of their attempt then will result in an API call to the Question History backend to save the attempt details. The implementation of the saving of the attempt is shown below.

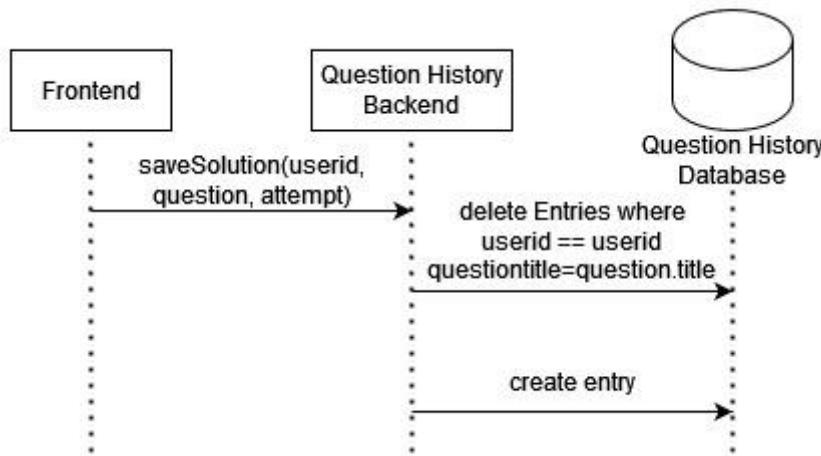


Figure 7.6.1 Saving entries in the Question History Service

When saving a solution, the service also records the user ID and a copy of the question into the database. The latter is useful if the question is changed, allowing the user to access the version of the question they attempted. The user and question title also serve as unique identifiers, and we only permit the user to store their latest attempt for a single given question. A workaround exists by writing multiple solutions before saving.

The point of the Question History backend is for users to be able to also view their attempt history, and so a user can do so by heading to the Question History page. Upon visiting the page, the Question History backend retrieves the entries for question attempts containing the user's userID and returns it to the user. This is demonstrated in the figure below:

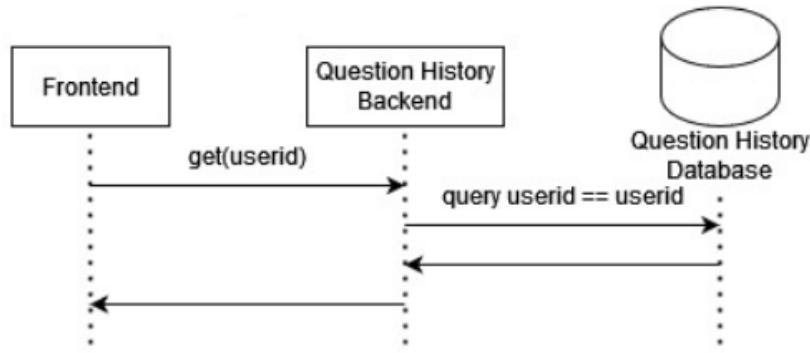


Figure 7.6.2 Viewing entries in the Question History Service

Upon user unregistration, the Question History backend also deletes all information related to that user. This is done by deleting all the attempts in the Question History with the associated user id. The implementation is shown below:

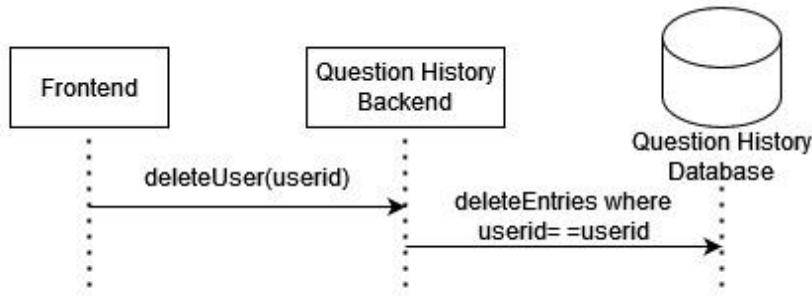


Figure 7.6.3 Sequence diagram of Question History Service reaction to a deleted user

In line with the principles of microservices architecture, we have decided to isolate the Question History backend from the Question Backend and User Profile Backend. This separation is driven by two key considerations:

- Service Autonomy:** The Question History represents a distinct domain of functionality. In a microservices architecture, it is essential to encapsulate different functionalities into separate, loosely coupled services. This ensures each service is responsible for a single aspect of the application, enhancing modularity and making the system more scalable and easier to maintain.
- Reduced Coupling and Increased Cohesion:** Integrating the Question History with either the Question Backend or User Profile Backend would increase coupling, violating the microservice principle of high cohesion within services and low coupling between services. By keeping Question History as a standalone service, we can maintain a cleaner, more organised codebase with improved readability and easier maintenance that is more resilient to changes.

By adhering to these microservice design principles, we can build a system that is more scalable, maintainable, and adaptable to evolving requirements.

## 7.7 Testing of Backend Services

In order to test our backend services, Postman and ThunderClient proved to be vital in our testing. Along with MongoDB Atlas and Postgres Database extension in VSCode, we could then verify whether entries are stored successfully.

