

CS3219 Final Report

Group 50

Source Code:

<https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g50>

Deployment Link:

<http://ec2-13-214-194-165.ap-southeast-1.compute.amazonaws.com:3000/>

Member	Chua Wen Quan	Ng Zhi Cai	Ong Wei Xiang	Qin Yifan
Student Number	A0221068M	A0199988N	A0185833W	A0201944L

Table of Contents

Individual Contributions	4
1. Introduction	5
1.1 Background	5
1.2 Purpose of PeerPrep	5
2. Developer Documentation	6
2.1 Screenshots	6
2.2 Architecture	8
2.2.1 Architectural Diagram	8
2.2.2 Architectural Decisions	9
2.2.2.1 Modularity with Micro-Services	9
2.2.2.2 Choice of Database	9
2.3 Technology Stack	10
3. Project Management	13
4. Frontend	15
4.1 Functional Requirements	15
4.2 Interactions and Workflow	16
4.2.1 Interactions between pages	16
4.2.2 Messaging patterns for interview page	17
4.2.3 Middleware Routes	18
4.2.3.1 Middleware route for existing user session	18
4.2.3.2 Middleware route for private routes	19
5. Backend	20
5.1 User Service	20
5.1.1 Functional Requirements	20
5.1.2 API Endpoints	20
5.1.3 Schema Design	22
5.2 Question Service	23
5.2.1 Functional Requirements	23
5.2.2 API Endpoints	23
5.2.3 Schema Design	24
5.3 Matching Service	25
5.3.1 Functional Requirements	25
5.3.2 Web Socket Events	25
5.3.3 Activity Diagram	26
5.4 Collaboration Service	27
5.4.1 Functional Requirements	27
5.4.2 Web Socket Events	27
5.4.3 Activity Diagram	28
5.5 Chat Service	29
5.5.1 Functional Requirements	29

5.5.2 Web Socket Events	29
5.5.3 Activity Diagram	30
5.6 Non-Functional Requirements	31
5.6.1 Quality Attributes Prioritization Matrix	32
6. Design Patterns	33
6.1 Pub Sub	33
6.2 Singleton	34
6.3 Decorator	34
6.4 Model View Controller (MVC)	35
6.5 Dependency Injection	36
7. CI/CD	37
7.1 Continuous Integration	37
7.2 Continuous Deployment	38
7.3 Local Staging Environment	40
8. Future Improvements	41
9. Conclusion	42

Individual Contributions

Chua Wen Quan	Report: <ul style="list-style-type: none">- Frontend- Future Improvements
	Code: <ul style="list-style-type: none">- Frontend
Ng Zhi Cai	Report: <ul style="list-style-type: none">- Matching Service- Collaboration Service- Chat Service- Design Patterns- Future Improvements
	Code: <ul style="list-style-type: none">- Matching Service- Collaboration Service- Chat Service
Ong Wei Xiang	Report: <ul style="list-style-type: none">- Technology Stack- User Service- Question Service- CI/CD- Future Improvements
	Code: <ul style="list-style-type: none">- User Service- Question Service- CI/CD- Deployment
Qin Yifan	Report: <ul style="list-style-type: none">- Architecture- Project Management- Future Improvements
	Code: <ul style="list-style-type: none">- Frontend

1. Introduction

1.1 Background

Computer Science students often encounter challenging technical interviews when applying for jobs in the software industry. While there are existing resources online that help students practise and prepare for such interviews, repeatedly practising difficult questions alone can be tedious and monotonous.

Students do not have access to an engaging platform where they are able to communicate and discuss their thought processes with each other. Articulating the problem and programming with peers is important because it helps students understand the given problem better.

To solve this issue, our team decided to create an interview preparation platform and peer matching system called PeerPrep, where students can find peers to practise whiteboard-style interview questions together.

1.2 Purpose of PeerPrep

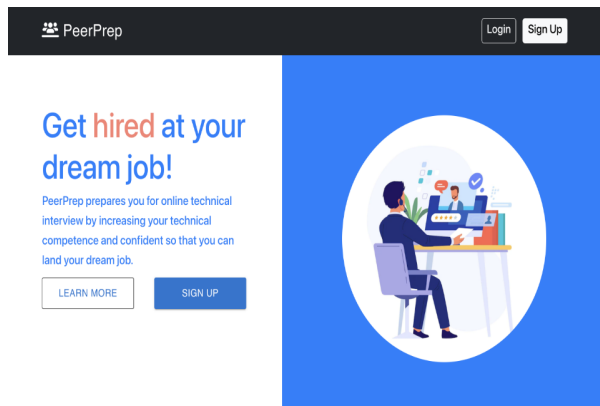
PeerPrep helps students prepare for technical interviews by using a peer learning system so students can learn from each other and break the monotony of revising alone. Deployed in the form of a WebApp, PeerPrep supports functionalities such as user authentication, matchmaking and real-time collaboration. The questions are fetched from a question bank which are categorised into 3 difficulties - easy, medium and hard.

Through PeerPrep, students can not only increase their technical knowledge and confidence in whiteboard interviews, but also their soft skills such as communication to better convey their ideas and train of thoughts to the interviewer.

