CS3219 Software Engineering Principles and Patterns

Group 61 Co-Coder Project Report

Team

Name	Matric No.
Sean Lum	A0199758Y
Wong Zhi, Chester	A0217718Y
Ng Lingshan	A0222456J
Rebecca Lau Jing Xuan	A0223029N

GitHub Repo: https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g61

Website Link: https://cocoder.link

Contributions

Name	Technical Contributions	Non-Technical Contributions
Chester	- User service backend. - User service frontend integration. - Communication service frontend integration. - Communication service backend.	- Presentation - Report - Testing - Reformatting - Diagrams
Lingshan	- Collaboration service backend. - Collaboration service frontend logic - Set up CI/CD pipeline - Configured local deployment - Configured auto-scaling - Deployed to AWS	- Presentation - Report - Conducted load testing
Rebecca	- Signup page frontend. - History service frontend. - Snackbars	- Presentation - Report - Diagrams
Sean	- Question service backend. - History service backend. - Matching service frontend and backend. - Account page frontend. - Interview page frontend. - Chat and video call frontend.	- Presentation - Report - Wireframe - Diagrams

1 Introduction

1.1 Background

Increasingly, students face challenging technical interviews when applying for jobs which many have difficulty dealing with. Issues range from a lack of communication skills to articulate their thought process out loud to an outright inability to understand and solve the given problem. Moreover, grinding practice questions can be tedious and monotonous.

To solve this issue, we have created an interview preparation platform and peer matching system called Co-Coder, where students can find peers to practise interview questions together.

1.2 Purpose

We aim to create a web application that helps students better prepare themselves for technical interviews. The web application will be a peer learning system so students keen on preparing for their technical interviews can learn from each other and break the monotony of revising alone.

1.3 Project Scope

1.3.1 Target User Profile

Our project focuses primarily on university students in Computer Science courses in Singapore that:

1. have a need to practise technical coding interviews that most companies include in their hiring process for software developers.
2. would like to practise interview and communication soft skills with another user of a similar standard.
3. wish to solve coding questions using common programming languages, namely: Java, Python, JavaScript, C, C++, and C#.
4. is familiar with or knows of Leetcode and/or similar websites and wishes to practice articulating their thought process to a peer while attempting a question from Leetcode.

2. Overall Description

2.1 Product Perspective

We propose the development of a real-time interview preparation platform, Co-Coder, that aims to match students with each other in order to practise whiteboard-style interview questions together. Students will be matched in real-time with another online student based on the selected level of difficulty (easy, medium, hard). They can collaborate on tackling the algorithmic problems via an online IDE, live chat, and a live video.

A student who is keen to prepare for his technical interviews visits the site. He creates an account and then logs in. After logging in, the student is navigated to the home page where he selects the question difficulty level he wants to attempt today (easy, medium, or hard). The student then waits until he is matched with another online student who has selected the same difficulty level as him. If he is not successfully matched after 30 seconds, he times out and will be notified. If he is successfully matched, the student is provided with the question and a free text field in which he is expected to type his solution. This free text field should be updated in real-time, allowing both the student and his matched peer to collaborate on the provided question. The matched students may choose to use the real-time chat and/or video call to communicate and practise their interview soft skills while attempting the question. After the students finish working on the question and are ready to end the session, any of them clicks on a “Finish” button which returns each student to the home page. From this page, the student can choose to log out.

2.2 Product Features

The following product features are implemented:

- User authentication
- Real-time matching of peers based on question difficulty levels
- Real-time sharing of code editor
- Real-time chat
- Real-time video streaming
- Interview questions from Leetcode
- Historical and statistical data of user’s attempt

2.3 Operating Environment

Our development takes place in both Windows and Linux environments. The server-side components are able to operate within Linux and Windows operating system environments. Whereas, the client-side components operate within typical web browser environments such as Mozilla Firefox 94+ and Google Chrome 95.

3 Software Requirement Specification

3.1 Functional Requirements

3.1.1. User Service

ID	Functional Requirements	Priority
FR-US-1	The service should allow users to create an account with a username and password.	High
FR-US-2	The service should ensure that every account created has a unique username.	High
FR-US-3	The service should allow users to log into their accounts by entering their username and password.	High
FR-US-4	The service should allow users to log out of their accounts.	High
FR-US-5	The service should allow users to delete their accounts.	High
FR-US-6	The service should allow users to change their password.	Medium
FR-US-7	Account passwords should follow the basic OWASP guideline for password length and be at least 8 characters but not more than 64 characters long.	Low

3.1.2. Matching Service

ID	Functional Requirements	Priority
FR-MS-1	The system should allow users to select the difficulty level of the questions they wish to attempt.	High
FR-MS-2	The system should be able to match two waiting users with similar difficulty levels and put them in the same collaboration room.	High
FR-MS-3	If there is a valid match, the system should match the users within 30 seconds.	High
FR-MS-4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	High
FR-MS-5	The matchmaking management should allow users to stop matching.	Medium

3.1.3 Collaboration Service

ID	Functional Requirements	Priority
FR-COL-1	The service should allow users to collaborate on a code editor	High
FR-COL-2	The service should allow users to quit the practice sessions and return to the home page.	High
FR-COL-3	The service should allow users to view the connection status of the other user.	Medium
FR-COL-4	The service should allow users to choose the shared programming language of the code editor.	Medium
FR-COL-5	The service should allow users to continue alone even if the partner disconnects.	Medium
FR-COL-6	The service should allow users to see the cursor location of the other user.	Medium
FR-COL-7	The service should allow the changes in the code editor to be synchronised between both users.	Medium
FR-COL-8	The service will show relevant popups based to notify when the partner disconnects or changes IDE language.	Low

3.1.4 Communication Service

ID	Functional Requirements	Priority
FR-COM-1	The service should allow users to join a video call with their partnered user.	High
FR-COM-2	The service should allow users to leave a video call with their partnered user.	High
FR-COM-3	The service should allow users to chat with their partners in real time.	Low

3.1.5 Question Service

ID	Functional Requirements	Priority
FR-QS-1	The service should provide a random question from Leetcode based on the difficulty selected (Easy, Medium, Hard)	High
FR-QS-2	The service should provide a fallback question in the event Leetcode APIs fails to response based on the difficulty selected	Medium

3.1.6 History Service

ID	Functional Requirements	Priority
FR-HS-1	The system should maintain a record of the past attempted questions.	Medium
FR-HS-2	The system should allow users to retrhieve all past data from the system	Medium
FR-HS-3	The system should allow users to retrieve historical statistical data from the system. (e.g. number of easy questions solved, total questions solved)	Low
FR-HS-4	The system should delete users history upon account deletion	Medium

3.1.7 Application Client

ID	Functional Requirements	Priority
FR-AC-1	The Application Client should provide a landing page for users to sign in or sign up for an account	High
FR-AC-2	The Application Client should provide the user an interface to select the difficulty of the question	High
FR-AC-3	The Application Client should provide the user an interface to start matching for an interview sessions	High
FR-AC-4	The Application Client should provide the user an interface to view a list of past interviews details	High
FR-AC-5	The Application Client should provide the user an interface to change password and delete account	High
FR-AC-6	The Application Client should provide the user an interface to code and view the questions	High
FR-AC-7	The Application Client should provide the user with a way to navigate between pages.	High
FR-AC-8	The Application Client should provide the user an interface to look at overall statistics	Medium
FR-AC-9	The Application Client should allow users to know the connection status of the other. (i.e., a snack bar to know language changed)	Low
FR-AC-10	The Application Client should provide users a way to chat with one another	Medium

FR-AC-11	The Application Client should provide the user a way to video call with one another	Medium

3.1.8 Deployment

ID	Functional Requirements	Priority
FR-D-1	The application can be deployed on a local machine (e.g., laptop) using native technology stack.	High
FR-D-2	The application can be deployed on a staging environment (e.g. Docker-based, Docker + Kubernetes).	High
FR-D-3	The application is deployed on a production system (e.g. AWS cloud platform)	High
FR-D-4	The deployed application should demonstrate easy scalability.	Medium
FR-D-5	The application should have an API gateway that redirects requests to the relevant microservices.	Medium
FR-D-6	The application should demonstrate service discovery or implement a service registry.	Medium
FR-D-7	Demonstrate effective usage of CI/CD in the project.	High

3.2 Non-Functional Requirements

	Score	Availability	Compatibility	Performance	Scalability	Security	Usability
Availability	2		^	<	^	^	<
Compatibility	1			^	^	^	^
Performance	4				<	<	^
Scalability	2					^	^
Security	3						^
Usability	4						

We prioritized our NFRs as per the table above. We are primarily focused on making our product look and feel good, thus, usability and performance NFRs are of higher importance.
Security is of next importance as we recognize that basic web security is necessary for our web application to ensure the privacy of users, especially since we are providing a video and audio service. Our application does not store any sensitive user data besides a username and password for our own domain.
Availability and scalability requirements are met with the help of Kubernetes and Docker deployment to AWS Cloud Services (see 8.1 Deployment).

3.2.1 Availability

ID	Non-Functional Requirements	Priority
NFR-AVA-1	The system should maintain an uptime of 99% during peak hours such as during the afternoon period.	High

3.2.2 Compatibility

ID	Non-Functional Requirements	Priority
NFR-COM-1	The web application should work on any latest two versions of commonly used browsers. Namely, Microsoft Edge, Google Chrome, Mozilla Firefox, and Apple Safari.	Medium

3.2.3 Performance

ID	Non-Functional Requirements	Priority
NFR-PER-1	The collaboration service should synchronise text and cursor activity with a latency of fewer than 2 seconds upon user input, with stable internet connection.	High
NFR-PER-2	The response to any user action should become visible within 5 seconds, with stable internet connection.	High
NFR-PER-3	In normal network conditions, any valid REST API request to backend services should take no longer than 3 seconds to receive a response if successful, and timeout within 30 seconds if the request failed.	Medium

3.2.4 Scalability

ID	Non-Functional Requirements	Priority
NFR-SCA-1	The system should be able to support at least 500 users accessing the website simultaneously	Low
NFR-SCA-2	The system should be able to host up to 250 pairs of matched users concurrently in the collaborative coding space	Low
NFR-SCA-3	The system should able to store information of at least 500 users	Low

3.2.5 Security

ID	Non-Functional Requirements	Priority
NFR-SEC-1	User passwords should be hashed and salted before storing in the database.	High
NFR-SEC-2	Sensitive microservices should be configured such that only authenticated parties and services are allowed to access the backend and databases.	Medium
NFR-SEC-3	Only the authorised user (or peers) should have access to their owned peer-to-peer connections and peer-matched functions (i.e., the interview room should only be accessed by the two matched users).	Medium
NFR-SEC-4	The web application should use HTTPS protocol to transmit data securely.	Medium

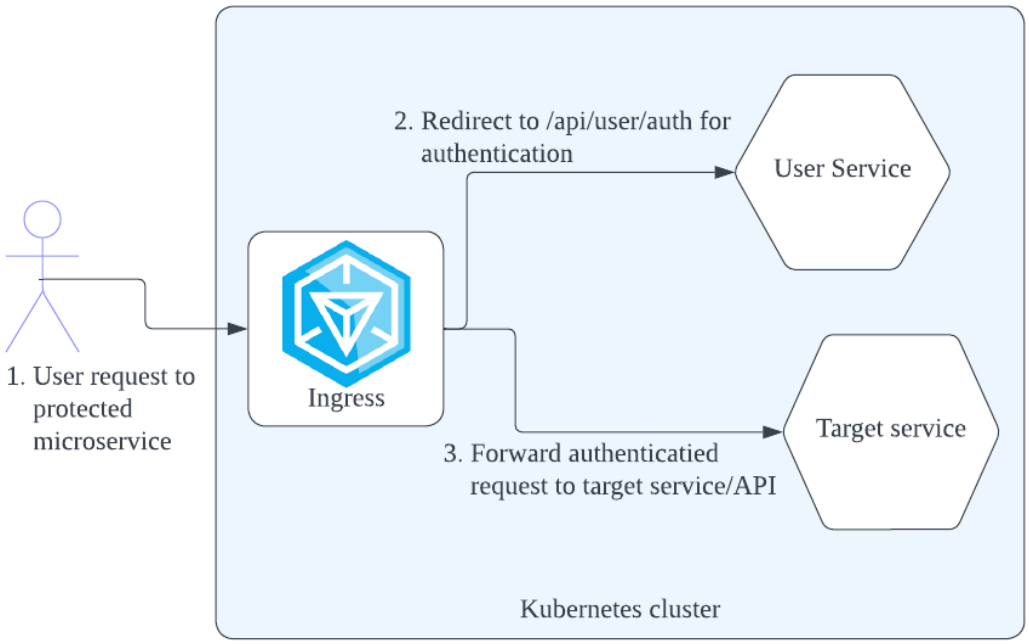
3.2.5.1 Implementation Details

NFR-SEC-1:
We use the bcrypt library as per OWASP guidelines to salt and hash passwords before storing them securely in MongoDB.

```
//schema middleware to apply before saving
UserModelSchema.pre<IUserDTO>('save', async function (this: IUserDTO, next) {
  //hash the password, set hash cost to 10
  this.password = await bcrypt.hash(this.password, 10);
  next();
});
```

Code snippet showing usage of bcrypt password hashing function.

NFR-SEC-2:
All backend services sit behind an NGINX Ingress Controller which will authenticate sensitive requests before re-routing them to their respective services. (FR-D-5) The diagram below shows the authentication flow.



Simplified architecture diagram to show user request authentication flow.

Sensitive requests to authenticated endpoints require a Cookie request header with the user's JWT token. I.e., Request Headers: Cookie: jwt=<cookie value>. **Only** the following endpoints are designed as public and do not require this request header: User Login, User Logout, Create User, and User Public Information.

NFR-SEC-3:
Users must be authenticated to access their peer-to-peer connections and peer-matched functions. This allows the respective services to confidently check if the current user is indeed an authorised owner of the connection (i.e., one of the users of the interview room).

NFR-SEC-4:

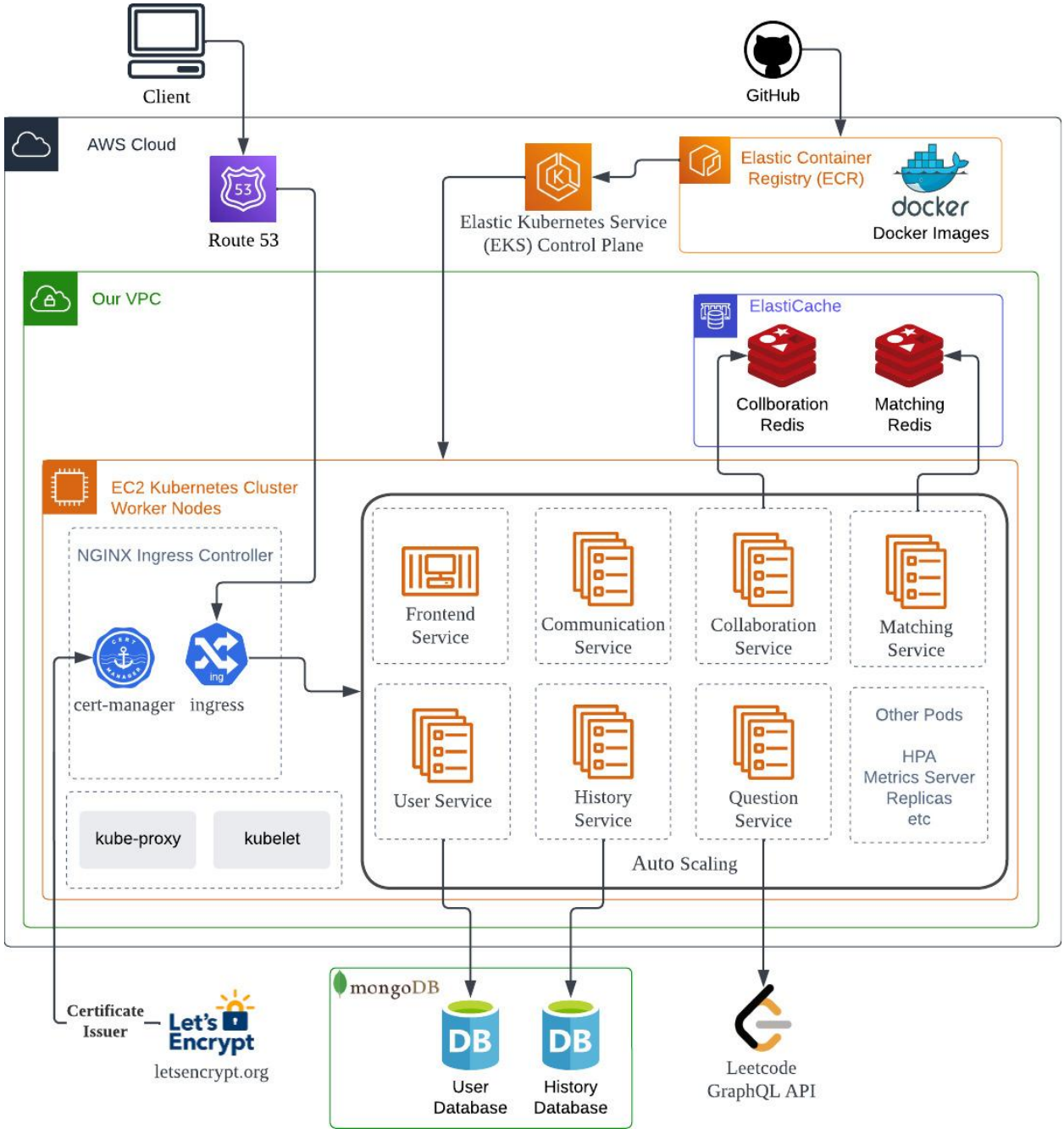
Our website is secured using HTTPS protocol with a certificate issued by Let's Encrypt (see also: Architecture Diagram). This would ensure that internet traffic is secured by TLS.