The figure consists of two side-by-side screenshots. On the left, the Postman interface shows a successful POST request to `http://localhost:8000/api/questions/`. The body contains a JSON object with fields `id`, `title`, `categories`, `complexity`, and `description`. The response status is 200 OK with a duration of 129 ms and a size of 265 B. On the right, the MongoDB Atlas interface shows the `questions` collection with one document added. The document has fields `complexity` (value "EASY"), `description` ("Given a target value and a list of numbers, find two numbers that add ..."), `_id` (value `ObjectId('65531f790924b594a2b6ef82')`), `id` (value 4), `title` ("Travelling Salesman Problem"), `description` ("Find him"), `categories` (array [ "BASIC" ]), `complexity` (value "HARD"), and `_v` (value 0).

Figure 7.7.1 Question successfully added to the database with valid token

The screenshot shows the Postman interface with a failed POST request to `http://localhost:8000/api/questions/`. The body is identical to the successful request in Figure 7.7.1. The response status is 400 Bad Request with a duration of 4 ms and a size of 314 B. The error message in the body is "1 You do not posess login token. Please sign in".

Figure 7.7.2 Question failed to be created due to lack of token

## 8. Frontend

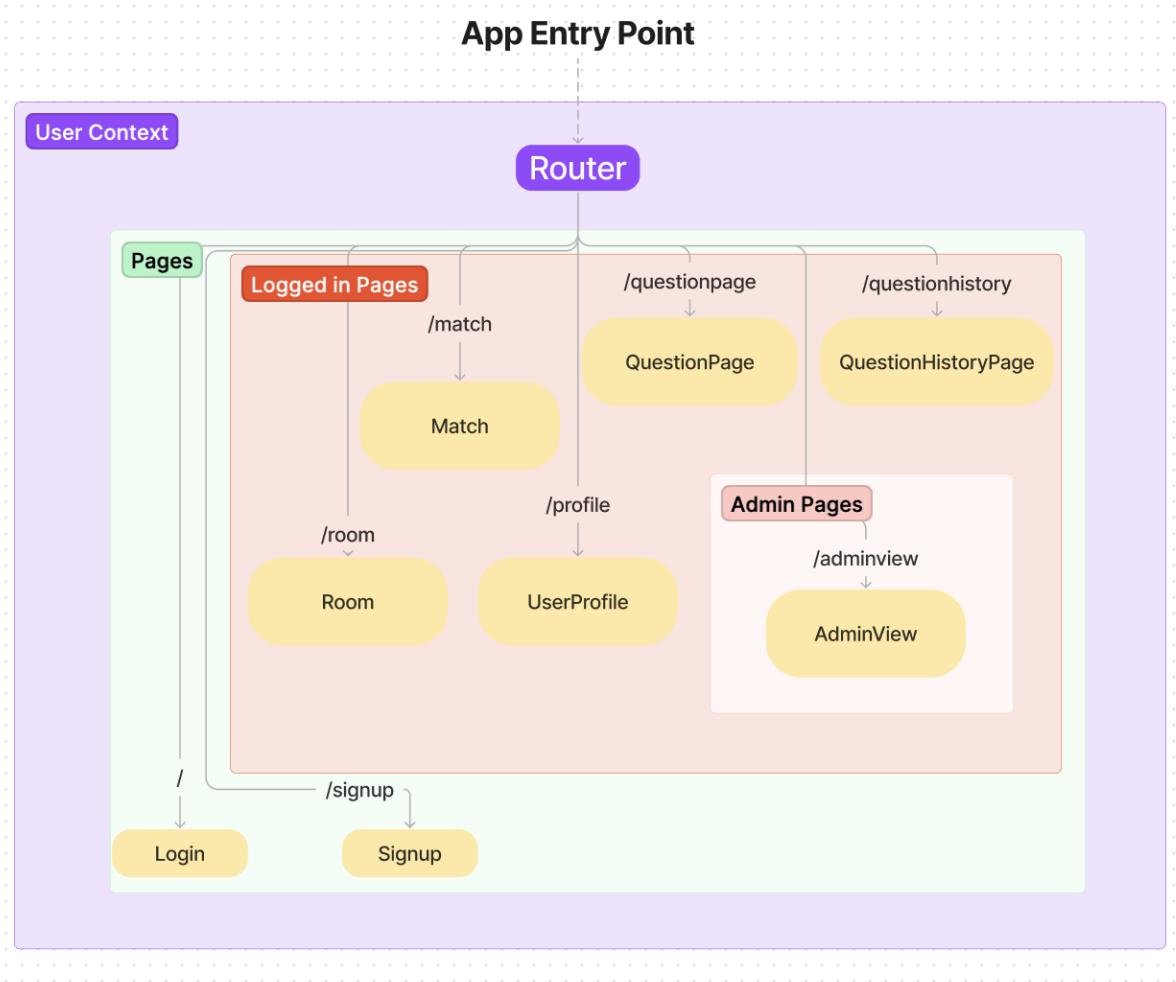


Figure 8.1 High-level frontend component layout of frontend

The frontend relies on React. Its component-based architecture breaks down the UI into smaller reusable components. This allows our codebase to be more modular and simpler. The components exhibit versatility, having their own state and employing React Hooks to obtain behaviours. React uses a Document Object Model – like structure, which allows for CSS integration. In addition, React has an active community that offers many easy to integrate modules. For instance, Material UI was used for its extensive component library, and Quill a text editor with the ability to record changes.

