

CS3219 Group 58 Project Report



LeetWithFriend!

Huang ZhenXin A0205018U
 Liu Yongliang A0214023B
 Marcus Tang Xin Kye A0217934Y
 Ngo Ngoc Phuong Uyen A0219848M

1. Project Introduction	3
1.1 Background	3
1.2 User Stories	3
1.3 Generic User Flow	4
2. Software Requirements	5
2.1 Functional Requirements	5
2.2 Non-Functional Requirements	8
2.3 Service Requirements	10
3. Developer Documentation	15
3.1 Architecture overview	15
3.2 Tech stack	17
3.3 Frontend	17
3.3.1 Authentication	17
3.3.2 Fancy UI	18
3.3.2.1 Dark Mode	18
3.3.2.2 Clear & Vibrant Design	19
3.3.2.3 Question Rendering	19
3.3.2.4 Conducive Interview Preparation Room	19
3.4 User Service (Authentication)	20
3.4.1 Design Considerations	22
3.4.1.1 Handling JWT on the Frontend	22
3.4.1.2 Cater for Persisting User Login	22
3.4.1.3 Backend Storage	23
3.4.1.4 Alternative Token Storage Strategies	23

3.5 Matching Service	23
3.5.1 Design Considerations	23
3.5.1.1 Room ID generation	23
3.5.1.2 Real Time Match Making System using Socket.IO	24
3.5.1.3 Redis Pub/Sub	24
3.6 Room Service	25
3.6.1 Design Considerations	25
3.6.1.1 Communication between users	25
3.6.1.2 Video Call Implementation	26
3.6.1.3 Forward and Backward Navigation of Questions	26
3.6.1.4 Realtime Notification of events in the room	27
3.6.1.5 Redis Pub/Sub	27
3.7 Question Service	28
3.8 Collaboration Service	29
3.9 History Service	30
3.10 Deployment	31
3.10.1 Local Environment	31
3.10.1.1 Native Technology Stack	31
3.10.1.2 Docker-Compose	31
3.10.1.3 Kubernetes IN Docker (KIND)	31
3.10.2 Production Environment	32
3.11 API Gateway	34
3.12 Scalability	35
3.13 Continuous Integration & Delivery	37
4. Future Improvements & Enhancements	40
4.1 User Service	40
4.2 Matching Service	40
4.3 Deployment	40
5. Project Management	40
5.1 Timeline	40
5.2 Development Process	41
5.3 Version control	43
5.4 Role Allocation & Challenges	43
5.5 Individual Contributions	44
6. Reflections & Learning Points	45
6.1 Ensuring Deployment Success	45
6.2 Importance of Clear Communication	45

1. Project Introduction

1.1 Background

Technical interviews play an important part in the hiring process of technical professionals, and virtually all industry roles (such as software developer, data analyst, software tester, and systems engineer among others) require candidates to go through some technical questions.

When applying for jobs and internships, many students find it difficult to tackle technical interviews. Issues range from a **lack of communication skills** to articulate their thought process out loud, to an **inability to understand and solve** the given problem. Moreover, practising for these questions (colloquially called “grinding”) can be **tedious and monotonous**.

To tackle these issues, we are creating an interview preparation platform with a peer matching system called LeetWithFriend, where users can pair up to practise technical interview questions together. The application will also provide functionalities such as question bank, realtime chat, recording learning history and more.

1.2 User Stories

As a student practising for my technical interview, I want...

- (1.2.1) to create an account on the website.
- (1.2.2) to login to the website.
- (1.2.3) to select appropriate difficulty levels and/or question categories to practise.
- (1.2.4) to get matched with students with the same criteria (level of difficulty and question category) as me.
- (1.2.5) to know if my match is successful after requesting it, so that I know if I should change the difficulty or wait longer.
- (1.2.6) to be provided with coding interview questions during the practice.
- (1.2.7) to be able to type out my solutions during the practice.
- (1.2.8) what I wrote to be visible in near-real time to the matched peer, so that we can collaborate over the solution.
- (1.2.9) to communicate with my matched peer over voice and video.
- (1.2.10) to change the question for my peer and I during the session, so that we can try new questions or skip ones that we do not like.
- (1.2.11) to save the questions that we have completed, as well as our answer during the session, for future reference.
- (1.2.12) to be able to end the session.
- (1.2.13) to see all questions the website has to offer.
- (1.2.14) to logout from the website.

1.3 Generic User Flow

1. A student who is keen to prepare for his technical interviews visits the site.
2. He creates an account and then logs in.
3. After logging in, the student selects the question difficulty level he wants to attempt (easy, medium, or hard). He can also select the question category (such as questions about Array, Dynamic Programming, String, etc.)
4. The student then waits until he is matched with another online user who has selected the same difficulty level and/or question category as him.
- 5.1 If he is not successfully matched after 30 seconds, he is prompted to try again or adjust his criteria.
- 5.2 If he is successfully matched, the student and his peer are provided with the question and a free text field in which they can type their solution.
6. This free text field should be updated in near-real time, allowing both the student and his matched peer to collaborate on the provided question.
7. They can also choose to use voice/camera features for communication.
- 8.1. After the students finish working on the question, either student can click on a “Completed” button to save the question and their solution for future reference.
- 8.2 The students can also ask the system to provide a new question, using the original criteria they were matched with.
9. Once they are ready to end the practice, one of them clicks on a “Finish” button which returns each student to the question difficulty selection page.
10. To review the questions he has practised so far, he visits his learning history page.
11. To view all the questions available, he visits the question bank page.
12. Once done, the student logs out.

2. Software Requirements

2.1 Functional Requirements

(Definitions)

US - User Service

MS - Matching Service

CS - Collaboration Service

QS - Question Service

RS - Room Service

HS - History Service

Requirements	Priority	Sprint
US User Service - a Node.js server to handle user authentication and management		
US-FR1 The system allows users to manage their account		
<ul style="list-style-type: none"> US-FR1.1 The system should allow users to create an account with username and password. US-FR1.2 The system should ensure that every account created has a unique username. US-FR1.3 The system should allow users to delete their account. US-FR1.4 The system should allow users to change their password. 	H H H M	1 1 2 2
US-FR2 The system should manage user log in/out		
<ul style="list-style-type: none"> US-FR2.1 The system should allow users to log into their accounts by entering their username and password. US-FR2.2 The system should allow users to log out of their account. US-FR2.3 The system should provide an option for users to persist login across manual browser refresh. 	H H H	1 1 3
US-FR3 The system provides a graphical interface for user profiles		
<ul style="list-style-type: none"> US-FR3.1 The system should provide a central location for users to view and perform user profile changes. US-FR3.2 The system should give users a random profile image. 	H L	2 3
US-FR4 The system should enforce user password strength of minimally 8 characters and must include uppercase and lowercase letters, a number and a special character	H	4
MS Matching Service - a Node.js server to handle user matching		
MS-FR1 - The system should manage user matching		
<ul style="list-style-type: none"> MS-FR1.1 The system should match two waiting users with the same difficulty levels and put them in the same room. MS-FR1.2 The system should match the users within 30s if there are available users to match. 	H H	1 1

<ul style="list-style-type: none"> MS-FR1.3 The system should inform the users that no match is available if a match cannot be found within 30 seconds. MS-FR1.4 The system should allow a user to stop searching for a valid match. MS-FR1.5 The system should prevent users from being matched to more than 1 user at a time. MS-FR1.6 The system should match two waiting users with the same question types and put them in the same room. MS-FR1.7 The system should remove disconnected users from the matchmaking queue. 	H	1
	M	2
	L	3
	L	9
	L	10
MS-FR2 - The system provides a graphical interface for users to view/use		
<ul style="list-style-type: none"> MS-FR2.1 The system should allow users to select the difficulty level of the questions they wish to attempt. MS-FR2.2 The system should display a timer to inform users about the search time. MS-FR2.3 The system should allow users to select the question types together with difficulty they wish to attempt 	H	2
	M	4
	L	9
RS Room Service - a Node.js server to manage the room and communication for matched users		
RS-FR1 - The system allows the users to manage the provision of questions in a room.		
<ul style="list-style-type: none"> RS-FR1.1 The system should allow matched users to get a new question. RS-FR1.2 The system should prevent duplicated questions when users get new questions where possible. RS-FR1.3 The system should allow users to go back to previous questions. 	M	4
	L	6
	L	9
RS-FR2 - The system allows users to manage the room.		
<ul style="list-style-type: none"> RS-FR2.1 The system should provide a means for the user to leave a room once matched and notify both users in the room. RS-FR2.2 The system should provide voice/video chat for the users to communicate. 	H	4
	M	6
QS Question Service - a node.js server to manage the questions		
QS-FR1 - The system allows management of the questions.		
<ul style="list-style-type: none"> QS-FR1.1 The system should store questions indexed by difficulty or type QS-FR1.2 The system should allow admins to create/add question QS-FR1.3 The system should retrieve a question of appropriate difficulty level to the shared workspace(room) once the participants are matched QS-FR1.4 The system should allow updating questions QS-FR1.5 The system should allow deletion questions. QS-FR1.6 The system provides a means to filter and select the questions based on question names, types or difficulty. 	H	2
	H	2
	H	3
	H	4
	M	4
	M	5