3.2.6 Usability

ID	Non-Functional Requirements	Priority
NFR-USA-1	90% of users should be able to figure out any application functionality on their first attempt.	High
NFR-USA-2	For input fields with a limited set of values, it must be possible for the user to select the value from a list.	High
NFR-USA-3	The user can match with another user within 3 mouse clicks from the sign-in page.	High
NFR-USA-4	Icons measure at least 1 square centimetre on a 10.1-inch display	Low

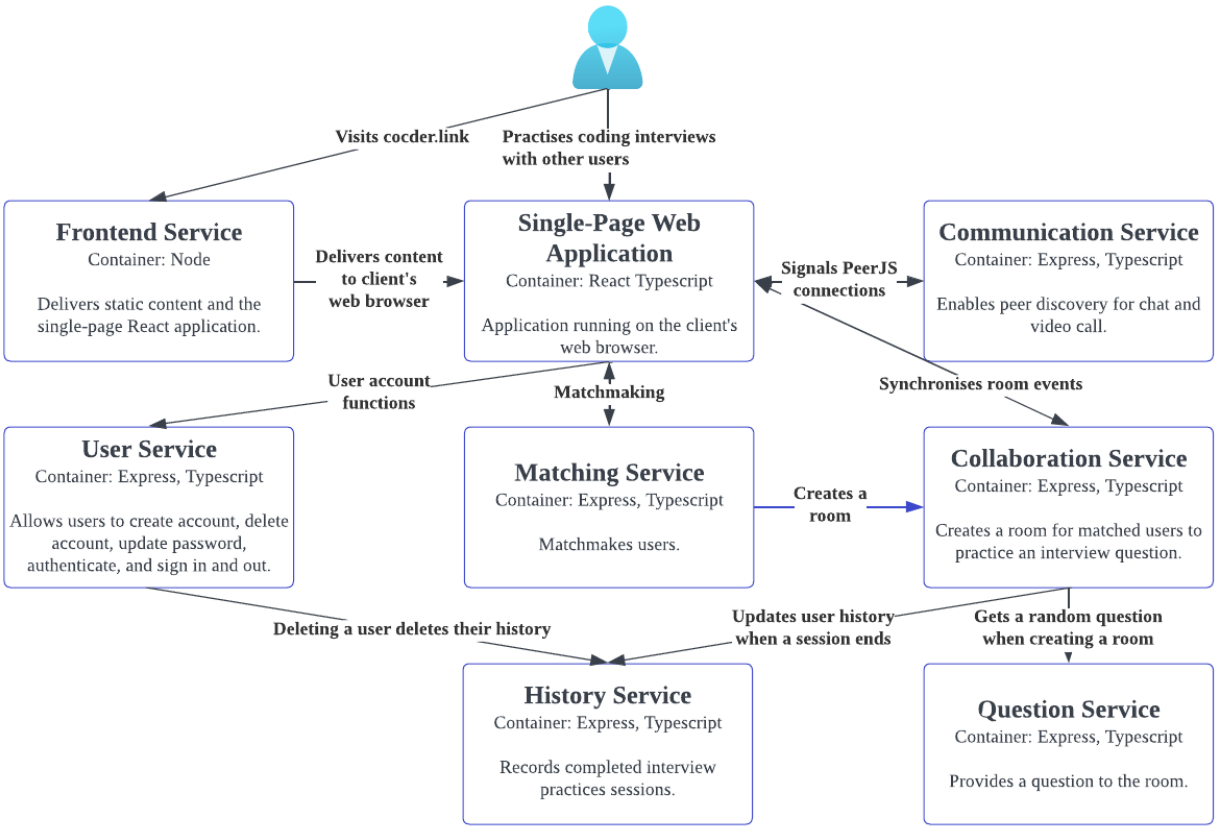
4 Architecture

4.1 Architectural Diagram



Our application utilises a microservice architecture, where each microservice has its own separate technology stack and own database. There is an NGINX Ingress controller which acts as a single entry point for the microservices, handling authentication and routing of traffic to the services. Our application is being deployed on the AWS Cloud.

A closer look at how the services interact:



Interactions between services.

4.2 Architecture decisions

4.2.1 Monolith vs Microservice

Consideration of architecture design to use		
	Microservice (chosen)	Monolith
Pros	- Reduces coupling between each logical component. - Easier to test each service separately. - Easier to scale for each individual service. - Easier to specify load requirements - Simplifies designating work by assigning members to be in charge of specific bounded contexts. - Relatively more reliable as an outage of a single service might not affect every other service due to less coupling and splitting of responsibilities.	- Easier continuous development. - Reduced latency as compared to microservice as it does not involve many services communicating with each other over a network - Ease and speed of development and configuration.
Cons	- Additional complexity. - Additional setup.	- Less reliable since most of the time, it is deployed as a large service. If any part fails, the whole thing fails. - The code base might become large and complicated.
Key considerations/ deciding factors: Reliability, separation of concerns and segregation of responsibility, and ease of extension.		
Final Decision: Microservice architecture allows us to decouple our backend system such that each service can focus on what it needs to do. This allows better separation of concerns as well as segregation of responsibilities among the members of the group. Each service would be easier to manage and team members can work independently. Additionally, we feel that microservice architecture allows new services to be added easily without much coupling. Though microservice architecture does have its own complexity of setting up, we feel that the advantage of using it outweighs the cost of implementing it.		

4.2.2 Microservice: Shared database vs Database per service

Comparison is referenced from https://microservices.io/patterns

Consideration of microservice database architecture to use		
Database	Database per service (chosen)	Shared database
Pros	- Helps ensure that the services are loosely coupled. - Each service can use the type of database that is best suited to its needs. - Lower throughput per database	- Can use familiar and straightforward ACID transactions to enforce data consistency. - A single database is simpler to operate.
Cons	- Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don't support them. - Implementing queries that join related data in multiple databases is challenging. - Complexity of managing multiple SQL and NoSQL databases	- Development time coupling: a developer working on, for example, "Service A" will need to coordinate schema changes with the developers of other services that access the same tables. - Runtime coupling: because all services access the same database they can potentially interfere with one another. For example, if a long-running transaction from "Service A" holds a lock on the table then another "Service B" might be blocked. - Single database might not satisfy the data storage and access requirements of all services. (e.g., row-based security) - Greater throughput per database
Key considerations/ deciding factor: Appropriate database for each service, reduced coupling of databases, and throughput of data.		
Final Decision: Database per service is chosen as one of our key considerations was having the appropriate database for each service. For instance, Redis' speed might suit the use case of a certain service better compared to a normal RDMS, or vice versa. Furthermore, the loose coupling of the design allows the team to develop, deploy, and scale the databases independently and ease the replacement/migration of a database for any particular service in the future. As the team capitalises on the microservice architecture to delegate work to individual members, using the database per service architecture also greatly improves our development speed. The owner of a microservice can work on it completely independently without having to liaise and wait for other busy team members to configure or grant database access, etc. Moreover, with the relatively simple data requirements of our project, there is no specified use case that necessitates a complex joining of data owned by multiple services. Using shared databases will also result in larger data throughput and storage requirements, leading to possible network congestion and a lower threshold of utilising free-tier storage options.		

4.2.3 Database: SQL vs NoSQL (MongoDB)

Consideration of database to use for User Service and History Service		
Database	MongoDB (chosen)	SQL
Pros	- Multi-record ACID transactions. - Supports both vertical and horizontal scaling. - Simple queries on de-normalized data. - Flexible schema. - High availability for free (MongoDB Atlas: 99.995% SLA) - More suitable for distributed database systems. - Consistency, availability, and performance can be traded based on the needs of the application (CAP Theorem)	- Multi-record ACID transactions. - Easy to scale vertically. - Strong data consistency. - Can handle complicated queries. - Reduced data duplication and redundancy.
Cons	- Weaker data consistency (due to eventual consistency model).	- Hard to scale horizontally. - Potentially less performant owing to strong consistency properties (notably when database locks are used). - Relational model requires a well-defined schema. - Creating relational data models takes time. - Slower queries if complicated joins are required. - Typically requires an object-relational mapping (ORM)
Key considerations / deciding factors: Development ease, future proofing, and frequent application changes driven by AGILE and DevOps practices.		
Final Decision: MongoDB is used for both user service and history service. The current data requirements for the above two services are fairly simple; personal user information for user service, and user statistics for history service. However, the data model is likely to change in future plans. NoSQL allows the team to easily make changes to the database as requirements change due to its flexible schema, allowing us to iterate quickly and continuously integrate new application features to provide value to users faster. It also allows us to store data in the same way it is used in our application code. This means less code, faster development time, and fewer bugs. Furthermore, using NoSQL helps future-proof the application by providing an added dimension of scalability. Firstly, adding additional machines through horizontal scaling will increase the application's overall performance threshold. Secondly, horizontal scalability means being able to add cheaper		

commodity servers whenever there is a need, and exploiting more free-tier services. Thirdly, it is easier to achieve greater topographic distribution. NoSQL is simply more suitable for web-scaling applications

Theoretical performance differences are not a key consideration as it is largely dependent on the underlying hardware (which should be as cheap as possible until we develop a viable business model).

We also do not foresee having to use complicated queries in our application.

4.2.4 Container Orchestration: AWS EKS vs ECS

Consideration of tool to use for container orchestration on AWS		
	Elastic Kubernetes Service (EKS)	Elastic Container Service (ECS)
Pros	- Provides and manages a Kubernetes control plane on its own - Can easily run tooling and plugins developed by the Kubernetes open-source community. - Kubernetes platform is available on all of the major public clouds and can easily be extended or migrated out of AWS - Kubernetes comes with built in DNS features and service discovery	- ECS integrates better with other Amazon cloud services because it's native to the platform and relies on a proprietary control plane - Full control over instance types
Cons	- Kubernetes deployment requires expert knowledge and more time to configure	- Cannot extend it into other clouds or move hosted workloads directly to another location - ECS service discovery uses Route 53 resources that comes at a cost
<u>Key considerations / deciding factors:</u> Portability, community support and resources		
<u>Final Decision:</u> We decided to use AWS EKS to deploy our application as we wanted to leverage on Kubernetes which comes with a plethora of tools and Kubernetes-compatable software that can be used to improve and extend our application architecture. We can also run our containerised application anywhere without needing to change our operational tooling. For example, we can test our application on a local kubernetes cluster and directly deploy it to the AWS cloud without having to make any code changes. In future, if we were to decide to extend or move our application to other cloud platforms, the migration will be much smoother. Although there is a learning cost that comes with Kubernetes, this is not an issue as there is sufficient online resources and time for us to learn to use it. Another benefit of using EKS is that Kubernetes comes with built-in free of charge service discovery features in contrast with the ECS service discovery which comes with a cost, albeit the latter provides a more robust solution. However, our application does not require complex service discovery features so our final decision was to use AWS EKS to deploy our application.		

4.2.5 API gateway: NGINX Ingress Controller

Consideration of NGINX Ingress Controller as an API gateway (FR-D-5)
Key considerations of API gateway • Can route API requests to the different microservices • Can authenticate and authorise API requests • Can support TLS authentication • Websockets should be supported to allow the matching, collaboration and communication services to work • Should work with services deployed on Kubernetes
Final Decision: The NGINX Ingress Controller was used as an API gateway as it is a fairly mature ingress solution for deployments on kubernetes and comes with a lot of out-of-box features. It is a specialized proxy that gets traffic into Kubernetes and to the services based on routing rules that can be configured. It also supports authorization-based routing whereby API requests need to be successfully authenticated by the user service before being forwarded to other services. It is compatible with protocols like HTTP/HTTPS and WebSocket. Moreover, TLS can easily be configured for ingress resources with support for automated certificate management. Cert-manager automatically requests missing or expired certificates from a range of supported issuers (including Let's Encrypt) by monitoring ingress resources, ensuring the security of the application (NFR-SEC-4).

5 Design patterns

5.1 Publisher-Subscriber Pattern

This pattern is used in both our communication and collaboration service to enable real-time updates for the chat and coding platform. This pattern is used in Socket.IO to allows bidirectional communication whereby users and microservices can publish and subscribe to messages sent over a WebSocket. There is less coupling as the endpoints do not need to know about each other (Frontend/backend) and both simply just need to abide by the contract of publishing a certain “event” and subscribing to a certain “event”.

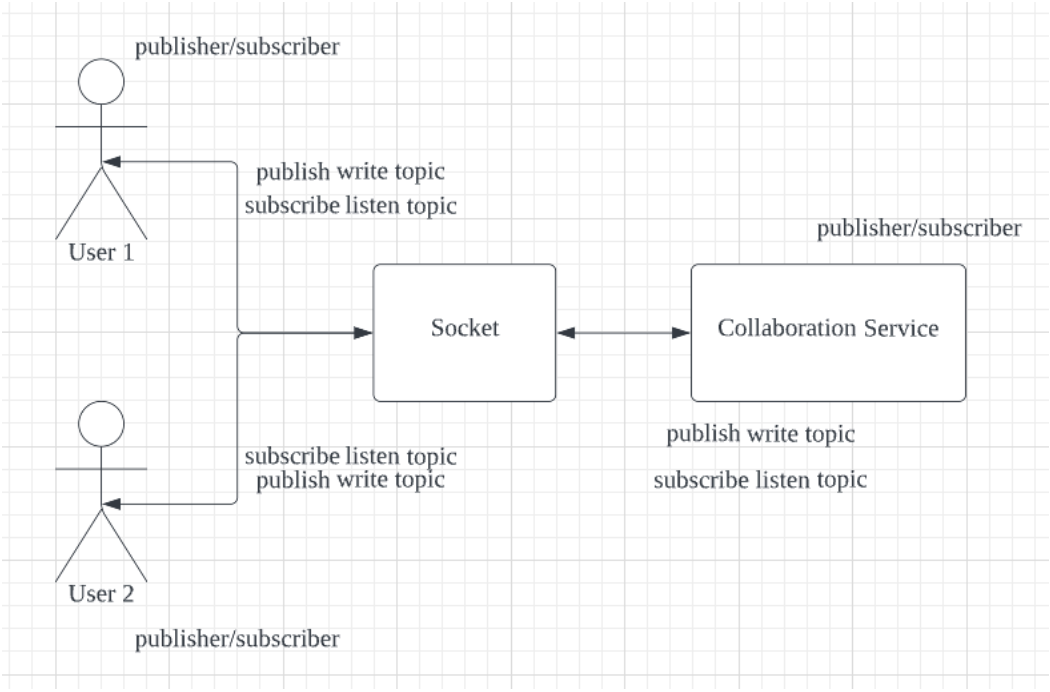


Diagram illustrating implementation of pub-sub pattern

5.2 Repository-Service-Controller Pattern

This pattern is used in our backend services as it provides a clean separation of concerns within each backend service. The controller can be thought of as the interface of this whole service as its sole responsibility is to expose the functionality of a particular service and handle REST requests/responses. The service layer is a more well-defined layer that handles the business logic of the microservice such as making calls to the repository to fetch or manipulate some data. The repository layer itself is an interface that allows the service layer to manage persistence. The repository layer will act as the data layer which communicates with the actual database. You can view this as a “layered architecture” whereby only the top layer will talk to the lower layer and the lower layer doesn't know anything about the top layer. This allows for a cleaner separation of concerns as each layer handles its own responsibility. Furthermore, in the event there is a change to the database in the future, the interface of the repository layer does not change, thus, other layers are not affected. In backend services where database operations are trivial, the repository layer is omitted.

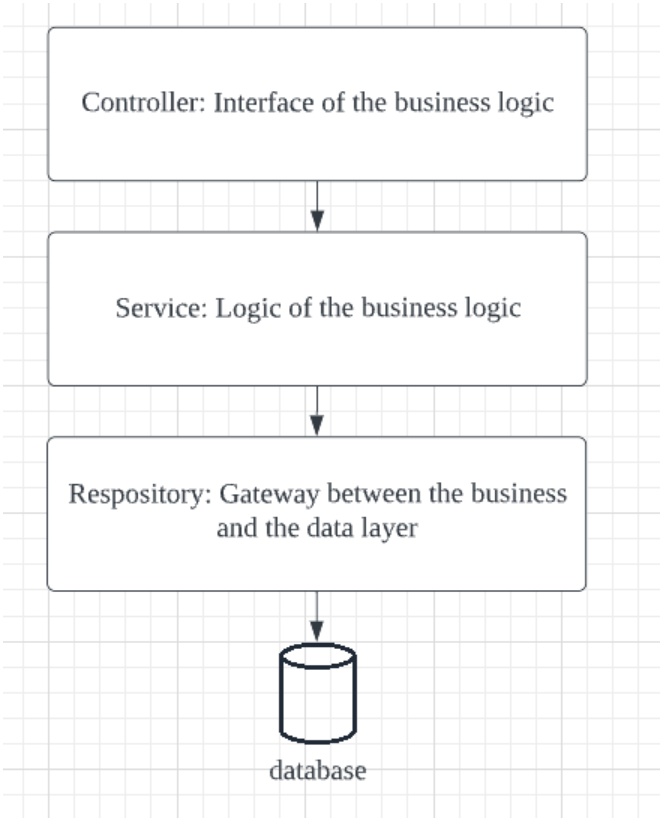


Diagram illustrating of repository-service-controller pattern

5.3 Synchronous Request/Response pattern

A use case of such a pattern will be between User Service and History Service microservices. This pattern is adopted to ensure that whenever a User is deleted before it is actually removed from persistence, we will clear all the history regarding a user and then delete the user itself. By having such a pattern, we are able to wait for a successful history delete response before deleting the user itself. However, we would like to note that there are many other patterns or architectures which actually provide better data consistency but due to the complexity of implementation and time constraints, we decide to go with the Request/Response pattern.

Other considerations:

1) Saga-pattern (Choreography / Orchestration)
    a) This allows eventual data consistency even in times of failed operation. This is due to the usage of a messaging queue that stores each event and is retractable or by an orchestrator whose responsibility is to ensure data consistency and that each operation is executed.
2) Using UUID instead of username as primary key (this is because the username is given to be unique as a specification) By using UUID, we can ensure that deleting a user will still ensure uniqueness when we query from the history service. Furthermore, we can in fact reduce coupling between both services by scheduling a cron job to run checks in user service and delete missing entries inside the history service.

5.4 Observer Pattern

This pattern is used in the frontend to reduce coupling between the components. An example usage of this pattern is used in displaying the video call screen. Both users are able to see each other as the video is mounted in both browsers. However, upon one user disconnecting, instead of re-rendering the whole DOM tree in passing through states, we use the observer pattern to simply remove the video DOM element. Note that though we used the words `subscribe` and `publish`, the implementation of the observer is not following the pub-sub model. `subscribe` is akin to addObserver and publish is akin to notifyAllObserver. The below code shows an example of the usage:

```
useEffect(() => {
   const subscriber1 = observer.subscribe('partnerCloseCall', () => {
     removeVideoStream(remoteVideoRef.current!);
   });

   const subscriber2 = observer.subscribe('partnerOpenVideo', (stream: any) => {
     const refinedStream = stream as MediaStream;
     addVideoStream(remoteVideoRef.current!, refinedStream);
   });

   return () => {
     subscriber1.unsubscribe();
     subscriber2.unsubscribe();
   };
 }, []);
```

Our observer subscribes to the above events and is notified when a connected peer opens or closes the video stream over PeerJS. The pattern allowed the video call component to be separated into the observer component above concerned with mounting and unmounting the video stream in the UI, and a PeerJS WebRTC component concerned with providing peer-to-peer connectivity. This allows a clean separation of concerns and less coupling between the UI and PeerJS.

6 Services

6.1 User Service

6.1.1 Overview

This service facilitates user creation, user deletion, and updating user information. It also facilitates login and authentication via JWT cookie.