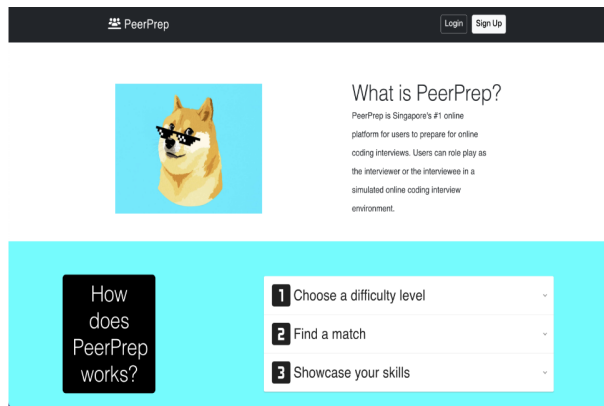
2. Developer Documentation

2.1 Screenshots

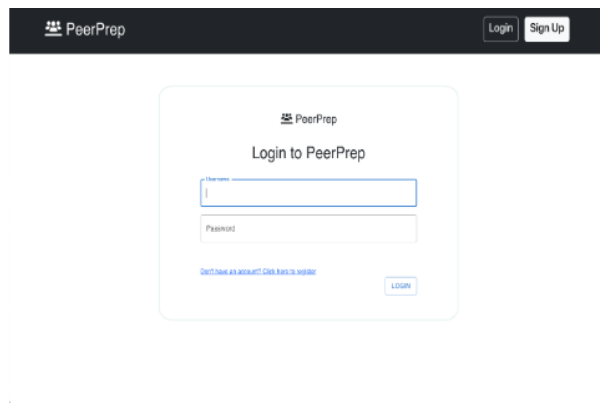
Landing Page



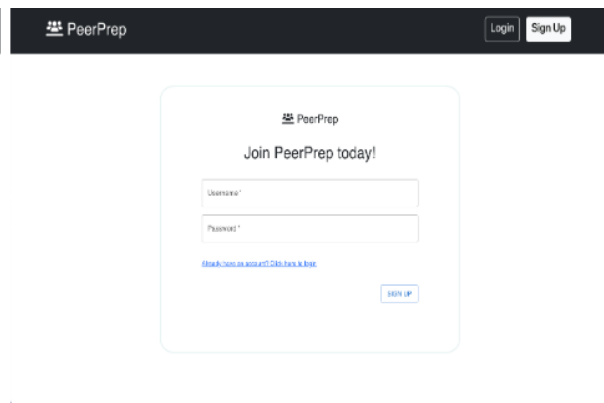
About Page



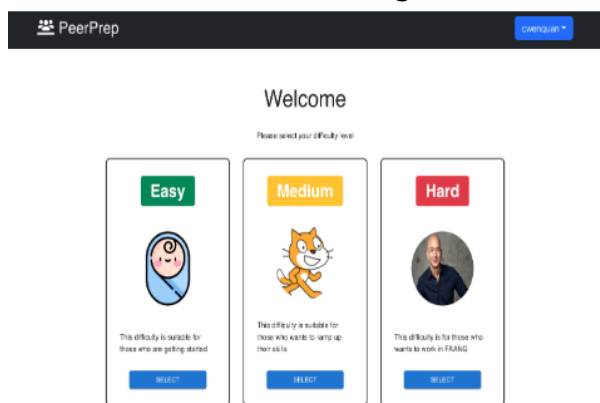
Login Page



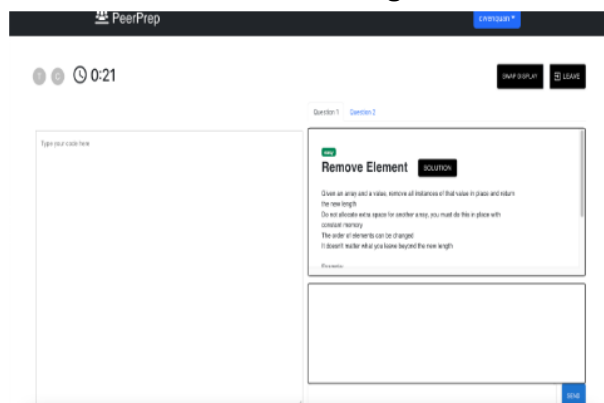
Registration Page



Dashboard Page



Interview Page




Profile Page

PeerPrep

Download PDF

Change Password

Logout



cwenquan

Change Password

404 Page

PeerPrep

Login

Sign Up

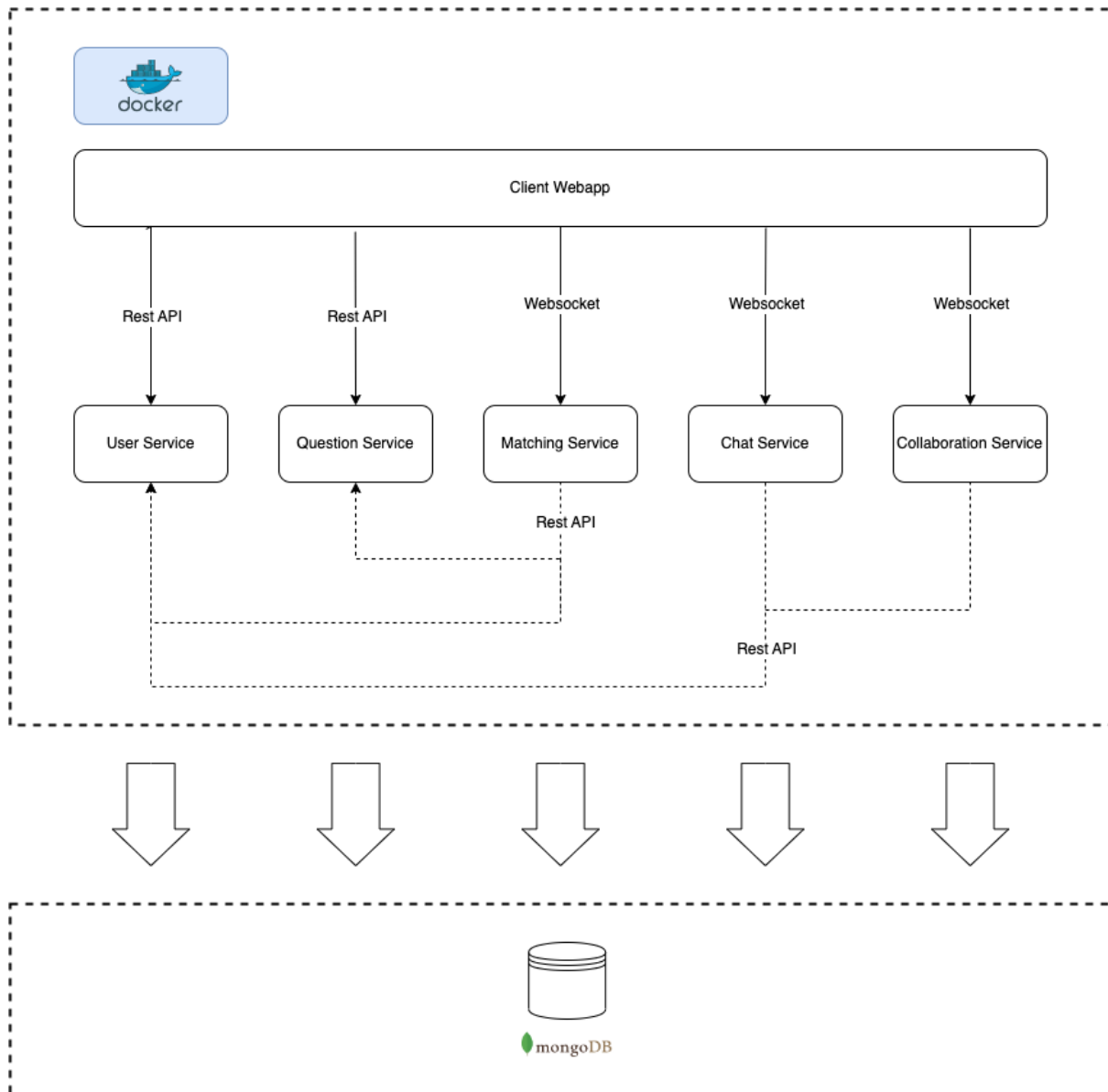
404

Page not found

The page you requested could not be found

2.2 Architecture

2.2.1 Architectural Diagram



2.2.2 Architectural Decisions

2.2.2.1 Modularity with Micro-Services

For the architecture of the backend of our application, we have decided to use microservices due to several benefits it would bring to our application as stated in the table below:

Benefits	Elaboration
Reduced coupling	Each micro service acts as a facade where other micro services can make requests to it based on an interface provided without having to reason about the underlying implementation, reducing coupling as a result.
Improved resiliency	When a single container fails inside a cluster of a micro service, we would be unaffected as traffic could simply be routed to other containers within a single cluster.
Increased horizontal scalability	Within a single cluster, more containers could be spun up when there is a larger load in order to handle a temporary increased number of requests.

2.2.2.2 Choice of Database

For the database of our application, we have decided to use a NoSQL database, MongoDB, over an SQL database or other NoSQL databases for some reasons below:

Reason	Elaboration
Horizontal scalability through sharding	MongoDB supports sharding and this allows data to be spread out across several different nodes, supporting horizontal scalability of the database for future needs.
Flexible schema	When comparing SQL vs NoSQL databases, SQL databases enforces a data structure that cannot be changed. We foresee that the data could change in schema as we develop the application further and we decided to use MongoDB which allows a flexible data schema that could change over time to support the needs of our application.
No need for ACID compliance	One of the strong reasons for SQL databases would be the ACID compliance of an SQL database that is built in. ACID compliance protects the integrity of the database by allowing prescription of exactly how to deal with transactions in the database. However, for our application, transactionality of database updates is not mission-critical, so we decided on using a NoSQL database.
Prebuilt cloud solution	MongoDB comes with their own pre-built cloud solution, MongoDB Atlas which is scalable and provides support for horizontal scaling out of the box. Given the short runway required to deliver the application, this allowed us to develop our application rapidly without having to worry about scaling.

2.3 Technology Stack

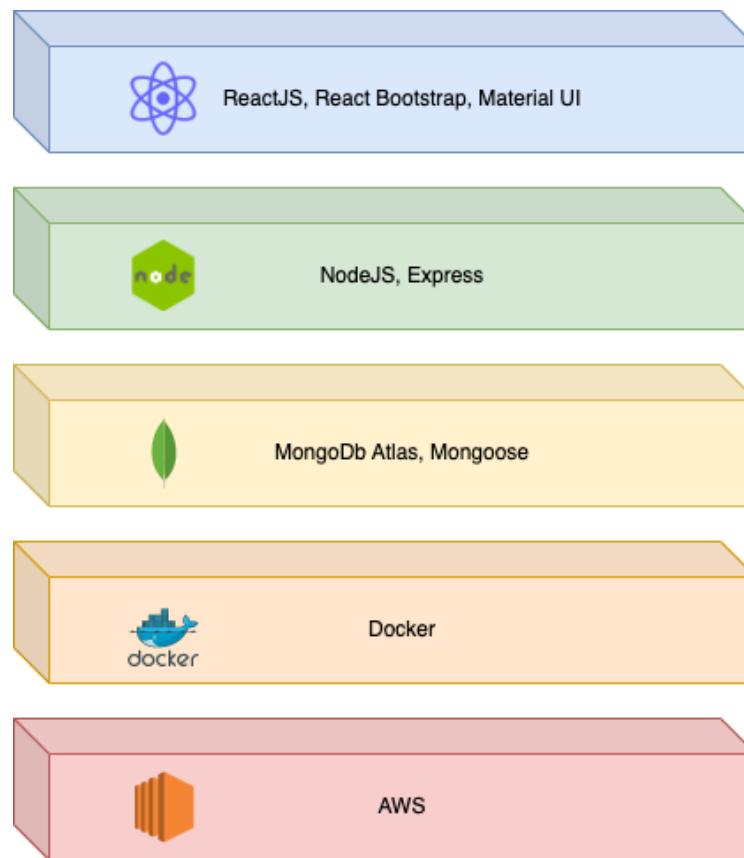


Figure: Technology Stack for PeerPrep

In this section, we will elaborate on the reasons as to why we chose certain technologies to be used in our project.

We based our decisions around 2 core reasons: practicality and community support.

Due to the short timeframe given to complete the project, we need a highly straightforward implementation process. This means that instead of prioritising learning new technologies, we should be focusing on implementing the functional requirements that we stipulated for the project.

Therefore, we want to use technologies that our team is familiar with and have good community support to reduce the amount of obstacles we face in the development process. Even if we encounter bugs, the community support for the technologies should be able to help us resolve the problems as quickly as possible, so that we can always be working on the prioritised features.

System	Technologies	Justification
Linting / Code Formatting	ESLint Prettier	Usage of ESLint and Prettier helped to enforce a consistent styling between all code written, improving code readability across the entire project.
Frontend	React React Bootstrap Material UI	<p>React is the first frontend technology that everyone of us learnt. Therefore, this easily became the top choice as we are all fluent with it. This means that any of us can help out with the frontend if necessary (even though we may have been assigned to backend development).</p> <p>This has proven to be useful as we were able to come together to debug problems when integrating our frontend with the microservices.</p> <p>Usage of UI Component libraries, React Bootstrap and Material UI, allows us to implement a consistent user experience throughout the entire application so our users know what to expect.</p>
Backend	Node.js Express.js	<p>Node.js and Express.js are the backend technologies introduced to us in the module. The OTOT tasks helped us to become familiar with the syntax and quirks.</p> <p>We wanted to bring over the knowledge that we gained to this project, and make effective use of our time/effort spent to learn the language/framework.</p> <p>Furthermore, by using the same language (JavaScript) for both frontend and backend, we are able to understand the implementation on both sides much more easily, as it becomes a language both subteams (frontend and backend) can understand.</p>
Database	Mongoose MongoDB Atlas	<p>Mongoose allows us to model the schema and manage entity relationships much more easily.</p> <p>When used on top of MongoDB which is one of the most popular database solutions that our team is familiar with, we are able to iterate quickly on the project without worrying about storage-related issues.</p> <p>Beyond technical reasons, these are the technologies that we have used before in our side projects so we were able to reference past implementations and adapt them to our use-case</p>

		in this project.
Containerization	Docker	<p>Likewise, this was a familiar tool that we used for our OTOT tasks.</p> <p>As we researched options for our deployment system, we discovered that Docker integrates well with AWS Elastic Container Registry / AWS Elastic Container Services.</p> <p>Furthermore, we are able to simulate the production environment locally, by setting up a staging environment where our local machine hosts the microservices containers and the frontend application container.</p>
CI/CD	GitHub Actions	<p>GitHub Actions is directly supported by GitHub and has solid integration with GitHub, the version control hub that we are using.</p> <p>Furthermore, there is a very good community support for the various workflows that we require that can be found in GitHub Actions marketplace (e.g. AWS authentication).</p> <p>With such extensive support and good integration with AWS (due to the community), we decided to go with GitHub Actions for our CI/CD system.</p>
Deployment	<p>AWS Elastic Container Registry (ECR)</p> <p>AWS Elastic Container Services (ECS)</p>	<p>Amazon Web Services (AWS) is known to be the most popular cloud infrastructure solutions in the market. This makes the decision-making process rather straightforward as the development documentation and community support is one of the best out there.</p> <p>Furthermore, with further extensions for our product in mind, it makes sense to use AWS in case we need any of the solutions provided in their wide category list (e.g. Elasticache). With so many solutions provided by AWS, we are able to develop our product with an ease of mind as we know that most, if not all, of our future ideas can be supported by AWS's cloud solutions.</p>
Pub-Sub	Socket.io	<p>We chose to go with Socket.io as it is by far the most popular library on Node Package Manager (npm), with 43k stars on the GitHub repository.</p> <p>With such substantial support for the library, we didn't have to consider much, as we only required the basic functionalities to connect users in persistent sessions. Thus, we went with the most popular solution by default.</p>