QS-FR2 - The system provides a graphical view of questions		
<ul style="list-style-type: none"> • QS-FR2.1 The system should display the question contents. • QS-FR2.2 The system should allow users to view all available questions. • QS-FR2.3 The system should allow users to view any specific question even if they are not in the room. • QS-FR2.4 The system should allow users to type in a search box for the question they are looking for in the question bank. 	H M M L	4 6 7 9
CS Collaboration Service - a service to handle concurrent code editing		
CS-FR1 The system provides a code editor for matched users.		
<ul style="list-style-type: none"> • CS-FR1.1 The system should accept text input from the user for the code editor. • CS-FR1.2 The system should update the code editor to show new changes made by the user / the partner. • CS-FR1.3 The system should show the status of the partner (their cursor and text selection) on this user's editor. • CS-FR1.4 The system allows users to choose the programming language (which enables syntax highlighting) in the code editor. • CS-FR1.5 The system allows users to choose tab size. • CS-FR1.6 The system should provide code completion. • CS-FR1.7 The system adapts the editor theme to dark/light mode. 	H H H M M L L	2 3 3 3 3 4 7
CS-FR2 The system should persist text throughout the room lifetime, e.g. text content should be persisted when user rejoins room after navigating away/Internet connection lost, etc.	H	2
HS History Service - a service to manage users' learning history		
HS-FR1 The system keeps track of the user learning process during sessions		
<ul style="list-style-type: none"> • HS-FR1.1 The system allows either user to mark a question as completed during a session. • HS-FR1.2 The system captures the solution of users when they mark a question as completed. • HS-FR1.3 The system records the timestamp when a question is marked as completed. • HS-FR1.4 The system records the name of the partner who completed this question with the user. • HS-FR1.5 The system should notify the user when they have successfully marked the question as completed. 	H H H H M	7 7 7 7 8
HS-FR2 The system provides a graphical interface to display learning history		
<ul style="list-style-type: none"> • HS-FR2.1 The system allows the user to view their entire learning history, including the name of questions completed, their solutions, 	H	8

the timestamp of completion, as well as the partner whom they solved the question with.		
<ul style="list-style-type: none"> • HS-FR2.2 The system should provide a way for users to navigate to a page with detailed question contents. • HS-FR2.3 The system should allow users to view the learning history in chronological and reverse-chronological order 	M	8
	L	9

Table 1 FR summary

2.2 Non-Functional Requirements

Requirements	Priority
System Requirement	
SR-NFR1 The product must be usable on all major operating systems: Windows, macOS, Linux on a device with a browser.	M
SR-NFR2 The product must support modern browsers: Chrome, Firefox, Safari.	M
Security/Privacy requirements	
S/P-NFR1 User should not be able to access another person's account.	H
S/P-NFR2 Users' passwords should be hashed and salted before storing in the database.	H
S/P-NFR3 Users' authentication tokens should expire automatically after 1 week.	H
S/P-NFR4 Users not belonging to the room should not be allowed to enter it.	M
S/P-NFR5 Non authenticated users should not be allowed to enter the matching process.	L
S/P-NFR6 Users should not be allowed to create, update or delete existing questions.	L
S/P-NFR7 Users should not be able to see another person's learning history.	H
Performance requirements	
P-NFR1 Matchup time when a valid pair is available should be less than 1s.	H
P-NFR2 The delay for code-editor changes (including updates to the text content, cursor movement, and text selection) to apply for both ends should be less than 0.5s	H
P-NFR3 Time of retrieval & selection of questions should be kept under 1s with around 1000 questions stored.	H
P-NFR4 Notification time to notify the other user when a user leaves the room should	H

be less than 1s.	
P-NFR5 Notification and redirection of users to rooms should be less than 1s.	M
P-NFR6 The loading of question content in the frontend should be kept under 1s.	M
P-NFR7 Video call latency between 2 users should be less than 100ms.	L
P-NFR8 The web-page should load in under 4s even with more than 10 concurrent users.	L
Constraints	
C-NFR1 The product is offered as a free online service for users until 12 November 2022.	
Quality Attributes	
QA-NFR1 Performance <ul style="list-style-type: none"> Refer to Performance Requirements above 	H
QA-NFR2 Security <ul style="list-style-type: none"> Refer to Security/Privacy Requirements above 	H
QA-NFR3 Reliability <ul style="list-style-type: none"> The mean time between failures of any services shall be at least 90 days. 	M
QA-NFR4 Availability <ul style="list-style-type: none"> The system should be 95% available on weekdays between 6:00 A.M. and midnight SGT. And at least 99% available on weekdays between 1:00 A.M. and 6 A.M. 	M
QA-NFR5 Robustness <ul style="list-style-type: none"> Code-editor contents are saved automatically to prevent loss when refreshing the page, connection failure or accidentally navigating to a different page. The user can re-enter the room after connection loss or accidentally navigating to a different page. Users who are in the matching queue but then leave the site, are removed and can reinitiate matchmaking when they enter the site again. 	M
QA-NFR6 Scalability <ul style="list-style-type: none"> The website shall be able to handle a page-view growth rate of 10 percent per week for at least one month without user-perceptible performance degradation. 	L
QA-NFR7 Usability <ul style="list-style-type: none"> The average learning time for new users should be less than 3 min. The system notifications must be timely and obvious to provide good feedback. 	L

Table 2 NFR summary

2.3 Service Requirements

The service requirements defined are geared towards providing a high level description of the microservices planned, which follow closely the mentioned FRs in [section 2.1](#) (as well as module project requirements). See [section 3.1](#) for the overall architecture and the design decisions of the microservices.

Microservice Requirement Specification	
User Service	<p>Description</p> <ul style="list-style-type: none"> - The User Service provides an API for creating, authenticating & authorising, and removing users. Other microservices & the frontend can make use of the User Service to validate and retrieve relevant user information for their usage. <p>Capabilities</p> <ul style="list-style-type: none"> - User management <p>Consumer Tasks</p> <ul style="list-style-type: none"> - Frontend web pages that allow users to perform user related services (register, login/logout, amend user information) - Other microservices to retrieve information of a user, and update user record if necessary, to persist core user states <p>Interface</p> <ul style="list-style-type: none"> - Query <ul style="list-style-type: none"> - Get user info - Check username duplication - Commands <ul style="list-style-type: none"> - Register - Login - Logout - Update user info - Delete user account <p>Qualities</p> <ul style="list-style-type: none"> - Security - Availability - Reliability - Scalability <p>Logic/Rules</p> <ul style="list-style-type: none"> - Use JWT to allow for user management across servers/microservices - By default, all users have the role of “user”, which can be adjusted in the future for additional role and access management <p>Data</p> <ul style="list-style-type: none"> - NoSQL data store (MongoDB) to allow for horizontal scaling - User basic information - User profile related information <p>Dependencies</p> <ul style="list-style-type: none"> - Redis

Matching Service	<p>Description</p> <ul style="list-style-type: none"> - The Matching Service matches users via requirements. <p>Capabilities</p> <ul style="list-style-type: none"> - Real time user matching <p>Consumer Tasks</p> <ul style="list-style-type: none"> - Frontend web pages that allow users to perform matching <p>Interface</p> <ul style="list-style-type: none"> - Commands <ul style="list-style-type: none"> - Find Match - Cancel Match - Socket.IO Event Emitted <ul style="list-style-type: none"> - Matching - Match Success - Match Fail - Match Timeout - Match Unavailable (For difficulty/type combinations that do not exist in Question Service) - Socket.IO Event Listened <ul style="list-style-type: none"> - Match Find - Match Cancel - Disconnect <p>Qualities</p> <ul style="list-style-type: none"> - Performance - Availability - Reliability - Scalability <p>Data</p> <ul style="list-style-type: none"> - Users waiting for a match <p>Dependencies</p> <ul style="list-style-type: none"> - Redis - PostgreSQL - Socket.IO
Room Service	<p>Description</p> <ul style="list-style-type: none"> - Stores the state of the room the users are in and provides communication between the users <p>Capabilities</p> <ul style="list-style-type: none"> - Video Communication - Forward and Backward Navigation of current page question <p>Consumer Tasks</p> <ul style="list-style-type: none"> - Display current room to user (with question and editor) - Communicate with other user <p>Interface</p> <ul style="list-style-type: none"> - Query <ul style="list-style-type: none"> - Get room by Id - Get room by userId - Commands

	<ul style="list-style-type: none"> - Next question for room id - Previous question for room id - Create Room - Delete Room - Socket.IO Event Emitted <ul style="list-style-type: none"> - User Connected - Room Update - Room End - Socket.IO Event Listened <ul style="list-style-type: none"> - Join Room - Disconnect <p>Qualities</p> <ul style="list-style-type: none"> - Performance - Availability - Reliability - Scalability - Security <p>Data</p> <ul style="list-style-type: none"> - Room States <p>Dependencies</p> <ul style="list-style-type: none"> - Redis - PostgreSQL - Socket.IO - PeerJS
Question service	<p>Description</p> <ul style="list-style-type: none"> - The Question Service provides an API for adding questions and retrieving questions index by difficulty level and types. The frontend can make use of the Question Service to retrieve question name & content & type. <p>Consumer Task</p> <ul style="list-style-type: none"> - Frontend or other microservices to retrieve questions or any additional information about questions <p>Interface</p> <ul style="list-style-type: none"> - Query <ul style="list-style-type: none"> - Get Question - Get Next Question - Get All Questions - Get Question Types - Get Question Names - Commands <ul style="list-style-type: none"> - Create - Update Question - Delete Question <p>Capabilities</p> <ul style="list-style-type: none"> - Management of questions <p>Qualities</p>

