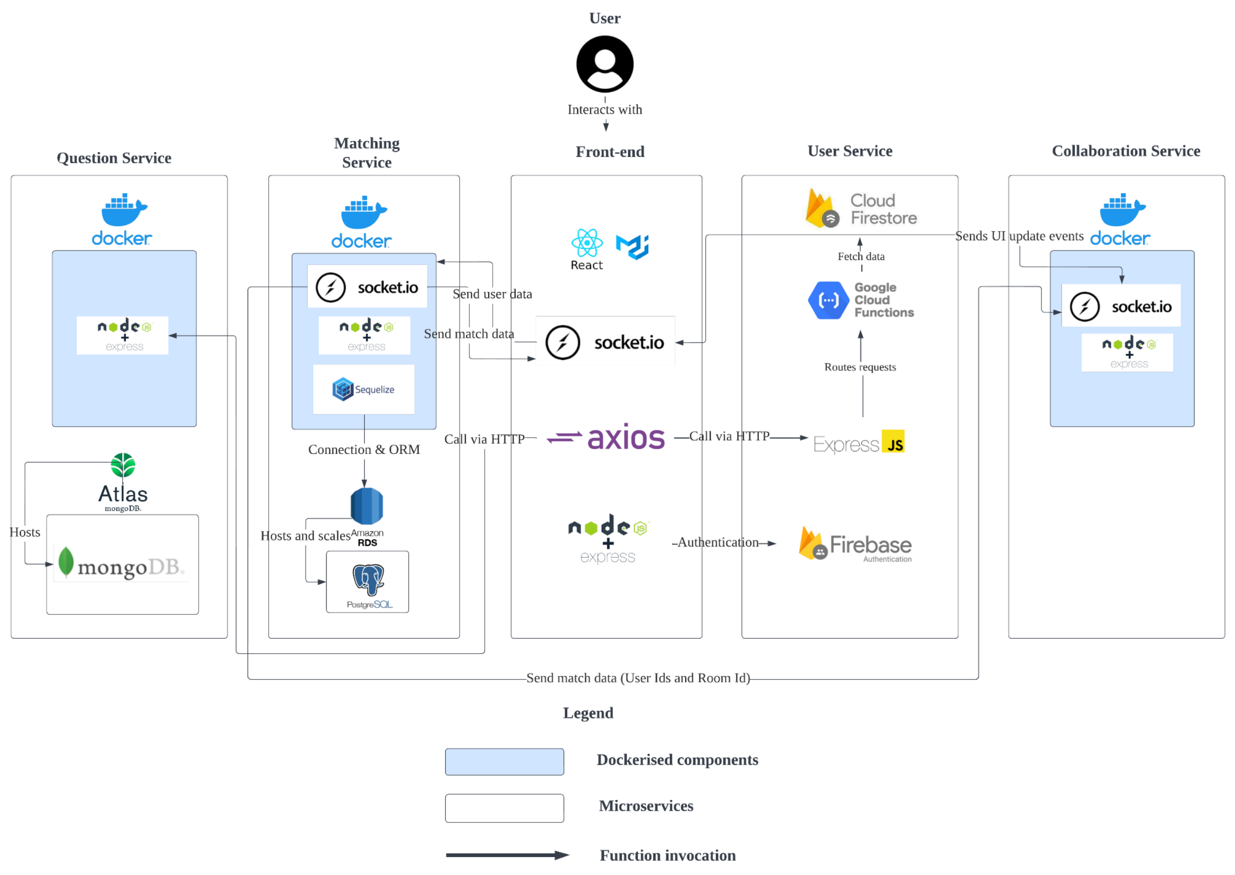
CS3219
AY 2022/2023 Semester 1
Group Project - Peerprep
Group: 26

Full Name	**Matriculation Number**
Euzin Tan	A0201847H
Ryan Tan	A0217493Y
Marcus Tan	A0204867X

Table of Contents

Design Documentation

Architecture and Tech Stack



As seen from the above diagram, our software architecture uses a microservice architecture, where we have 4 distinct microservices, the question service, matching service, user service and collaboration service, along with a front-end that pulls data from the relevant microservices.

Architectural Decision: Monolith vs Microservice

We decided that a microservice architecture suited our needs more than a Web MVC Layered Application (Also known as a Monolith). We wanted our system to be easily scalable and loosely coupled.

Scalability

A microservice architecture is easily scalable as each microservice contains its own independent mini-ecosystem of appropriate resources separate from other microservices.

A Web MVC Layered application will not be as easily horizontally scalable as the entire monolith must be duplicated when a particular service inside the monolith is under heavy load which is more expensive and less responsive. On the other hand, for a microservice architecture, each microservice can be independently scaled according to demand with relatively low latency with orchestration services such as Kubernetes or other AWS/Google Cloud Services.

Separation of Concerns

We also chose a microservice architecture as it reduces the most harmful form of coupling, implementation coupling and enforces Separation of Concerns at an architectural level. Since each microservice handles different logic and is worked on by a different programmer, reduced implementation coupling allows for easier testing and debugging of the program as it can be broken down into small components and tested in isolation.

Backend Microservices

User Service

The user service's purpose is to enable users to sign in before using the application so that their progress can be saved and their identity can be established.

Components

The code for user service is divided into two main components:

1. functions/api/user/route.js
2. functions/helper/auth.js

route.js

Allows the system to automatically perform read and write operations to the user's profile information in order to save user progress like number of questions completed, breakdown of difficulty of questions attempted, saved code, etc.

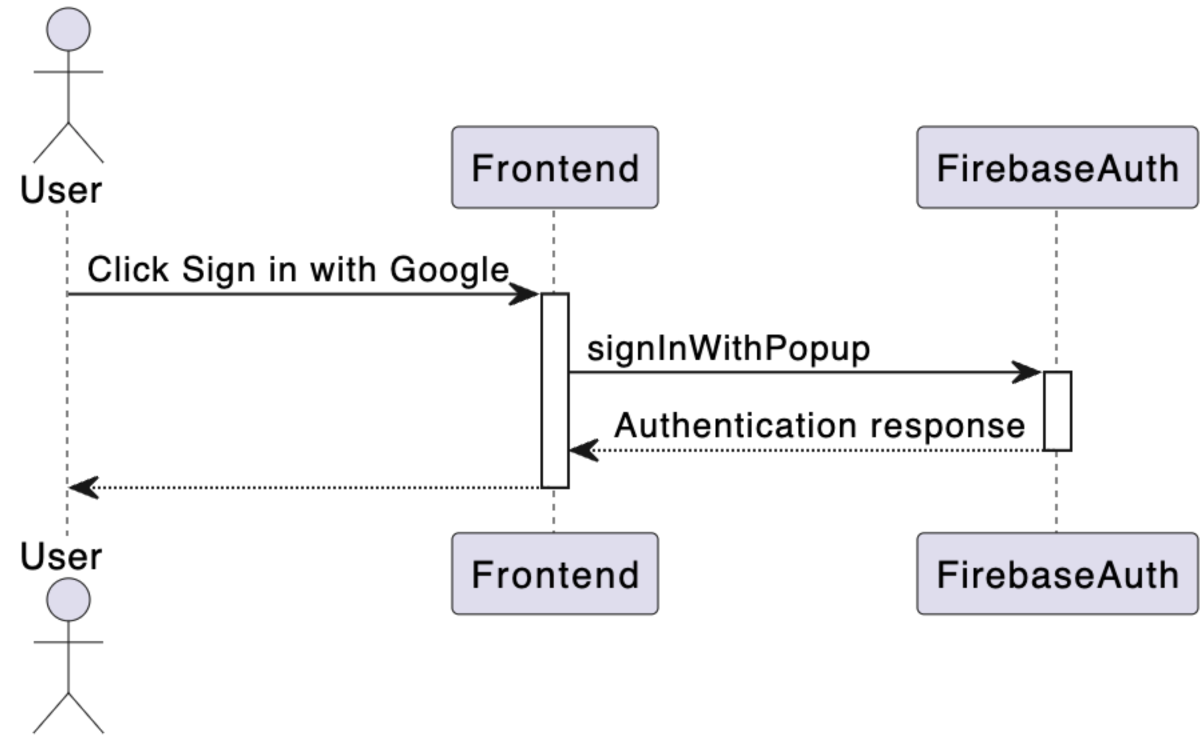
auth.js

Allows the system to perform authentication when a request is sent to the backend. This module will use Firebase admin SDK to validate the authorization token obtained from the header of the API request.