3. Project Management

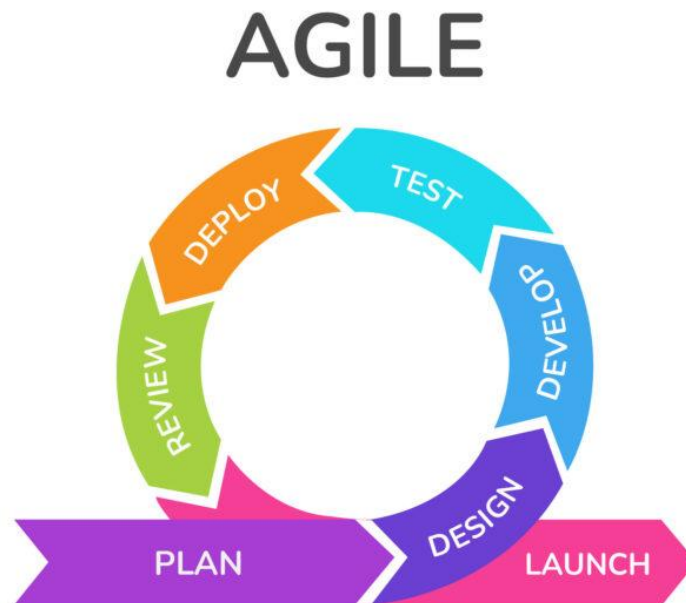


Figure: Agile Development Workflow

We adopted an agile development framework as shown by the figure above. There is a weekly standup every Thursday held at 10pm. This meeting is used to plan the features that we will be implementing throughout the week, as well as assign the workload to the individual team members.

We intentionally held the meeting after every lecture so that we are able to implement any new insights of design considerations from the lessons taught in our project. After assigning tasks, we will check in on the overall progress of the development and what has been built so far, taking note of any blockers encountered. This helps to ensure accountability for every team member to push out the features necessary for the milestones.

Throughout the rest of the week. We design, develop, and build unit tests for our respective features, pushing our commits to our own respective GitHub accounts incrementally. This allows us to work in parallel and integrate our services with each other more easily. On the coming Tuesday, we pull and review each other's pull requests to ensure that there are no bugs and that all the functional requirements have been met.

After deploying our WebApp to AWS, we invited our batchmates to conduct user testing to ensure that our application satisfies all defined functional and non functional requirements, user goals, and has a smooth user experience.

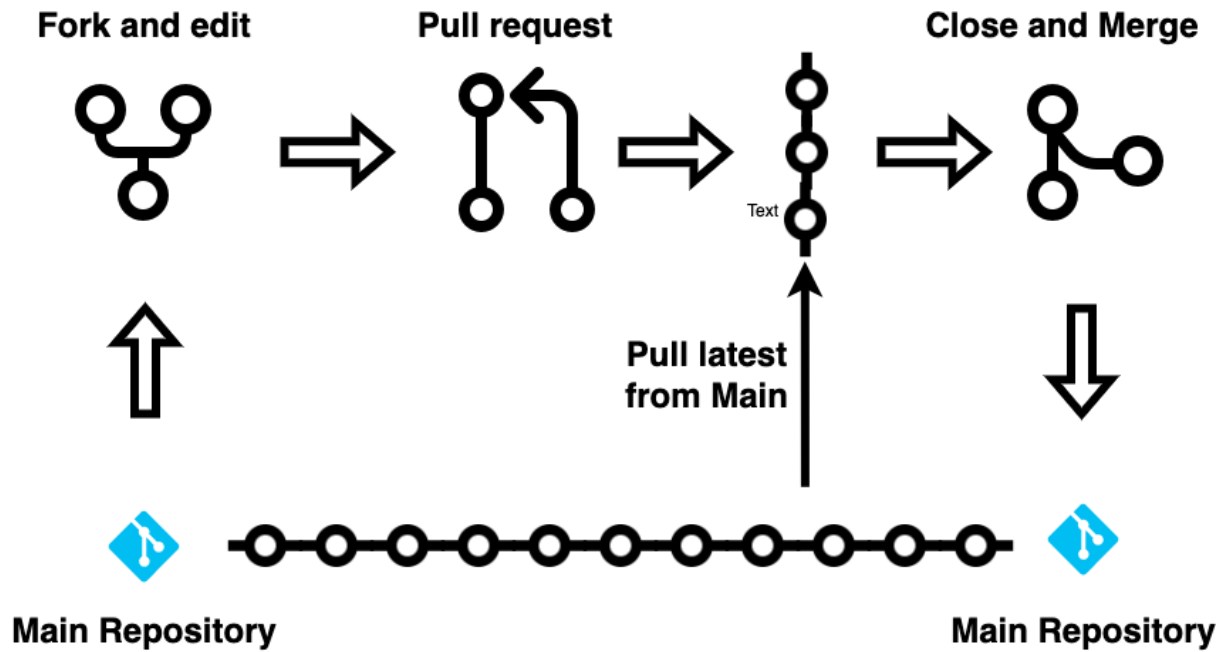


Figure: Forking workflow

Our team has employed a standard git workflow to ensure that work is completed in a consistent and productive manner. We have used the forking workflow whereby each member has their own private repository, and any changes are pushed to this private repository which is then merged to the main branch of our team's repository after pulling and rebasing.

When we are reviewing and merging pull requests, we use Zoom to do ***pair programming*** to ensure that we can review and fix bugs at the same time. We also use a project Discord Channel primarily for work communication.

4. Frontend

4.1 Functional Requirements

S/N	FR	Priority	Completed
F1	Must persist user data using cookies when page is refreshed.	High	Y
F2	Users are able to chat with a chat interface.	High	Y
F3	Users should be able to find solutions to the problems.	High	Y
F4	There should be a timer so that users can swap roles.	High	Y
F5	The timer should be synced between the 2 users.	High	Y
F6	The system should have two questions for each user to take turns as interviews.	High	Y
F7	The system should have a text editor that is supported by the collaboration service.	High	Y
F8	The system should have a UI for registering and logging in.	High	Y
F9	The system should allow the users to role-play (i.e. interviewer and interviewee).	Medium	Y
F10	Notification when the other user leaves the room.	Medium	Y
F11	Allow the user to return back to the interview session if he navigates away.	Medium	Y
F12	Save the state of chat messages when the user navigates away from the interview page.	Medium	Y
F13	Save the state of the code editor when the user navigates away from the interview page.	Medium	Y

4.2 Interactions and Workflow

4.2.1 Interactions between pages

The pages in the frontend displayed to the users are made up of React components, which are able to navigate between each other. The App component, together with the React Router library, in our React code acts as the controller class of our application.

We try to design our UI as though it is a product for real end users, therefore we have implemented a landing page (or a hero section) to show the users our unique selling points, and there is also an about page for users to learn more about our application.

The figure below illustrates the interaction between different React components.

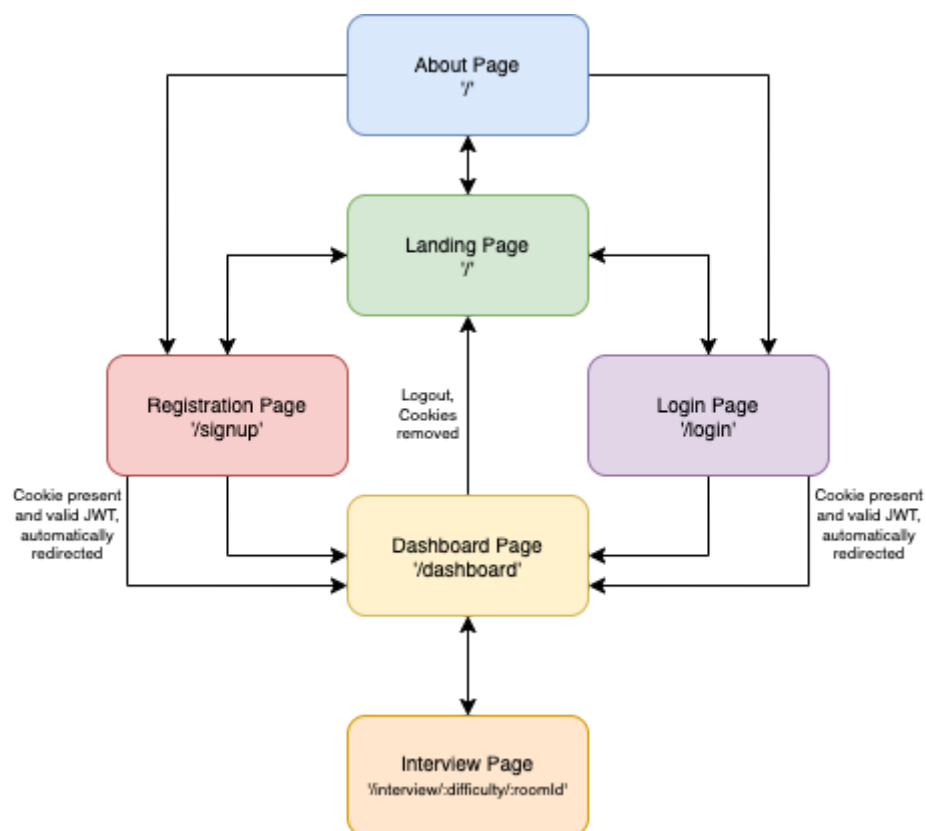


Figure: Interactions between React components

4.2.2 Messaging patterns for interview page

In the interview page, the chat box and text editor relies on the chat and collaboration service to communicate between different clients. On the client side, a client socket is created for both the chat box and text editor when the user enters the interview page, which will then connect to the socket implemented in the chat and collaboration service. The following diagram illustrates the interaction between the client and the services.