In designing the architecture of this web application, we made three key considerations to ensure that it adheres to robust software engineering principles:

## **1. Security-First User Context Management**

In architecting the application, we prioritized security above all. We did not want unauthorized users to access or modify our data. By leveraging cookies for storing user context—specifically, the user's login and admin states—we mitigate security risks. The cookies are set with `HttpOnly` and `Secure` attributes, which are crucial for preventing access through client-side scripts and ensuring transmission only over secure, encrypted connections. This decision embodies a security-first mindset, crucial for user trust and regulatory compliance.

## **2. Modular Page Architecture:**

Adopting a modular approach for page components directly aligns with the principles of scalable and maintainable software design. This modularity facilitates parallel development and testing, enabling agile workflows and reducing interdependency issues. It's a strategic choice that anticipates the application's growth and evolution, ensuring that enhancements or fixes can be implemented with minimal impact on the overall system.

## **3. Centralised Routing with Separation of Concerns:**

Implementing a centralised Router was a deliberate decision to abstract the navigation logic from the application's business logic. This aligns with the separation of concerns principle, fostering a codebase that is easier to understand, maintain, and extend. By directing traffic based on user state, the Router simplifies the user experience while decoupling page components from the underlying routing mechanism, thus enhancing the application's robustness and flexibility.

## 9. DevOps

### 9.1 Testing

Testing is a crucial aspect of the development of PeerPrep, ensuring that the code behaves as expected and identifying potential issues before they reach production.

#### 9.1.1 Testing Frameworks and Tools

##### Mocha & Chai

Mocha is a widely-used JavaScript test framework that provides a flexible and feature-rich environment for testing asynchronous code. Chai is an assertion library that works seamlessly with Mocha, offering various assertion styles.

The sample code utilises Mocha and Chai to structure and assert the behaviour of the addQuestion controller.

```
describe('addQuestion Controller', () => {
  afterEach(() => {
    sinon.restore()
  })

  it('should add a new question', async () => {
    const req = [
      body: {
        id: 1,
        title: 'Test Question',
        description: 'Test Description',
        categories: ['Category1', 'Category2'],
        complexity: 'Medium',
      },
    ]
    const res = {
      send: sinon.stub(),
    }

    // Stub Question model's findOne method to simulate no existing title
    const findOneStub = sinon.stub(Question, 'findOne').resolves(null)

    // Stub Question model's save method
    const saveStub = sinon.stub(Question.prototype, 'save').resolves()

    await addQuestion(req, res)

    // Assertions
    expect(findOneStub.calledOnceWithExactly({ title: 'Test Question' })).to.be.true
    expect(res.send.calledWith('Success')).to.be.true
  })
})
```

Figure 9.1.1.1 Test for Question backend using Mocha and Chai

##### Sinon

Sinon is a powerful library for creating spies, stubs, and mocks in JavaScript tests. In the provided code, Sinon is employed to create stubs for methods of the Question model, allowing controlled testing scenarios.

## **NYC\_Istanbul and c8 v1.3**

NYC (Istanbul) is a code coverage tool, and C8 is a fast coverage tool built on top of Istanbul. These tools help measure the effectiveness of test suites by providing insights into which parts of the code are covered by tests.

### **9.1.2 Testing Strategy**

Our test cases demonstrate a comprehensive testing strategy, covering positive scenarios, error handling, and edge cases related to duplicate titles. The use of Sinon allows for controlled testing, and Mocha and Chai facilitate expressive and readable test cases.

### **9.1.3 Code Coverage Analysis**

To assess the effectiveness of the test suite, code coverage tools like NYC\_Istanbul and C8 are employed. These tools provide insights into which parts of the code are covered by tests, aiding developers in identifying areas that require additional testing. In our testing, we implemented a check for 70% loc.

In Figure 9.1.3.1, we show that even if all tests pass, test coverage can still fail, thus signalling to developers to include more tests.

```
"test-with-coverage": "c8 --reporter=text mocha --exit && c8 check-coverage --lines 70"
"test-with-coverage": "nyc --reporter=text mocha --exit && nyc check-coverage --lines 70"
```

Figure 9.1.3.2 specification of coverage rules to be followed

```
(base) [REDACTED] MacBook-Air-2 user_profile_backend % npm run test-with-coverage
> user_profile_backend@1.0.0 test-with-coverage
> nyc --reporter=text mocha --exit && nyc check-coverage --lines 70
User server is running on port 4000

Check User Admin
  ✓ should check if a user is an admin
  ✓ should check if a user is not an admin
  ✓ should handle user not found

User Server API
  ✓ should get a user by id (70ms)
  ✓ should login a user with the correct username and password (114ms)
  ✓ should login a user with the correct email and password (110ms)
jwt must be provided
  ✓ cannot get a user by the id
  ✓ should not login a user with the wrong password (100ms)
  ✓ should not login a user with the wrong username

Set User Admin
  ✓ should set a user as an admin
  ✓ should handle user not found

Update User Information
  ✓ should update user information successfully
  ✓ should handle user not found
  ✓ should handle unique username constraint violation
  ✓ should handle password complexity constraint violation
  ✓ should handle internal server error

16 passing (430ms)

File          | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #
---           | ---     | ---      | ---     | ---     |
All files    | 68.93   | 52.04   | 51.85  | 61.41  |
user_profile_backend | index.js | 93.33   | 50       | 50      | 100
user_profile_backend/controller | checkUserAdmin.js | 93.33   | 50       | 50      | 100
user_profile_backend/controller | createUser.js | 58.23   | 55.81  | 57.14  | 58.57
user_profile_backend/controller | deleteUser.js | 9.09    | 0       | 75      | 100
user_profile_backend/controller | getUser.js | 67.79   | 50       | 55.37  | 67.79
user_profile_backend/controller | updateAdmin.js | 64.15   | 66.66  | 100     | 84.15
user_profile_backend/controller | updateUser.js | 76.74   | 80.55  | 100     | 76.74
user_profile_backend/database | db.js | 100     | 100     | 100     | 100
user_profile_backend/middleware | tokenUtils.js | 33.33   | 20      | 25      | 33.33
user_profile_backend/middleware | validateAdmin.js | 85.71   | 100     | 56      | 85.71
user_profile_backend/middleware | validateUser.js | 28.57   | 25      | 0       | 28.57
user_profile_backend/routes | user-profile-router.js | 19.84   | 16.66  | 0       | 19.84
user_profile_backend/routes | user-profile-router.js | 100     | 100     | 100     | 100
user_profile_backend/routes | user-profile-router.js | 100     | 100     | 100     | 100

ERROR: Coverage for lines (61.41%) does not meet global threshold (70%)