	<ul style="list-style-type: none"> - Performance - Availability - Reliability - Scalability <p>Data</p> <ul style="list-style-type: none"> - PostgreSQL for storage of arrays for question-types - Question basic information (name, content) - Question Categories (types and difficulty) <p>Dependencies</p> <ul style="list-style-type: none"> - PostgreSQL
Collaboration service	<p>Description</p> <ul style="list-style-type: none"> - Provides the mechanism for near real-time collaboration between two matched users to write up their solution. - Specifically, it provides concurrent code editing. <p>Capabilities</p> <ul style="list-style-type: none"> - Concurrent code editing and editor content management <p>Consumer Tasks</p> <ul style="list-style-type: none"> - Frontend service: consumes Collaboration service to update the code editor display. <p>Qualities</p> <ul style="list-style-type: none"> - Availability - Reliability - Scalability - Robustness <p>Data</p> <ul style="list-style-type: none"> - Editor content and cursor position of each user <p>Dependencies</p> <ul style="list-style-type: none"> - Firebase
History service	<p>Description</p> <ul style="list-style-type: none"> - Maintains a record of the user's learning history: the questions they have completed, their partner name, their answers, time completed. <p>Consumer Tasks</p> <ul style="list-style-type: none"> - Frontend service: to retrieve learning history and display on the webpage. - Matching service: to initialise a learning history associated with the room <p>Interface</p> <ul style="list-style-type: none"> - Query <ul style="list-style-type: none"> - Get history for user - Command <ul style="list-style-type: none"> - Initialise history for new room - Add a completed question to the learning history <p>Qualities</p> <ul style="list-style-type: none"> - Availability

	<ul style="list-style-type: none"> - Reliability - Scalability <p>Data</p> <ul style="list-style-type: none"> - History data: question ID, partner name, solution content, timestamp of completion <p>Dependencies</p> <ul style="list-style-type: none"> - MongoDB, Question service
--	---

Table 3 Service requirement summary

3. Developer Documentation

Our code is hosted at this [Github Repo](#). In this section, we explain the overall architecture and the technical details of the microservices developed.

3.1 Architecture overview

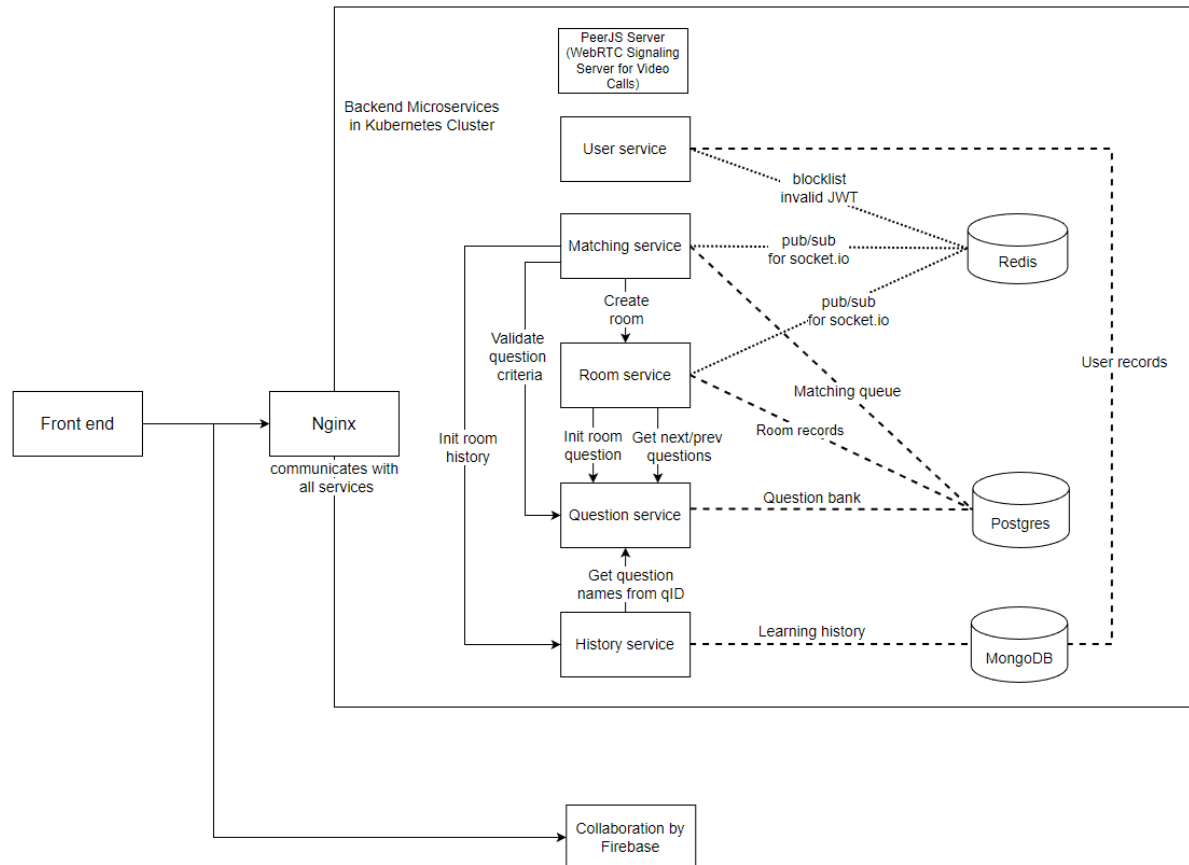


Diagram 1 Architecture overview

We are using the **Microservices** architecture, employing a **Domain Driven Design** to decompose the application into supporting microservices. To identify the services, we decompose by resources – each microservice is responsible for all operations on a given entity in our application. For example, the User Service handles all aspects of managing the user and their account, while the Question Service manages the question bank and individual questions.

To keep dependencies simple and manageable for the ease of development and debugging, we strived to make the services independent of each other and have a one way relationship. The services hence are all able to work with the frontend and have their own database supporting their storage needs. The interactions between the services are minimal as we decided to go for **orchestration** instead of choreography. The frontend, acting as the entry point of our user interaction, also serves as the service that talks to the rest of the services for information. For example, the frontend contacts the user service to acquire user

information, which it relays to the rest of the services when necessary e.g. when the Matching service requires user information to perform matching.

However, as seen in Diagram 1, we do have choreography elements in the interactions between Matching, Room and Question service. This is done for two reasons:

1. The Matching Service was initially set out to perform the matching logic and initialise a room for matched users. In subsequent iterations, with more features added, we recognised that “room” should be regarded as an entity on its own. Thus, the Matching Service was split into two new microservices: a new Matching Service that exclusively handles matching, and a Room Service that oversees a room and its operations, such as fetching next/previous question for the room. This ties in well with the concepts of *Aggregates* and *Bounded Contexts* in Domain Driven Design. As the application evolves and more Aggregates emerge, services are appropriately refactored and decomposed further.
2. On top of the above, we decided to keep this interaction over changing to let the frontend perform multiple round trips across the services just for the creation of a room, for the sake of speed and efficiency.

See [section 2.3](#) for more details about each microservice’s requirements, which include the entity that they manage and their associated operations.

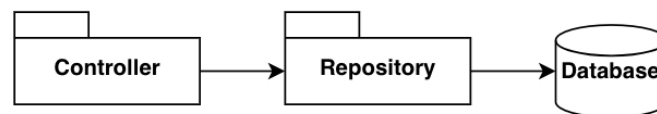


Diagram 2 General architecture of each microservice

In general, we decided to create the backend microservices as a Node.js Express server, having the same structure of controller-repository pattern. This decision is made to facilitate development and cross-review of code implementation across our team, as members are individually responsible for different services but still review and help others when necessary. For the servers, a request first arrives at the controller. Controller performs application logic if any, and invokes the repository to access the database. Repository interaction with the database is through ODM/ORM (particularly Mongoose for MongoDB and Sequelize for PostgreSQL), which isolates domain objects from data objects (consistent with the spirit of domain driven design).

For certain microservices such as the Matching and Room services, there is a Service layer between the Controller and the Repository as well. In such cases, as the business logic is more complicated, it is moved to Service instead of Controller. Controller becomes the gateway between the incoming request and the domain logic – it handles request validation, makes calls to the Service for business logic, and outputs the response. This adheres to the Single-responsibility principle, where Controller is singly responsible for HTTP aspects (parameter validation, response status code, response body, etc.) and Service focuses on serving the logic of the request. In contrast, for microservices such as the User service which have simpler business logic and mostly interact with the Repository directly, we remove the Service layer to avoid pass-through function calls.

The details of internal designs of each microservice, including the technologies used, is explained in the following sections.