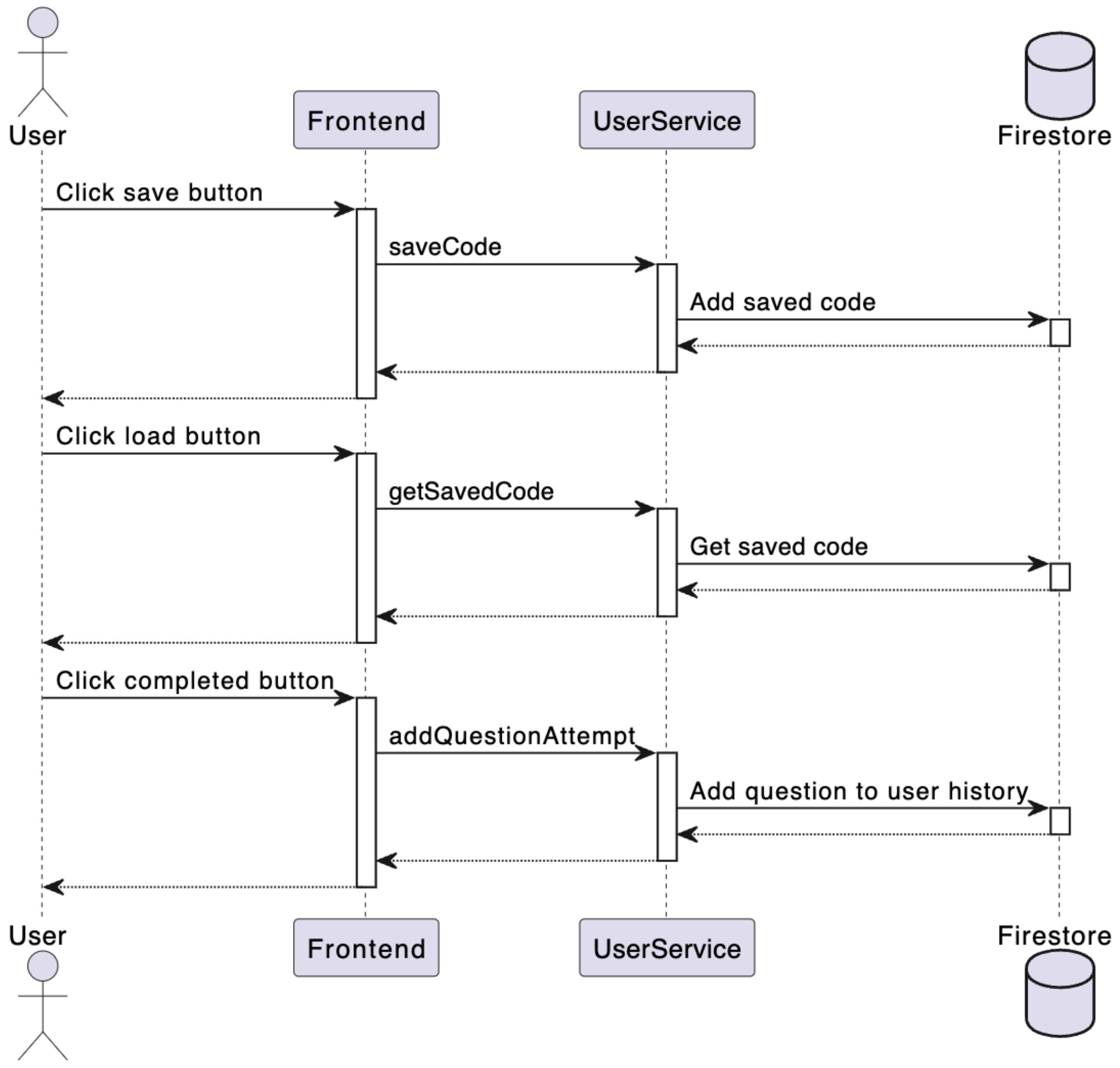
External Library: **Firebase Admin SDK**
This SDK allows the app to generate and verify firebase auth tokens as well as allow users to sign in different providers like Google sign in.

Internal Sequence Flow

Signing in

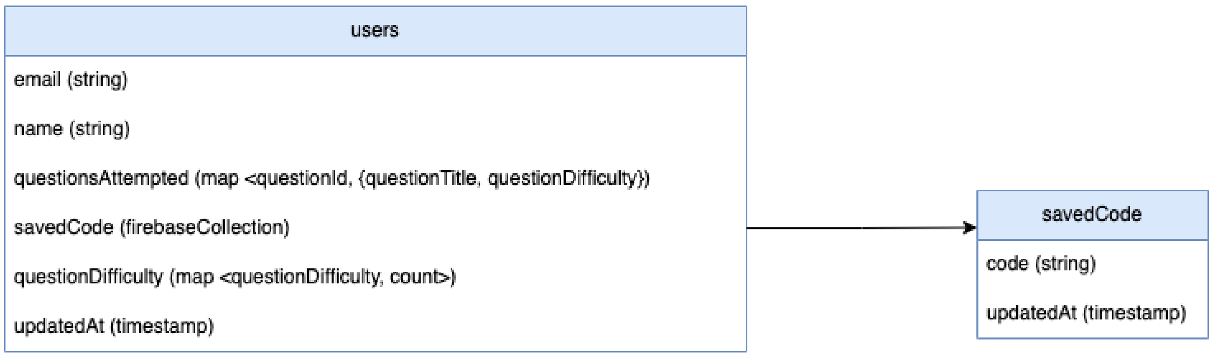


Saving and Loading code in the Code editor
When a user is in the code editor, the average use case will result in 3 main actions - saving code progress, loading previously saved code and completing the question attempt. These will result in API calls to the User Service which reads and writes to the Firebase Firestore database.



Database Models

Below are the database models used in the User Service:


users
email (string)
name (string)
questionsAttempted (map <questionId, {questionTitle, questionDifficulty})
savedCode (firebaseCollection)
questionDifficulty (map <questionDifficulty, count>)
updatedAt (timestamp)
savedCode
code (string)
updatedAt (timestamp)

Note: User Service uses Cloud Firestore, which is a NoSQL, document-oriented database

Matching Service

The matching service's purpose is to enable users to select the difficulty of questions they want to attempt and get matched with other users who selected the same difficulty.

Components

The code for the backend Matching Service is divided into three components (ES6 modules)

1. database.js
2. time.js
3. index.js

Database.js

The database.js module is used to query the postgres database containing the Waiting model and the Socket model. These models will be elaborated on in the Database Models section

External Library: **Sequelize**
The database.js module uses Sequelize for ORM, to connect to a Postgres database.

The waiting table in the database containing the Waiting model acts as a persistent stack to contain the data of all users currently waiting for a match. The module also contains query logic to determine if a match is found and to return that data to the Node.js layer of the Matching Service. The socket model and correspondingly the socket table, keeps track of the socket ids for all sockets currently connected across all instances of the matching-service so that these ids can be used to connect the sockets across web servers/nodes.

Time.js

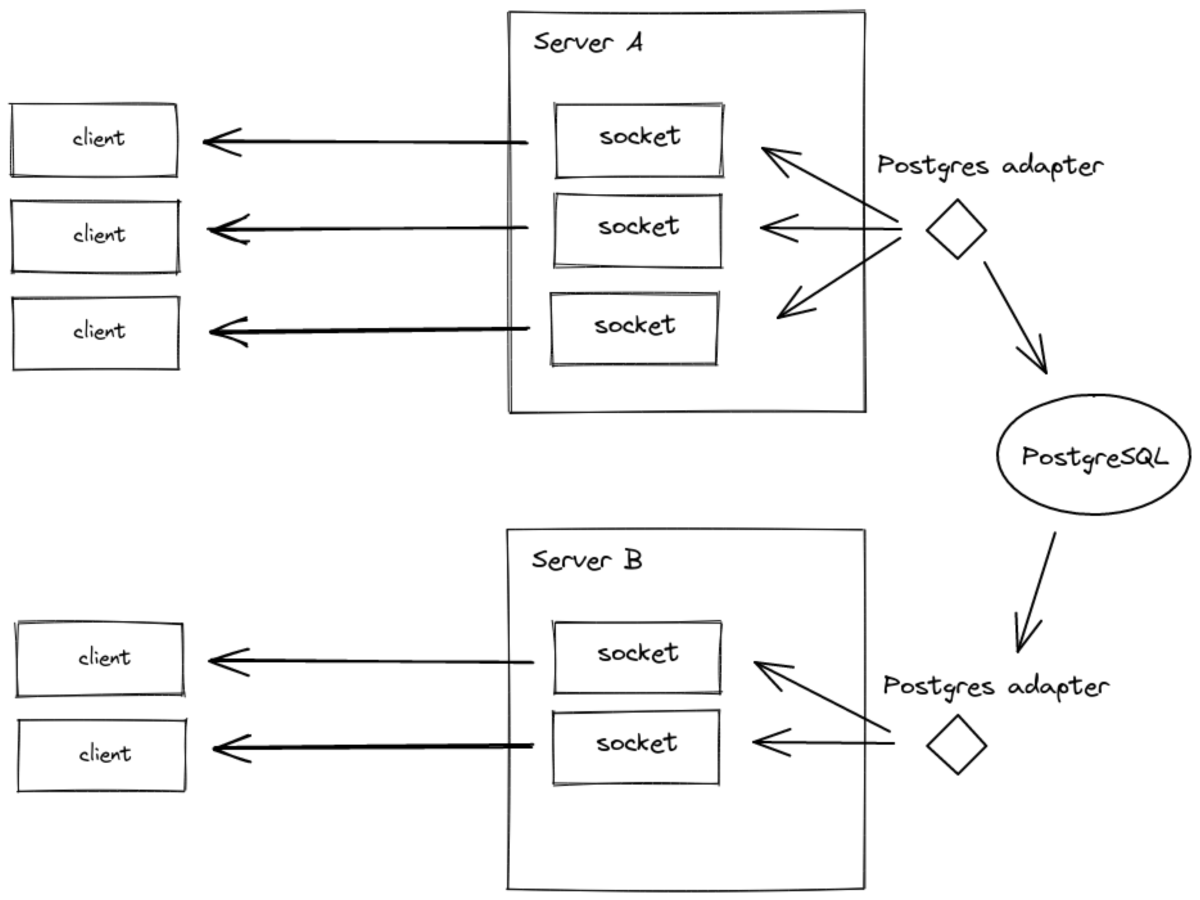
The time.js module simply obtains the time in SGT from the client and formats it to supply to the match model.