```

Figure 9.1.3.1 Test coverage failure despite every test cases passing

## 9.2 Local development

For local development, we utilised docker and docker compose to provide all users with the same environment. This meant that there was a consistent working environment for all members and that as long as a member had downloaded docker on their device, they would be able to work on the application. The use of Docker is well suited for the microservices architecture as we are able to develop, test and run individual microservices locally. After defining the individual Dockerfiles for each service, using docker-compose, we are able to run multiple containers on our local device, with a virtual network that allows the different services to communicate with one another. We can also monitor when a container is not running to see which microservice is failing.

```
[+] Building 84.3s (61/61) FINISHED
--> [user_profile_backend internal] load .dockerignore
--> transferring context: 66B
--> [user_profile_backend internal] load build definition from Dockerfile
--> transferring dockerfile: 151B
--> [matching_backend internal] load metadata for docker.io/library/node:18
--> [history-backend internal] load build definition from Dockerfile
--> transferring dockerfile: 156B
--> [history-backend internal] load .dockerignore
--> transferring context: 66B
--> [communication-backend internal] load .dockerignore
--> transferring context: 2B
--> [communication-backend internal] load build definition from Dockerfile
--> transferring dockerfile: 157B
--> [mongodb_container internal] load build definition from Dockerfile
--> transferring dockerfile: 61B
--> [mongodb_container internal] load .dockerignore
--> transferring context: 2B
--> [mongodb_container internal] load metadata for docker.io/library/mongo:latest
--> CACHED [mongodb_container 1/1] FROM docker.io/library/mongo
--> [mongodb_container] exporting to image
--> exporting layers
--> writing image sha256:cdfa9d2de31af2d5db4062e731f2e66a6680cf349717c66c1ddc02f5ed14e9ca
--> naming to docker.io/library/mongo:latest
--> [collab-backend internal] load .dockerignore
--> transferring context: 66B
--> [collab-backend internal] load build definition from Dockerfile
--> transferring dockerfile: 157B
--> [user_profile_backend auth] library/node:pull token for registry-1.docker.io
--> [matching_backend 1/5] FROM docker.io/library/node:18@sha256:7ce8b205d15e30fd395e5fa4000bcdf595fcff3f434fe75822e54e82a5f5cf82
--> [communication-backend internal] load build context
--> transferring context: 209.84kB
--> [collab-backend internal] load build context
--> transferring context: 70.27kB
--> [history-backend internal] load build context
--> transferring context: 63.90kB
```