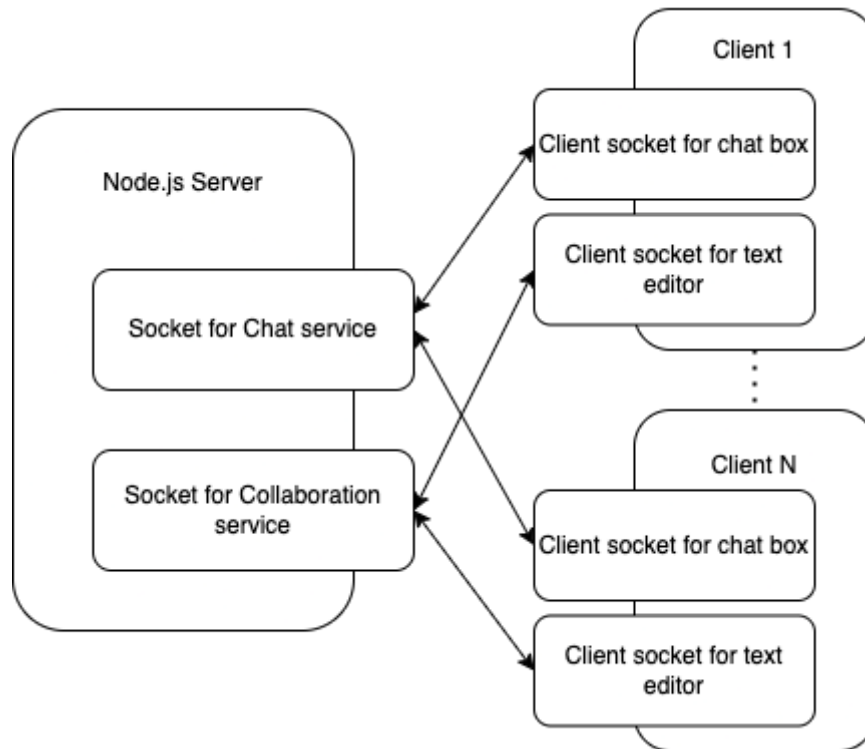


Figure: Messaging patterns between client and server

4.2.3 Middleware Routes

Middleware routes are routes that are first accessed before the user is redirected to the requested endpoints in the frontend, which works similarly to middlewares used in Express.

4.2.3.1 Middleware route for existing user session

The middleware route to check if there is already an existing user session (via browser cookie) is called ExistingAuth. This is mainly to improve user experience to automatically log the user into the application if the user has already logged in before and has a valid token stored in the browser as cookie.

ExistingAuth is used for the login and registration page with endpoints of '/login' and '/signup' respectively. The following activity diagram illustrates the flow of ExistingAuth.

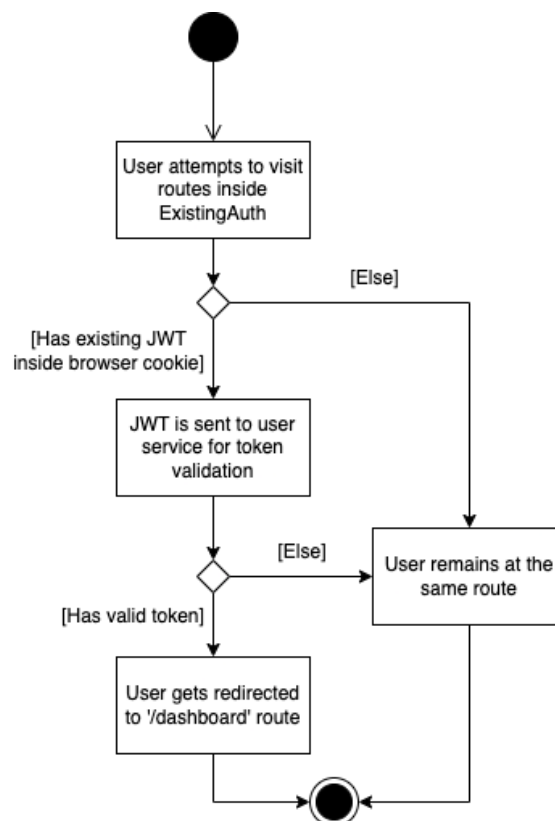


Figure: Activity diagram for ExistingAuth

4.2.3.2 Middleware route for private routes

The middleware route to check if the user is authorised to visit private routes (or routes that require authentication) is called `RequireAuth`. This is to ensure that the user is logged in and has an existing valid token before the user is able to use the application.

`RequireAuth` is used for the dashboard, profile and interview pages with the endpoints of `/dashboard`, `/profile` and `/interview/:difficulty/:roomId` respectively. The following activity diagram illustrates the work flow of `RequireAuth`.

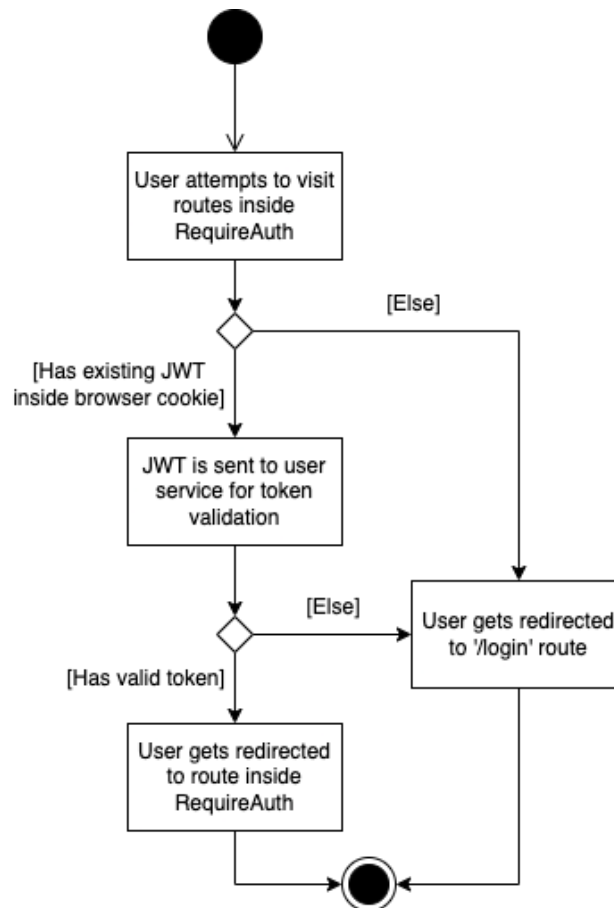


Figure: Activity diagram for `RequireAuth`

5. Backend

5.1 User Service

The user service mainly stores user credentials and allows basic authentication capability via JWT. These functionalities will be illustrated through the functional requirements, API endpoints, and User Schema as elaborated in the following sections.

5.1.1 Functional Requirements

S/N	FR	Priority	Completed
U1	The system should allow users to create an account with username and password.	High	Y
U2	The system should ensure that every account created has a unique username.	High	Y
U3	The system should allow users to log into their accounts by entering their username and password.	High	Y
U4	The system should allow users to log out of their account.	High	Y
U5	The system should store login credentials in the browser as cookies.	High	Y
U6	The system should allow users to change their password.	Medium	Y
U7	The system should validate the number of characters for registration.	Medium	Y
U8	The system should allow admins to delete the user's account.	Low	Y

5.1.2 API Endpoints

Route	Method	Payload (JSON)	Successful Response (JSON)	Description
/register	POST	username: string password: string	success: string	Returns a success message with the username if the registration is successful.

				Otherwise return validation/internal errors.
/login	POST	username: string password: string	success: string accessToken: string	Returns JWT access token to client-side if login is successful. Otherwise return validation/internal errors.
/logout	POST	- <i>Note: The request should contain the JWT access token.</i>	success: string	Returns a success message if the logout is successful. Will blacklist the JWT token on server-side. Otherwise return validation/internal errors.
/verify-token-or-role	POST	role: string (optional) <i>Note: The request should contain the JWT access token.</i> <i>If there is a role, it would validate the token role against the actual role as well, otherwise it would just validate the token.</i>	success: string username: string	Returns a success message if the JWT token is deemed as valid. Otherwise return authentication/authorization/internal errors.
/update	PUT	username: string newPassword: string <i>Note: The request should contain the JWT access token.</i> <i>If there is a role, it would validate the</i>	success: string	Returns a success message if the password update is successful. Otherwise return validation/internal error. Currently only updates

		<i>token role against the actual role as well, otherwise it would just validate the token.</i>		password. In the future, we may extend it to updating user information in general.
/delete	DELETE	username: string <i>Note: The request should contain the JWT access token.</i> <i>If there is a role, it would validate the token role against the actual role as well, otherwise it would just validate the token.</i>	success: string	Returns a success message if the user deletion is successful. Otherwise return validation/internal error.

5.1.3 Schema Design

```

UserSchema = {
  username: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  role: {
    type: Number,
    required: true,
  },
  refreshToken: {
    type: String,
    required: false,
  },
};

```

5.2 Question Service

The question service mainly stores our list of technical interview questions and provides questions among the list when requested according to the difficulty levels. These functionalities will be illustrated through the functional requirements, API endpoints, and Question Schema as elaborated in the following sections.

Once users are matched by the matching service, the matching service will make a call to the question service to retrieve 2 interview questions for the pair in the room.