Index.js

The index.js module contains all logic for the use of WebSockets with the external library Socket.io in order to communicate in real time with the frontend, for e.g to get a match.

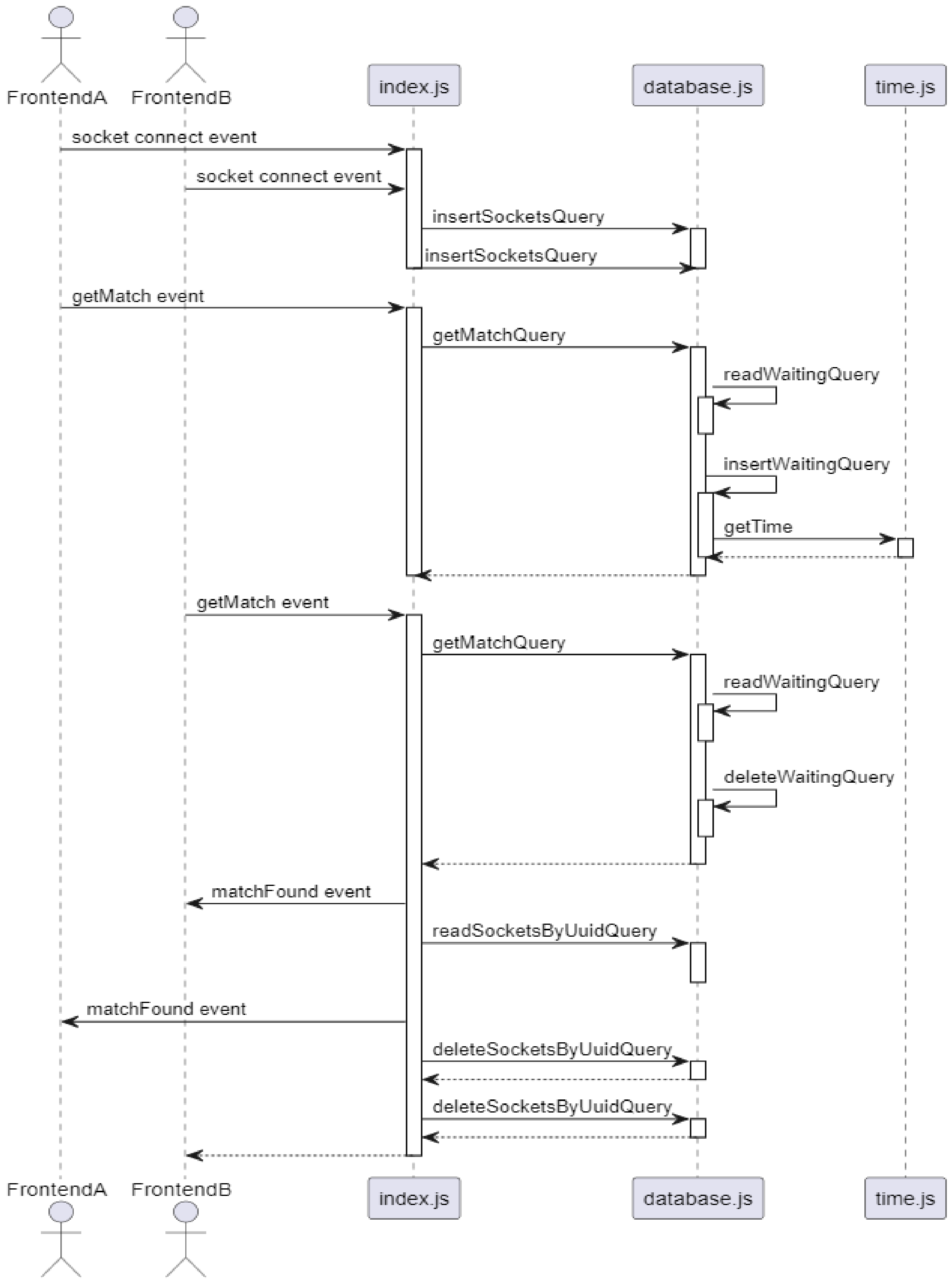
External Library: **Socket.io Postgres Adapter and pg**

A Socket.io adapter is also used in conjunction with Postgres in order to ensure that sockets running on different Node.js instances or different web servers can communicate with one another. This is important as during deployment we might run several instances of the matching-service, but logic for connecting two clients for a match depends upon sockets which could be across multiple web servers hence the need for sockets to communicate across server nodes/instances through the adapter. **This same setup is used for the Collaboration Service.**



Internal Sequence Flow

The average use case for the matching-service will be two frontend clients logging in with two different users will each request a match to the same difficulty: Easy, Medium or Hard. How the matching-service handles this request is shown in the sequence diagram below.



Database Models

Below are the database models used in the Matching Service which are stored in SQL tables:

Model: Waiting		
Fields	**Attributes**	**Description**
uuid	● Type: character varying ● Not null: True ● Primary Key?: True	uuid of the user waiting for a match
difficulty	● Type: character varying ● Not null: True	Difficulty that of the questions that the user wishes to attempt
time (time without timezone)	● Type: time without timezone ● Not null: True	Time at which the user requested for a match

Model: Sockets		
Fields	**Attributes**	**Description**
id	● Type: character varying ● Not null: True ● Primary Key?: True	Socket.io socket id of frontend socket used to connect to the matching-service
uuid (character varying)	● Type: character varying ● Not null: True	uuid of the user waiting for a match

Collaboration Service

The collaboration service employs socket.io to enable different clients to be able to persistently and synchronously code and communicate with each other. It serves four main functions:

1. Synchronisation of the code editors between users
2. Synchronisation of the questions between users
3. Chat service between the users
4. Joining and leaving of a session for either of the users

Components

index.js

The index.js module contains all logic for the use of WebSockets with the external library Socket.io in order for clients to communicate with each other through the UI in real time.

External Library: **Socket.io**
Socket.io is employed as the middleman for interaction between clients and their joining or leaving of sessions.