3.2 Tech stack

Area	Technology
Frontend	React TailwindCSS Chakra UI Socket.IO Axios charaka-ui-markdown-renderer PeerJS
Backend	Express.js, Node.js Mongoose Axios (Communication between microservices) Sequelize Socket.IO Node Scheduler (Timer for matching) PeerJS
Database	PostgreSQL, MongoDB, Redis
Deployment	GCP Kubernetes Engine
Pub-Sub Messaging	Socket.io
CI/CD	GitHub Actions
Tooling	Eslint, Prettier
Orchestration Service	Docker-Compose, Kubernetes
Project Management	GitHub Issues & milestones
Testing	Jest, Mocha & Chai, Supertest

Table 4 Tech stack

3.3 Frontend

3.3.1 Authentication

Together with our backend user service setup, the frontend client ensures that requests include the appropriate header to send over JWT tokens. Persistent login across browser

refresh is also done on the frontend by implementing an automatic retry policy to refresh the JWT token behind the scene.

3.3.2 Fancy UI

Our frontend application is scaffolded from create-react-app. The state-management is done with native react context and using use-state hooks for local states. To ease the amount of effort required for a consistent and modern user interface design, we use a combination of TailwindCSS framework and Chakra UI component library. The following is a list of other noteworthy points about our frontend UI.

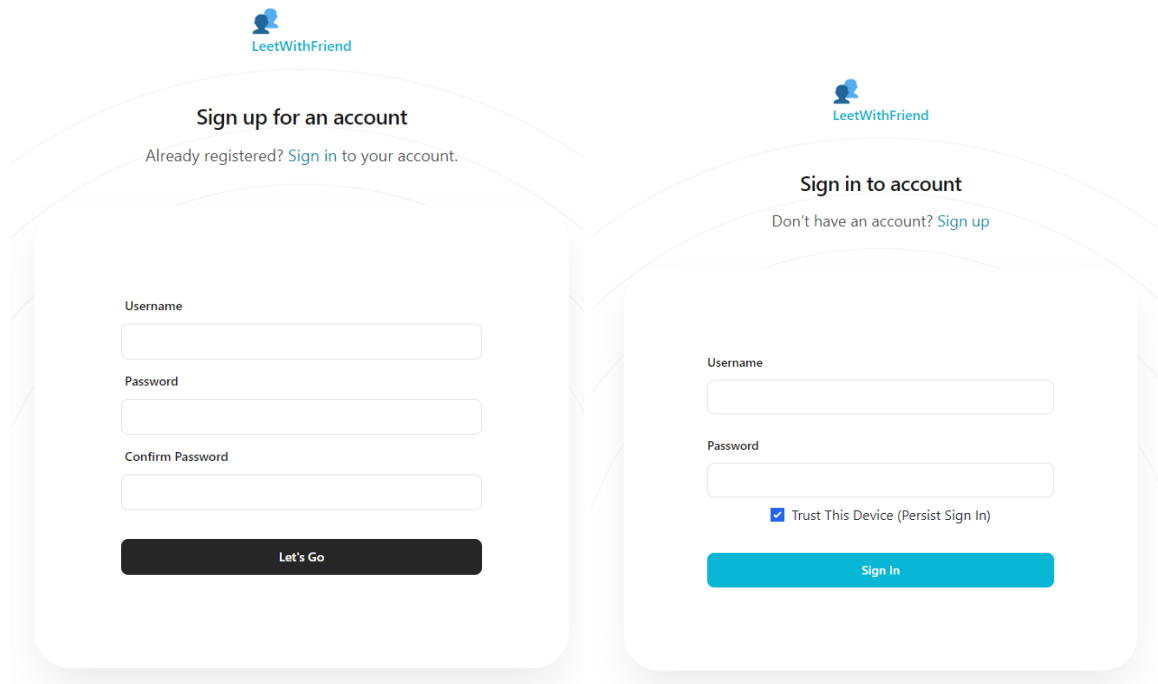


Diagram 3 Signup and login screens

3.3.2.1 Dark Mode

To accommodate different colour preferences, our frontend will apply the system-preference colour mode initially, and support changing of colour mode (light vs dark) on our navigation bar.

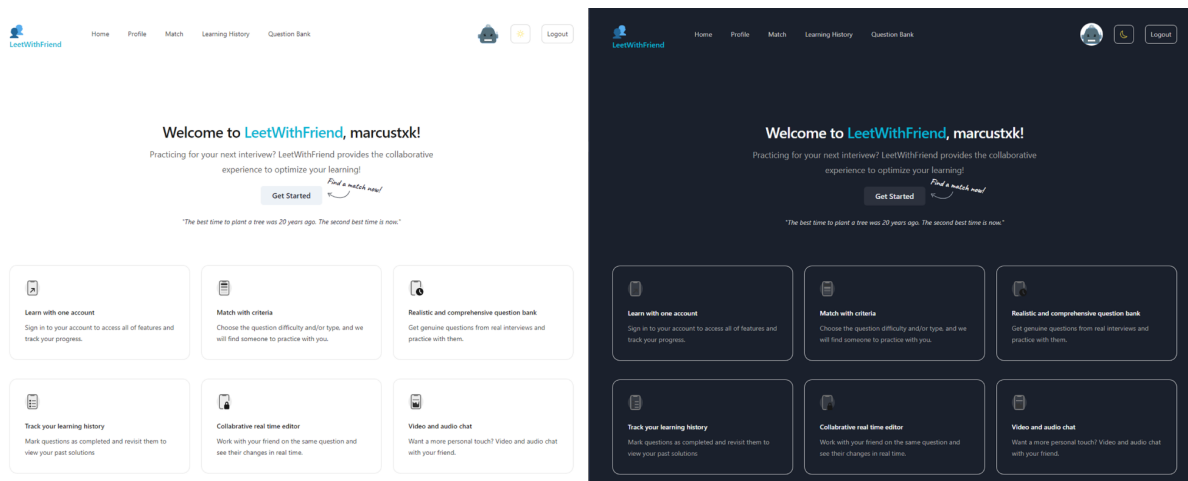


Diagram 4 Light and Dark mode comparisons for home page

3.3.2.2 Clear & Vibrant Design

Our home page includes a Hero Section describing features that our website has to offer them and to encourage users to start a match. Instructions with animations are also included, such as on how to use matching service as well as other features such as keeping track of history.

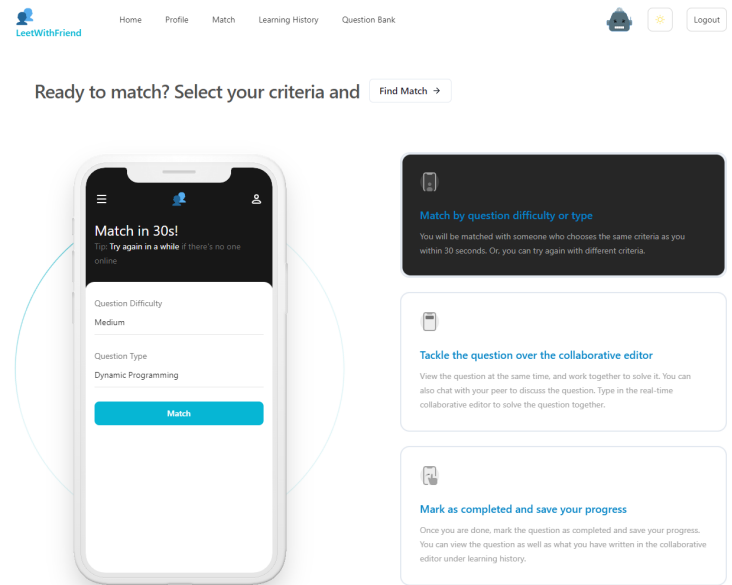


Diagram 5 Animated Phone with instruction for matching

3.3.2.3 Question Rendering

To allow proper rendering of interview questions (such as mathematical symbols and diagrams), we display the question source from Markdown to HTML.

Constraints:

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- **Only one valid answer exists.**

Follow-up: Can you come up with an algorithm that is less than $O(n^2)$ time complexity?

Diagram 6 Mathematical symbols displayed for questions

3.3.2.4 Conducive Interview Preparation Room

To provide a conducive environment for practising interviews, in a room, we have added a Question Pane displaying difficulty and type tags as well as offering forward and backward question navigation. A collaborative editor offering customisable syntax highlighting for multiple popular programming languages and configurable tab sizes for users with different preferences. Video Call functionality is also offered at the bottom right, with toggleable video and audio controls.

The screenshot shows the LeetCode interface for the problem "Number of Steps to Reduce a Number to Zero". The problem is labeled "EASY" and "Bit manipulation". The description states: "Given an integer `num`, return the number of steps to reduce it to zero. In one step, if the current number is even, you have to divide it by 2, otherwise, you have to subtract 1 from it."

Example 1:
Input: `num = 14`
Output: 6
Explanation:
Step 1) 14 is even; divide by 2 and obtain 7.
Step 2) 7 is odd; subtract 1 and obtain 6.
Step 3) 6 is even; divide by 2 and obtain 3.
Step 4) 3 is odd; subtract 1 and obtain 2.
Step 5) 2 is even; divide by 2 and obtain 1.
Step 6) 1 is odd; subtract 1 and obtain 0.