5.2.1 Functional Requirements

S/N	FR	Priority	Completed
Q1	The system should store a comprehensive list of technical interview questions.	High	Y
Q2	The system should be able to generate 2 questions of the chosen difficulty level for the users upon request.	High	Y
Q3	The system should allow the creation of new questions.	High	Y
Q4	The system should allow editing of existing questions.	Low	N
Q5	The system should allow the creation of new questions.	Low	N
Q6	The system should provide a new question when requested.	Low	N
Q7	The system should track which questions a user has attempted.	Low	N
Q8	The system should allow users to opt out of receiving questions previously attempted.	Low	N

5.2.2 API Endpoints

Route	Method	Payload (JSON)	Successful Response (JSON)	Description
/get-two-questions-by-diff	POST	-	success: string questionOne: string	Returns a success

		<p><i>Note: The request should contain the JWT access token.</i></p> <p><i>Will call user service for authentication/authorization.</i></p>	questionTwo: string	<p>message with the 2 questions.</p> <p>Otherwise return validation/internal errors.</p>
/create-question	POST	<p>name: string description: string difficulty: string examples: string</p> <p><i>Note: The request should contain the JWT access token.</i></p> <p><i>Will call user service for authentication/authorization.</i></p>	success: string	<p>Returns a success message with the question name.</p> <p>Otherwise return validation/internal errors.</p>

5.2.3 Schema Design

```

QuestionSchema = {
  name: {
    type: String,
    required: true,
    unique: true,
  },
  description: {
    type: String,
    required: true,
  },
  difficulty: {
    type: String,
    required: true,
  },
  examples: {
    type: String,
    required: true,
  },
};

```


5.3 Matching Service

5.3.1 Functional Requirements

S/N	FR	Priority	Completed
M1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High	Y
M2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	High	Y
M3	The system should inform the users that no match is available if a match cannot be found.	High	Y

5.3.2 Web Socket Events

The matching service consists of several web socket events that a client can emit where the server could carry out various actions based on the event emitted.

Event	Payload	Description	Server Action
match:find_match	{ difficulty: string }	Event emitted when a user searches for a potential match. There are only three Difficulty levels available - 'easy', 'medium' or 'hard'.	Place the user in a queue if there is no other user in the same queue for a difficulty level. If there is a match found, the server would emit a match:match_found event to notify the client.
match:cancel_find_match		Event emitted when a user cancels finding of a match.	Removes the user from the match queue.

The matching service emits several web socket events that a client would need to handle as well.

Event	Payload	Description	Client Action
match:match_found	{ roomId: string, difficulty: string, questions: string[] }	Event emitted from the server when a server finds a match for a user.	Redirect the user to the interview page and display the questions.

5.3.3 Activity Diagram

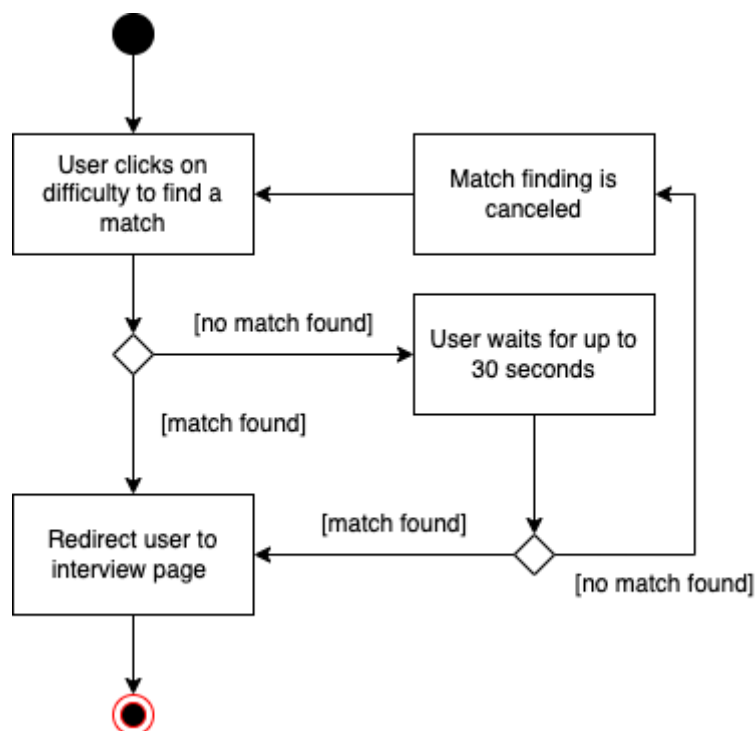


Figure: Activity Diagram for Matching Service

5.4 Collaboration Service

5.4.1 Functional Requirements

S/N	FR	Priority	Completed
C1	The system should support concurrent code editing for the users.	High	Y
C2	The system should support being able to show which users are currently in the room.	High	Y
C3	The system should support at least 2 instances of code editing window for every match.	High	Y
C4	The system should provide a means for the user to leave a room if they are in a session.	High	N
C5	The system should provide a means for the user to rejoin a room if they navigate away from the interview dashboard.	High	N

5.4.2 Web Socket Events

The collaboration service consists of several web socket events that a client can emit where the server could carry out various actions based on the event emitted.

Event	Payload	Description	Server Action
collaboration:join_room	{ roomId: string }	Event allows a user to join a room	Adds user to the room specified
collaboration:editor	{ message: string, roomId: string }	Event emitted when a user makes a change on the code editor	Broadcasts change in code editor to other users in the same room
disconnect	-	Event emitted when a user leaves a room	Broadcasts to other users in the room that the user has left

The collaboration service emits several web socket events that a client would need to handle as well.

Event	Payload	Description	Client Action
collaboration:leave_room	{ users: string[] }	Event emitted when a user leaves the room. The payload contains the users left in the room.	Notify the client that a user has left and display the remaining users in the room.

5.4.3 Activity Diagram

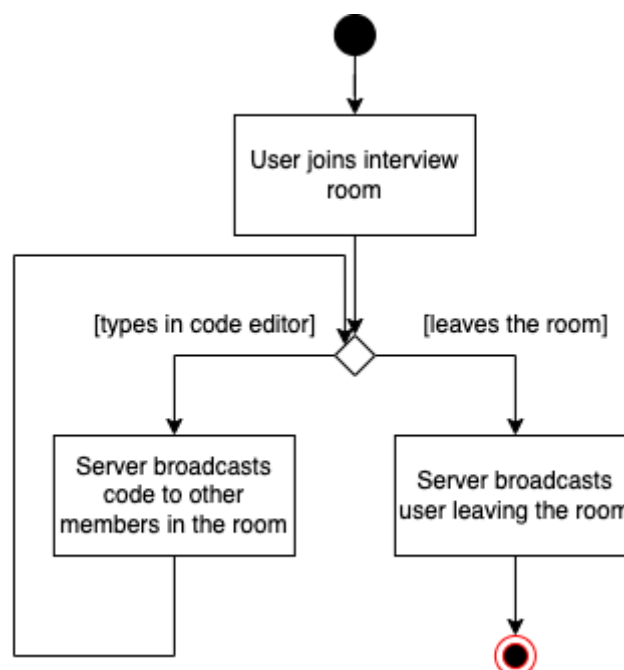


Figure: Activity Diagram for Collaboration Service

5.5 Chat Service

5.5.1 Functional Requirements

S/N	FR	Priority	Completed
H1	The system should allow users to join a chat room.	High	Y
H2	The system should allow other room participants to receive a message sent by a user.	High	Y
H3	The system should allow a user to leave the room when they are done.	High	Y
H4	The system should allow a user to rejoin the same chat room if they navigate away from the room and back.	High	Y

5.5.2 Web Socket Events

The chat service consists of several web socket events that a client can emit where the server could carry out various actions based on the event emitted.

Event	Payload	Description	Server Action
chat:join_room	{ roomId: string }	Event allows a user to join a room	Adds user to the room specified
chat:message	{ message: string, roomId: string }	Event emitted when a user sends a message	Broadcasts message to other users in the same room
disconnect	-	Event emitted when a user leaves a room	Broadcasts to other users in the room that the user has left

The chat service emits several web socket events that a client would need to handle as well.

Event	Payload	Description	Client Action
chat:leave_room	{ users: string[] }	Event emitted when a user leaves the room. The payload contains the users left in the room.	Notify the client that a user has left and display the remaining users in the room.

5.5.3 Activity Diagram

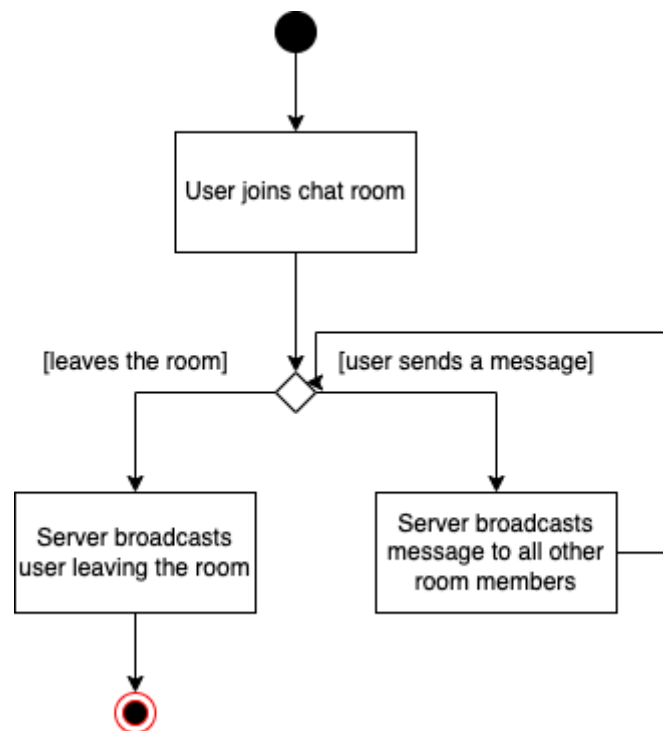


Figure: Activity Diagram for Chat Service

5.6 Non-Functional Requirements

Performance
The questions should load within 2 seconds.
Users should be sent to the room within 1s if a match is found.
The code editor should be updated within 500ms.
Users should receive chat messages within 500ms.
Users should be able to see if the other user leaves the room in 500ms.