Technology Stack	- TypeScript - bcrypt - Express - Dotenv - MongoDB - Mongoose
Backend Schema	MongoDB NoSQL document User Document: { username: string, password: string }

6.1.2 API Endpoints

6.1.2.1 Authentication

GET /api/user/auth

- Provides basic authentication of a user via JWT authentication cookie, following ingress-nginx authentication standards.
- Any sensitive request will pass through this endpoint and will only be forwarded upon a successful OK response.
- The frontend calls this endpoint to set the user context to show protected routes and set personalised content.

6.1.2.2 User Login

POST /api/user/login

- Logs the user in and returns a secure JWT authentication cookie (FR-US-3).

Request JSON Body: { “username”: “string”, “password”: “string” }

Example response header:

- Set-Cookie:jwt=<cookie value>; Path=/; Expires=<1 day>; HttpOnly; Secure; SameSite=None

6.1.2.3 User Logout

GET /api/user/logout

- Logs the user out and invalidates the user’s JWT authentication cookie (FR-US-4).

Example response header:

- Set-Cookie: jwt=”loggedout”; Path=/; Expires=<1 day>; HttpOnly; Secure; SameSite=None

6.1.2.4 Create User

POST /api/user/user

- Creates a new user (FR-US-1) with a unique username (FR-US-2) and basic password validation (FR-US-7). This endpoint is used to implement user sign-up.

Request JSON Body: { “username”: “string”, “password”: “string” }

6.1.2.5 User Public Information

POST /api/user/user/{user}

- Retrieves a user’s public information to display on the user’s account page.

6.1.2.6 Update User Password

PUT /api/user/user/{user}

- Updates user password (FR-US-6) with basic password validation (FR-US-7) if the old password provided is correct.

Request JSON body: { “oldPassword”: “string”, “password”: “string” }

6.1.2.7 Delete User

DELETE /api/user/user/{user}

- Irreversibly deletes the specified user (FR-US-5).

6.2 Matching Service

6.2.1 Overview

This service facilitates the matching of two users who have selected the same question difficulty.

Technology stack	- TypeScript - UUID - Express - Dotenv - Socket.io - Redis
Backend Schema	Redis key-value pair Key: difficulty (since this is based on question difficulty) Value: { username: { username, roomID } }

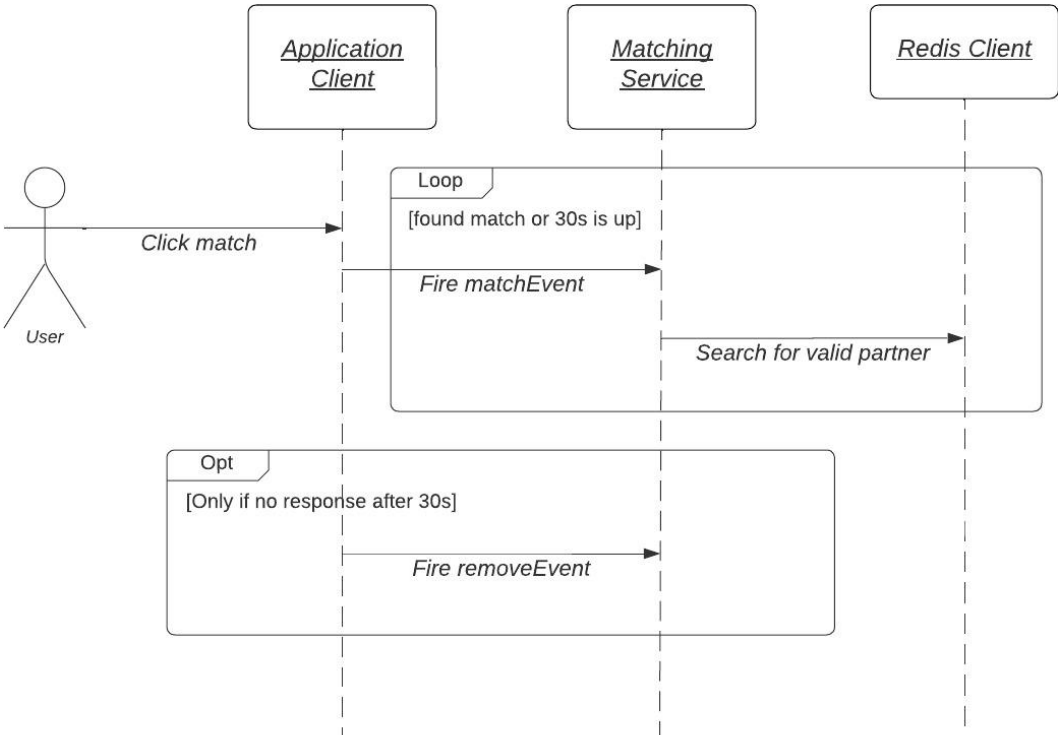
6.2.2 Exposed Socket Events

The frontend creates a web socket connection to the matching service using Socket.IO.

Events	Functionality
matchEvent	Fired when a user is attempting to match. The service will respond with a **matchSuccessEvent** if matched or a **matchFailureEvent** after 30s without a match. (FR-MS-1)
matchSuccessEvent	Upons successful match found in the backend within 30s, fires **matchSuccessEvent** to the frontend in a specific room to notify users of that room that they are matched (FR-MS-2 and FR-MS-3)
matchFailureEvent	After 30 seconds of searching in the backend for a match for the same difficulty, if no such exist, fires a **matchFailureEvent** to the frontend to notify the user who is searching that no match found. (FR-MS-4)
removeEvent	Frontend will also fire a **removeEvent** in the event when the backend does not respond in the time during the 30s to terminate the matching process (FR-MS-5)

6.2.3 Implementation Detail

When a user click find match, it fires the **matchEvent** to the **MatchingService** which will **createMatch** in the temporary in memory Redis storage. For 30 seconds, **MatchingService** will continue searching the Redis Client for valid partner based on the difficulty selected. If upon successful match, **MatchingService** will create a room and delete the two candidates from the pool of candidate and fire a **matchSuccessEvent** to the **application client**. Else, it will simply delete the two candidates from the pool and fire a **matchFailureEvent.** In order to ensure each room is unique, the UUID package is used to ensure unique string ID is generated for each room. In the event that the **MatchingService** did not managed to send an event to notify the **application client** that the 30s has ended, we enforced the **application client** to also count till 30s and fire a removeEvent to the **MatchingService**.



Sequence diagram of matching service

6.2.4 Redis vs SQLite

Consideration of database to use			
Database	Redis (chosen)	SQLite	Memcached
Pros	- Simple key-value store. - In memory. - Fast lookup time. - Easy to set up. - Easily scalable by using existing out-of-the-box services like Amazon ElastiCache for Redis. - Provides expiration of keys which adds another layer to ensure 30s timeout on users. - High availability.	- Fast lookup time - Can be in memory/acts as normal DBMS - Persistency if implemented as a normal DBMS - High availability	- Simple key-value store. - In memory - Fast lookup time.
Cons	- Transient, thus data stored inside might not persist after a restart.	- Larger complexity in setting up without additional benefits - Less scalable than Redis.	- Least familiar within the team
Key considerations / deciding factors: Speed, ease of setting up, ease of scalability and availability.			
Final Decision: Since speed is key in our design, we choose Redis or SQLite. Since we don't expect too many unmatched users since we are just matching on difficulty i.e 3 possible fields, based on the research here, it shows that Redis outperforms SQLite Also, Redis is easier to set up compared to SQLite and there are out-of-the-box services to scale Redis easily. Comparing Redis to Memcached, it isn't as reliable and scalable as Redis, thus ultimately Redis as the database/cache is chosen.			

6.3 Collaboration Service

6.3.1 Overview

This service facilitates user-to-user communication and acts as an orchestrator in the room creation/deletion process by fetching questions from the question service and updating user data in the history service. It provides a publisher-subscriber service for room events as well as augments CodeMirror with enhanced functionality to provide a feature-rich collaborative coding experience.

Technology Stack	- Typescript - Express - Dotenv - Socket.IO - Redis

Backend Schema	Redis key-value pair
	Key: roomId
	Value: {
	users: {username, connected}[];
	text,
	questionData,
	language
	};

6.3.2 API Endpoints

6.3.2.1 Create Room

POST /api/collaboration/createRoom

- This endpoint is used by the matching service upon successful matchmaking. A new, unique room will be created in the backend which stores information on the room users and question.

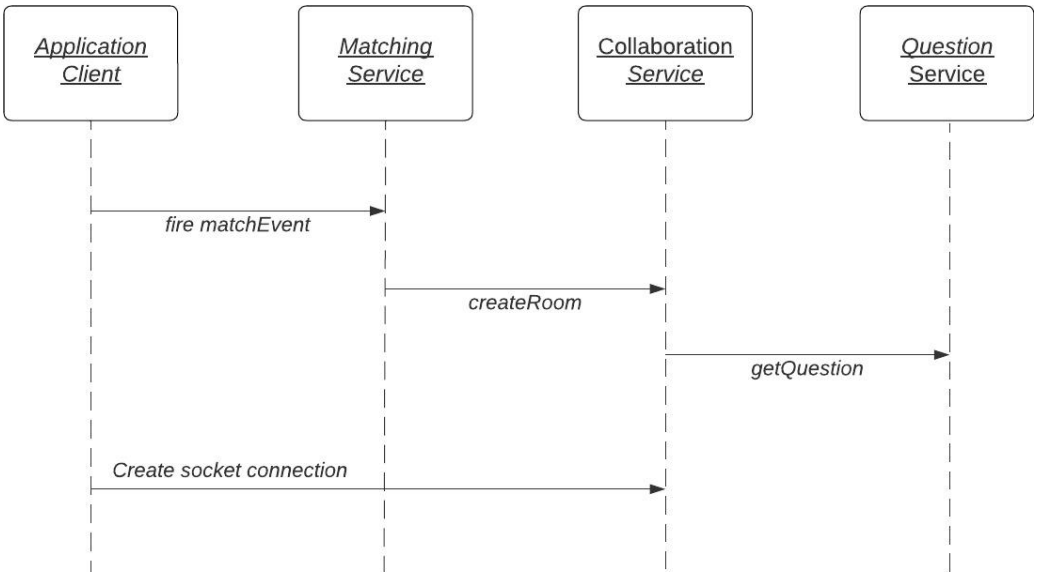
Request JSON Body: { “roomId”: “string”, “users”: [“string”], “difficulty”: “string” }

6.3.3 Exposed Socket Events

The frontend creates a web socket connection to the collaboration service using Socket.IO.

Events	Functionality
JoinRoomEvent	Frontend will fire a joinRoomEvent to attempt a socket connection to the given roomId. Collaboration service checks if the username belongs to the room and fires a **joinRoomSuccess** event if valid, otherwise it fires a **joinRoomFailure** event.
ExitRoomEvent	Frontend will fire an exitRoomEvent to disconnect a user from the room and the collaboration service will register the user as disconnected. In the case when both users are disconnected after a period of time, the room data will be deleted from the Redis database.
FetchRoomEvent	Frontend will fire a fetchRoomEvent to retrieve room data and the server will fire back the roomUsersChangeEvent, roomLanguageChangeEvent, roomQuestionEvent and codeInitEvent with the information of users, code language, question and room text from the Redis database.
CursorChangeEvent	Frontend fires this event when a user's cursor position changes and the server will fire a cursorChangeEvent to notify the other room user of the new cursor position of this user.
CodeInsertEvent, CodeReplaceEvent, CodeDeleteEvent	The frontend fires these events when a user makes a change to his editor. The server then forwards an event to update room users with information about the change, thus allowing code to be synced across room users.
CodeSyncEvent	Frontend fires this event in regular intervals for the collaboration service to write the new room text changes to the Redis database.
RoomLanguageChangeEvent	Frontend fires this event when the user changes the coding language and the server will fire an event to update the other room users on the change and also update the room language in the Redis database.

6.3.4 Implementation details



Sequence diagram of the room creation and socket connection to collaboration service

When the matching service successfully matches a pair of users, it invokes the collaboration service to create a room in the redis database. The collaboration service will retrieve a question from the question service, and store the question and user information in the room. Thereafter, the frontend will create a socket connection to the collaboration service.

The frontend will regularly update the Redis datastore with the text in the code editor via **CodeSyncEvent**. This is to allow

1. the saving of code data to history; and
2. to ensure that in the event that a user disconnects, he will be able to recover the code in the code editor when he reconnects.

The main difficulty with collaborative text editing is handling conflicting edits as it is possible for users to make changes at the same time, which have to be reconciled in some way when synchronizing everybody up again. To handle this concurrency issue, the backend acts as a central authority and provides a singular source of truth for the room users. There are multiple other techniques such as Operational transformation and CRDT that can solve this issue, however, in order to simplify the implementation and increase real-time performance, we simply send snapshots in backend.

To implement real-time cursor tracking (FR-COL-6), we use a CodeMirror extension provided by convergencelabs and synchronise cursor positions through the collaboration service backend.

6.3.5 Redis vs MongoDB

Consideration of database to use		
Database	**Redis (chosen)**	**MongoDB**
Pros	- In memory - Fast read and write speeds. - Easy to set up.	- Document database with robust mapping of data types. - Supports more complex query structures. - Very easily scalable as MongoDB does this for you (at a cost).
Cons	- Transient, thus data stored inside might not persist after a restart. - Much more difficult to scale.	- Much slower than Redis.
<u>Key considerations/ deciding factor:</u> Performance		
<u>Final Decision:</u> The collaboration service requires frequent writing of room data to the database and reading from it to enable fast synchronisation when users are collaborating on the code editor. In addition, the room data only needs to be cached for the duration of the collaboration session. Redis provides better performance than MongoDB and lack of data persistence is not an issue, hence Redis was chosen.		

6.4 Communication Service

6.4.1 Overview

This service provides a signalling service and peer discovery mechanism in order to establish peer-to-peer communication over WebRTC using PeerJS for users in the same collaboration room.

Technology Stack	- Typescript - Express - Socket.io - PeerJS

6.4.2 Exposed Socket Events

The frontend creates a web socket connection to the communication service using Socket.IO.

Events	**Functionality**
joinRoomEvent	Joins a signalling room and broadcasts a “peerConnected” event to others in the room. This event is fired when a user enters an interview page and it is only fired when the peer connection has been set up for communication for video and chat. (FR-COM-3)
joinCallRoomEvent	Broadcasts a “peerCallConnected” event to others in the room. Requires the user to already be in a room. This event is fired when a user clicks the “Join Call” button in the interview page. (FR-COM-1)
leaveCallRoomEvent	Broadcasts a “peerCallDisconnected” event to others in the room. Requires the user to already be in a room. This event is fired when a user clicks the “Leave Call” button in the interview page. (FR-COM-2)
disconnect	Broadcasts a “peerDisconnected” event to others in the room and removes the disconnected user from the room.

6.4.3 Implementation Details

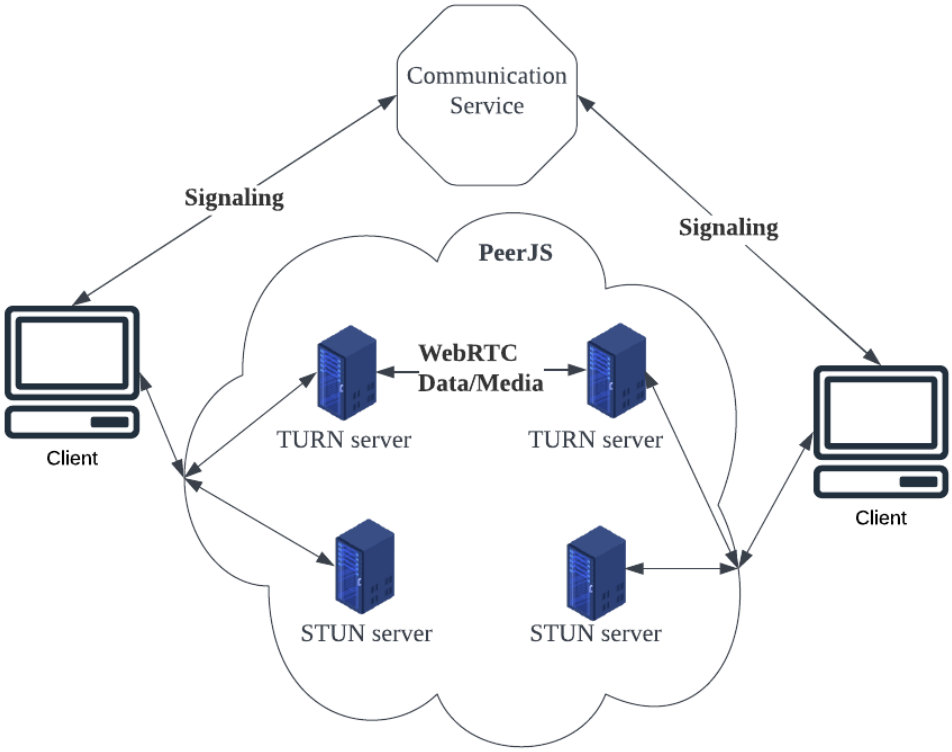
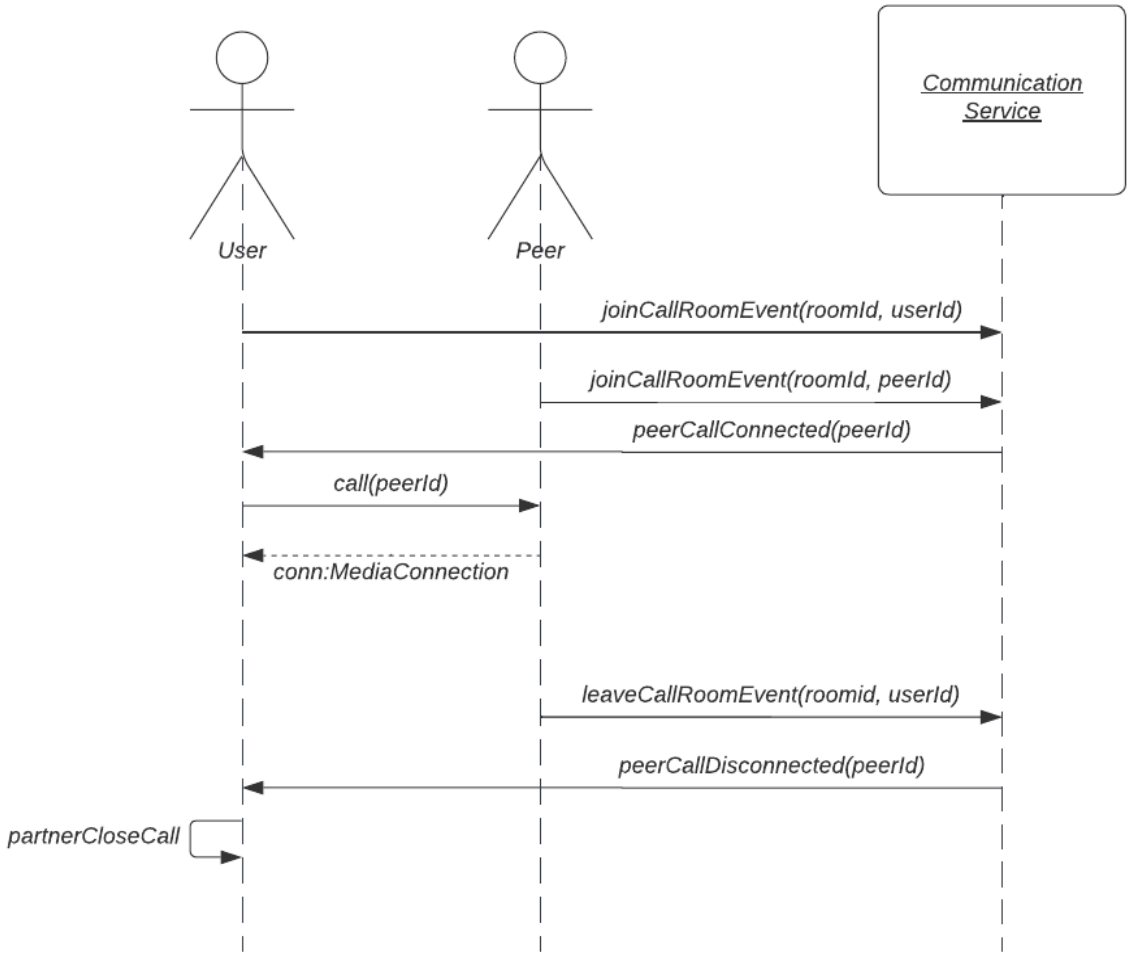


Illustration of how the app achieves communication over WebRTC

Upon peer discovery via the communication service, the clients' browsers establish a peer-to-peer connection with each other with the help of the PeerJS library.