Figure 9.2.1 Local deployment using Docker Compose

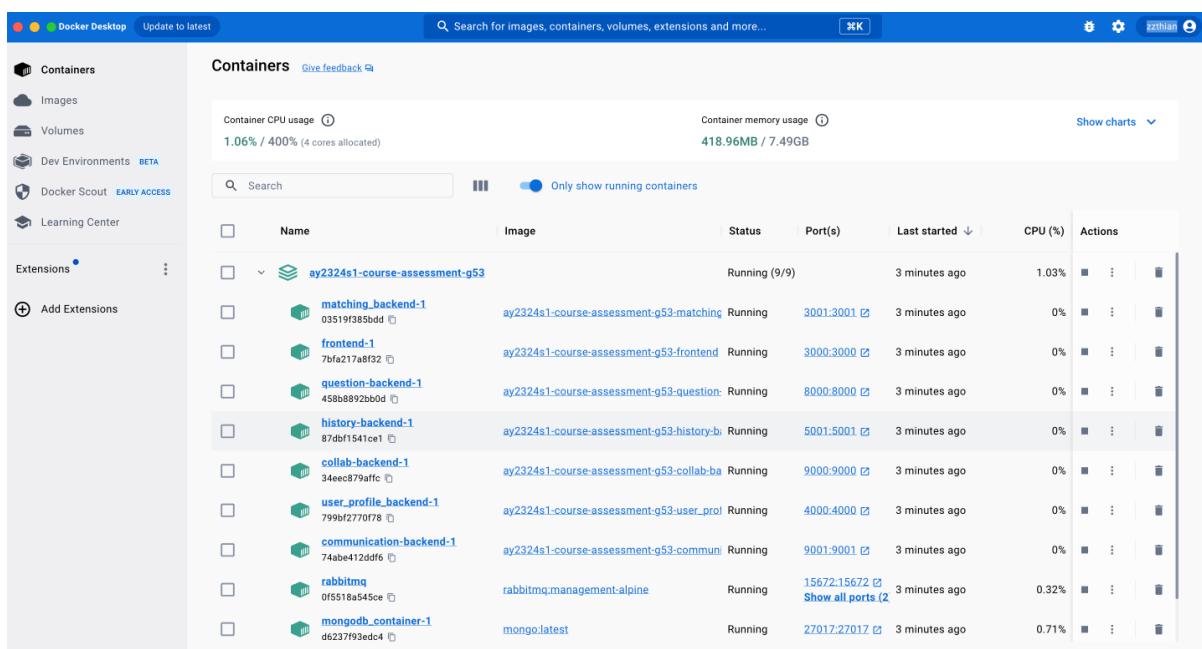


Figure 9.2.2 Screenshot of Docker Desktop

Each dockerfile specifies how an image should be built. Similar dockerfiles are used in the deployment on AWS ECS which means that our local development closely replicates the deployment environment and so any issues can be dealt with in the local environments before the deployment online. Likewise, any issues faced with the online deployment will be easily replicated locally to be dealt with.

```

frontend > 🎨 Dockerfile > ⚙ FROM
  1  FROM node:18
  2  WORKDIR /src/app
  3  COPY package*.json ./
  4  RUN npm install
  5  COPY . .
  6
  7  RUN npm run build
  8  RUN npm i -g serve
  9  EXPOSE 3000
10  CMD ["serve", "-s", "build"]

```

Figure 9.2.3 Sample Dockerfile

## 9.3 Deployment

We deployed our application on AWS Elastic Container Service (ECS) which is a container management service. Below are some concepts that are relevant to ECS.

### 9.3.1 Task and Task definition

In order to run our containers on ECS we need to define task definitions for each service which specifies key information about the containers we want to run. This includes CPU resources, port mappings, Container images and number of containers to be used for a single service (if more than one are needed). We provide our task definitions in our github repository so that it is visible to our workflows for Continuous development. Like how the container is an instance of an image, a task is a running instance of the task definition. Thus the task is the environment and containers that are being maintained and run by AWS.

### 9.3.2 Elastic Container Registry (ECR)

ECR is a registry that stores Docker container images much like DockerHub. Images can be pulled when a task is being provisioned as long as the task definition specifies the repository from which the image can be found.

### 9.3.3 Overview of deployment process

PeerPrep has one frontend service and six backend microservices. We create an ECS service for each of these seven services altogether. The services become publicly available through a public facing load balancer that is shared by all of the services. In order to deploy one service, the following steps take place:

1. Build container image of relevant service and push to a repository in ECR
2. Create task definition that identifies which container image to pull for the task
3. Create service based off of the task definition in step 2

4. Create target group associated to task in EC2 dashboard and add target group to application load balancer (ALB)
5. Configure ALB rules according to service's needs

## 9.4 Monitoring of deployed services

Each time a new version of the application is deployed, we can check the status of the deployment and whether each task is running successfully from the AWS console. As seen in figure 9.4.1, all tasks are running successfully.

<input type="checkbox"/>	Service name	Status	ARN	Service...	Deployments and tasks	Last...
<input type="checkbox"/>	Peerprep-Question-Backend	Active	arn:aws:ec...	REPLICA	<div style="width: 100%; background-color: #2e7131; height: 10px;"></div> 1/1 tasks ru...	
<input type="checkbox"/>	Peerprep-Matching-BE	Active	arn:aws:ec...	REPLICA	<div style="width: 100%; background-color: #2e7131; height: 10px;"></div> 1/1 tasks ru...	
<input type="checkbox"/>	PeerPrep-FE	Active	arn:aws:ec...	REPLICA	<div style="width: 100%; background-color: #2e7131; height: 10px;"></div> 1/1 tasks ru...	
<input type="checkbox"/>	Peerprep-User-Backend	Active	arn:aws:ec...	REPLICA	<div style="width: 100%; background-color: #2e7131; height: 10px;"></div> 1/1 tasks ru...	
<input type="checkbox"/>	Peerprep-Collab-Backend	Active	arn:aws:ec...	REPLICA	<div style="width: 100%; background-color: #2e7131; height: 10px;"></div> 1/1 tasks ru...	
<input type="checkbox"/>	Peerprep-Comm-Backend	Active	arn:aws:ec...	REPLICA	<div style="width: 100%; background-color: #2e7131; height: 10px;"></div> 1/1 tasks ru...	
<input type="checkbox"/>	Peerprep-History-Backend	Active	arn:aws:ec...	REPLICA	<div style="width: 100%; background-color: #2e7131; height: 10px;"></div> 1/1 tasks ru...	

Figure 9.4.1 AWS console showing running services

Currently we have specified only one task per service as the application is still in its infancy stage. Not having an auto scaler, we will need to manually increase task instances manually and this will come at a great cost when run in the long run. Hence we have opted to keep it to one instance per task.