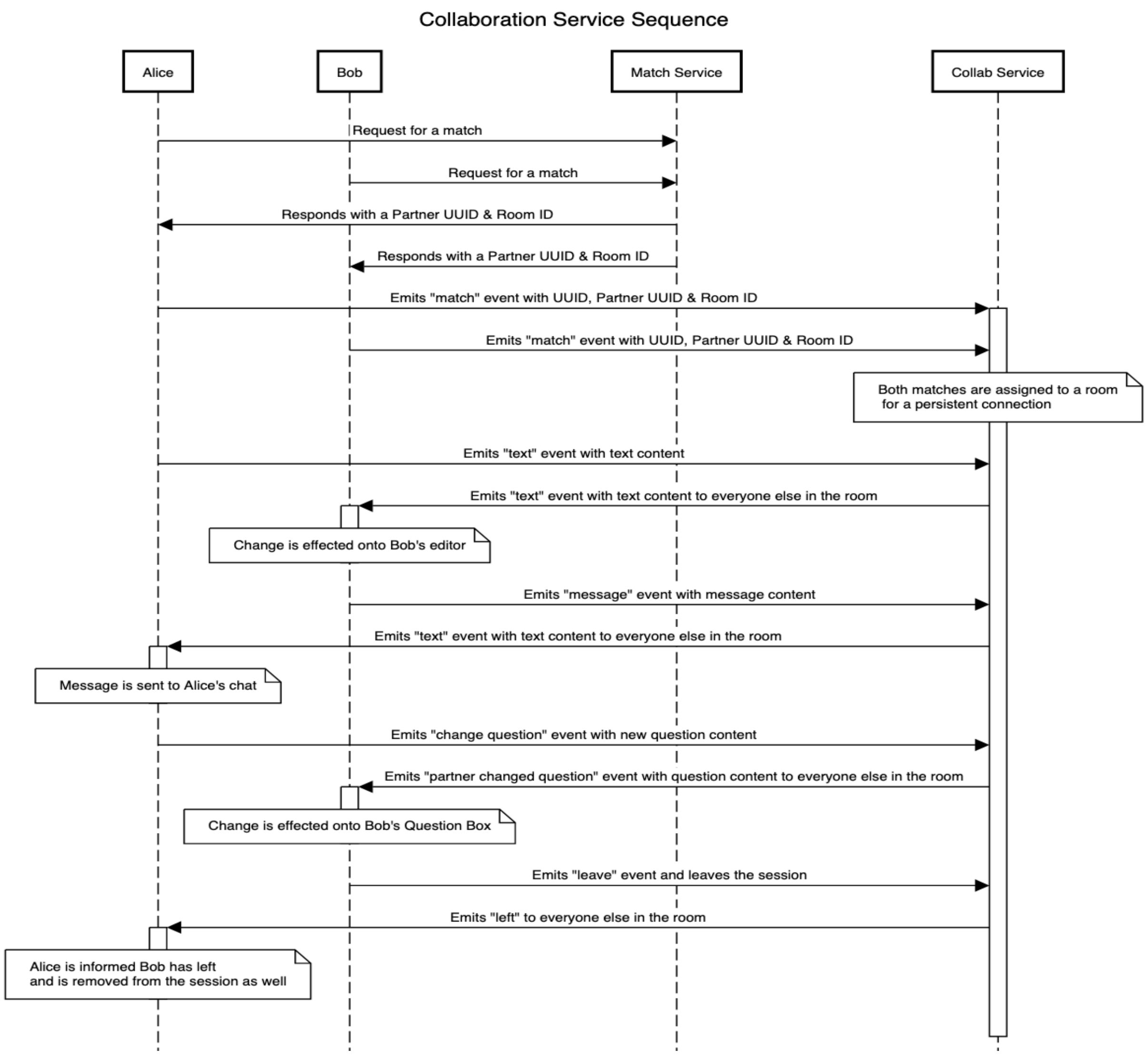
When a user emits a ‘match’ event with a roomID as a parameter, they are added to a specific ‘room’ by socket.io. Two users are expected to join the room at the same time. Henceforth, all communication to and fro these users will be limited to within that room.

The socket also acts as a middleman for interaction by receiving event emissions from clients and processing them and emitting them to other specific clients. There are 3 types of emissions ‘text’, ‘message’ and ‘leave’ which respectively indicate the user’s intention to code, send a message or leave the session. This intention is then conveyed to the other client in the room.

The frontend Common Editor, Chat and QuestionBox components are then updated based on the events received by the frontend.

External Library: **Socket.io Postgres Adapter**
A Socket.io adapter is also used in conjunction with Postgres in order to ensure that sockets running on different Node.js instances or different web servers can communicate with one another as elaborated above regarding the Matching Service

Internal Sequence Flow


Collaboration Service Sequence
Alice
Bob
Match Service
Collab Service
Request for a match
Request for a match
Responds with a Partner UUID & Room ID
Responds with a Partner UUID & Room ID
Emits "match" event with UUID, Partner UUID & Room ID
Emits "match" event with UUID, Partner UUID & Room ID
Both matches are assigned to a room for a persistent connection
Emits "text" event with text content
Emits "text" event with text content to everyone else in the room
Change is effected onto Bob's editor
Emits "message" event with message content
Emits "text" event with text content to everyone else in the room
Message is sent to Alice's chat
Emits "change question" event with new question content
Emits "partner changed question" event with question content to everyone else in the room
Change is effected onto Bob's Question Box
Emits "leave" event and leaves the session
Emits "left" to everyone else in the room
Alice is informed Bob has left and is removed from the session as well

Question Service

The question service's purpose is to enable users to obtain questions from the question bank to be used to practise programming.

It employs a basic Model-View(Frontend)-Controller pattern. It is a simple REST API connected to a database that enables basic question fetching operations.

Components

questionModel.js - Model

questionModel.js contains the Schema created via Mongoose. It manages the data structure and logic of the question-service. The database model can be found below in a later section.

questionController.js - Controller

questionController contains the functions employed by the Question Service. It takes in the user's input and converts it to commands for the questionModel. It also manages sending responses to the user.

server.js

server.js exposes two endpoints, GET /questions and GET /allQuestions.

GET /questions allows the user to either:

1. Get a specific question based on the id provided by the user
2. If not ID is provided, get a random question of a specific difficulty provided by the user

GET /allQuestion simply retrieves every question from the database. It is employed by the ChangeQuestionModal to allow users to choose a specific question to attempt.

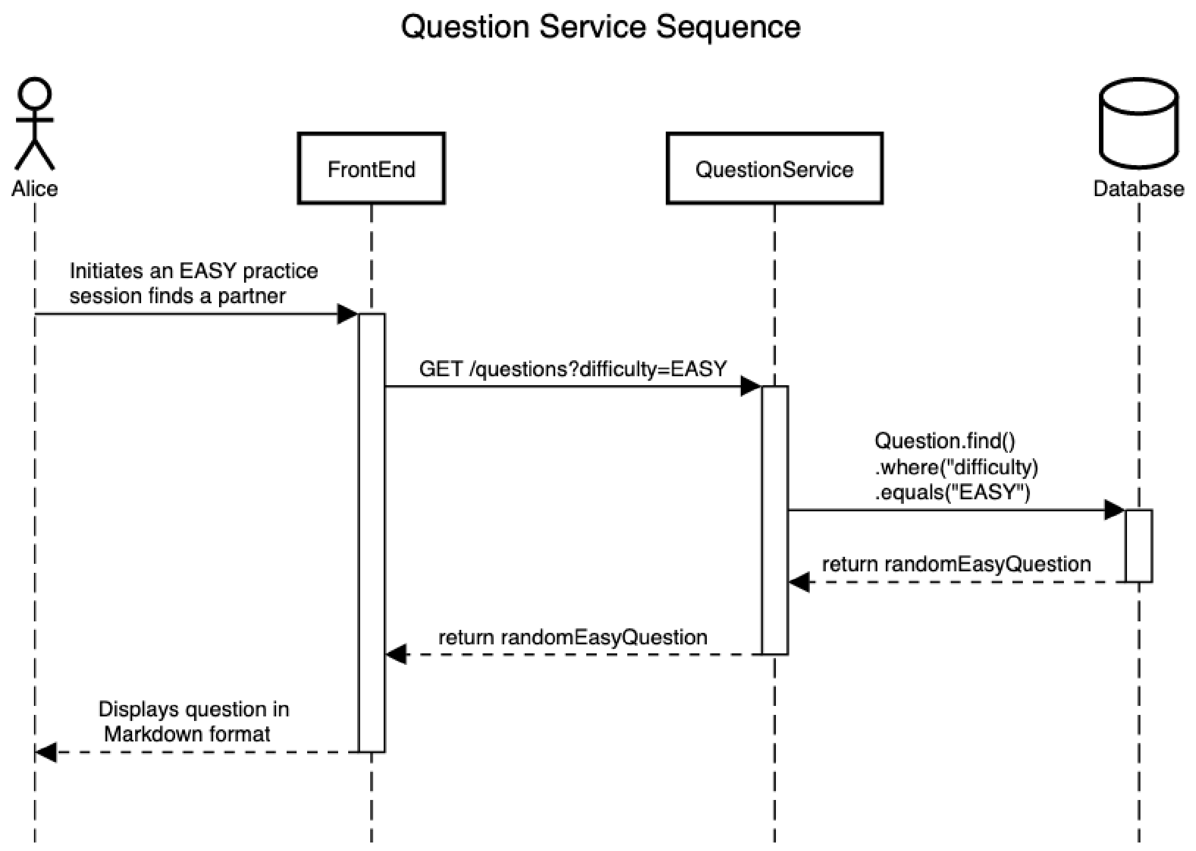
Database Models

Below is the database model used for the Question Service which as mentioned before.

Model: Question		
Fields	**Attributes**	**Description**
_id	• Type: character varying • Not null: True • Primary Key?: True	uuid of the question
name	• Type: character • Not null: True	
difficulty	• Type: character varying • Not null: True	Difficulty that of the question: EASY, MEDIUM or HARD
question	• Type: character • Not null: True	The question itself in markdown language to be displayed to users

Internal Sequence Flow

This sequence diagram depicts how the average use case of the Question Service would look like


Question Service Sequence
Alice
FrontEnd
QuestionService
Database
Initiates an EASY practice session finds a partner
GET /questions?difficulty=EASY
Question.find() .where("difficulty) .equals("EASY")
return randomEasyQuestion
return randomEasyQuestion
Displays question in Markdown format

Frontend

The front-end of the Collaboration service is contained in Practice.js, which will be routed from MatchingPage.js .