PeerJS wraps the browser's WebRTC implementation to provide a complete, configurable, and easy-to-use peer-to-peer connection API. Equipped with nothing but an ID, a peer can create a peer-to-peer data or media stream connection to a remote peer. It abstracts away Interactive Connectivity Establishment (ICE) logic and helps us connect to a peer without having to worry about NAT, TURN, and STUN servers.

The signalling process is as such: when a user joins an interview room, he connects to the Communication Service and joins a signalling room. Other users already in the room are notified and they then initiate a connection with the newly joined user. This means that whichever user connects to the Communication Service first will initiate the connection. Once the users are connected, they can send real-time chat messages to each other. When a user chooses to join a video call by clicking the "Join Call" button, he is connected to his peer in a way that is synonymous with the above. This peer discovery process is illustrated in the diagram below.



Sequence diagram of the signalling process when a user and his peer join a call and the peer leaves.

Since our application is intended for users to practice their interview soft skills (i.e., code presentation skills), the chat function could be argued to be out-of-scope. However, we do recognise that it is a nice quality-of-life feature to add, despite the fact that the application would be completely usable without it. Since it is fairly easy to implement together with video calls using PeerJS, we include a bare-bones real-time chat as a low-priority functionality.

Note to testers:
Technical limitation when testing using a single computer - Video call when testing using two different browsers on the same computer will not display video due to how video streams work. Make sure that your camera is not already claimed/used by a different software/browser (i.e., test using the same browser, with one account on an incognito window and another on a normal window)

6.5 History Service

6.5.1 Overview

This service manages the history of user activities. It allows querying of a user's history as well as providing APIs for statistical data

Technology Stack	- Mongoose - MongoDB - TypeScript - Express

Backend Schema	`export type HistoryData = {`
	`  partner: string;`
	`  startTime: Number;`
	`  date: Date;`
	`  duration: string;`
	`  questionDifficulty: QuestionDifficulty;`
	`  questionID: string;`
	`  questionURL: string;`
	`  code: string;`
	`  questionName: string;`
	`  questionContent: string;`
	`  language: string;`
	`};`
	
	`export type IHistoryModel = {`
	`  me: string;`
	`  historyInfo: HistoryData[];`
	`};`

6.5.2 API Endpoints

6.5.2.1 Save to History

PUT /api/history/{username}

- Stores the user result (FR-HS-1)

Request JSON body: { "partner": "string", "duration": "string", "startTime": number, "date": Date, "questionDifficulty": "string", "questionID": "string", "questionURL": "string", "code": "string", "questionName": "string", "questionContent": "string", "language": "string"}

6.5.2.2 Get History

GET /api/history/{username}

- Get all past history information of the past attempts of a user such as partner, question difficulty,

code snippets etc (FR-HS-2)

6.5.2.3 Get Statistical History

GET /api/history/statistic/{username}

- Gets formatted result of the user's statistics such as the number of each question type attempted, main programming languages used, and information of past attempts (FR-HS-3)

6.5.2.4 Delete History

GET /api/history/statistic/{username}

- Deletes all information regarding a particular user (FR-HS-4)

6.6 Question Service

6.6.1 Overview

This service facilitates fetching questions from Leetcode. In the event that the API from Leetcode fails, it will serve static questions randomly based on difficulty to users.

Technology Stack	- TypeScript - Express - GraphQL (Leetcode Query)

6.6.2 API Endpoints

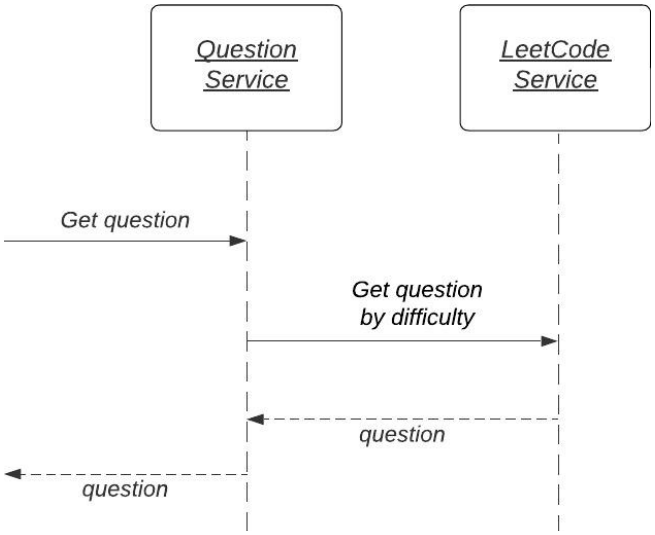
6.6.2.1 Get Leetcode Question

GET /api/question

- Gets a random question of the specified difficulty from Leetcode Graphql API (FR-QS-1). In the event that the Leetcode APIs fail, this API will default to a list of fallback questions (FR-QS-2)

Request Query Params: (i.e /api/question/?difficulty=HARD)

1. difficulty: string



Sequence diagram of retrieving a question from the question service.

7 Frontend

7.1 Frontend framework

Our team chose React in the frontend framework as it best suits our use case of development. It has lesser boilerplate code as well as ease of starting up a project.

Consideration of frontend framework to use			
Frontend technology	**React (chosen)**	**Vue**	**Angular**
Pros	- Easy to learn. - Easy to set up. - New hooks syntax which allows cleaner code reuse.	- Easiest to learn. - Easy to set up. - New composition API which allows for better reactive.	- Lots of boilerplate is provided for routing, state management, etc.
Cons	- Less boilerplate, key functionality is provided by libraries (e.g., routing).	- Smaller community compared to React and Angular. - Less available plugins compared to React and Angular.	- Difficult to learn due to verbosity. - Tougher to set up.
Key considerations/ deciding factor: Ease of setup and team's familiarity and adaptation			
Final Decision: We chose React in the end as most of the team is familiar with it and we believe that it is relatively easier to set up. Also, there are many plugins available in React that help in the development of this project.			

7.2 Frontend Libraries

Library	**Reason**
MaterialUI	It is an enterprise-grade library that provides a large number of components that are used in our project and it is compatible with React as well
SocketIO	To provide a pub-sub messaging system with the backend for Matching, Collaboration, and Communication services
React Context	As we only need to store user information in the project itself, we chose to use React Context instead of other popular state management tools like Redux, Jotai, and Zustand, as it is more lightweight overall.

CodeMirror	A code editor component for the web. Used to implement a text/code input field with support for many editing features.

Consideration of coding platform to use			
Platform	**Codemirror (chosen)**	**Quill**	**AceEditor**
Pros	- Provides almost everything needed for a coding platform - Provides out-of-the-box collaborative editing functionality. - Cursor syncing plugin.	- Simple to work with - Multiple libraries available to work with Quill to become a coding platform, but additional work is required.	- Provides almost everything needed for a coding platform.
Cons	- Complicated to develop with.	- Very little functionality.	- Complicated to develop with. - Fewer plugins provided as compared to CodeMirror.
<u>Key considerations/ deciding factor:</u> Functionality			
<u>Final Decision:</u> Despite CodeMirror being difficult to implement, it provides almost everything needed for our coding platform. What sets it apart from AceEditor is its out-of-the-box collaborative editing, as well as the availability of a cursor-syncing plugin provided by the community.			

8 DevOps

8.1 Deployment

Our application can be deployed locally for testing and is deployed to the AWS Cloud in production.

Technology Stack	- Docker - Kubernetes - kind - Amazon Elastic Kubernetes Service - Amazon Elastic Container Registry - Amazon ElastiCache for Redis - Github Actions

8.1.1 Local Deployment

Before deploying our application to the production environment, we test our changes by running the application locally in a containerised environment using Docker, Kubernetes and kind (FR-D-2).

Technology used:

- Docker is an open source platform for building, running, and managing containers on servers and the cloud.
- Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.
- kind is a tool for running local Kubernetes clusters using Docker container

For local deployment, we use Docker Compose to build the multi-container application and run it in a local Kubernetes cluster created with the kind tool.

Steps to deploy the application to a local stagin environment:

1. In the root of the repository, run “docker compose build” to build the new docker images for each service.
2. Run the “start.sh” script which does the following:
   2.1. Create a local Kubernetes cluster using Kind.
   2.2. Load the docker images into the kind cluster.
   2.3. Create Kubernetes resources from the configuration files in k8s/manifests folder.
3. The application can then be accessed at http://localhost.

Alternatively, run the application using Docker without a local Kubernetes cluster:

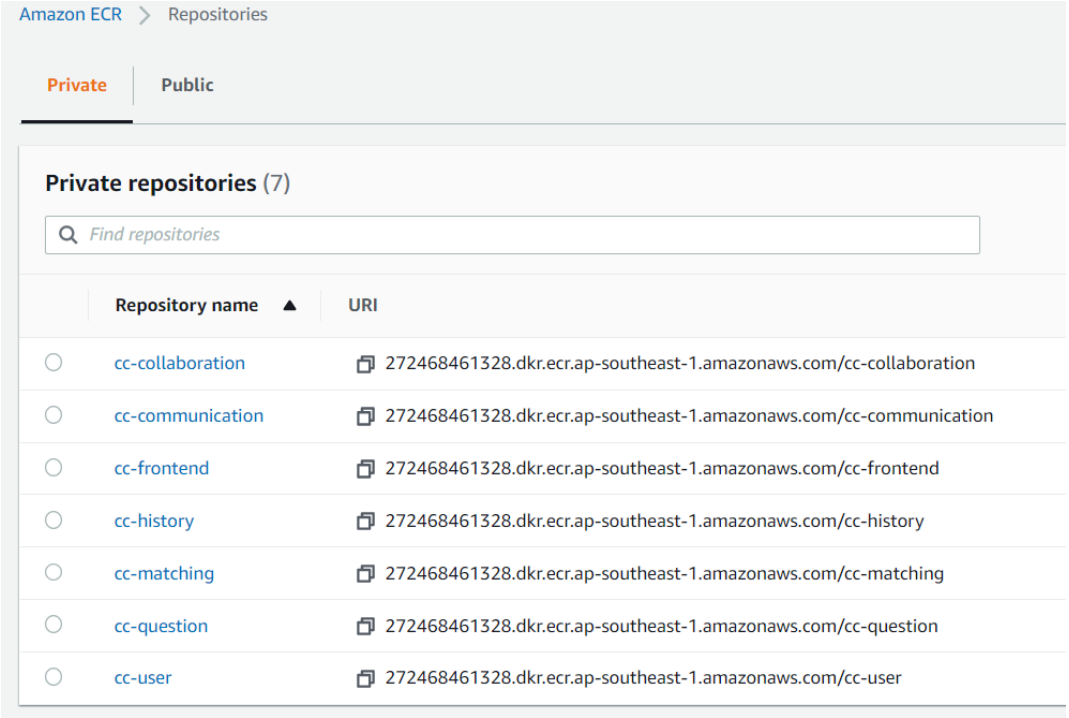
1. In the root of the repository, run “docker compose up --build”.
2. The application can then be accessed at http://localhost.

8.1.2 AWS Deployment

Our application is deployed on the Amazon Elastic Kubernetes Service (EKS) with the container images of our services stored on the Amazon Elastic Container Registry (ECR) (FR-D-3).

With Amazon EKS, we can easily create a Kubernetes cluster that is provisioned with a scalable and highly-available Kubernetes control plane responsible for scheduling containers, managing application availability and storing cluster data. The cluster nodes leverage on Amazon Elastic Compute Cloud (EC2) and take advantage of all the performance, scale, reliability, and availability of AWS infrastructure, as well as integrations with AWS networking and security services such as the AWS Virtual Private Cloud (VPC) support for pod networking. The cluster nodes are managed to allow updates and graceful terminations which ensure that our application remains available. Amazon EKS runs the Kubernetes control plane across multiple AWS availability zones while the worker nodes have also been configured to reside across multiple subnets in separate availability zones. This eliminates a single point of failure while and ensures high availability (NFR-AVA-1).

Alongside EKS, we use Amazon ECR which is a secure and scalable container image registry service. For each service of our application, we used a private repository to store and manage its container images.



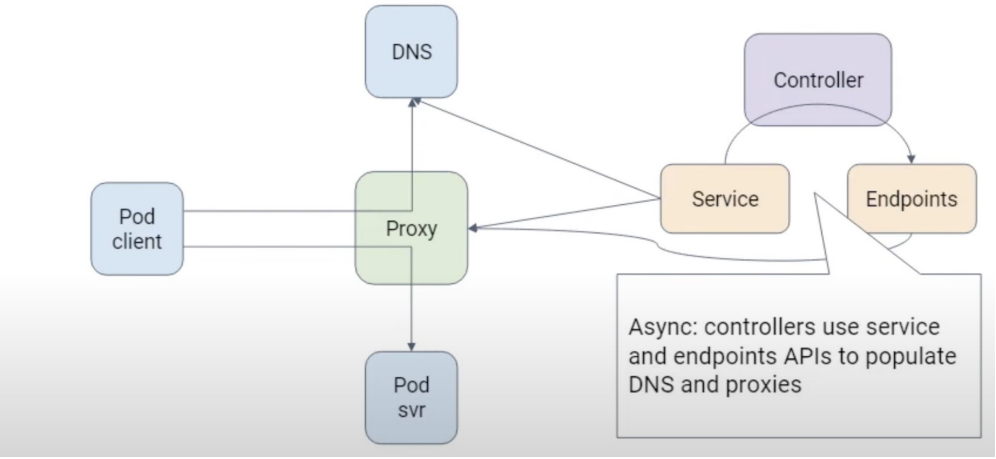
ECR repositories.

The Redis caches used by the Collaboration service and Matching service are hosted on Amazon ElastiCache. ElastiCache provides a high-performance and scalable caching solution. Once a cluster is provisioned, Amazon ElastiCache automatically detects and replaces failed nodes, providing a resilient system that mitigates the risk of overloaded databases. Automated backups can also be performed to ensure high reliability of the system.

The Amazon Route 53 DNS service is used to handle routing requests from the http://cocoder.link domain to our application.

8.1.3 Service Discovery

In order for services to communicate within our microservice application, we used the built in Kubernetes service discovery which uses kube-proxy and a cluster-aware DNS server, CoreDNS (FR-D-6).



Service Discovery in Kubernetes using kube-proxy and DNS

A Kubernetes service object is a stable endpoint that points to a group of pods based on label selectors. It proxies requests to the backend pods using labels and selectors. Since the pods can come and go dynamically in Kubernetes, a service object serves the purpose of never changing the endpoint or IP address that will point to the list of running pods. The requests are also load-balanced over a set of pods if multiple pods are running in the same application.

The clients can use the DNS name of the Kubernetes service and CoreDNS in Kubernetes handles the mapping of service names to service IP addresses. A kube-proxy instance running on each worker node handles the underlying implementation of Kubernetes Service.

An example of using Kubernetes service discovery in our application is in our matching service. It makes a call to the collaboration service by using its service name “http://collaborations-svc:8002”. This will be resolved to the IP address of the collaboration service and allow requests to be forwarded from the matching service to the collaboration service.

```
containers:
  - name: matching-svc
    image: 272468461328.dkr.ecr.ap-southeast-1.amazonaws.com/cc-matching:latest
    imagePullPolicy: Always
    ports:
      - containerPort: 8001
    env:
      - name: REDIS_URL
        value: 'redis://redis-matching.opyeey.ng.0001.apse1.cache.amazonaws.com:6379'
      - name: URI_COLLAB_SVC
        value: 'http://collaboration-svc:8002'
```

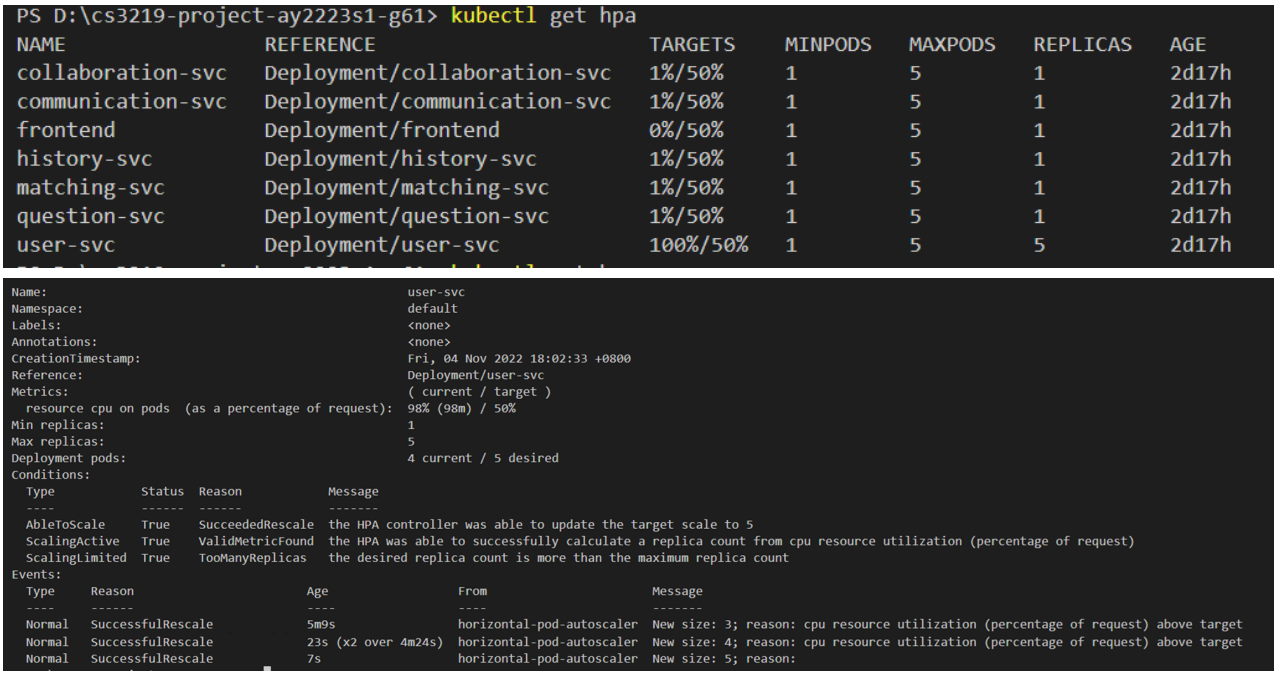
Snippet of matching-deployment.yaml

8.1.4 Scalability of Deployed Application

Our application can be easily scaled to meet growth in demand (FR-D-4).