## 9.5 Continuous Integration and Continuous Deployment

The workflow of the continuous development begins whenever there is a new push made to the master branch of the upstream repository. This only occurs when there is a Pull Request being merged. The pull request is approved by a member who has scrutinised the changes to the code before it can be merged. The github actions are then triggered and we begin regression testing with the test files we have created. Below are some figures showing one or two instances of our tests being run after code has been pushed to the master branch.

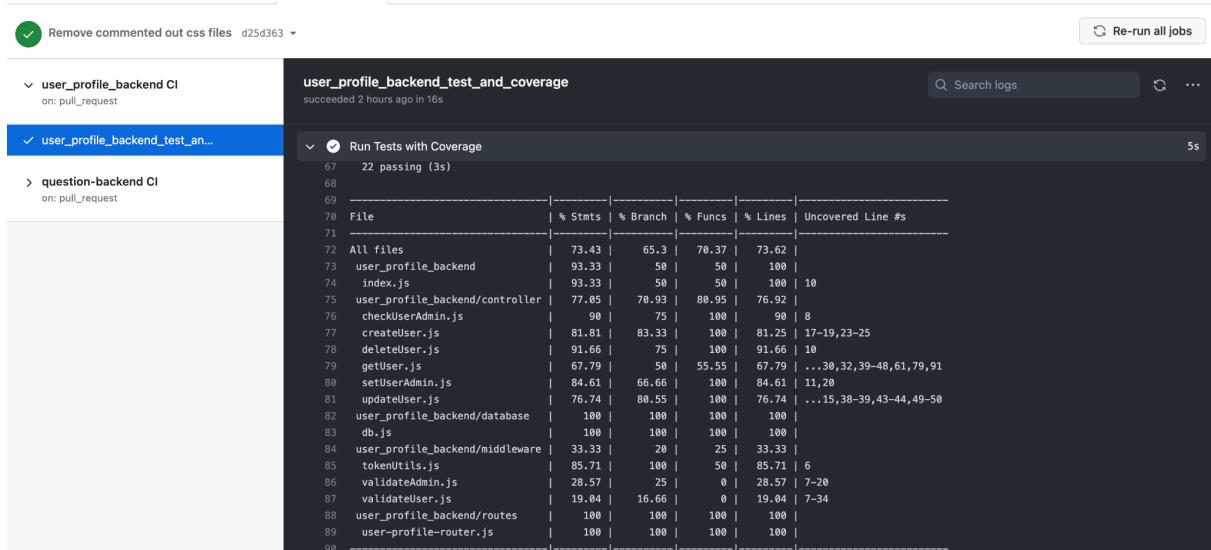


Figure 9.5.1 Example of successful CI test

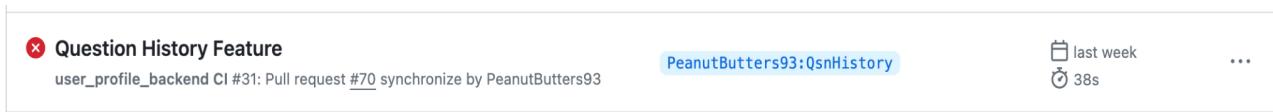


Figure 9.5.2 An example of feedback given when a test fails

In addition, we begin building each container image for each of the microservices, and then pushing the images to the AWS Elastic Container Registry (ECR). The task definitions of each service specifies the container for which the service's image should be found and subsequently, run. Github then instructs AWS to redeploy PeerPrep using the new images for each service. This is all automated so as to reduce the repetitive nature of manually redeploying the application each time there is a change to the code base

The screenshot shows a list of GitHub workflows. On the left, there's a sidebar with "Actions" and "New workflow" buttons, and a "All workflows" tab selected. Below this is a list of triggers: "Deploy Collab BE to Amazon ECS", "Deploy Comm BE to Amazon ECS", "Deploy History BE to Amazon ECS", "Deploy Matching BE to Amazon ECS", "Deploy Qn BE to Amazon ECS", "Deploy to Amazon ECS", "Deploy User BE to Amazon ECS", "question-backend CI", and "user\_profile\_backend CI". The main area lists five workflow runs for "Merge pull request #73 from kohkaixun/master". Each entry includes the author, commit hash (db405cb), pushed by zzthian, the target branch (master), and the timestamp (4 days ago, 4m 1s).

Figure 9.5.3 Workflows that are triggered on a push to master branch

## 10. Workflow

### 10.1 Scrum Methodology

Adapting the Scrum methodology for our project, project meetings were conducted on Zoom at the end of each week. The frequency of meetings was reduced due to other student commitments. During these sessions, we forecasted the project timeline, reviewed the weekly progress, and established targets for the upcoming week. Post-meeting, detailed minutes were recorded in Google Docs for reference, and the identified targets were added to Jira.

#### Planning for UI using GoodNotes

We utilised GoodNotes as a means for drawing up our plans for the User Interface. We chose this over other technologies like Figma due to the more personalised and faster styling we could accomplish over a shorter period of time, given the technical capabilities of the team requiring more time implementing the product rather than planning.