Routing Parameters:

1. UUID1: The current User's ID
2. UUID2: The User's assigned partner's user ID
3. roomID: The Room ID assigned to the user by the Match Service API
4. Difficulty: The difficulty level chosen by the user during the matching phase

Practice Page

Question Box

The Question Box will fetch a question from the Question Service API based on the desired difficulty level. The Questions Service will provide a string containing the question written in **Markdown Language.** Users are also able to choose a specific question they would like to

External Library: **React Markdown**
To display the string fetched from the Question Service as Markdown Language, we are using React Markdown.

Change Question Modal

This is a Modal that appears when users click on a button to change the question being collaborated on. In this modal, users can specify the difficulty of the question they would like to attempt, and subsequently choose a specific question to attempt. The selected question is then shown in Markdown inside the modal. Clicking on the "Confirm Question Change" button will result in a change in question for both users in the session.

Common Editor

The Common Editor is where both users in a room will be coding. Its key feature is that it is always editable by and synchronised between both users in the same room. It also allows for saving of code for loading in future sessions and marking a question as completed.

External Library: **Socket.io Client**
To enable a persistent connection between the clients for synchronisation, we are using Socket.io Client to interface with the Collaboration Service API, which is also powered by Socket.io. Upon rendering of the Editor, the front-end client initiates a connection with the Collaboration Service API and is henceforth listening for changes in the Common Editor by their partner.

External Library: **Monaco Editor for React**
Monaco Editor is a code editor that powers VS Code. This library handles the setup process of a monaco editor in the front-end client and provides a clean interface for interacting.

Load Code Modal

This modal appears when the user indicates an interest in loading code saved from a previous attempt by hitting the "Load Code" button in the Practice page. The modal will show the user their previously saved attempt and if the user hits the "Confirm Restore" button, the saved code will be loaded onto both Common Editors.

Partner Leave Modal

This modal automatically appears when the user's partner has left the room. The modal informs the user their partner has left and will give the user the option of Saving or marking his attempt as Completed, or simply leaving the room.

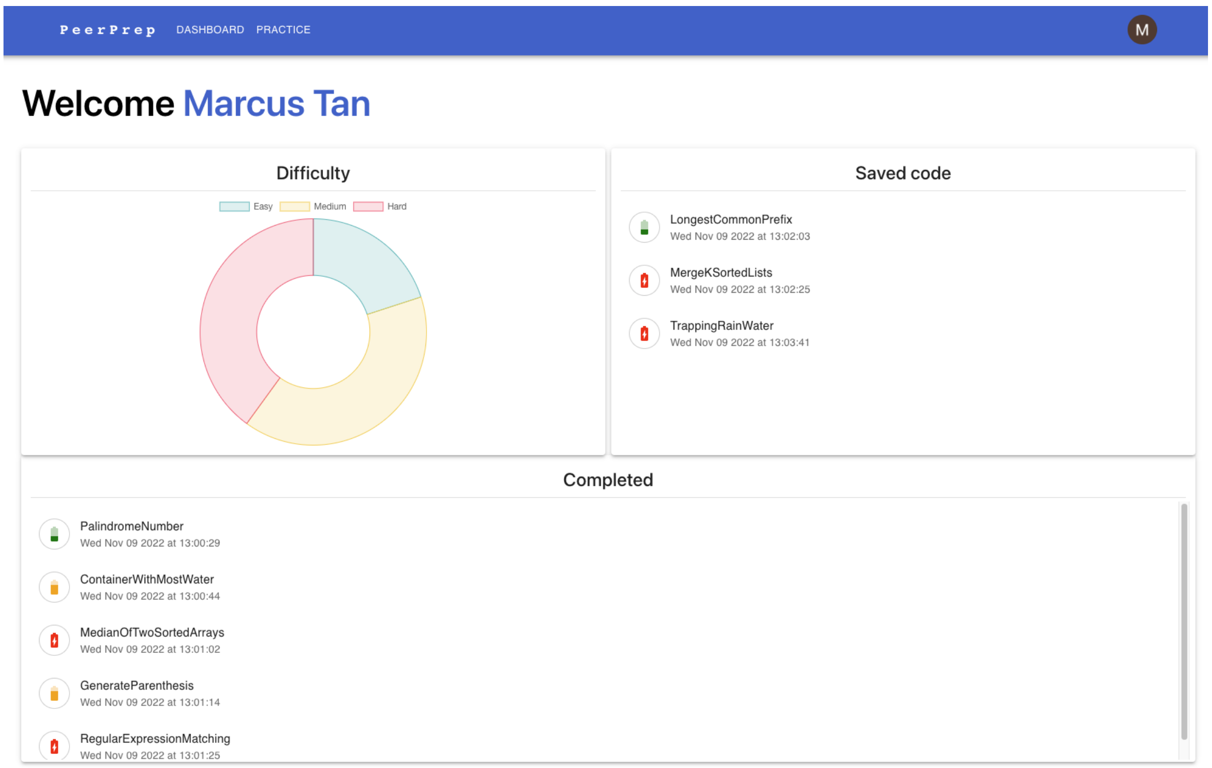
Difficulty Select Page

The Difficulty Select is where users will select their difficulty and wait for a match. Users will be connected to the matching-service via sockets which will match users and publish events to the front-end to synchronise their connection to collaboration service.

<u>External Library: **Socket.io Client**</u>
To enable a persistent connection between the clients for synchronisation, we are using Socket.io Client to interface with the Matching Service API, which is also powered by Socket.io. Upon rendering of the Difficulty Select, the front-end client initiates a connection with the Matching Service API. The front-end client then updates the Matching Service API via socket events, for e.g getMatch. The Matching Service API also updates relevant front-end clients when users are matched.

High Usability with Good UI/UX design

In this project we made high usability and good UI/UX design a priority through an NFR (see more below in Requirements).



Welcome Marcus Tan

LongestCommonPrefix
Wed Nov 09 2022 at 13:02:03

MergeKSortedLists
Wed Nov 09 2022 at 13:02:25

TrappingRainWater
Wed Nov 09 2022 at 13:03:41

PalindromeNumber
Wed Nov 09 2022 at 13:00:29

ContainerWithMostWater
Wed Nov 09 2022 at 13:00:44

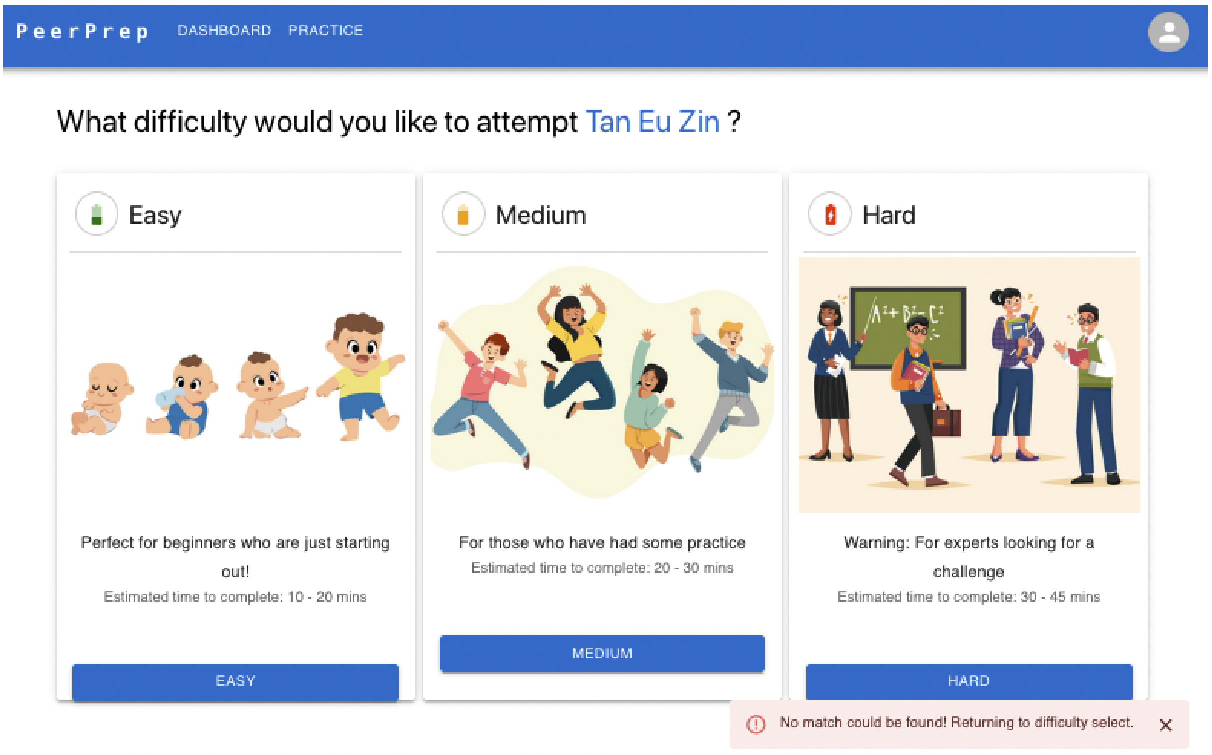
MedianOfTwoSortedArrays
Wed Nov 09 2022 at 13:01:02

GenerateParenthesis
Wed Nov 09 2022 at 13:01:14

RegularExpressionMatching
Wed Nov 09 2022 at 13:01:25

As seen from the above screen shot of the dashboard and the demo, we attempt to make the user experience as good as possible with a fancy UI, as well as notifying the users about the current status of their actions through notifications such as snack bars and timers.