Usability
The service should handle exactly 2 people in the room only.
Intuitive user interface.
Interface should allow users to begin using the platform intuitively without tutorial.
Users that are unable to find a match should be below 20%.
Users should not receive repeated questions.

Security
Username and password must be at least 6 characters long.
Passwords should be hashed and salted before storing in the DB.
Authentication should be handled by a secure mechanism (e.g JWT or sessions).

Portability
Must be cross-browser compatible (Chrome, Safari, Firefox, Brave).

Availability
Should be available to access 24/7.

5.6.1 Quality Attributes Prioritization Matrix

Attribute Score		Performance	Security	Usability	Portability	Availability
Performance	4		<	<	<	<
Security	1			^	<	^
Usability	3				<	<
Portability	0					^
Availability	2					

After establishing the above prioritisation matrix, our team has decided to focus on the following three main attributes arranged in priority: 1) Performance, 2) Usability, 3) Availability.

As a real-time technical interview platform, performance is of utmost importance to ensure a smooth experience for the user. Without good performance, it would most likely affect the usability attribute as well, because there is typically a low user tolerance level for latency problems for a supposed real-time application.

Usability is also crucial in order to attract new users to the platform and encourage them to stay for the long-term. This means that we should have a clean and user-friendly interface to reduce their barrier of entry to the platform.

Lastly, we want to prioritise availability so that users know that they can always rely on the presence of the platform to help them prepare for technical interviews regardless of when their interviews are scheduled.

6. Design Patterns

6.1 Pub Sub

Services that required real-time response between two users utilised the Publisher Subscriber (Pub Sub) pattern.

An implementation of the pub sub pattern is a web socket which provides a full duplex communication channel between a client and a server over a single TCP connection. In our services, we decided to use a popular web socket library, socket.io library which implements the web socket connection in JavaScript.

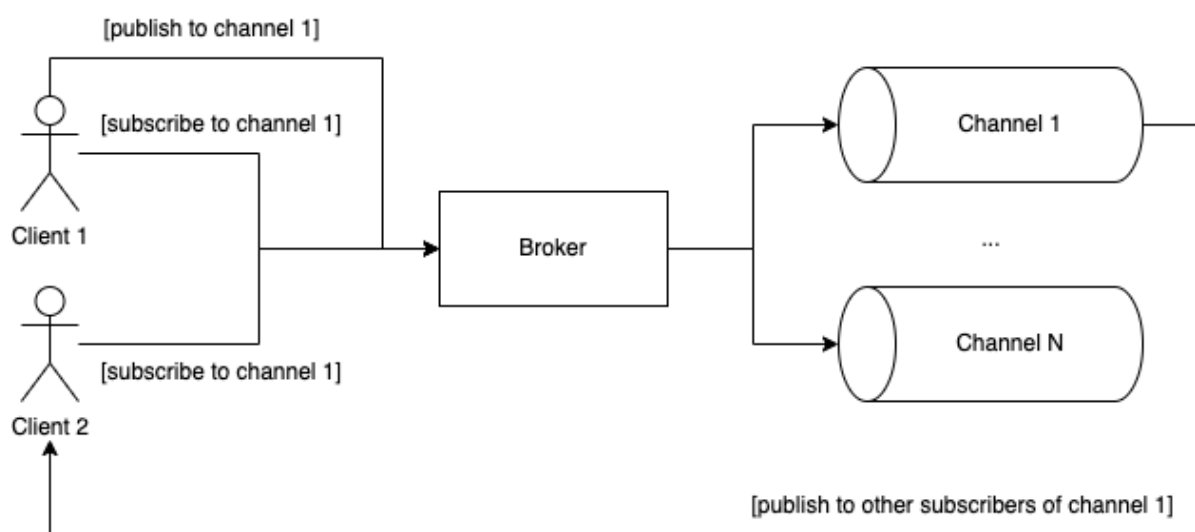


Figure: Pub Sub Pattern with Brokers

In the Matching Service, a web socket connection is established between a client, the user, and the server. This allows the server to send a message back to the user when another user who has requested a match of the same difficulty is found.

In the Collaboration Service, a room is created when the user joins the room. Multiple users would be able to join a single room. When a user makes a change to the code editor, this change is broadcast to all other code editors. Additionally, when a user rejoins the room, existing users who are in the room would broadcast their code editor states to the new user, allowing the user to have an updated state in their code editor.

In the Chat Service, a room is created when a user joins the room. Multiple users are able to join a single room. When one of the users in a room sends a message to the room, the service broadcasts this message to all other users who are in the same room, allowing users to contribute between each other within the room.

6.2 Singleton

Certain services utilised a Singleton pattern, such as the matching, chat and collaboration services, to allow for a single global access point to an instance of an object for various classes.

The services utilised a singleton pattern as part of dependency injection. In the definition of a MatchSocket, we require some implementation of a user service client and a question service client in order to communicate with the other microservices in the operation of the MatchSocket. Singletons were created for the implementation of user and question service clients so they could serve as a global access point between various classes which required the client to utilise the same instance of user and question service clients.

6.3 Decorator

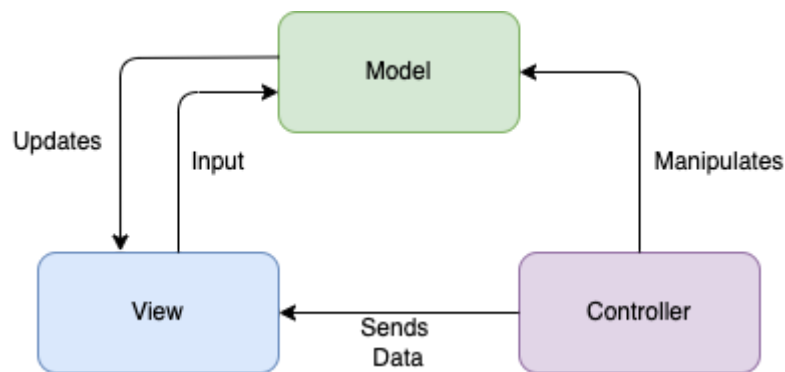
The decorator pattern is used in the construction of the API endpoints in certain backend services as it allows attaching of additional responsibilities to an object dynamically.

```
@Route('healthcheck')
@Tags('Info')
export class HealthcheckController extends Controller {
  @Get()
  @SuccessResponse(204)
  get(): void {}
}
```

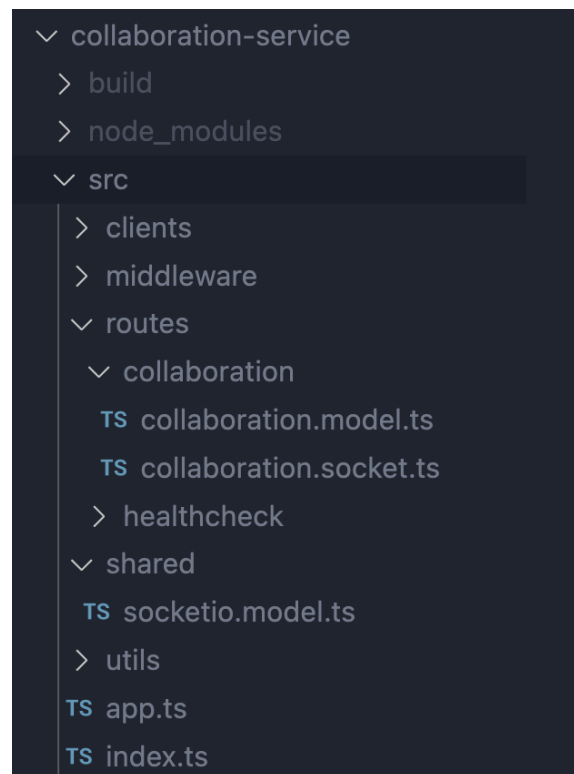
Figure: HealthcheckController which uses decorators from the `tsa` library

As shown above, the decorator pattern is a part of a JavaScript library named tsa that allows users to create OpenAPI Compliant REST endpoints with ease. Usage of the `@Route` decorator allows the specification of the URL endpoint of a particular controller, whereas the `@Get()` decorator indicates that the function binds to the HTTP request GET request and the `@SuccessResponse(204)` decorator provides the hint that if it's successful, a HTTP Code 204 is returned to the client as part of the response.

6.4 Model View Controller (MVC)



The MVC pattern was used in certain backend services to separate the concerns between the model, view and the controller. In this case, the model refers to all the various types of objects being passed around between the backend service itself, the view refers to the request and response sent between the front end and the back end and the controller refers to the handler that delegates the request to various classes on the backend.



The MVC pattern is shown as we separate the different models used by the collaboration socket into a `collaboration.model.ts` file and the business logic is contained within `collaboration.socket.ts`, serving as the controller.

Additionally, the separation between services utilised a component-style pattern where each route is a component containing its own controller and model to contain its business logic and types.

Figure: Folder structure for Collaboration Service

6.5 Dependency Injection

The dependency injection pattern was utilised in certain services utilised in the implementation of the web sockets across a few services. Dependency injection allowed better separation of concerns between the usage of a specific object and the creation of the object.

```
export class MatchSocket {  
  private io: Server  
  private userServiceClient: UserServiceClient  
  private questionServiceClient: QuestionServiceClient  
  
  constructor(  
    io: Server,  
    userServiceClient: UserServiceClient,  
    questionServiceClient: QuestionServiceClient  
  ) {
```

Figure: Source code from Matching Service, `src/routes/match.socket.ts`

In the server code which would run at runtime, dependency injection allows the user to pass any instance of `UserServiceClient` and `QuestionServiceClient` as long as it adhered to the required interface specified for the clients, reducing the coupling between the usage of the client and the construction of the client.

In test code, dependency injection allows a mock implementation of the clients to be passed in during testing, without having to amend any code in the implementation of the client from how it is being used at runtime, which would not have been possible without dependency injection.