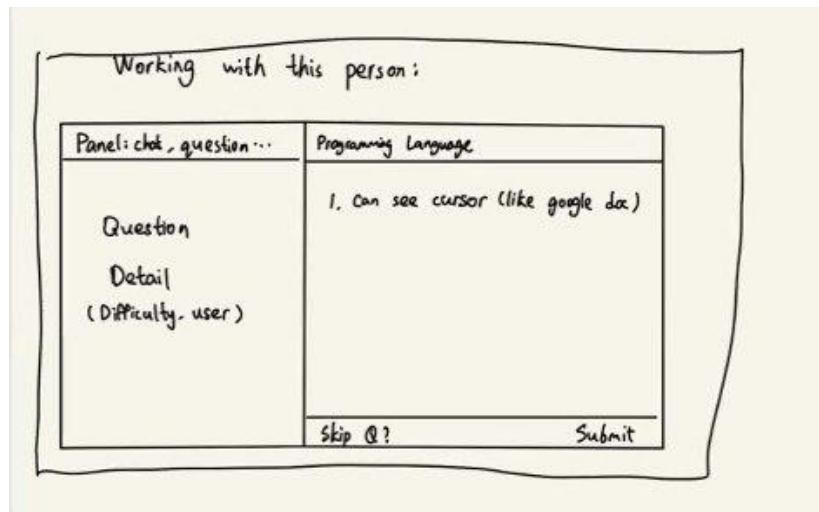


Figure 10.1 Sample diagram for planning of a UI page, Room.js

## 10.2 Jira

Jira provided indispensable benefits in our Agile project management. Firstly, it streamlined work item management by allowing the creation and organisation of user stories and epics. This helped us capture and refine user requirements efficiently.

The screenshot shows the Jira software interface for a project named 'My Scrum Project'. The left sidebar includes links for Planning, Backlog (which is selected), Development, and other project settings. The main area displays the backlog under 'Assignment 1 Sprint 2' (15 Sep – 22 Sep). There are 21 issues listed, each with a title, assignee (e.g., DC, KX, CL, LF, TZ), and status (e.g., TO DO, IN PROGRESS). A 'Create sprint' button is visible at the top right of the backlog section.

Figure 10.2 Snapshot of our Jira Backlog

Jira's robust backlog management system ensured a structured and prioritised list of tasks, facilitating clear planning and alignment with project goals.

Moreover, Jira supported each Agile iteration through its sprint functionality, enabling our team to visualise, plan, and execute our work effectively. Task assignment and tracking features enhance accountability and transparency, allowing team members to know who's responsible for each task and its current status.

Lastly, Jira fosters collaboration by serving as a centralised platform for communication. Its capability for comments, attachments, and mentions ensures seamless team interaction, promoting clarity and swift decision-making.

Throughout the duration of the project, we frequently updated our progress and work schedules on Jira especially during team meetings, and it streamlined task management, prioritisation, execution, and communication, ultimately contributing to enhanced productivity and successful project delivery.

## 10.3 Github

GitHub played a dual role as a version control system and code management tool, enhancing collaboration by enabling parallel work on branches. Before new code would be added, a Pull Request had to be made, followed by peer review and testing, before being integrated into the main branch. As shown in Section 9.3, in the latter part of the project, GitHub actions automated portions of the testing process.

## 11. Proposed Extensions

### 11.1 Image Support

835. Image Overlap

Medium    1.3K    467      

Companies

You are given two images, `img1` and `img2`, represented as binary, square matrices of size  $n \times n$ . A binary matrix has only `0`s and `1`s as values.

We **translate** one image however we choose by sliding all the `1` bits left, right, up, and/or down any number of units. We then place it on top of the other image. We can then calculate the **overlap** by counting the number of positions that have a `1` in **both** images.

Note also that a translation does **not** include any kind of rotation. Any `1` bits that are translated outside of the matrix borders are **erased**.

*Return the largest possible overlap.*

**Example 1:**

1	1	0
0	1	0
0	1	0

img1

0	0	0
0	1	1
0	0	1

img2

**Input:** `img1 = [[1,1,0],[0,1,0],[0,1,0]]`, `img2 = [[0,0,0],[0,1,1],[0,0,1]]`  
**Output:** 3  
**Explanation:** We translate `img1` to right by 1 unit and down by 1 unit.

Image 11.1 Screenshot of LeetCode, another interview preparation platform

Currently, PeerPrep questions only support text. Incorporating image support to add visuals to complex questions, or TeX for mathematical expressions and symbols would help improve question clarity.

## 11.2 CQRS

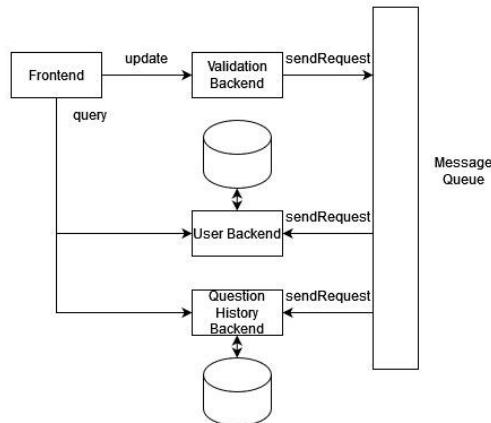


Figure 11.2 An example of how CQRS could be implemented

CQRS stands for Command-Query-Responsibility Segregation. It is an architectural pattern that separates the write (commands) and read (queries) paths, which allows them to be optimised independent from each other.

In PeerPrep, the user deletion process involves updating the user and question history database. Implementing CQRS would have reduced code reuse in transmitting the information from the frontend to the services. However, CQRS was not implemented as it would require more microservices, and greater hosting costs.