Example 2:
Input: `num = 8`
Output: 4
Explanation:
Step 1) 8 is even; divide by 2 and obtain 4.
Step 2) 4 is even; divide by 2 and obtain 2.
Step 3) 2 is even; divide by 2 and obtain 1.
Step 4) 1 is odd; subtract 1 and obtain 0.

Example 3:
Input: `num = 123`
Output: 12

Constraints:
• $0 \leq \text{num} \leq 10^6$

The right side of the image shows a collaborative editor with a Java solution:

```
1 // This is my answer!!!
2
3 class Solution {
4     public int numberOfSteps (int num) {
5         int ans = 0;
6         for (; num > 0; ans++)
7             if (num % 2 == 1) num--;
8             else num /= 2;
9         return ans;
10    }
11 }
```

At the bottom right, there is a video call overlay showing two participants, one labeled "yongliang".

Diagram 7 Room with Question, Collaborative Editor and Video Call

3.4 User Service (Authentication)

We use JWT to handle user authentication.

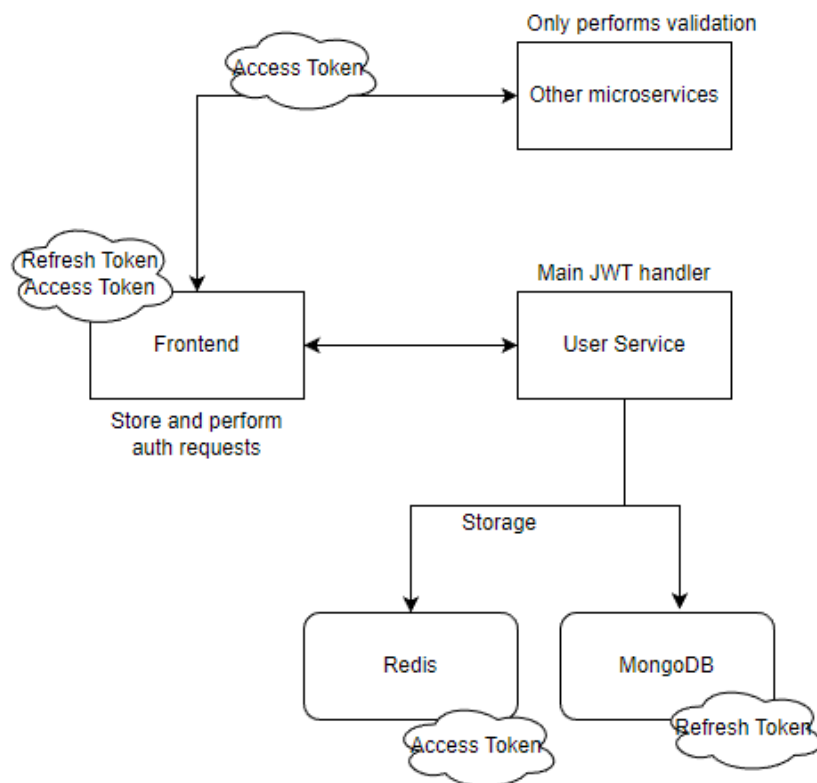


Diagram 8 JWT and service interactions



```
const UserModelSchema = new Schema({
  username: { // Identifier for the user
    type: String,
    required: true,
    minLength: 1,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  email: { // For communication purposes
    type: String,
  },
  school: {
    type: String,
  },
  role: { // For future proofing, to add roles to users
    type: String,
    default: "user",
  },
  refreshToken: {
    type: String,
  },
  createdAt: {
    type: Date,
    immutable: true,
    default: () => Date.now(),
  },
  updatedAt: {
    type: Date,
    default: () => Date.now(),
  },
})
```

Diagram 9 User Model Schema

In the JWT authentication flow, users first register for an account by providing an unique username and a relatively strong password combination. The system hashes the password and stores the user details into the database. When a user wants to login, he/she provides the username-password pair to retrieve two JWT tokens: Access token & Refresh token.

The two tokens work together in slightly different ways to cater for security concerns.

For the Access Token, it is sent via JSON in the response body. The client stores the token in memory (as opposed to storing in localStorage or cookie). The client then uses the token for API access until expiry/browser refresh (which will clear the token in memory). Access

token is sent via HTTP Authorization request header, which will be verified in the backend middleware to check for validity and retrieves the user information encoded within. A new access token is issued when a user issues a refresh request. Once a user logs out, the access token is blacklisted in the redis cache for 60 seconds. This is to ensure that the token loses the access immediately even if it is still valid for a few more seconds.

For the Refresh Token, it is sent via httpOnly cookie by using a `set-cookie` directive. The client stores (and sends automatically) the token in the cookie when making API requests. The token is therefore not accessible via Javascript (e.g. via document.cookie). The refresh token also expires after 7 days such that if the user does not logout, he/she ultimately needs to re-login again. When refreshing the access token, the system also re-issues a refresh token, which means users can indefinitely be logged in if he/she is active within 7 days. The refresh token is stored in the database tagged to the user, hence once a user logs out of the system, the token is no longer valid. When the client wants to logout, the cookie is set to empty to clear the refresh token stored in the client browser.

3.4.1 Design Considerations

3.4.1.1 Handling JWT on the Frontend

Frontend calls the refresh token route immediately on mount. If the backend sees a valid refresh token in the cookie, the backend will send back the JWT access token for use. Else, the frontend redirects users to login.

To deal with access token expiry, frontend encapsulates the request library (axios) to refresh the token behind the scene if the current request fails. This is done by setting a check that if the response is 403 (token expired), the frontend issues a request to refresh and retry the original request. On the user side, no observable effect is shown. An alternative strategy to refresh the authentication tokens is to set a loop to silently refresh the token periodically (whenever it is nearly expired).

3.4.1.2 Cater for Persisting User Login

The use of JWT poses a problem that when a user refreshes a page manually, the user state stored in the context is lost and the website treats the user as an unknown actor, therefore requiring a login again.

To resolve this issue, two mechanisms are introduced:

1. A wrapper function that ensures routes that have to persist login will silently refresh once landed on those pages. This is done by retrieving the accessToken and the user using the existing refresh token.
2. A “persist” option is added in the login page. When signing in, it allows the user to indicate his/her trust in the device to keep the user logged in. This option is remembered via localStorage and will make the user experience smooth for devices that are trusted by the user.

3.4.1.3 Backend Storage

Refresh token is stored alongside the user information in MongoDB, to ensure that at any one time, only one refresh token is valid for the user. This mitigates the risk of refresh token being used indefinitely to refresh itself, as users or the system can invalidate the refresh token immediately if needed.

Access token is checked against a blacklist to ensure that it loses validity once the user is logged out

The amount of user information stored in the database is limited to the core set of user details. Additional details of the user should be stored separately in other microservices, linking back to the user via his/her username. This will ensure that the user-service does not get overwhelmed by further additions of functionalities and data storage requirements. Other microservices can also easily access information that is relevant for them as they are stored alongside them.

3.4.1.4 Alternative Token Storage Strategies

Storage decisions are affected by security concerns (such as XSS and CSRF attack)

1. For Access Token: it is also possible to set it as httpOnly cookie.
2. For Refresh Token: it is also possible to send it as JSON and let the client store it in localStorage.

3.5 Matching Service

The matching service is designed using Socket.IO, allowing clients to find matches via the "match" event. Socket.IO is a library that enables low-latency, bidirectional and event-based communication between a client and a server.

3.5.1 Design Considerations

3.5.1.1 Room ID generation

When 2 users are matched, it is important for the generated room id to be unique as multiple other services depend on it being unique. By studying the source code of how other libraries generated unique IDs, such as Socket.IO, where they use an npm package called [base64id](#) to generate a 20 character 64 bit string. However, base64id is not ideal as it uses the conventionally base64, which includes any illegal symbols in URL, namely = and /, which is not ideal considering the roomId will be used in a URL.

[Nanoid](#) was used instead, which also uses base 64, but with a different set of symbols (A-Za-z0-9_-) instead, which are url safe. The room ID is generated with the following logic: a 7 character randomly generated base64 ID e.g. COBiZQ7, is concatenated with Unix epoch time in ms, which would result in an ~20 character ID (depending on the length of Unix epoch time). An example roomId would be: COBiZQ71665716994899. This prevents roomIds from the future from ever colliding, which is important for match history which requires roomIds to be unique. The chance of collisions is very low, since the only way it

would occur is if both rooms are being created at the exact same millisecond, and with approximately $(1/64)^7$ chance, which is an acceptable probability of collision as it can be considered to be negligible.