7. CI/CD

The illustration below depicts our CI/CD process and development workflow.

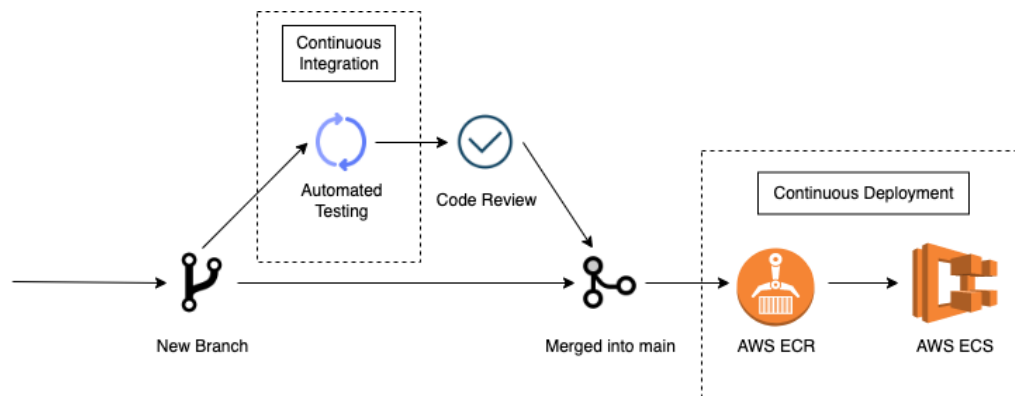


Figure: CI/CD and Forking Workflow

There are 2 parts to our entire development operations workflow: 1) continuous integration, 2) continuous deployment. Note that we are currently using GitHub Actions for our CI/CD workflow.

7.1 Continuous Integration

Whenever a pull request is made by one of our team members, it would trigger our CI GitHub actions workflow, where tests for every microservices would be run to ensure that the existing functionalities are working.

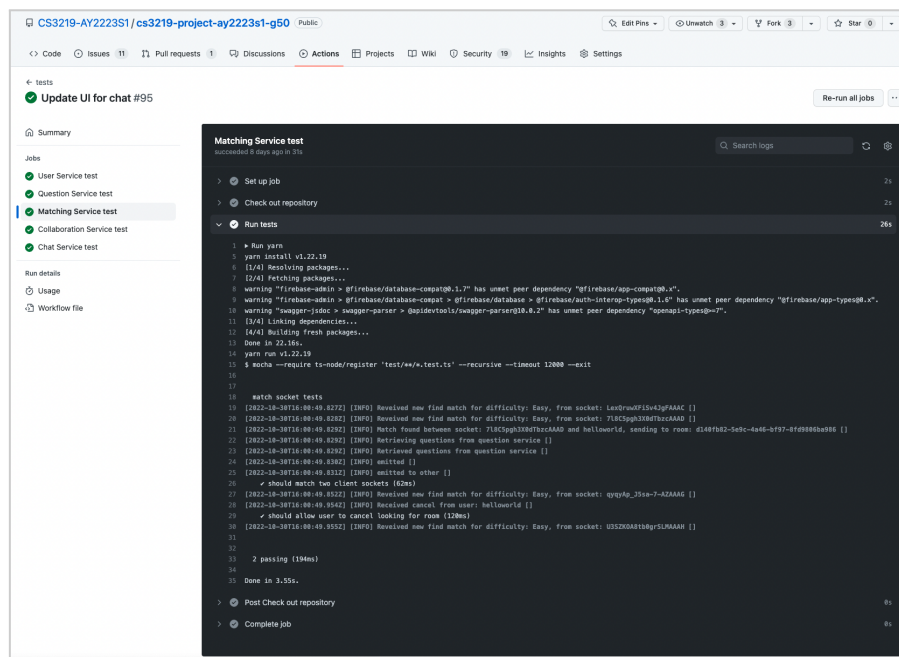


Figure: Tests running in continuous integration for each microservice

Once the request is reviewed and accepted, with all the tests passing, then it can be merged into the main branch.

7.2 Continuous Deployment

The CD process can be broken down into 2 steps.

Step 1: Build & Push Image (ECR)	Step 2: Deploy every image from their respective registry to containers in EC2 instance (ECS)
frontend_ecr	service: cs3219-project-ay2223s1-g50-service-new cluster: cs3219-project-ay2223s1-g50-cluster-new container-name: frontend container-name: user-service container-name: question-service container-name: matching-service container-name: collaboration-service container-name: chat-service
user_service_ecr	
question_service_ecr	
matching_service_ecr	
collaboration_service_ecr	
chat_service_ecr	

Note that Step 2 of the CD process requires all of the images to be pushed successfully in step 1. If step 1 fails, step 2 will not be triggered.

When the pull request is merged, it will first trigger the step 1 to build docker images of all our different services and push each of them to their respective container registry in AWS. After each successful push, it will output the image ID in the environment so that it can be used by other jobs.

	Repository name ▲	URI	Created at ▼	Tag immutability	Scan frequency	Encryption type	Pull through cache
○	cs3219-project-ay2223s1-g50-chat	582969597664.dkr.ecr.ap-southeast-1.amazonaws.com/cs3219-project-ay2223s1-g50-chat	October 26, 2022, 06:56:57 (UTC+08)	Disabled	Manual	AES-256	Inactive
○	cs3219-project-ay2223s1-g50-collaboration	582969597664.dkr.ecr.ap-southeast-1.amazonaws.com/cs3219-project-ay2223s1-g50-collaboration	October 23, 2022, 02:55:26 (UTC+08)	Disabled	Manual	AES-256	Inactive
○	cs3219-project-ay2223s1-g50-frontend	582969597664.dkr.ecr.ap-southeast-1.amazonaws.com/cs3219-project-ay2223s1-g50-frontend	October 29, 2022, 04:04:33 (UTC+08)	Disabled	Manual	AES-256	Inactive
○	cs3219-project-ay2223s1-g50-matching	582969597664.dkr.ecr.ap-southeast-1.amazonaws.com/cs3219-project-ay2223s1-g50-matching	October 23, 2022, 02:55:15 (UTC+08)	Disabled	Manual	AES-256	Inactive
○	cs3219-project-ay2223s1-g50-question	582969597664.dkr.ecr.ap-southeast-1.amazonaws.com/cs3219-project-ay2223s1-g50-question	October 03, 2022, 15:58:14 (UTC+08)	Disabled	Manual	AES-256	Inactive
○	cs3219-project-ay2223s1-g50-user	582969597664.dkr.ecr.ap-southeast-1.amazonaws.com/cs3219-project-ay2223s1-g50-user	October 03, 2022, 18:24:51 (UTC+08)	Disabled	Manual	AES-256	Inactive

Figure: AWS ECR Repositories for microservices

Once all the images are successfully pushed to ECR, step 2 of our CD workflow will commence and retrieve the image ID from the environment output of step 1, so that it can update the deployment definition with the latest image ID for each container in our deployment. Once the update is finished, it will then deploy to the appropriate cluster/service in AWS ECS.