8.1.4.1 Horizontal Pod Autoscaler

To automate scaling of our application, we used Horizontal Pod Auto-scalers (HPA) in Kubernetes which automatically scale the number of pods for each service to match demand. A Metrics Server is used to provide resource usage metrics to the HPA controller, such as CPU and memory usage for nodes and pods. When the utilisation of a resource exceeds the limits set, more Pods are automatically deployed to scale it up horizontally. As seen in the screenshots below, there is a HPA for each service which automatically adjusts the number pods for that service. The user service is experiencing an increased load so the HPA scales up the number of pods.



```
PS D:\cs3219-project-ay2223s1-g61> kubectl get hpa
NAME                REFERENCE                          TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
collaboration-svc   Deployment/collaboration-svc       1%/50%      1         5         1          2d17h
communication-svc   Deployment/communication-svc       1%/50%      1         5         1          2d17h
frontend            Deployment/frontend                0%/50%      1         5         1          2d17h
history-svc         Deployment/history-svc             1%/50%      1         5         1          2d17h
matching-svc        Deployment/matching-svc            1%/50%      1         5         1          2d17h
question-svc        Deployment/question-svc            1%/50%      1         5         1          2d17h
user-svc            Deployment/user-svc                100%/50%    1         5         5          2d17h
```

```
Name:                                                  user-svc
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           <none>
CreationTimestamp:                                     Fri, 04 Nov 2022 18:02:33 +0800
Reference:                                             Deployment/user-svc
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  98% (98m) / 50%
Min replicas:                                          1
Max replicas:                                          5
Deployment pods:                                       4 current / 5 desired
Conditions:
  Type            Status  Reason            Message
  ----            ------  ------            -------
  AbleToScale     True    SucceededRescale  the HPA controller was able to update the target scale to 5
  ScalingActive   True    ValidMetricFound  the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
  ScalingLimited  True    TooManyReplicas   the desired replica count is more than the maximum replica count
Events:
  Type     Reason                        Age                 From                       Message
  ----     ------                        ----                ----                       -------
  Normal   SuccessfulRescale             5m9s                horizontal-pod-autoscaler  New size: 3; reason: cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale             23s (x2 over 4m24s) horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal   SuccessfulRescale             7s                  horizontal-pod-autoscaler  New size: 5; reason:
```

Screenshots from load tests.

8.1.4.2 Load Testing

To ensure that our application is able to accommodate increasing number of users, data and network traffic without compromising performance, we performed a few load tests.

NFR-SCA-1

We load tested with a total of 1000 request to our website https://cocoder.link with 100 requests per second for 10 seconds. The return response for all 1000 requests are 200 OK which means that our system is able to support at least 500 users accessing the website simultaneously.

```
cs3219-project-ay2223s1-g61$ hey -z 10s -c 50 -q 2 -m GET https://cocoder.link/

Summary:
  Total:        10.1068 secs
  Slowest:      0.8765 secs
  Fastest:      0.0063 secs
  Average:      0.1863 secs
  Requests/sec: 98.9437

  Total data:   862000 bytes
  Size/request: 862 bytes

Response time histogram:
  0.006 [1]     |
  0.093 [518]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.180 [97]    |■■■■■■■
  0.267 [89]    |■■■■■■■
  0.354 [126]   |■■■■■■■■■■
  0.441 [98]    |■■■■■■■■
  0.528 [29]    |■■
  0.615 [18]    |■
  0.702 [0]     |
  0.790 [18]    |■
  0.877 [6]     |


Latency distribution:
  10% in 0.0289 secs
  25% in 0.0808 secs
  50% in 0.0916 secs
  75% in 0.2792 secs
  90% in 0.4026 secs
  95% in 0.5002 secs
  99% in 0.7788 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0061 secs, 0.0000 secs, 0.2197 secs
  DNS-lookup:   0.0017 secs, 0.0000 secs, 0.0379 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0032 secs
  resp wait:    0.1761 secs, 0.0061 secs, 0.8764 secs
  resp read:    0.0001 secs, 0.0000 secs, 0.0008 secs

Status code distribution:
  [200] 1000 responses
```

NFR-SCA-1 Load test result

NFR-SCA-2

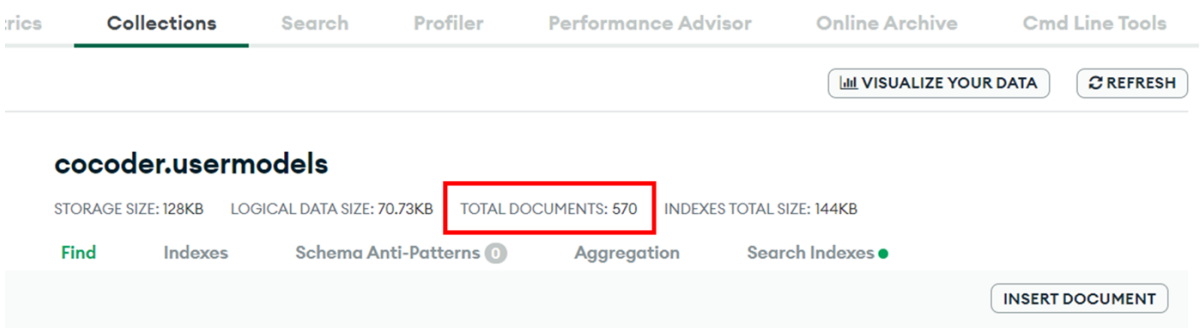
We sent 50 request second for 10 seconds to our collaboration service API endpoint. All 500 packets responsed with 200 status code meaning that they have successfully ping the endpoint. Hence, this shows that our system is able to support 500 users concurrently in the collaboration space.

```
cs3219-project-ay2223s1-g61$ hey -z 10s -c 50 -q 1 -m GET -H "Accept: application/json" -H "Cookie: jwt=eyJhbGciOiJIUzI1
NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjYzMTcwYWQxOWYzMDNmNWM2ZTE3NjIyYiIsImlhdCI6MTY2NzgxNjkzMSwiZXhwIjoxNjY3OTAzMzMxfQ.QzKOrGoT
5TdL9GAVNwdqh6NYYQ2WGxWmLDsPVWxE6eM; Path=/; Secure; HttpOnly;" https://cocoder.link/api/collaboration/

Summary:
  Total:        10.4599 secs
  Slowest:      0.8638 secs
  Fastest:      0.0269 secs
  Average:      0.3093 secs
  Requests/sec: 47.8018

  Total data:   19000 bytes
  0.111 [38]    |■■■■■■■■■■■■
  0.194 [97]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.278 [88]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.362 [85]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.445 [122]   |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.529 [36]    |■■■■■■■■■■■■
  0.613 [14]    |■■■■■
  0.696 [16]    |■■■■■
  0.780 [2]     |■
  0.864 [1]     |


Latency distribution:
  10% in 0.1344 secs
  25% in 0.1735 secs
  50% in 0.3226 secs
  75% in 0.4292 secs
  90% in 0.4553 secs
  95% in 0.5327 secs
  99% in 0.6365 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0203 secs, 0.0000 secs, 0.2313 secs
  DNS-lookup:   0.0010 secs, 0.0000 secs, 0.0108 secs
  req write:    0.0001 secs, 0.0000 secs, 0.0033 secs
  resp wait:    0.2886 secs, 0.0267 secs, 0.8634 secs
  resp read:    0.0001 secs, 0.0000 secs, 0.0005 secs

Status code distribution:
  [200] 500 responses
```

NFR-SCA-2 Load test result

NFR-SCA-3

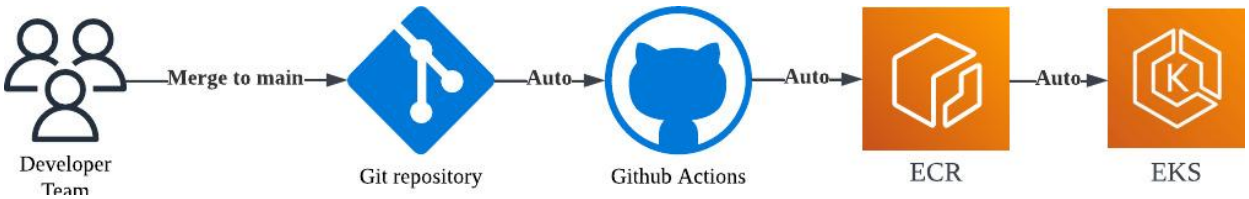
We load tested our mongoDB database with information of 570 users which shows that it is capable of storing supporting at least 500 users.



NFR-SCA-3 Load test result

8.2 CI/CD

Continuous deployment of the application is achieved via GitHub Actions. The Actions tests, builds and then pushes updated Docker images to AWS ECR. EKS has been configured to then deploy these images (FR-D-7).

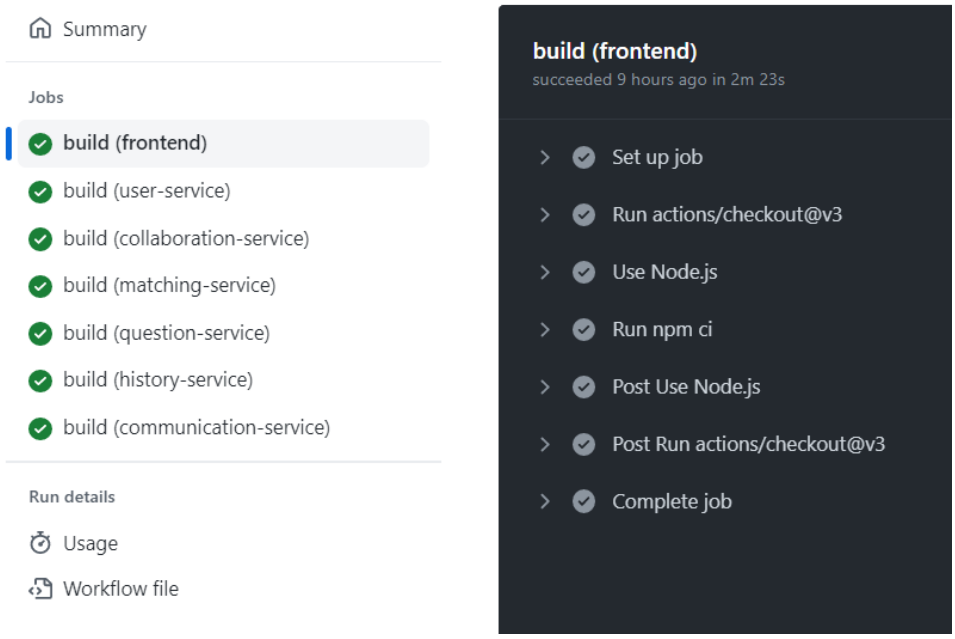


Continuous deployment architecture.

We have 2 Github Action workflows, one for testing and checking build success, and another for deployment.

Test and Build Success Check Workflow

On creation of a Pull Request and subsequent updates, GitHub Action will run tests and check for the build success of each service. This will ensure that any code that is committed is of good quality.

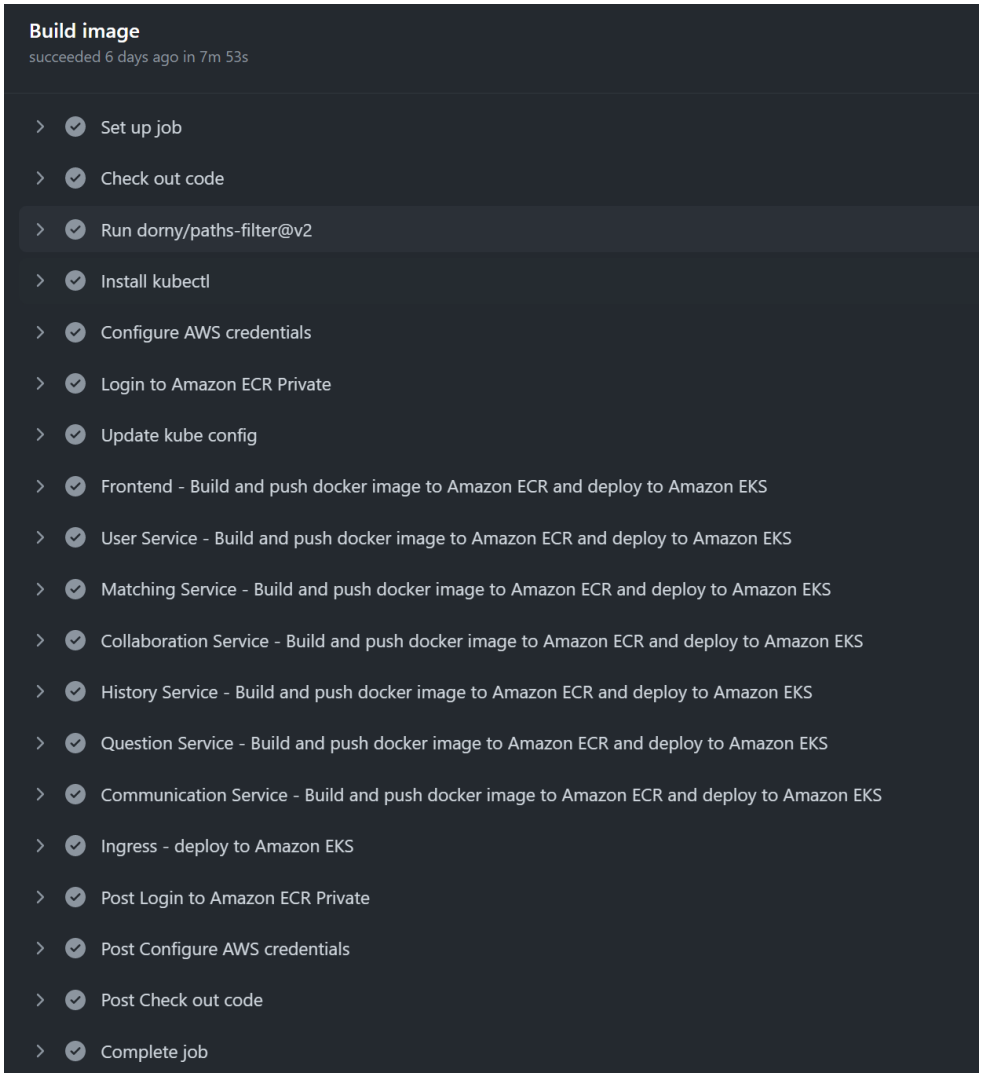


Automated Test and Build workflow

Deployment Workflow

When a code is merged into the main branch, the deployment workflow executes will the following steps:

1. Check for the microservices that have been changed.
2. Configure AWS credentials.
3. Logins to Amazon ECR.
4. Build, tag, and push a new docker image to Amazon ECR for each service changed.
5. Reapply Kubernetes configuration files.



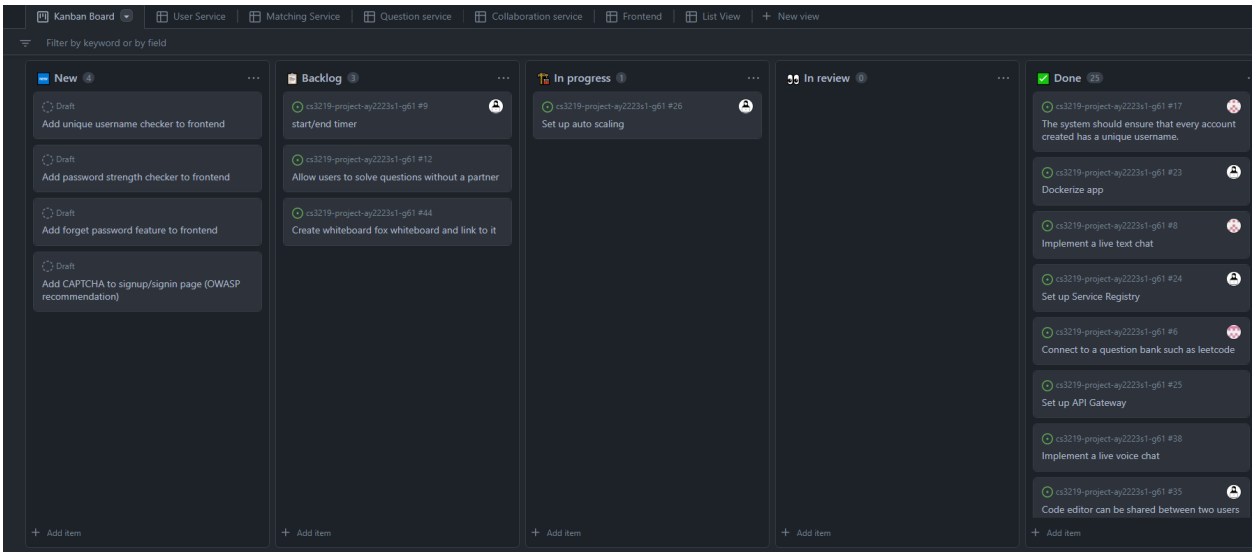
Automated deployment workflow

AWS EKS handles the rest of the deployment by pulling new images from the ECR repositories, then tearing down and setting up new pods while ensuring limited downtime of the services.

As such, we are able to automate the software release process from build to deploy. The repeatable and automated CI/CD process enables us to quickly and reliably release new changes and this consistency can help improve scalability.

8.3 Team workflow

Our team adopted an agile software development process for our project, specifically, Scrum. We had 3 sprints in total, 1 for each milestone. In the first week, we discussed the features that we aimed to develop according to the FRs of the application. Then, we added their respective priority levels based on the difficulty of the task and its value added to our product. Thereafter, we allocated them to their respective sprints and assigned a team member to each feature. Our team decided to allocate roles and tasks according to each member's strengths and familiarity with the related libraries. We used GitHub Projects for issue tracking and project management, as shown below, and updated it regularly so that everyone was updated on the overall progress of the project.