For instance, the Difficulty Select component when unable to match a user will notify the user that a match was not found during matching.



What difficulty would you like to attempt Tan Eu Zin ?

Easy

Perfect for beginners who are just starting out!

Estimated time to complete: 10 - 20 mins

EASY

Medium

For those who have had some practice

Estimated time to complete: 20 - 30 mins

MEDIUM

Hard

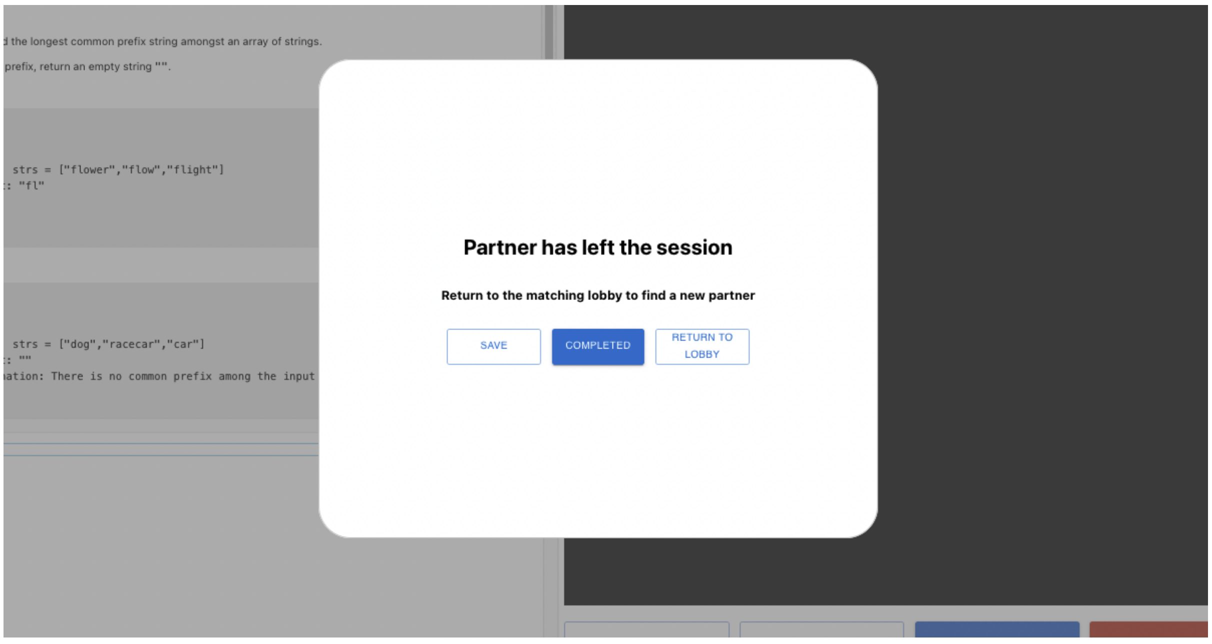
Warning: For experts looking for a challenge

Estimated time to complete: 30 - 45 mins

HARD

No match could be found! Returning to difficulty select.

Another instance of the effort we made in UI/UX design is when the Partner Leave Modal will notify the user about the fact that their partner has left the session. It also gives the user the opportunity to save or mark their code as completed before returning to the lobby.



```
d the longest common prefix string amongst an array of strings.
prefix, return an empty string "".

 strs = ["flower","flow","flight"]
: "fl"

 strs = ["dog","racecar","car"]
: ""
ation: There is no common prefix among the input
```

Partner has left the session

Return to the matching lobby to find a new partner

Deployment Details

Matching, Question and Collaboration Service

The matching, question and collaboration services are Node.js backend services, and therefore have to be deployed on web server environments.

Containerization with Docker

For ease of deployment, the Matching Service, Question Service and Collaboration Service have been containerized, following the motto “write once, run anywhere”. With these containers, we can run the services on a variety of architectures without having to rewrite code. Containerization also allows the services to be scaled easily with container orchestration technologies.

Local Deployment: Docker Compose

For ease of local testing, we have also included a local deployment setup that involves docker compose.

Ensure the docker daemon is running locally on your computer. Then, we run simply `docker compose --profile local build` to build the images locally and then run `docker compose --profile local up` in the root folder to set up all three services running in containers locally.

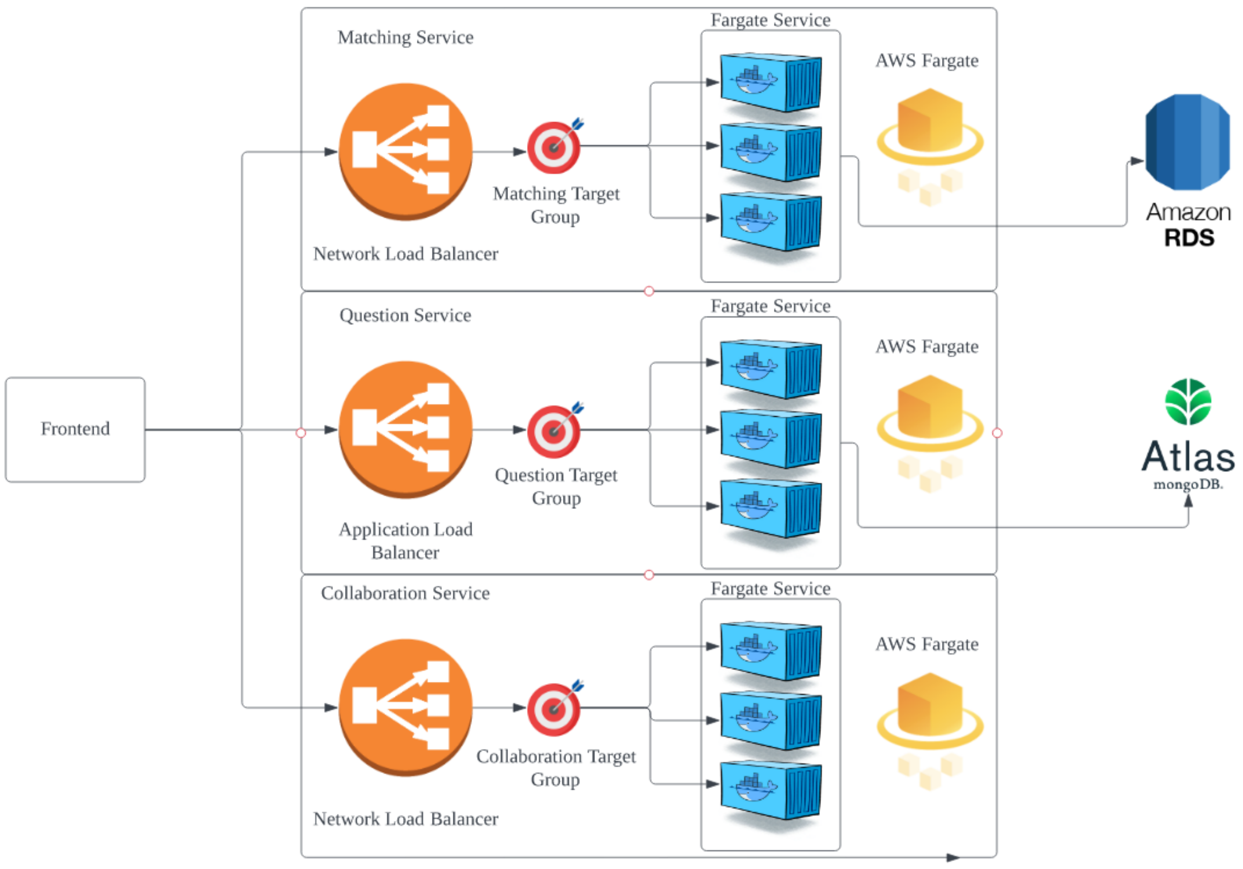
More information on how to run a complete local deployment of our system can be found in our README on GitHub.

Cloud Deployment Tech Stack

The Matching Question and Collaboration Services make use of the following technologies for a production cloud deployment:

1. Amazon Elastic Container Service (ECS)
2. Amazon Elastic Load Balancer (ELB)
3. Amazon Fargate
4. Docker
5. Docker Hub
6. Amazon RDS
7. Mongo Atlas

Cloud Deployment Environment



The Docker containers are pushed to an online repository for container images called Docker Hub. Amazon ECS then pulls these images from Docker Hub and deploys these images to services running in Amazon Fargate (more on this in the Continuous Deployment Section), which comes with its own proprietary container orchestration, where we use one service for each microservice. We then create a target group per microservice that points to instances created by the Fargate services, and attach an AWS Network Load Balancer to each target group, so that TCP traffic can be easily divided between instances running on Fargate.