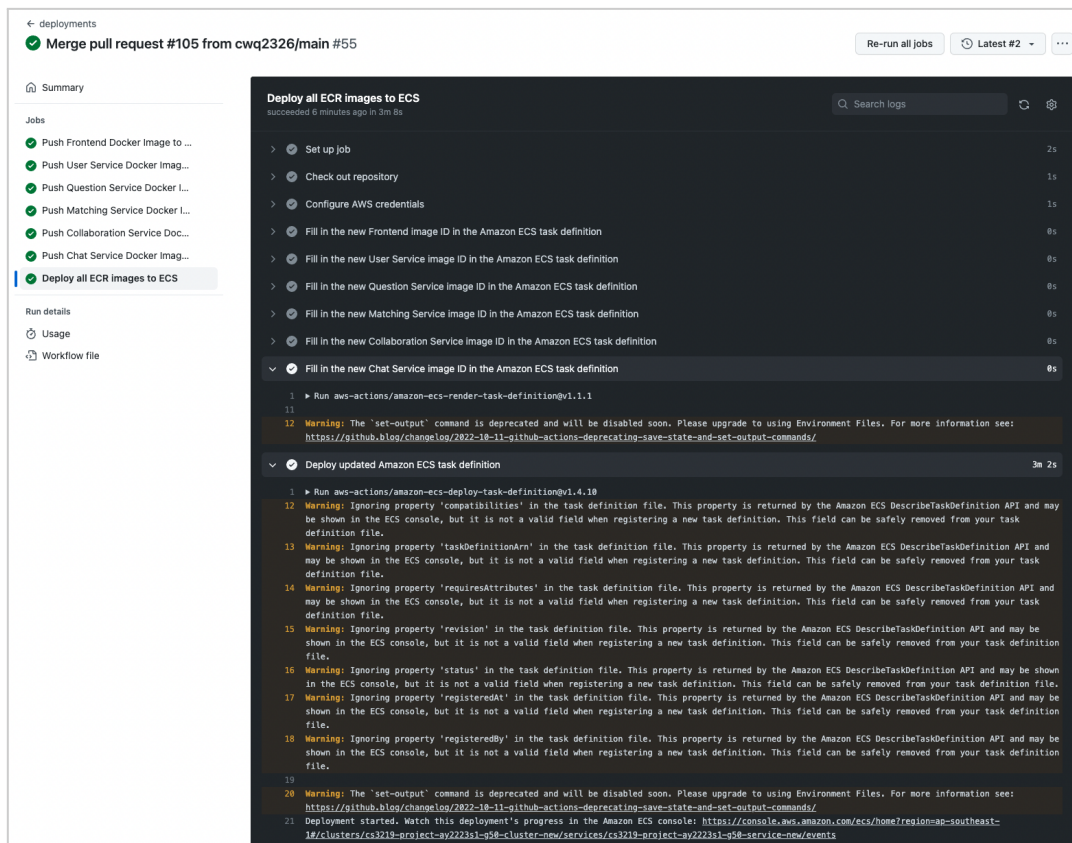


Figure: Continuous deployment for each microservice

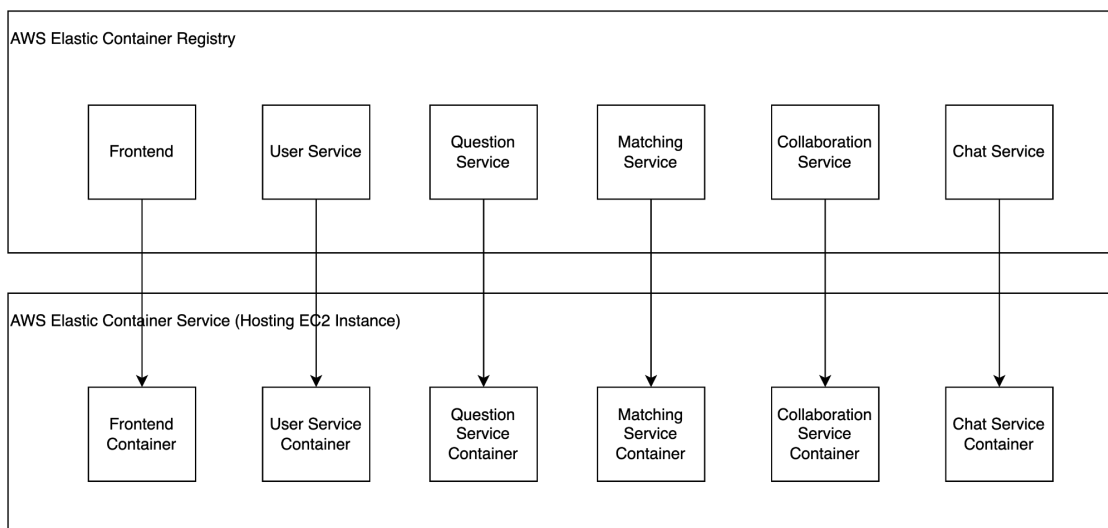


Figure: Interaction between AWS ECR and ECS

Whenever a service container or the EC2 instance goes down, AWS will spin up a new container and/or EC2 instance respectively to ensure availability of our service, so that our desired service count is met.

Do note that we currently only have 1 service instance that maps to 1 EC2 instance which hosts all the different microservices in containers as we have not prepared for our

microservices to scale (all microservices have to be deployed together). Furthermore, to scale properly, we also need to implement an API gateway / service discovery facade so that the client can be routed to the services correctly regardless of how many EC2 instances we have.

We understand that this is an anti-pattern, and we will work to improve it in the future once the microservices are refactored to scale. Once the microservices are ready to scale, we will deploy each microservices independently so that they can scale without coupling with one another – this will be the most appropriate way to deploy and scale microservices.

Ideally, we will want our future deployment architecture to roughly look like something below.

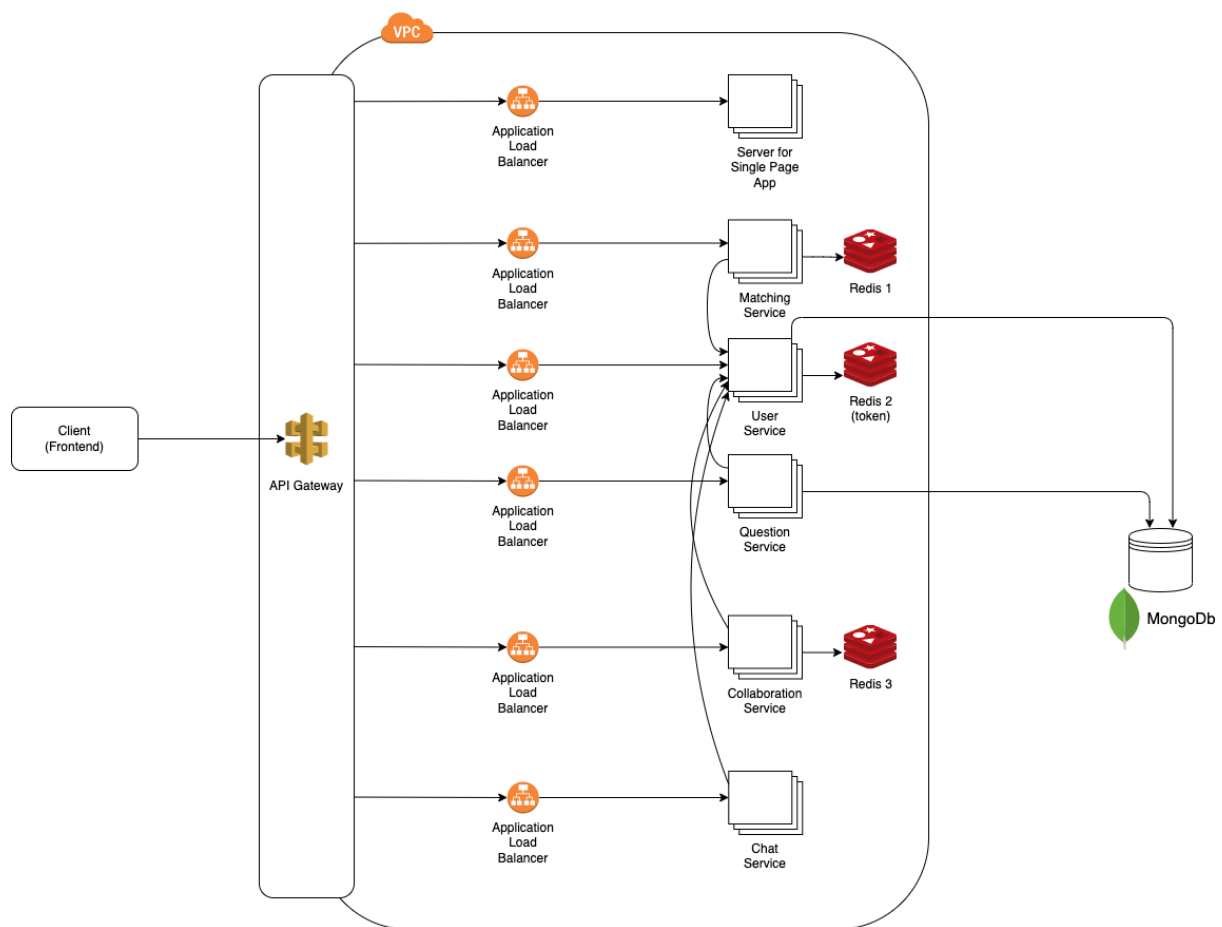


Figure: Future deployment architecture

7.3 Local Staging Environment

For the local staging environment, we use Docker to simulate our production environment where our local machine will host the different containers for all our microservices.

Therefore, it will give us a good confidence in how the production environment will behave, simply by spinning up the docker containers locally using `docker-compose` to test out our application in isolation.

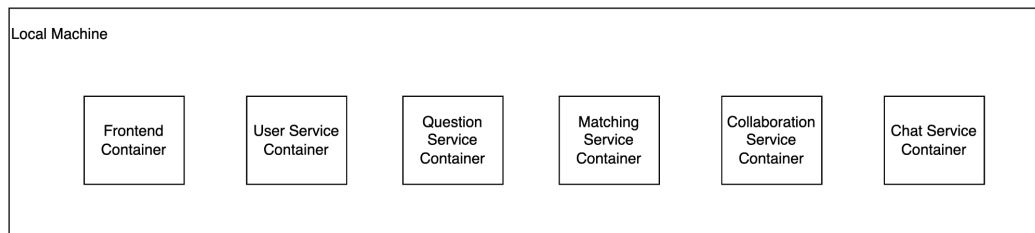


Figure: Local Docker Staging environment

8. Future Improvements

S/N	Improvements
8.1	Increase the coverage of unit test code across the code bases.
8.2	Implement end-to-end testing on the frontend and between backend microservices.
8.3	Implement additional features such as friend lists, matchmaking history, being able to match with friends as well as allowing more than 2 interviewers per session (as there may be more than 1 interviewer in a real interview).
8.4	Syncing of chat messages.
8.5	Implement features for scalability as currently our application scales vertically.
8.6	Implement API gateway (facade) and application load balancer in front of each microservice, and configure networking capabilities properly, so that we are able to scale each microservice separately.
8.7	Seek feedback from end-users so that the UI/UX can be improved.
8.8	Perform load testing on our application to determine performance metrics under load.
8.9	Implement more integration tests within each service and the frontend.
8.10	Implement an API Gateway into a Virtual Private Cloud (VPC) in order to have a reverse proxy into our backend services.
8.11	Implement horizontal scalability for our services to individually be encapsulated in a scalable cluster.
8.12	Implement feedback service with persistence so users are able to record their sessions and look through past feedback.
8.13	Implementing video and call functionality within the interview room.

9. Conclusion

In conclusion, our team has developed a web application that aims to aid Computer Science students in acing their technical interviews. We have applied the lecture contents from CS3219 in this project, such as different software development processes, specifying software requirements, software architecture design, and design principles and patterns.

Throughout this project, not only have we gained technical skills in software engineering, we have also learned and developed soft skills such as communication, conflict resolution, rapport-building and listening, problem solving, decision making, accountability, and organisation and planning skills.

Throughout the development process, our team has:

1. Identified functional requirements based on user requirements.
2. Adopted agile development framework with a 2 weeks sprint as the software development process model.
3. Developed the application using the micro-services architecture design.
4. Applied design principles and patterns in our code.
5. Deployed our code base onto a remote endpoint/ production environment using principles of CI/CD.
6. Applied tech stacks learnt from OTOT tasks.
7. Identified areas of improvements for future iterations.
8. Gain better appreciation of the design and analysis of software products.

Beyond completing all must-haves, our team has also achieved some nice-to-haves such as:

1. Deploying the application on a local machine (e.g. laptop) using native technology stack.
2. Deploying the application to the (local) staging environment (eg. Docker-based, Docker + Kubernetes).
3. Deployment of the application to a production system (eg. AWS).
4. Demonstrate effective usage of CI/CD in the project.

As a closing note, this project turned out to be harder and more time-consuming than expected amidst our busy schedule. Nevertheless, it was a fruitful and fulfilling experience that makes us more competent and prepared for future projects.