During the week, we would focus on designing, developing and self-testing our features. We would also review and test each other's PRs frequently, and merge them to master as soon as we ensure that it has fulfilled our FRs. Our team also had weekly meetings to update each other on our development process and progress. During these meetings, we brought up issues that we faced during the week, and potential issues that we foresaw might happen. We then added them to our backlog for the upcoming sprint. Additionally, for every sprint, we would also have a sprint planning meeting, where we reviewed our progress during the previous sprint, and discussed our plans for the upcoming sprint. All meeting minutes were stored on Microsoft Teams.

During each meeting, we will cover these 4 agendas:

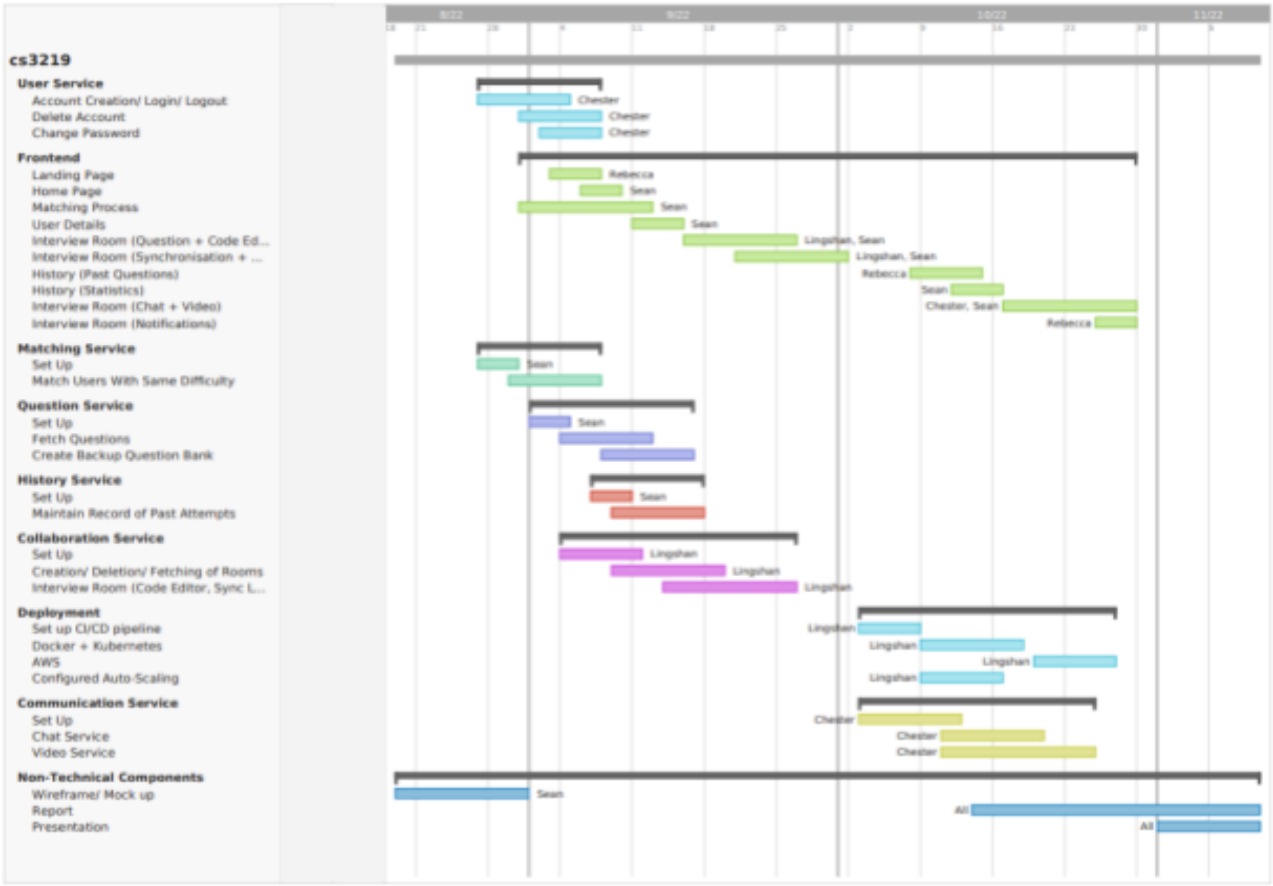
1. Progress sharing by each member
2. Key design decisions whenever it is made
3. Assignments of tasks and issues for the week/ sprint
4. Conduct integration of the new features from various members

Overall, our aim was to produce our MVP for Milestone 1, finish all the 'Must have' FRs by Milestone 2, and focus on deployment and the "Nice to have" tasks for Milestone 3. We were able to achieve this by using the Scrum model, which allowed us to collaborate more efficiently and effectively, and better structure our workload.

9 Project Plan

Milestone Number	Deliverables
Milestone 1	1) User service 2) Matching Service
Milestone 2	1) Collaboration service 2) History service 3) Question service 4) Effective usage of CI/CD 5) Documentation and report
Milestone 3	1) Communication service 2) Deployment and scaling 3) Testing of NFRs and FR 4) Documentation and report 5) Demo preparation

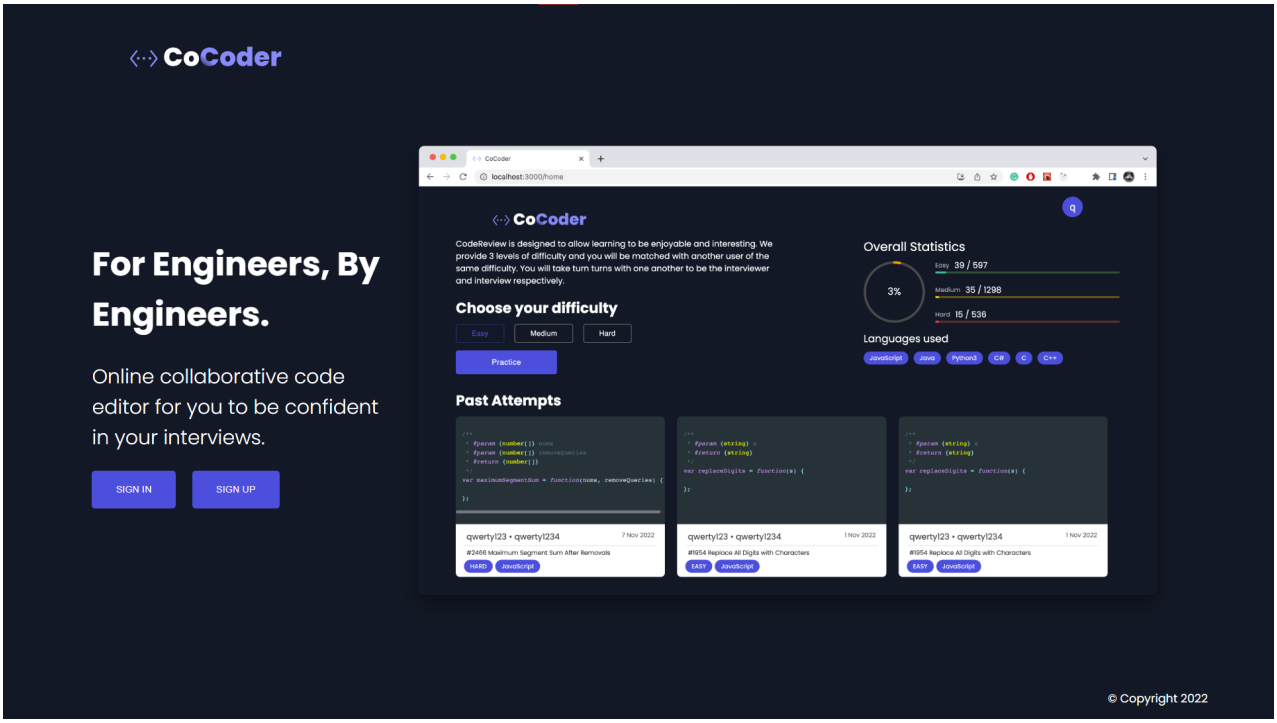
The Gantt chart below breaks down our team's development process over the 3-4 months.



Gantt Chart of Team's Development Timeline

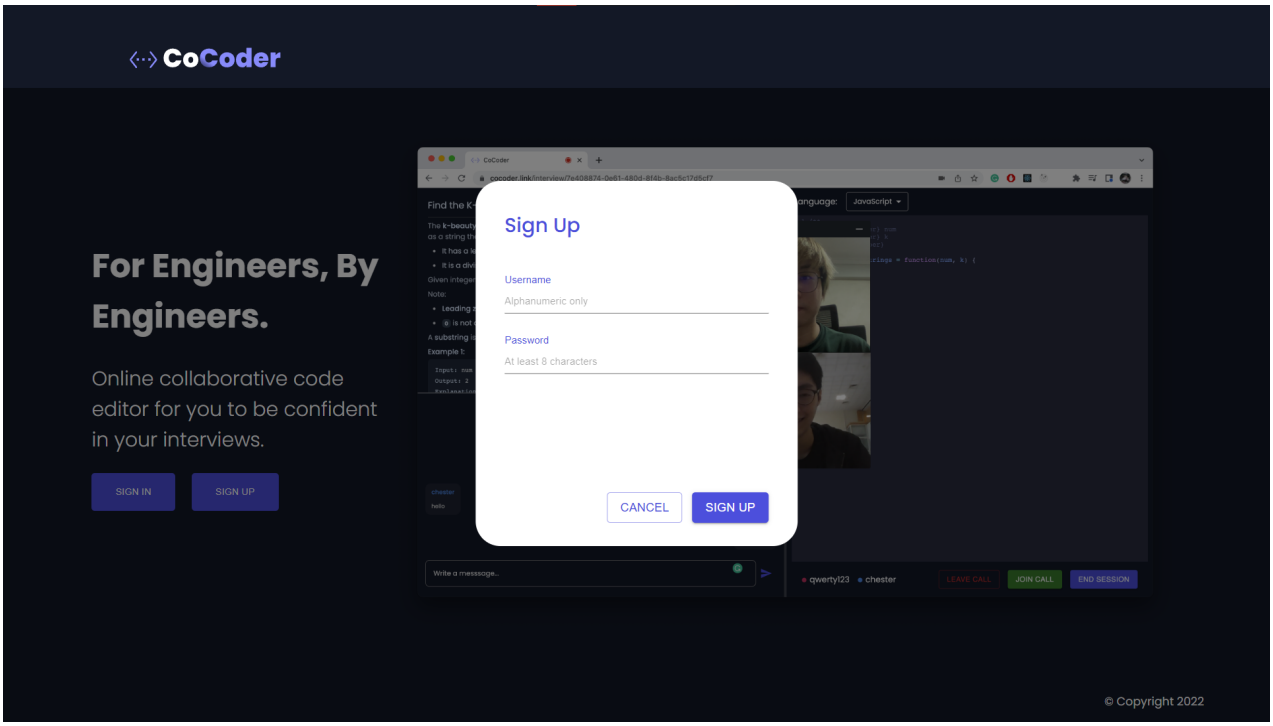
10 Application Screenshots

10.1 Landing Page

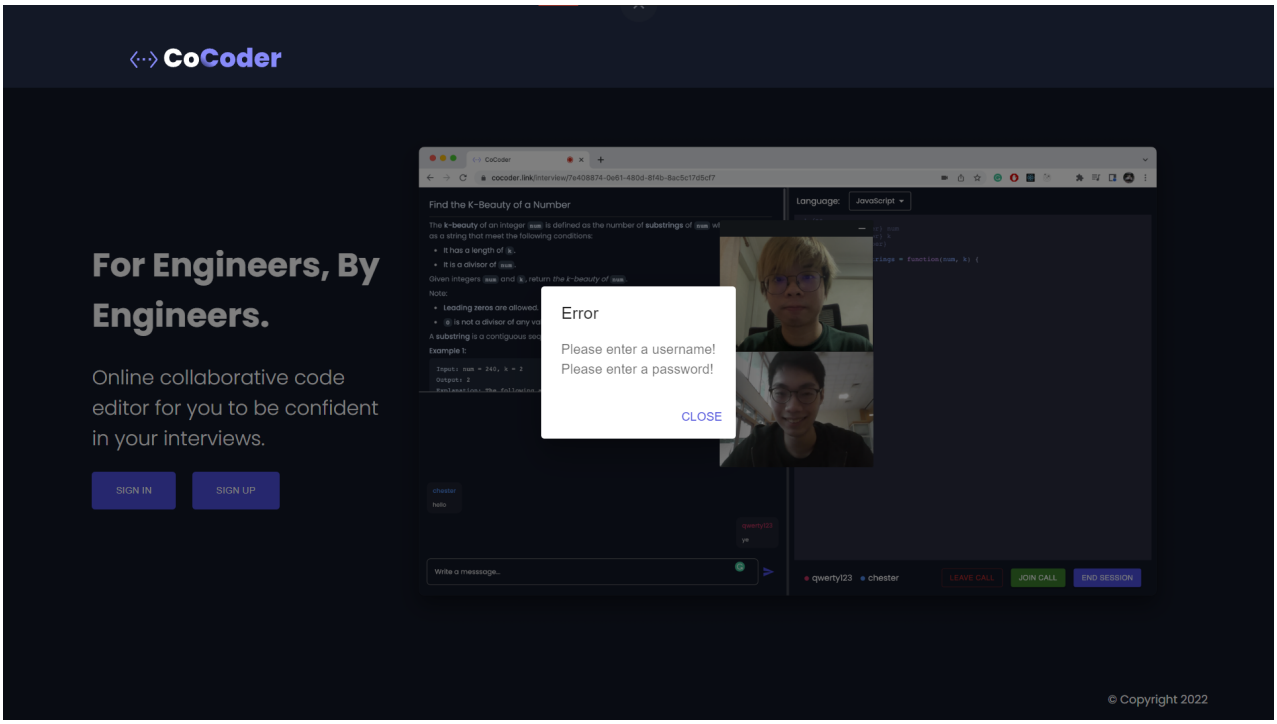


Landing page for users to sign in or sign up for an account (FR-AC-1)

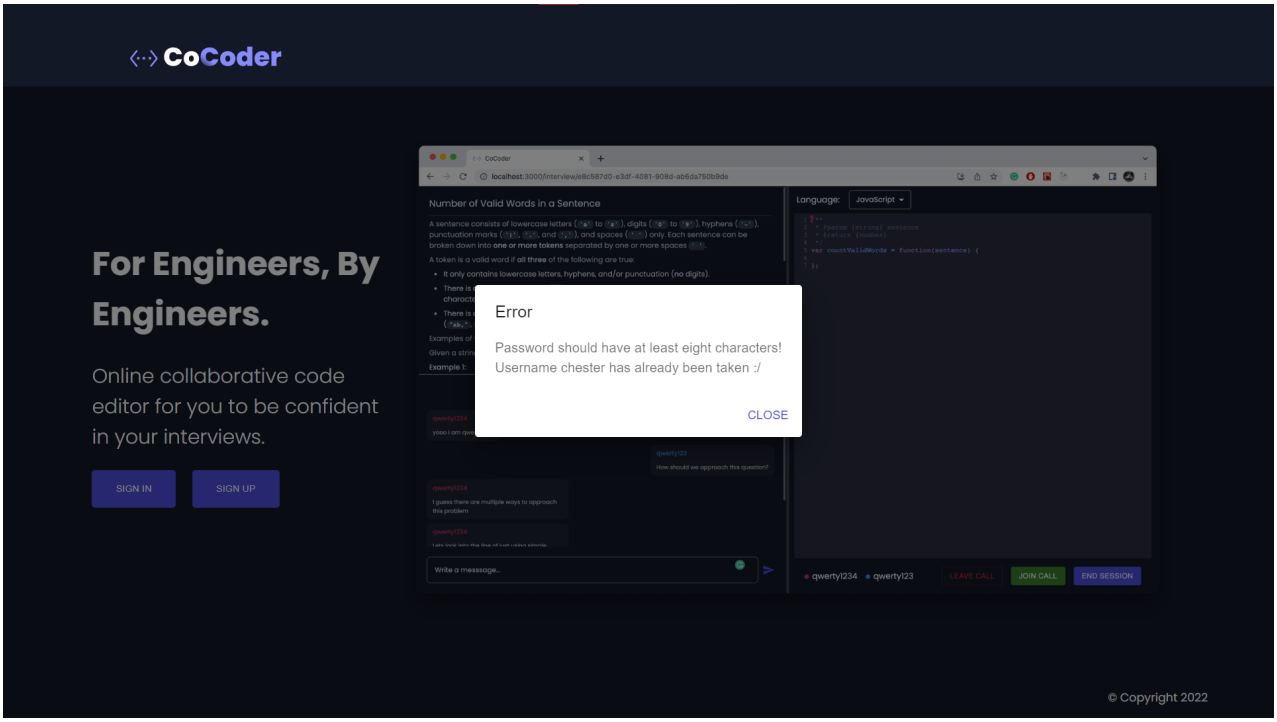
10.2 Sign Up



Sign up dialogue.

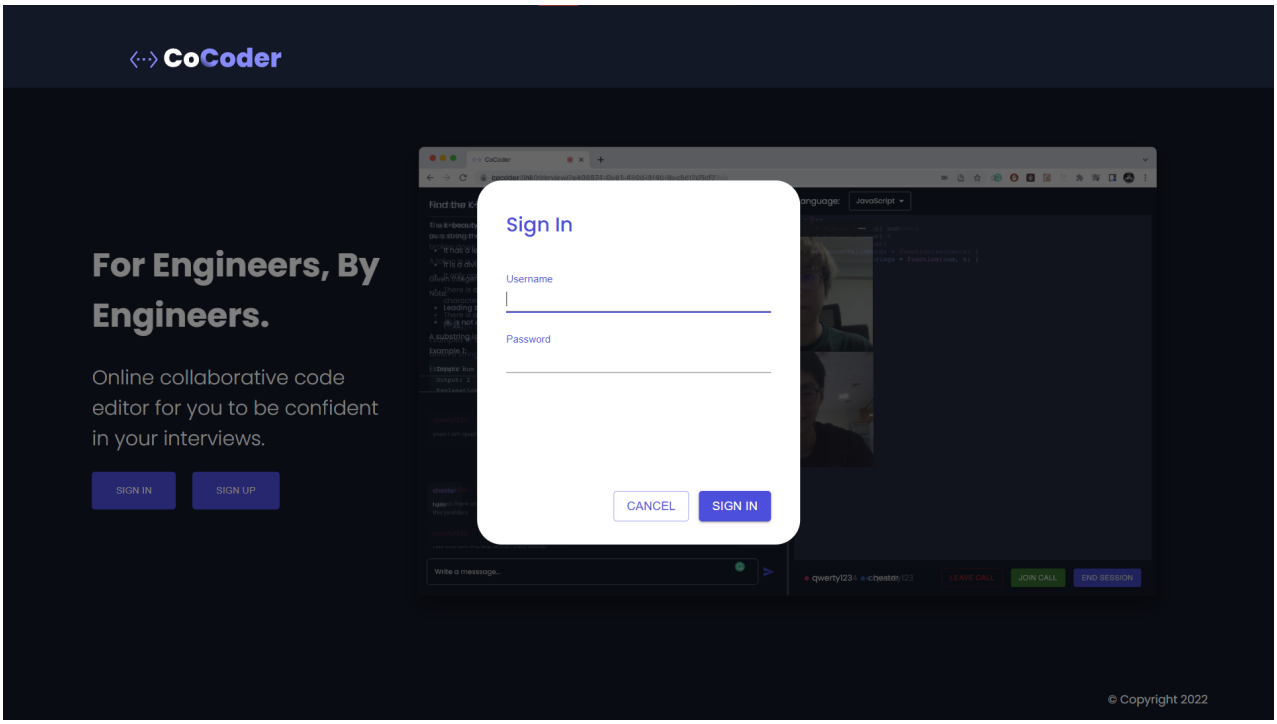


Sign up error dialogue 1.