## 11.3 Question Scraping

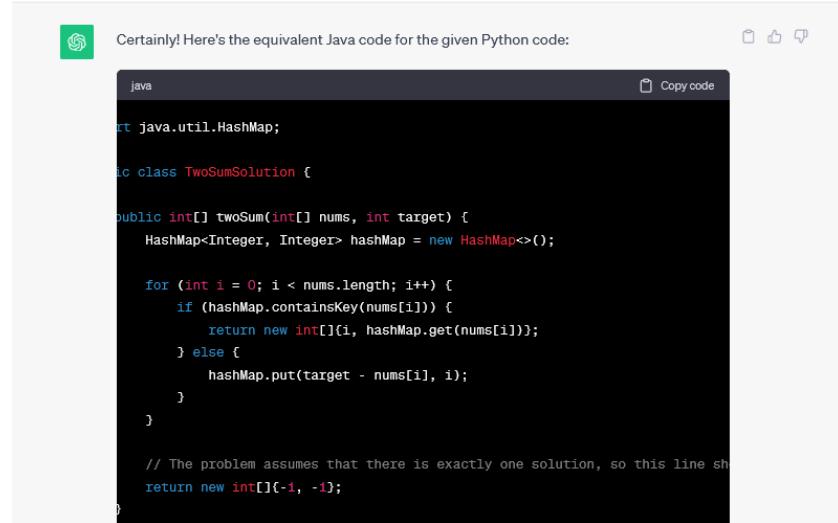


Figure 11.3 Number of Questions in LeetCode and Codeforces as writing

There exists numerous websites - such as LeetCode and Codeforces - with a wealth of coding questions. A future extension could be the creation of a serverless function that would periodically access those websites and integrate their questions into our database. This would ensure a consistent influx of diverse questions, and reduces the need for manual entry or creating questions. By embracing this automated solution, we optimise our time resources, offering our users an ever-expanding array of coding challenges to enhance their learning and practice experience on our platform.

## 11.4 Incorporation of Generative AI

```
def twoSum(self, nums: List[int], target: int) -> List[int]:  
    hash_map = {}  
    for i in range(len(nums)):  
        if nums[i] in hash_map:  
            return [i, hash_map[nums[i]]]  
        else:  
            hash_map[target - nums[i]] = i  
Convert this python code into java
```



Certainly! Here's the equivalent Java code for the given Python code:

```
java  
import java.util.HashMap;  
  
public class TwoSumSolution {  
  
    public int[] twoSum(int[] nums, int target) {  
        HashMap<Integer, Integer> hashMap = new HashMap<>();  
  
        for (int i = 0; i < nums.length; i++) {  
            if (hashMap.containsKey(nums[i])) {  
                return new int[]{i, hashMap.get(nums[i])};  
            } else {  
                hashMap.put(target - nums[i], i);  
            }  
        }  
  
        // The problem assumes that there is exactly one solution, so this line should never be reached.  
        return new int[]{-1, -1};  
    }  
}
```

Figure 11.4 ChatGPT converting Python code to Java

In the ever-evolving landscape of technology, the integration of generative AI, exemplified by tools like ChatGPT and Copilot, is becoming ubiquitous. Considering this, we propose the incorporation of generative AI capabilities into PeerPrep to empower our users with cutting-edge features. With generative AI, our platform can offer functionalities such as detailed code explanations and code translation between languages. This not only ensures that our users are well-acquainted with the tools and practices prevalent in the tech industry, but also provide PeerPrep with a new unique selling point.

## 12. Reflections

This project has been an extremely tough, but eye-opening experience for our group. As we were required to start the project from scratch, we were exposed to the entire software development cycle, most of us for the first time. We have gained a newfound appreciation for the planning phase of any project. While it does take up time that could have been used to immediately start on the project, it gave us a sense of direction that saved a lot more time when it came to the actual implementation.

We have also learnt just how fast-paced a project can be. As we were very new to most of the tech stack we used in our project, using a technology after just a day of learning or even learning as we implemented it became the norm. Thus, we were faced with a steep learning curve throughout this project, making it as daunting as it was fulfilling when we finished.

Another crucial takeaway from the project was the advantages of the microservices architecture. Not only did it improve the scalability and manageability of our code base, but it also allowed us to work on different features of the project concurrently. However, this caused us some setbacks when we realised that services required to communicate with each other were not doing so successfully due to differences in implementation. Nonetheless, we eventually acknowledged that this could have been avoided had we gone into further detail in the planning phase, thus further teaching us the importance of planning before implementation.

Implementing communication patterns was something else that taught us a lot. The event-driven nature of PeerPrep's architecture due to features like matching and a live collaborative code editor signalled a need for communication patterns to facilitate communication between users. When used correctly, the right pattern could increase efficiency and even reduce coupling. But likewise, before we can even implement communication between our services, we must ensure that communication first occurs between the different people implementing these services, once again signalling the importance of the planning phase.

Finally, we also learnt that there is no one right way to implement a project. As can be seen above in the section on Architecture, every design has its advantages and disadvantages. It is up to us to weigh the pros and cons as a team and decide which design to proceed with.

In conclusion, this has been a long and arduous journey riddled with setbacks and difficulties. But it is through these challenges where we have learnt so much about software engineering and about ourselves as a team. The lessons and mistakes we have learned from and lived through in this project will help pave our way to become better software engineers.