The frontend then pulls data through HTTP Requests or through a WebSocket connection established with the instances by first connecting through each microservice's load balancer.

The database for the Matching Service is run on Amazon RDS for Postgres whereas the database for Question Service is run on Atlas for MongoDB.

Horizontal Scalability

As mentioned above, we prioritised horizontal scalability in this project and have accordingly containerized each of the above microservices. We can scale the number of containers for each microservice with AWS Fargate Auto Scaling Groups, which increase the number of container instances running depending on metrics such as CPU load per instance. We then ensure that the load is correctly balanced within each service by using an AWS Network Load Balancer to divide TCP traffic between the Fargate Instances running in each service.

The use of Amazon RDS and Atlas also automatically provides provisioned databases with horizontal scaling by increasing the number of database instances running when load increases.

User Service

Local Deployment: Firebase Local Emulator Suite

Firebase provides us with a local emulator suite to test our apps locally, Cloud Firestore, Realtime Database, Cloud Storage for Firebase, Authentication, Firebase Hosting, Cloud Functions (beta), Pub/Sub (beta), and Firebase Extensions (beta).

For the user-service we make use of Firebase Functions. We simply deploy our functions locally with `npm run firebase` in the user-service directory to run the local emulator suite.

Deployment: Firebase Deploy

Firebase allows us to easily deploy our functions that we have tested locally to a Firebase Functions production environment with `firebase deploy --only functions`. Note that you will be unable to run this command as you are not authenticated.

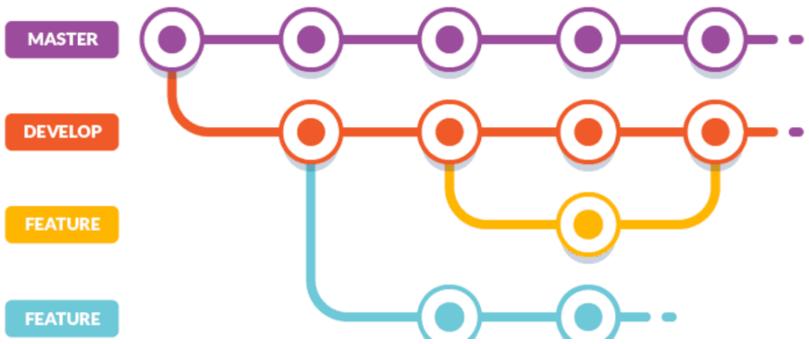
More information on how to run a complete local deployment of our system can be found in our README on GitHub.

Use of Firebase: Horizontal Scalability

Firebase automatically scales horizontally and vertically, featuring automatic multi-region data replication, strong consistency guarantees, atomic batch operations, and real transaction support.

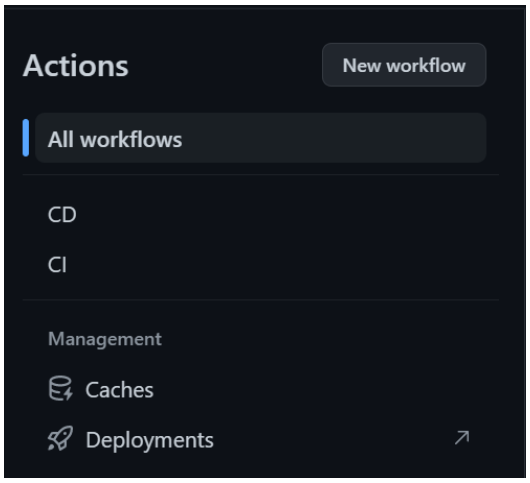
Development Process

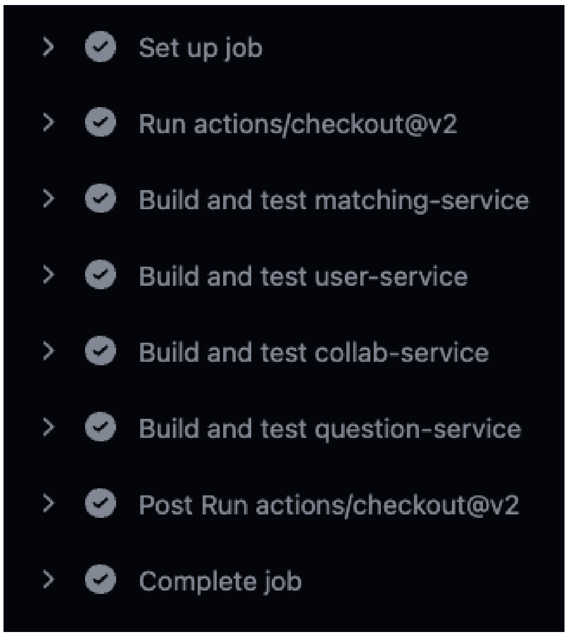
Feature Branch Workflow

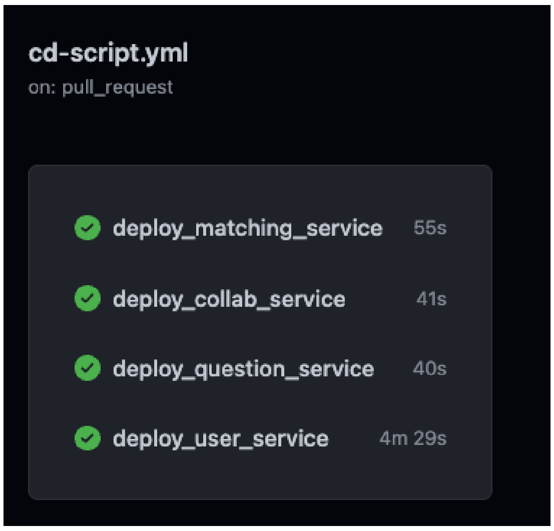


In this module, the team decided to go with a branching workflow. In the branching workflow, each team member working on a feature will branch off the main branch and write the feature in a new branch. Upon completion, the team member will then make a PR to merge the feature branch into the main branch.

CI/CD




Set up job
Run actions/checkout@v2
Build and test matching-service
Build and test user-service
Build and test collab-service
Build and test question-service
Post Run actions/checkout@v2
Complete job


cd-script.yml
on: pull_request

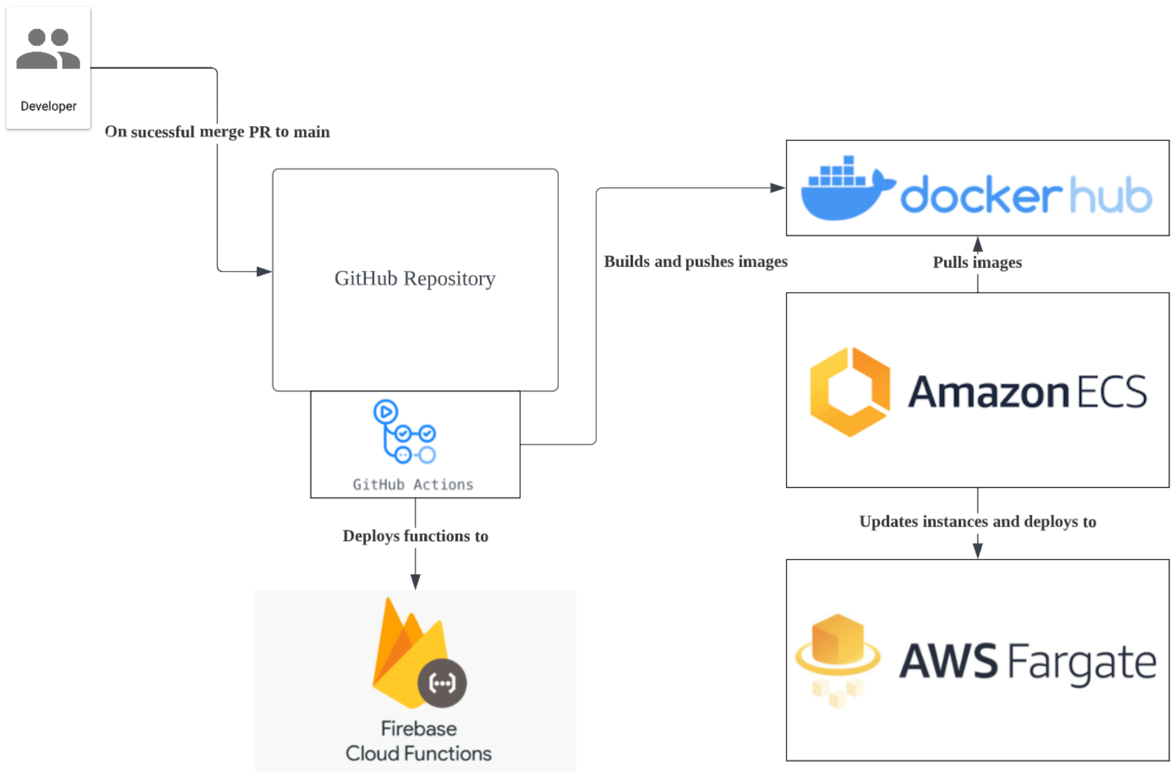
deploy_matching_service	55s
deploy_collab_service	41s
deploy_question_service	40s
deploy_user_service	4m 29s

Continuous Integration and Continuous Deployment is done with Github Actions. Two scripts are written for two workflows, CI and CD respectively.