3.5.1.2 Real Time Match Making System using Socket.IO

To implement a real time matchmaking system we utilised Websockets connections, made possible via the library Socket.IO. The match-making process is as follows:

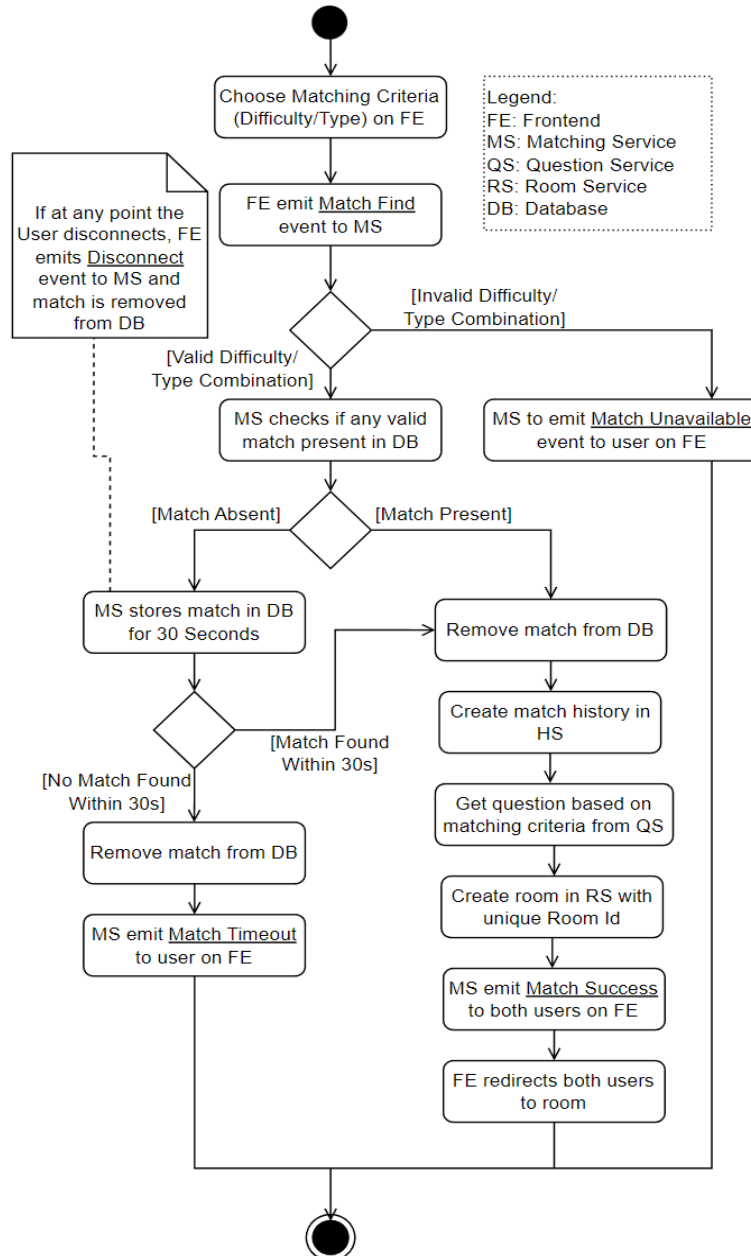


Diagram 10 Activity Diagram for Match Making Process

3.5.1.3 Redis Pub/Sub

Messages sent from the frontend can be received by different Socket.IO servers across multiple Pods in the production Kubernetes cluster (see [section 3.10](#) for details on

deployment), such as after an horizontal pod autoscaling event. The users might then experience communication issues as the servers are unaware of each other.

To resolve this issue, we ensure that the sessions are “sticky” and the messages are available to all Socket.IO servers with the help of the Redis Pub/Sub mechanism. Through a Redis Adapter, after a packet is sent to all matching clients connected to the current server, it is then published in a Redis channel, which allows it to be received by the other Socket.IO servers in the cluster which are subscribed to the channel.

3.6 Room Service

The room service’s main responsibility is to manage active rooms between users. It keeps track of the users and questions in the room, as well as facilitating communication between the users in the room. It is also responsible for notifying both users when the room ends.

3.6.1 Design Considerations

3.6.1.1 Communication between users

For communication between 2 users, the team had to decide between Video calls and Chat messaging.

Video Calls

Benefits	Demerits
Faster communication between two users.	Potentially harder to implement
More natural communication between two users, simulating a face to face conversation.	
Better mimics an online technical interview environment and thus enables the users to better prepare for it.	

Table 5 Video call analysis

Chat messaging

Benefits	Demerits
Potentially easier to implement	Does not mimic an online technical interview environment and thus does not enable the users to better prepare for it.
	Users could already potentially communicate via text in the collaborative editor.

Table 6 Chat messaging analysis

After weighing the benefits and demerits between the two, our team decided on using Video Calls.

3.6.1.2 Video Call Implementation

The Video communication was set up using two technologies, WebRTC and sockets. This was done by leveraging [PeerJS](#), a simple peer-to-peer WebRTC library that wraps the browser's WebRTC implementation to provide a complete, configurable, and easy-to-use peer-to-peer connection API.

In order to facilitate the peer-to-peer connection, a signalling server is required. A WebRTC signalling server is a server that manages the connections between devices. While there are online free signalling servers, such as the [PeerServer Cloud service](#), we chose to host our own signalling server, separating the production and localhost servers, as this would ensure no collisions of IDs for localhost testing and production, since users utilise userIDs to call, which can be the same across localhost and production. Managing our server will also allow us to ensure uptime, reducing dependency on external providers and manage any issues with scaling.

When a user first joins the call, a call will be initiated to the other user ID (with a "call" event) via a WebRTC peer-to-peer video call, sending a user's `usermedia` stream to the other party, where the other party can choose to answer it with their own usermedia stream, emitting a "stream" event. Upon receiving the stream event, the user can then show the video and audio of the other user in their browser. Toggling of video and audio input can be done by the user and modifies the media stream being streamed to the other party.

3.6.1.3 Forward and Backward Navigation of Questions

Room Service keeps track of questions that have been seen by the users in the room. This allows users to easily revisit previous questions such as if they accidentally click to the next question. This is done via storing an array of question IDs in the database (as questionIds) as well as a pointer for the current question, which allows a simple update of the pointer when users navigate forward.

Keeping track of all viewed questions in the room by the users also serves another important purpose. It avoids showing users duplicated questions for the difficulty and/or type that they have chosen during the matching phase. This is done by passing the IDs of the completed questions, the difficulty as well as the type through an API call via `axios` to the Question Service, where a question ID that is not viewed by users in the current room is returned. Only once all possible questions for the chosen difficulty and/or type have been viewed, the questions will start to repeat.



Diagram 11 Room Model Schema

3.6.1.4 Realtime Notification of events in the room

When a user leaves the browser page, terminates the room or updates the current question, the other user in the room needs to be notified. This was done with websockets via Socket.IO.

We chose to differentiate the handling of the events for terminating the room and leaving the browser page. This is to allow for fault tolerance, such as due to a user's intermittent connection issues or accidental closing of the webpage. A user leaving the room will be reflected to the other user as the video call feed of the disconnected user will be hidden. A room will continue indefinitely until either one of the users clicks the "End Session" button to terminate the session. Should the user try to initiate another match while still being in an existing room, they will not be allowed to do so, instead, they will be redirected to the room they are already in.

3.6.1.5 Redis Pub/Sub

This was implemented in an identical way to Matching Service, mentioned in [section 3.5.1.3](#), with the same rationale of allowing Socket.IO servers in a cluster to communicate with one another.

3.7 Question Service

The question service is responsible for storing the questions.

In terms of schema design, there are two options:

1. Store all attributes of a question in one table.
2. Splitting the question content and its category information into two tables.

Considering extensibility, we have chosen option 2 since in the future there could be more matching criteria implemented and there might be more ways to categorise a question. It would be better to separate the question content and its category information. The contents of questions are stored as Markdown and converted to HTML in the frontend.



Diagram 12 Category Model Schema



Diagram 13 Question Model Schema

3.8 Collaboration Service

The algorithm behind collaborative editing is [Operational transformation](#) (OT). Given a number of concurrent editing operations, we have to sequentially apply them to the document – OT is used to adjust the parameters of one operation to account for other concurrent operations that have been executed before it. The purpose of this transformation is to achieve the correct effect and maintain document consistency.

We have two options:

1. Use an editor package that comes with support for collaborative editing.
2. Use an editor without collaborative editing, and implement OT on our own backend.

However, OT is not trivial to implement, hence we decided to follow option (1).

A good collaborative editor candidate is [Firepad](#). It is a component that combines the [CodeMirror](#) text editor with Firebase, to support OT and collaborative editing. Firepad is used to build CoderPad, a popular live coding platform, and thus it is a reliable choice. However, the official Firepad is not actively maintained anymore (frozen ~2 years). Thus we decided to use a fork of Firepad (<https://github.com/lucafabbian/firepad>) since it is being maintained by the fork owner.

Additionally, we implemented cursor synchronisation between the users as this feature is not provided by Firepad. We also leverage Firebase to update the cursor data of each user.

Whenever user A's cursor changes, his cursor position is pushed to Firebase. User B's editor attaches a listener to the Firebase document, and on detecting a change it renders a new cursor to represent A.