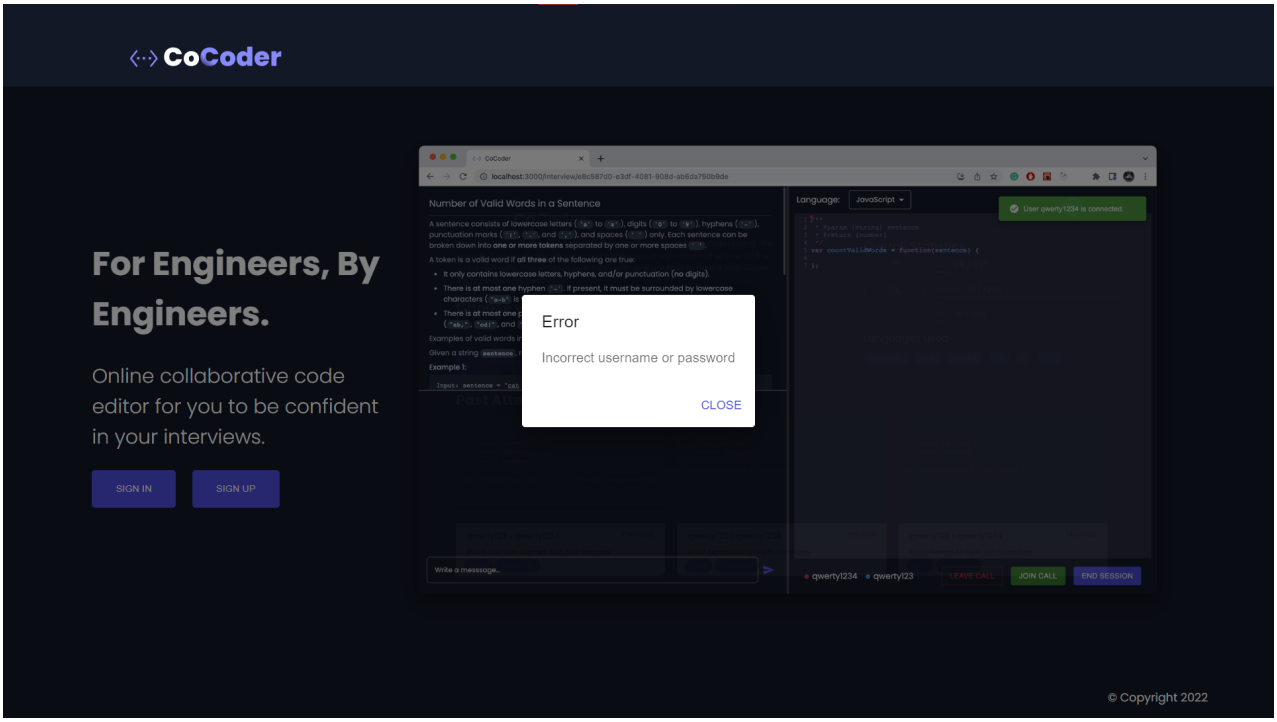


Sign up error dialogue 2.

10.3 Sign-In

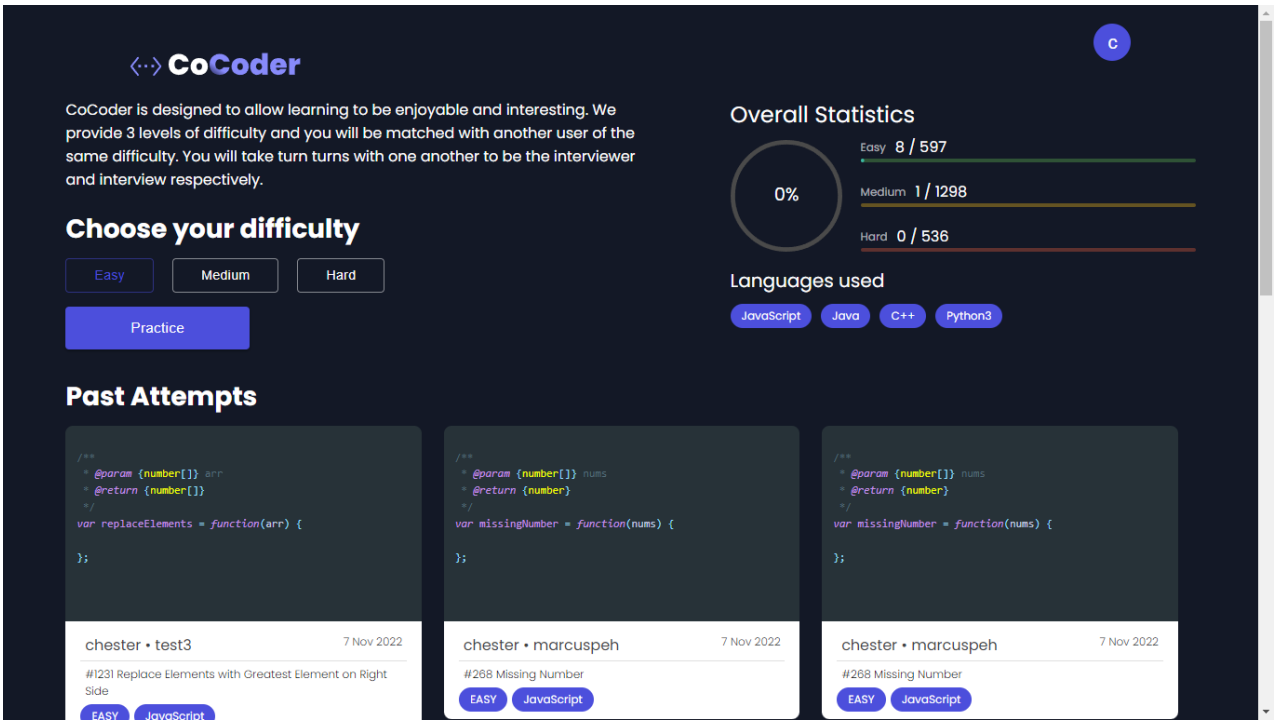


Sign-in dialogue.



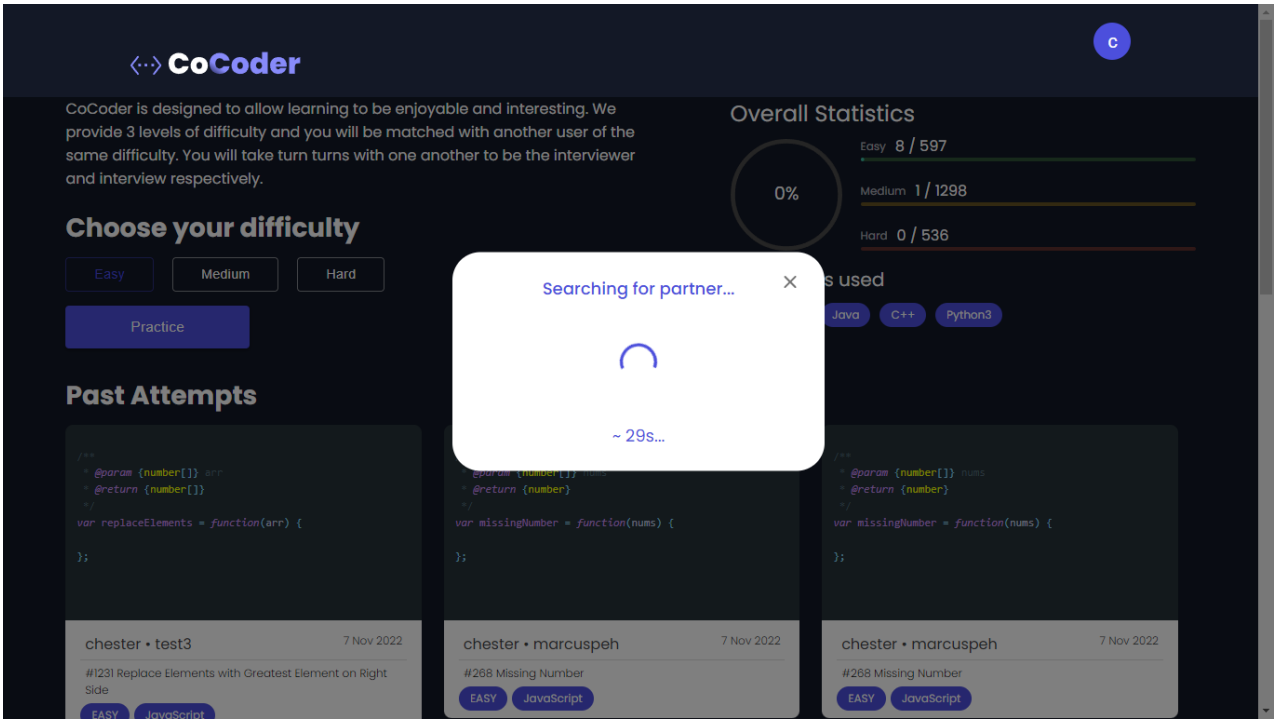
Sign-in error dialogue.

10.4 Home

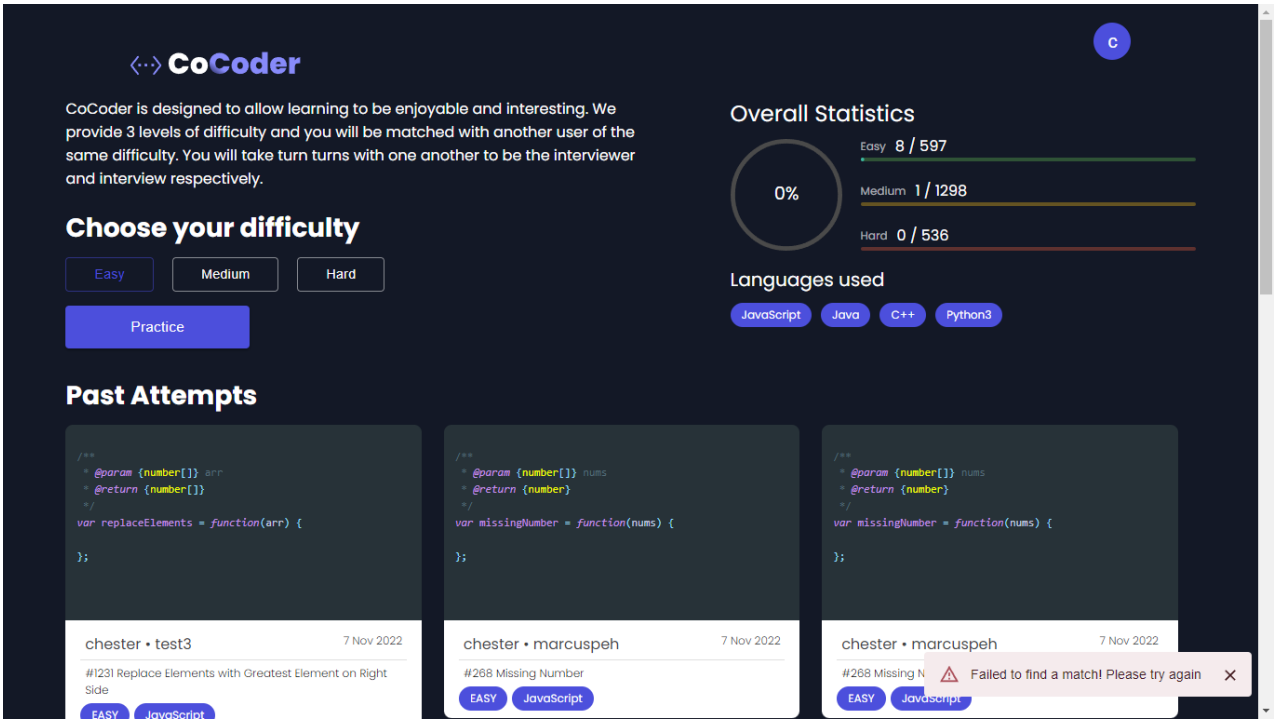


Home page.

This is where a user can search for a match based on difficulty (FR-AC-2 and FR-AC-3) and view his interview history (FR-AC-4) as well as statistical data (FR-AC-8). The user can navigate the application using the routes that popup under his avatar at the top right. (FR-AC-7)



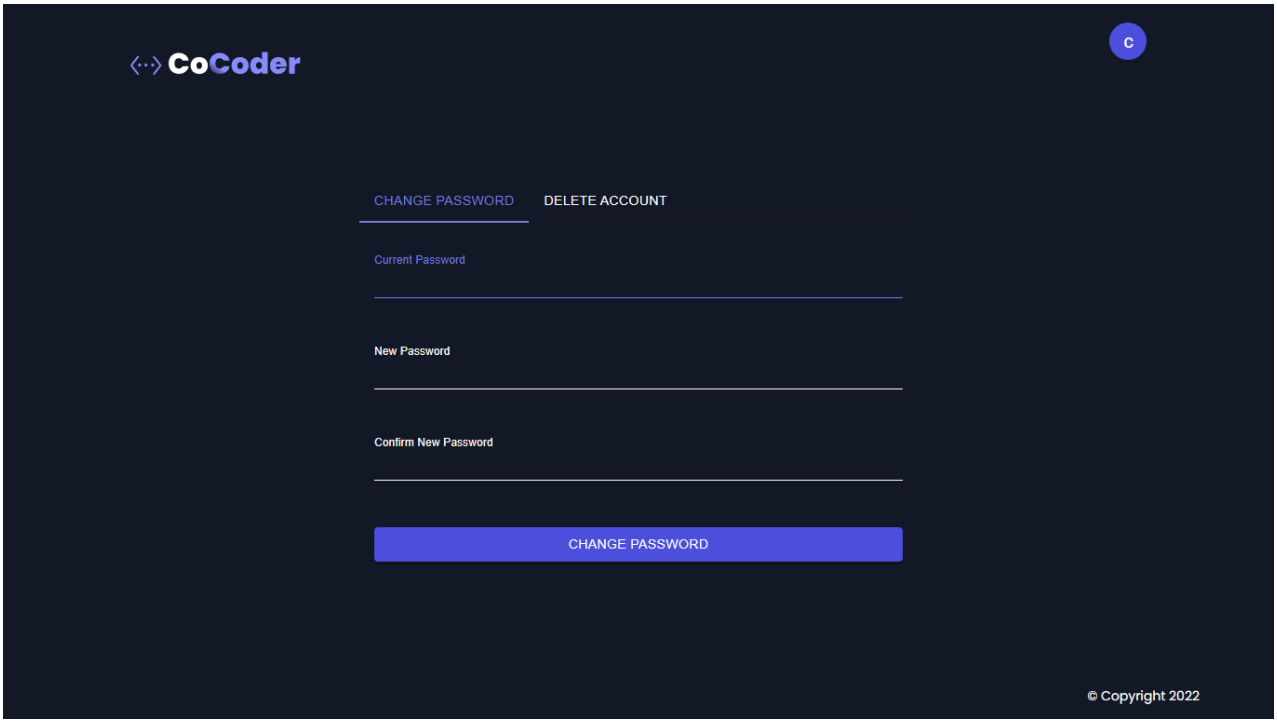
Finding match.



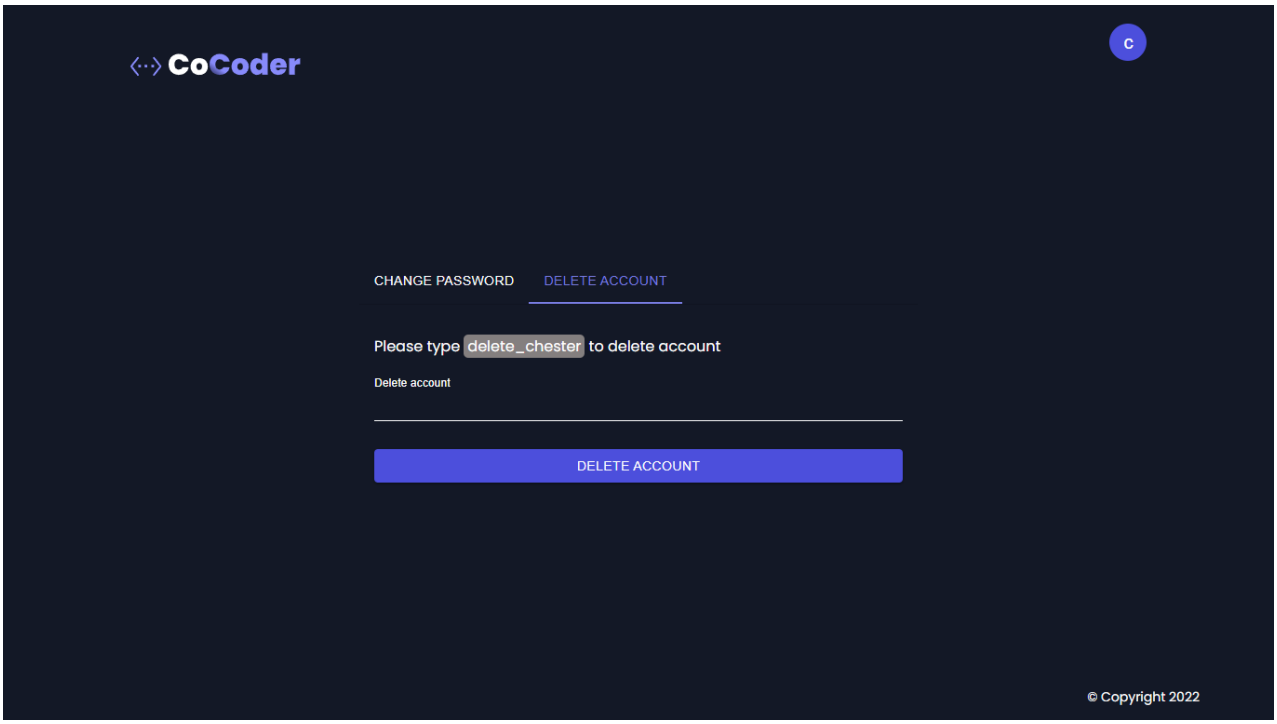
Notification when no match found after 30 seconds.