Continuous Integration

Whenever a Pull Request to the main branch is made, the CI workflow is run, automatically setting up the local Node.js testing environment and installing dependencies for each microservice. The CI workflow then runs the unit test suites for each microservice. This way, whenever a change is made to a feature or a new feature is written, we can ensure that there are no regressions to existing features, as PRs cannot be merged without first passing CI.

Continuous Deployment



If CI passes, and a PR is merged to the main branch, the CD workflow is run (shown above), which runs several Github Actions provided by Docker to build the Docker images for the Question Service, Collaboration Service and Matching Service, which are then pushed to a public image repository hosted on Docker Hub.

AWS Fargate then pulls these updated images from Docker Hub and updates the container instances running on AWS ECS. Through the use of AWS Target Groups, the load balancer then automatically updates the endpoints it targets to the newly created instances.

Through containerization with Docker and orchestration with AWS Fargate, you can see deployment is made easy and convenient.

For the User Service, a Github Action is run to directly deploy the tested Firebase functions (local) to Firebase Functions (production).

Project Management

Software Development Lifecycle

Our team chose to use an Agile SDLC to conduct our project. We conducted sprints which were usually around 2 weeks long where we iterated on our project. A weekly standup was conducted and team members elaborated on what they worked on since the previous week, and updated the Github Issues which serves as our backlog, containing most of the requirements that we wish to deliver for the project.

At the end of every sprint, we would conduct a sprint meeting to discuss what went well and what could be improved, as well as to update our Project Board and Github Issues which served as our backlog. (A Google Doc version of the Sprint Planning that we have done is shown below).

Since the completion of Milestone 1 we had a mostly complete product that served the basic functionality that we wished to provide. Every week since, we iterated on the product and gradually added features to it, adhering to the iterative nature of Agile development.

Requirements

Functional Requirements

FR ID	Component	Feature Description	Priority
FR 1.1	User Service	The system should ensure that every account created has a unique id	Must-have
FR 1.2	User Service	The system should allow users to log into their account via Google Sign in	Must-have
FR 1.3	User Service	The system should allow users to log out of their account	Must-have
FR 1.4	User Service	The system should allow users to save and load their unfinished question attempts	Nice-to-Have
FR 1.5	User Service	The system should allow users to saved finished question attempts	Should-have
FR 2.1	Matching Service	The system should allow users to select the difficulty of the questions they wish to attempt.	Must-have
FR 2.2	Matching Service	The system should be able to match two waiting users with similar difficulty levels and	Must-have

		put them in the same room	
FR 2.3	Matching Service	If there is a valid match, the system should match the users within 30s	Must-have
FR 2.4	Matching Service	The system should inform the users that no match is available if a match cannot be found within 30 seconds	Must-have
FR 2.5	Matching Service	The system should notify users of their waiting status while waiting for a match.	Must-have
FR 2.6	Collaboration Service	The system should provide a means for the user to leave a room once matched	Must-have
FR 2.7	Collaboration Service	The system should allow users to edit their code at the same time and have both editors be in sync	Must-have
FR 2.8	Collaboration Service	The system should allow users in the same room to chat with one another via real time chat messaging	Nice-to-Have
FR 3.1	Dashboard	The system should allow users to see which questions they have attempted before	Nice-to-Have
FR 3.2	Dashboard	The system should allow users to see which questions they have saved before	Nice-to-Have
FR 3.3	Dashboard	The system should allow users to view a breakdown of the questions attempted and saved based on difficulty levels	Nice-to-Have
FR 4.1	User Service	The system should allow users to continue their attempt from previously saved code	Should-have
FR 4.2	Question Service	The system should allow users to obtain specific questions from the question bank	Nice-to-Have
FR 4.3	Question Service	The system should select questions for users from a randomised set of questions for practice	Must-have

Non-functional Requirements

Below are the non-functional requirements that we strived to achieve in the project.

NFR 1.1	The system should be built using a microservice architecture.
NFR 1.2	The response time of the system should be fast. (All user actions should be responded to under 3 seconds)
NFR 1.3	The system should be horizontally scalable.

NFR 1.4	The system should be easy to deploy through containerization.
NFR 1.5	The system should feature high usability in terms of user experience.
NFR 1.6	The system components for each microservice should be deployed using cloud services.

Sprint Planning

Milestone 1

Sprint 1 (Week 5 and Week 6)

Person in charge	Status	FRs / NFRs
Ryan	Launched	FR 2.1
Ryan	Launched	FR 2.2
Ryan	Launched	FR 2.3
Everyone	Launched	NFR 1.1
Everyone	Launched	NFR 1.6
Eu Zin	Launched	FR 2.6
Eu Zin	Launched	FR 2.7
Marcus	Launched	FR 1.1
Marcus	Launched	FR 1.2
Marcus	Launched	FR 1.3

Milestone 2

Sprint 2 (Recess Week and Week 7)

Person in charge	Status	FRs / NFRs
Ryan	Launched	FR 2.4
Ryan	Launched	FR 2.5
Ryan	Launched	NFR 1.4

Person in charge	Status	FRs / NFRs
Eu Zin	Launched	FR 2.8
Eu Zin	Launched	FR 4.1
Eu Zin	Launched	FR 4.3
Marcus	Launched	FR 1.4
Marcus	Launched	FR 1.5
Marcus	Launched	FR 3.3

Sprint 3 (Week 8 and Week 9)

Person in charge	Status	FRs / NFRs
Everyone	Launched	NFR 1.5
Ryan	Shifted to n…	NFR 1.3
Eu Zin	Shifted to n…	FR 4.2
Marcus	Launched	FR 3.1
Marcus	Launched	FR 3.2

Milestone 3

Sprint 4 (Week 10 and Week 11)

Person in charge	Status	FRs / NFRs
Everyone	Launched	NFR 1.6
Everyone	Launched	NFR 1.2
Everyone	Launched	NFR 1.3
Eu Zin	Shift to next…	FR 4.2
Marcus	Launched	NFR 1.5

Final Sprint (Week 12 and 13)

Person in charge	Status	FRs / NFRs
Eu Zin	Launched	FR 4.2

Appendices

Appendix A: Grading nice-to-haves

Development

Nice-to-have outlined in Section 3.4 of PeerPrep Project Description	**Satisfied by which project feature outlined in this document**	**FRs / NFRs**
(Nice to have) Communication service – provides features such as chat, voice, and/or video calling among the participants in the room, once the users have been matched	Chat features in the collaboration-service	FR 2.8
(Nice to have) Front-end (fancy UI) – a reasonably good, user-friendly front-end UI for users to access the application's functionalities.	High usability in UI/UX	NFR 1.5
(Nice to have) Learning pathway/history service – maintains a record of the past attempts e.g., difficulty levels or questions attempted	Question service and user service providing users with a way to track their question history. In this case, we have many nice-to-haves.	FR 3.1 FR 3.2 FR 3.3

Deployment

Nice-to-have outlined in Section 3.4 of PeerPrep Project Description	**Satisfied by which project feature outlined in this document**	**FRs / NFRs**
(Nice to have) Deploying the app to the (local) staging environment (e.g., Docker-based, Docker + Kubernetes).	Docker-based containerization and local deployments of all services using Firebase Emulator and Docker Compose	NFR 1.4
(Nice to have) Scalability – the	Deployment on AWS Fargate with	NFR 1.3

deployed application should demonstrate easy scalability of some form. An example would be using a Kubernetes horizontal pod auto-scaler to scale up the number of application pods when there is a high load.	auto-scaling groups, keeping the system horizontally scalable.	
(Nice to have) The application should have an API gateway of some kind that redirects requests to the relevant microservices. An example would be using an ingress controller such as NGINX ingress controller if using Kubernetes (https://kubernetes.GitHub.io/ingress-nginx/)	Each service has an API Gateway (AWS ELB and Firebase) that redirects requests to the relevant instances in Fargate.	NIL
(Nice to have) Demonstrate effective usage of CI/CD in the project.	Usage of Github Actions to test and deploy each service.	NIL
(Nice to have) Deployment of the app on the production system (AWS/GCP cloud platform)	Deployment on production AWS Fargate for Question, Matching and Collaboration Service Deployment on production Firebase for User Service	NFR 1.6