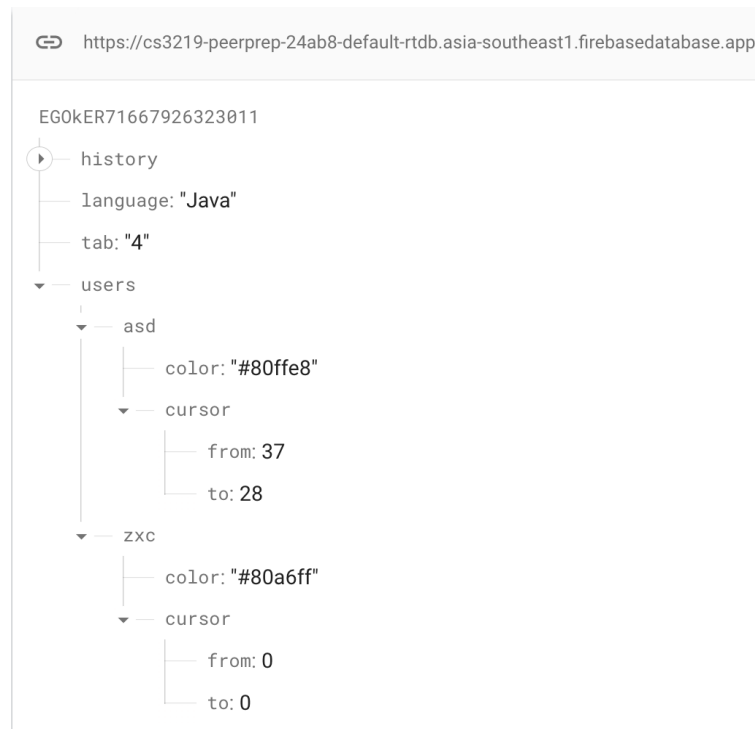


Diagram 14 Firebase schema for collaborative editor

Each room's editor is represented as one document on the Firebase Realtime database, and the document name is the room ID. Within, there is a "history" child document managed by Firepad, to store the text contents of the editor. To satisfy CS-FR1.2 and CS-FR1.3 which is to synchronise user cursor, a child document "users" contains information about two users and their cursor data. To satisfy CS-FR1.4 and CS-FR1.5 which is for choosing language and tab size, attributes "language" and "tab" are added under the parent document.

3.9 History Service

The History service stores the learning history of users, and its query interface needs to provide the questions completed, the partner name, the solution, and the time completed for a given user.

As can be seen in Diagram 15, our schema for history data is "room" oriented, i.e. the questions completed and their details are stored under each "room" document. However, since the query interface is "user" oriented, we need to transform the data to aggregate all questions completed by the given user across the rooms. This operation is done in the Service layer of History Service.



Diagram 15 History Model Schema

3.10 Deployment

3.10.1 Local Environment

To view a local version of the application (with all the services running), we provide three possible approaches.

3.10.1.1 Native Technology Stack

The services can run natively on Windows or macOS machines. This will require installation of dependencies such as those from NPM & Databases (PostgreSQL, MongoDB, Redis etc). To get help with environment setup, individual services have their own Dockerfile to allow for quick cross-platform development. Each service also contains an individual README that includes an installation guide.

3.10.1.2 Docker-Compose

A docker-compose script has been created to easily spin up all required services. Some of the databases in use will be seeded to insert mock data for manual testing and interaction with the application. Running the entire application will be as easy as running the following command: `docker-compose -f docker-compose.local.yml up --build -d`.

3.10.1.3 Kubernetes IN Docker (KIND)

As the production deployment is done via Kubernetes, we also provide instructions on how to run the application locally in a KIND cluster. This helps with debugging and testing the full application in an almost similar environment to production as we apply the Kubernetes configurations locally. The setup process for KIND is more involved but it generally includes the following steps:

- Create a local KIND cluster

- Setup Kubernetes ingress and secrets
- Apply Kubernetes configurations

3.10.2 Production Environment

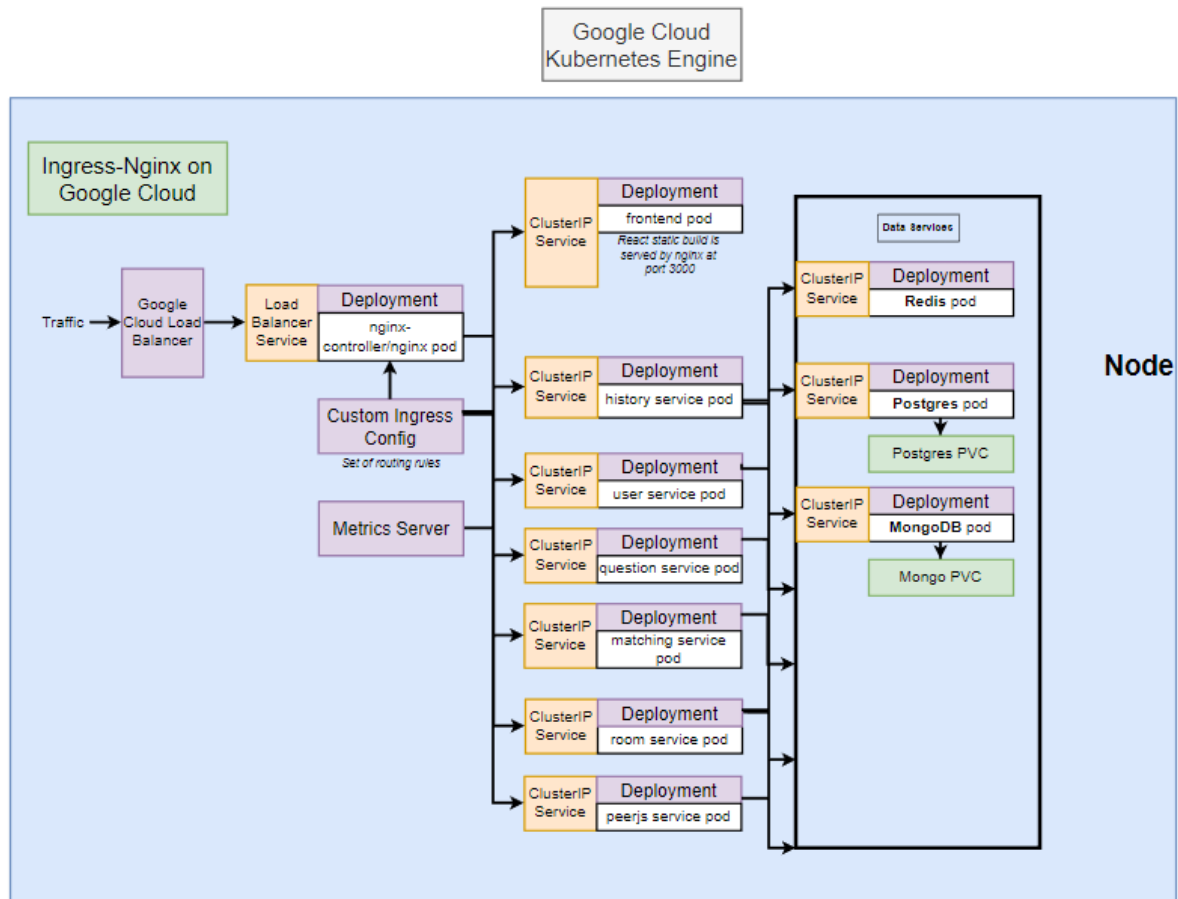


Diagram 16 Overview of our production structure

We decided to use the GCP cloud platform for the deployment of our complete application. The application will run in a Kubernetes cluster in GCP Kubernetes Engine. It will be accessible to the public via our purchased domain with HTTPS for better security.

Some one-time initial setups are required to create the production environment. They are specified here for developer reference.

1. Go to Google Cloud console and create a new project
2. Enable billing
3. Create new cluster
 - a. Choose zone near the region (asia-southeast1-a)
 - b. Enter name of cluster to be (cs3219-cluster), this is referenced in the script hence needs to be exactly the same
 - c. Connect to the cluster via Google Cloud Shell and run the required commands in `k8s/init.sh`. Namely:
 - i. Authorise yourself
 - ii. Create production secrets

- iii. Install Helm (to help with installing dependencies)
- iv. Install ingress-nginx

OVERVIEW

OBSERVABILITY

COST OPTIMIZATION

Filter

Enter property name or value

<input type="checkbox"/>	Status	Name ↑	Location	Number of nodes	Total vCPUs	Total memory
<input type="checkbox"/>		cs3219-cluster	asia-southeast1-a	3	6	12 GB

OVERVIEW

COST OPTIMIZATION

Filter

Is system object : False

Filter workloads

<input type="checkbox"/>	Name ↑	Status	Type	Pods	Namespace	Cluster	Pods Running
<input type="checkbox"/>	cert-manager	OK	Deployment	1/1	cert-manager	cs3219-cluster	1
<input type="checkbox"/>	cert-manager-cainjector	OK	Deployment	1/1	cert-manager	cs3219-cluster	1
<input type="checkbox"/>	cert-manager-webhook	OK	Deployment	1/1	cert-manager	cs3219-cluster	1
<input type="checkbox"/>	frontend-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	history-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	matching-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	mongo-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	my-release-ingress-nginx-controller	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	peerjs-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	postgres-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	question-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	redis-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	room-deployment	OK	Deployment	1/1	default	cs3219-cluster	1
<input type="checkbox"/>	user-deployment	OK	Deployment	1/1	default	cs3219-cluster	1

Diagram 17 Google Cloud Admin Panel

To populate the cluster with our services, we define an array of declarative Kubernetes configurations to specify the production cluster components. They are:

- Deployments: individual microservice will be deployed to one or more pods, while databases will be allocated to a single pod each.
- ClusterIP: IP address is allocated to the deployments to allow in-cluster communication
- Persistent Volume Claim: for PostgreSQL and MongoDB that require long lasting data storage across pod migration, they have been assigned with GCP block storage of 1Gi each.
- Ingress: the custom ingress config defines the rules of traffic routing after the Google Cloud load balancer brings in the request
- Issuer & Cert manager: together with the domain purchased, we can secure our website with digital certificates and SSL encryption

Details of the configurations are available in our code base under the `k8s` folder.