10.5 Account Page

Provides an interface for users to change password or delete an account (FR-AC-5)

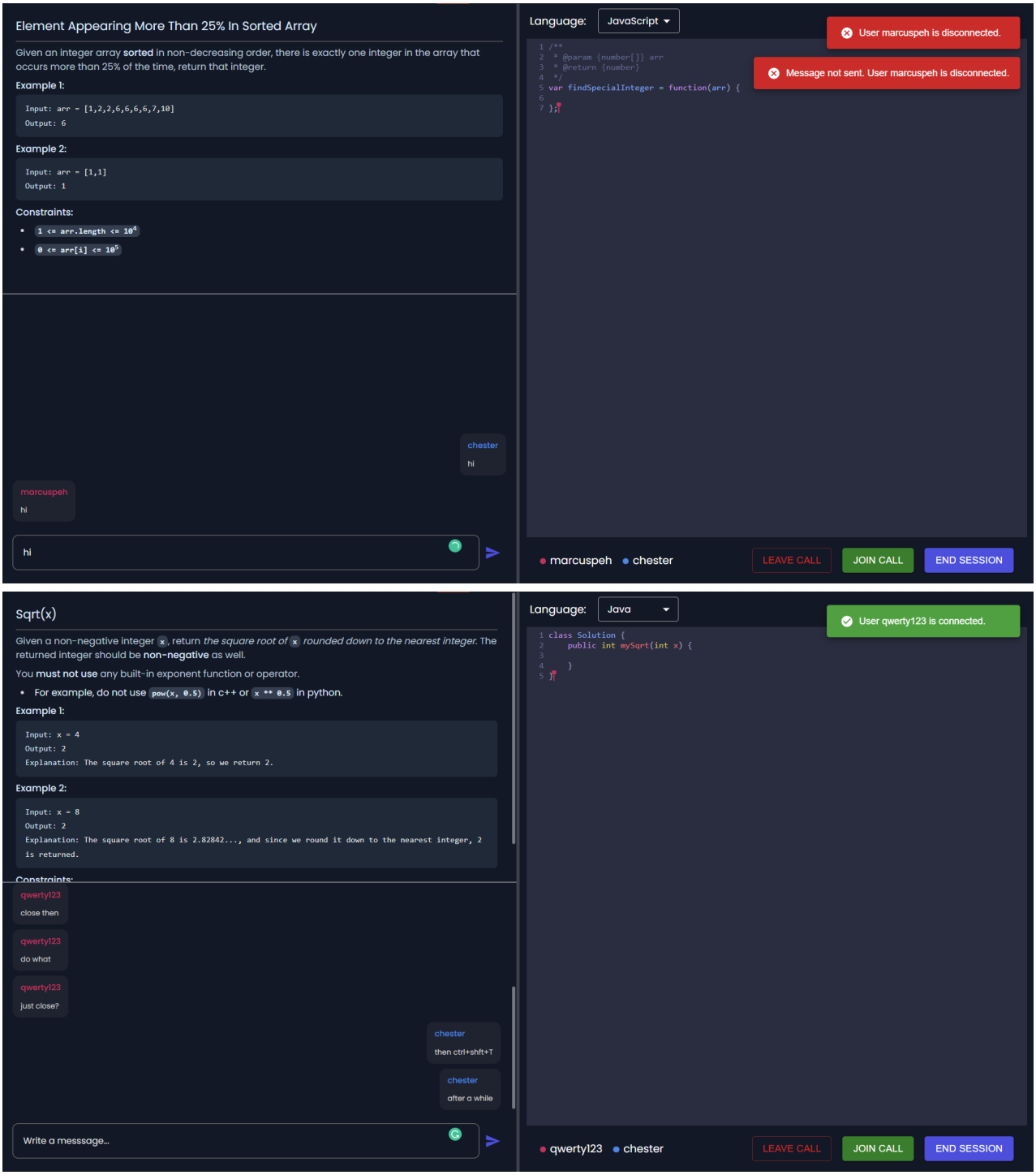


Change password page

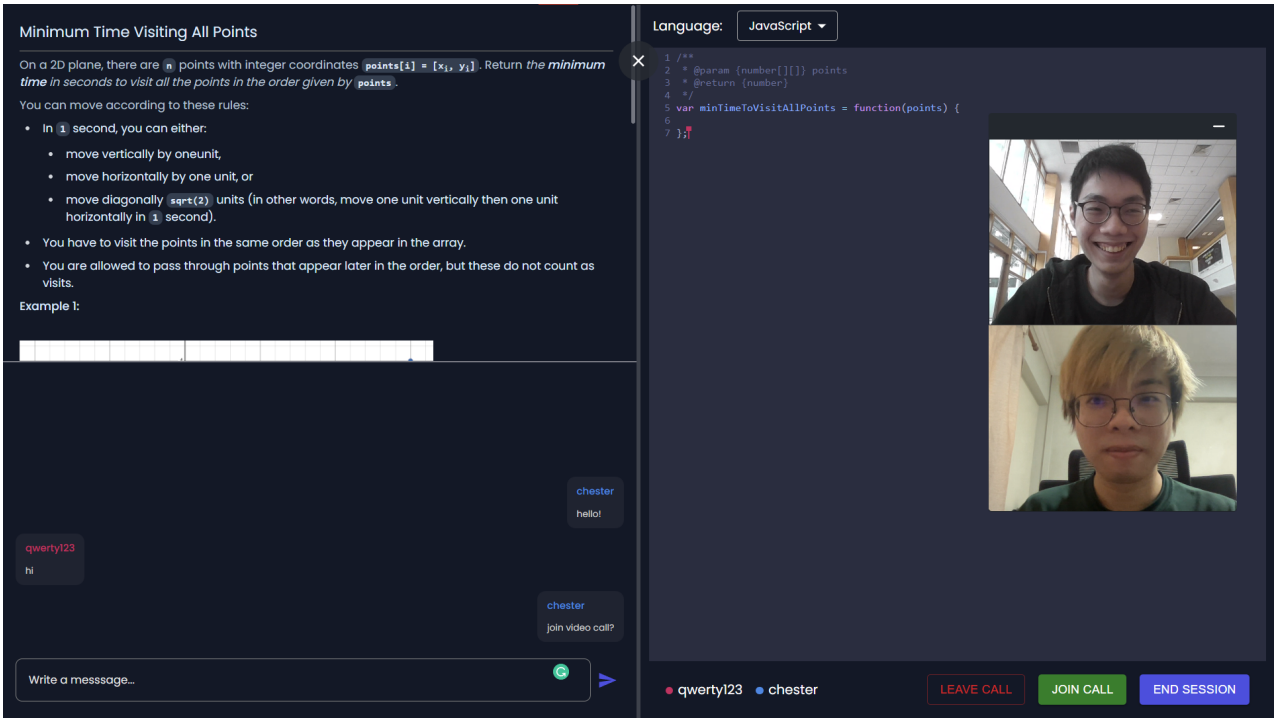


Delete account page

10.6 Interview Page



Snackbar notification examples (FR-AC-9)



Chat and video call available for users to communicate with one another (FR-AC-10 & FR-AC-11)

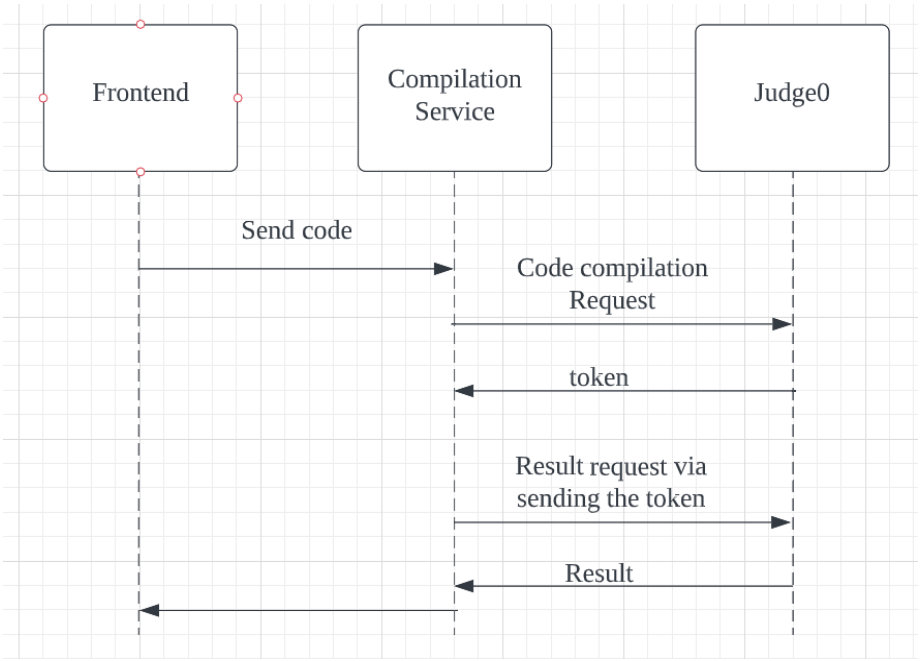
11 Extensions

11.1 Feature extensions

11.1.1 Code Execution

Currently our program doesn't support execution of code despite providing various language support such as C++, C#, Java, etc. Thus this purely relies on both participants to help each other proof read and understand the code functionality. This is indeed our initial intention as it helps to strengths the participants whiteboard coding skills which are essential. However, we believe that an additional code execution would definitely be useful for their own evaluation. As such we suggest using open platforms like Judge0 or Piston to assist in such an online code engine. We suggest Judge0 based on the reasons below:

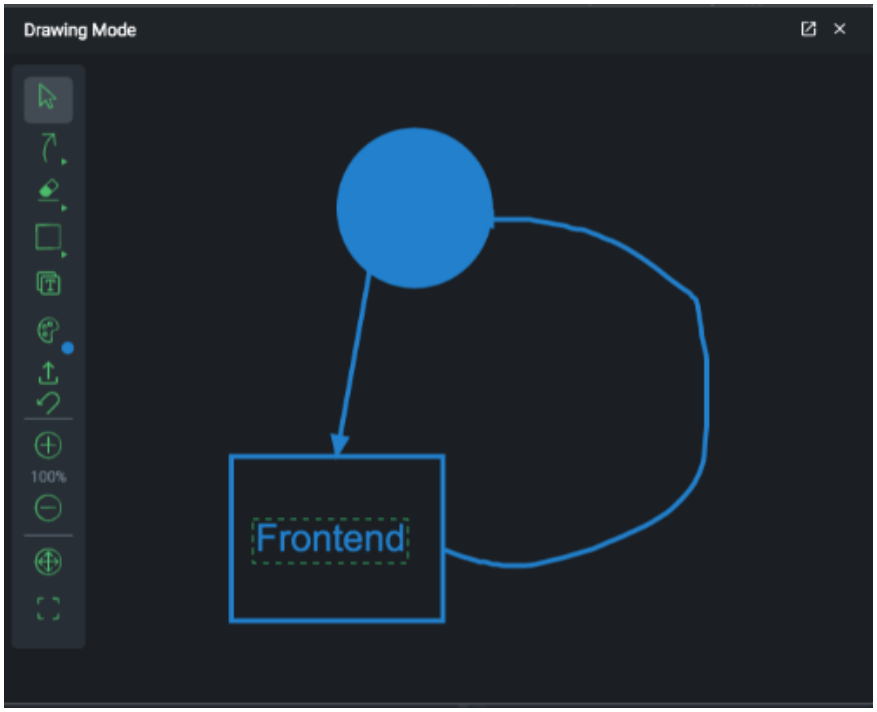
- Scalable
- Good documentation
- Strong community
- Sandboxing environment is provided during execution and compilation
- Supports a larger range of languages

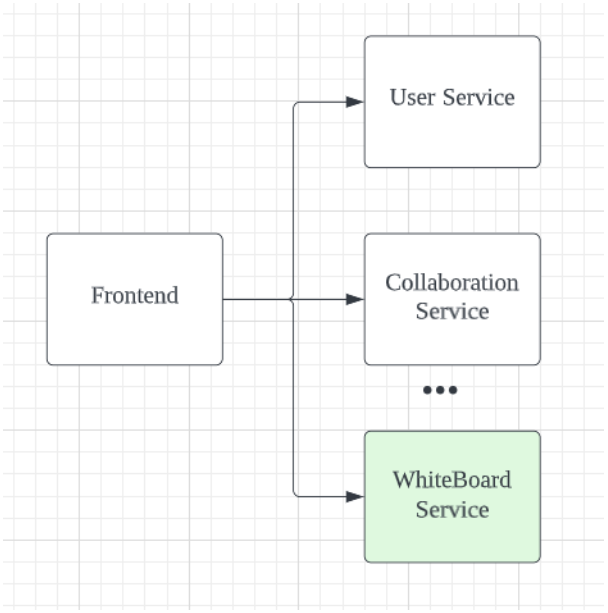


Suggested sequence diagram if using Judge0

11.1.2 Collaborative drawing whiteboard for discussion

It can be difficult to explain coed without diagrams. To better assist these participants in sharing their ideas clearly, we should provide them with a whiteboard popup/extensions to draw just like how we would in a normal whiteboard interview. An example of how the drawing mode could look like is shown below. Credits to CoderPad.





Suggested architecture diagram with the new WhiteBoard Service.

11.1.3 Multiple people in the room

Currently we limit only 2 person per room, but this can be easily be changed in our backend service to contain more than 2 participants. We initially set a hard requirement of 2 as we want it to be a pair collaborative session. But we do admit that having multiple people in the room models more closely to real life collaborative platforms as there can exist more than 2 people in the room per time.

11.1.4 Question extensions

Currently we support fetching of questions randomly from leetcode based on question difficulty. To allow users to have a greater exposure of different questions, we can considered querying from other popular coding platform such as Hackerrank.

1. Fetch a list of question information: https://www.hackerrank.com/rest/contests/master/tracks/algorithms/challenges?offset=0&limit=10&filters%5Bdifficulty%5D%5B%5D=easy
2. Fetch full questions using their name from the previous query (models > select an index > name): https://www.hackerrank.com/rest/contests/master/challenges/simple-array-sum/

11.1.5 Video Call extensions

Currently, we didnt support muting of video which might be distracting for the users during coding (sound from the keyboard). Also, currently it is more a draggable component within the app itself. We envision that we could actually use technologies like ElectronJS to create a video app by itself and linking to our original app.

11.1.6 CAPTCHA

Due to our business logic (unique usernames), there is some discrepancy factor which could allow an attacker to mount a user enumeration action against the application. A method to guard against this is to protect against brute-force attacks to prevent an attacker from applying enumeration at scale. Usage of CAPTCHA can be applied on our account sign up feature for which a generic error message cannot be returned because the user experience must be preserved.

11.1.7 Solo Practice and Matchmaking Extensions

Even though our platform promotes pair programming to help interviewees, including solo practice platform could be useful for users who want to do some preparation on their own. This allows them to use their time effectively when other users are not actively matching. To help users know when there are others finding a match, we would include a matching status component to the app that will indicate if there is another user is in the matching pool.

11.2 Implementation extensions

11.2.1 Using Message Queues to Decouple Services

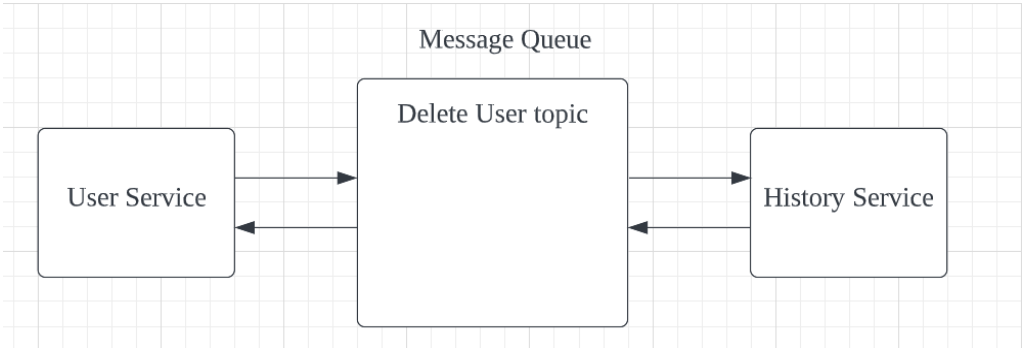


Diagram showing suggested message queue implementation

Currently we are using Synchronous Request pattern which is blocking in nature and not as reliable. We could definitely use a message queue to help us in such a situation. Firstly message queues will help to tackle the issue of coupling between the user service and history service by alerting the both services that the message has been received or processed. Furthermore the good thing about message queue is that even if there is a down time or failure, the message will not be lost and can be rescheduled later to ensure that the deletion of history will be successful.

12 Reflection

This project is indeed fulfilling and exciting as it allows us to explore various new technologies that we have not tried before and thinking about the trade offs for various implementations and technologies used. The projects allows us to wear multiple hats from planning the app design (through high-fidelity wireframe), to backend and frontend and even devops such as github actions and AWS, Docker and Kubernetes deployment. The team learned a lot thanks to our decision to create as many functionalities as possible by ourselves instead of relying on external libraries or services.

The following are some learning points from this project:

1) Live collaborative text editor is difficult to implement in reality as it involves synchronisation and ultimately, making sense of it. There are multiple research papers and algorithms that are presented to solve this problem which requires much knowledge in distributed systems. These are topics we are unfamiliar with and we implemented a best effort. Though we have to note that there are some libraries out there which help like Firepad and yjs.
2) We chose to build functionalities in-house as much as possible in the spirit of learning, but perhaps some things should not be self-implemented. A wise man once said: "Please everyone, stop writing your own user authentication code". Securing a web application is an endless journey, and if not done perfectly, can leave the application vulnerable to a multitude of attacks and exploits. Even if one is well-learned in the area of security, it is still better to leave it to the expert teams who dedicate their time to securing millions of users and fixing difficult zero-day vulnerabilities. That said, we consciously made the decision to store **no user identifying information** since we are aware of our own shortcomings.
3) It is better to analyze the problem/feature thoroughly with the team before implementing it and realising that there is a better approach and reimplementing the feature again. After delegating work, our members took it upon themselves to solve their own puzzles without consulting others, for fear of "disturbing" them. However, some problems are better solved from different perspectives. Problems in implementation can be caught with our peer review process, but more time could've been saved by putting in more initial thought into the design.
4) Communication is paramount to team success. The constant communication and updating in our team has allowed us to proceed more smoothly and keep track of each other's progression.
5) The learning curve was steep as many of us come from different backgrounds. Picking and learning multiple technologies that are relevant to the project within a short period is indeed difficult.
6) Crucial to link up the various services and frontend early. Our team enforces this by front-loading API design decisions. As such, we did not face any last minute issues delivering each milestone.