3.11 API Gateway

In our Kubernetes cluster, we implemented an API gateway to redirect requests to the relevant microservices. This is achieved with an NGINX ingress controller to route traffic to the respective microservices. For the deployment objects, we need to specify Cluster IP respectively such that the service can be reachable at their endpoints.

SERVICES

INGRESS

Services are sets of Pods with a network endpoint that can be used for discovery and load balancing. Ingresses are collections of rules for routing external HTTP(S) traffic to Services.

Filter

Is system object : False

Filter services and ingresses

<input type="checkbox"/>	Name ↑	Status	Type	Endpoints	Pods	Namespace	Clusters
<input type="checkbox"/>	cert-manager	✓ OK	Cluster IP	10.0.8.236	1/1	cert-manager	cs3219-cluster
<input type="checkbox"/>	cert-manager-webhook	✓ OK	Cluster IP	10.0.10.65	1/1	cert-manager	cs3219-cluster
<input type="checkbox"/>	frontend-cluster-ip-service	✓ OK	Cluster IP	10.0.4.123	1/1	default	cs3219-cluster
<input type="checkbox"/>	history-cluster-ip-service	✓ OK	Cluster IP	10.0.11.147	1/1	default	cs3219-cluster
<input type="checkbox"/>	matching-cluster-ip-service	✓ OK	Cluster IP	10.0.3.98	1/1	default	cs3219-cluster
<input type="checkbox"/>	mongo-cluster-ip-service	✓ OK	Cluster IP	10.0.4.223	1/1	default	cs3219-cluster
<input type="checkbox"/>	my-release-ingress-nginx-controller	✓ OK	External load balancer	34.143.186.41:80 34.143.186.41:443	1/1	default	cs3219-cluster
<input type="checkbox"/>	my-release-ingress-nginx-controller-admission	✓ OK	Cluster IP	10.0.6.112	1/1	default	cs3219-cluster
<input type="checkbox"/>	peerjs-cluster-ip-service	✓ OK	Cluster IP	10.0.2.201	1/1	default	cs3219-cluster
<input type="checkbox"/>	postgres-cluster-ip-service	✓ OK	Cluster IP	10.0.15.59	1/1	default	cs3219-cluster
<input type="checkbox"/>	question-cluster-ip-service	✓ OK	Cluster IP	10.0.1.146	1/1	default	cs3219-cluster
<input type="checkbox"/>	redis-cluster-ip-service	✓ OK	Cluster IP	10.0.11.7	1/1	default	cs3219-cluster
<input type="checkbox"/>	room-cluster-ip-service	✓ OK	Cluster IP	10.0.2.203	1/1	default	cs3219-cluster
<input type="checkbox"/>	user-cluster-ip-service	✓ OK	Cluster IP	10.0.1.52	1/1	default	cs3219-cluster

Diagram 18 a list of cluster IP objects

The current set of routing rules can be seen as follows:

```

paths:
- path: /?(.*)
  pathType: Prefix
  backend:
    service:
      name: frontend-cluster-ip-service
      port:
        number: 3000
- path: /api/v1/user?(.*)
  pathType: Prefix
  backend:
    service:
      name: user-cluster-ip-service
      port:
        number: 8000
- path: /api/v1/matching?(.*)
  pathType: Prefix
  backend:
    service:
      name: matching-cluster-ip-service
      port:
        number: 8001
- path: /api/v1/question?(.*)
  pathType: Prefix
  backend:
    service:
      name: question-cluster-ip-service
      port:
        number: 8500
- path: /socket.io/room?(.*)
  pathType: Prefix
  backend:
    service:
      name: room-cluster-ip-service
      port:
        number: 8022
- path: /socket.io/matching?(.*)
  pathType: Prefix
  backend:
    service:
      name: matching-cluster-ip-service
      port:
        number: 8001
- path: /api/v1/room?(.*)
  pathType: Prefix
  backend:
    service:
      name: room-cluster-ip-service
      port:
        number: 8022
- path: /api/v1/history?(.*)
  pathType: Prefix
  backend:
    service:
      name: history-cluster-ip-service
      port:
        number: 8080
- path: /myapp/peerjs?(.*)
  pathType: Prefix
  backend:
    service:
      name: peerjs-cluster-ip-service
      port:
        number: 9000

```

Diagram 19 Routing rules

3.12 Scalability

To achieve scalability, we use Kubernetes horizontal pod auto-scaler to scale up the number of application pods when there is a high load. We first need to deploy a metrics server that collects information about the utilisation rate of each deployment. With the information available, we then follow by specifying the CPU and memory request of each deployment. This estimates the amount of resources required for the deployment to run comfortably. The specified requests also help the Kubernetes schedulers to efficiently manage the pods.

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-hpa
  namespace: default
spec:
  metrics:
    - resource:
        name: cpu
        target:
          averageUtilization: 80
          type: Utilization
        type: Resource
  minReplicas: 1
  maxReplicas: 5
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: frontend-deployment

```

Diagram 20 HPA config for frontend

With the resource requests defined, we can utilise a Kubernetes object called “HorizontalPodAutoscaler”. We define for each deployment that we want to scale, a minimum and a maximum number of replicas (see Diagram 19 above). It will then automatically scale out the deployment should the resource metrics exceed the target. Should the load be lower than expected, the deployments will be scaled down to a minimum of 1 replica.

```

Name: room-hpa
Namespace: default
Labels: <none>
Annotations: <none>
CreationTimestamp: Wed, 02 Nov 2022 05:46:03 +0000
Reference: Deployment/room-deployment
Metrics:
  resource cpu on pods (as a percentage of request): 0% (0) / 80%
Min replicas: 1
Max replicas: 5
Deployment pods: 5 current / 5 desired
Conditions:
  Type          Status  Reason              Message
  ----          -
  AbleToScale   True    ScaleDownStabilized recent recommendations were higher than current one, applying the highest recent recommendation
  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
  ----          -
  Warning       FailedGetResourceMetric 7m4s (x2 over 7m19s) horizontal-pod-autoscaler no recommendation
  Warning       FailedGetResourceMetric 5m19s (x7 over 6m49s) horizontal-pod-autoscaler missing request for cpu
  Normal       SuccessfulRescale       5m4s             horizontal-pod-autoscaler New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal       SuccessfulRescale       4m49s            horizontal-pod-autoscaler New size: 5; reason: cpu resource utilization (percentage of request) above target

Name: user-hpa
Namespace: default
Labels: <none>

```

Diagram 21 scaling events for overload

```

Name: room-hpa
Namespace: default
Labels: <none>
Annotations: <none>
CreationTimestamp: Wed, 02 Nov 2022 05:46:03 +0000
Reference: Deployment/room-deployment
Metrics:
  resource cpu on pods (as a percentage of request): 0% (0) / 80%
Min replicas: 1
Max replicas: 5
Deployment pods: 1 current / 1 desired
Conditions:
  Type            Status  Reason                        Message
  ----            -
AbleToScale      True    ReadyForNewScale             recommended size matches current size
ScalingActive    True    ValidMetricFound             the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
ScalingLimited   True    TooFewReplicas               the desired replica count is less than the minimum replica count

Events:
  Type    Reason                  Age    From                      Message
  ----    -
Warning  FailedGetResourceMetric  9m50s  horizontal-pod-autoscaler  no recommendation
Warning  FailedGetResourceMetric  8m5s   horizontal-pod-autoscaler  missing request for cpu
Normal   SuccessfulRescale       7m50s  horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
Normal   SuccessfulRescale       7m35s  horizontal-pod-autoscaler  New size: 5; reason: cpu resource utilization (percentage of request) above target
Normal   SuccessfulRescale       2m34s  horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) below target

```

Diagram 22 scaling events for underload

3.13 Continuous Integration & Delivery

We use GitHub Actions to automatically run linting/testing whenever a new PR is made against the main branch. For continuous delivery, our Actions script will be triggered whenever there's a push to the production branch to rebuild the docker images and reapply the kubernetes configs to our production environment.

The screenshot shows the GitHub Actions interface. On the left, under 'Actions', there is a list of workflows: 'Deploy RepoSense report', 'Deploy To Production', 'Frontend', 'History Service Test', 'Matching Service Test', 'pages-build-deployment', 'Production Cleanup', 'Question Service Test', 'Room Service Test', and 'User Service Test'. The 'All workflows' tab is selected. On the right, under 'All workflows', it says 'Showing runs from all workflows'. Below this, there is a section '1,487 workflow runs'. Four recent runs are listed, all with a green checkmark indicating success. Each run is titled 'Fix production issues (#183)' and includes a commit hash and the user 'tlylt'. The runs are for the 'main' branch.

Diagram 23 Workflow overview

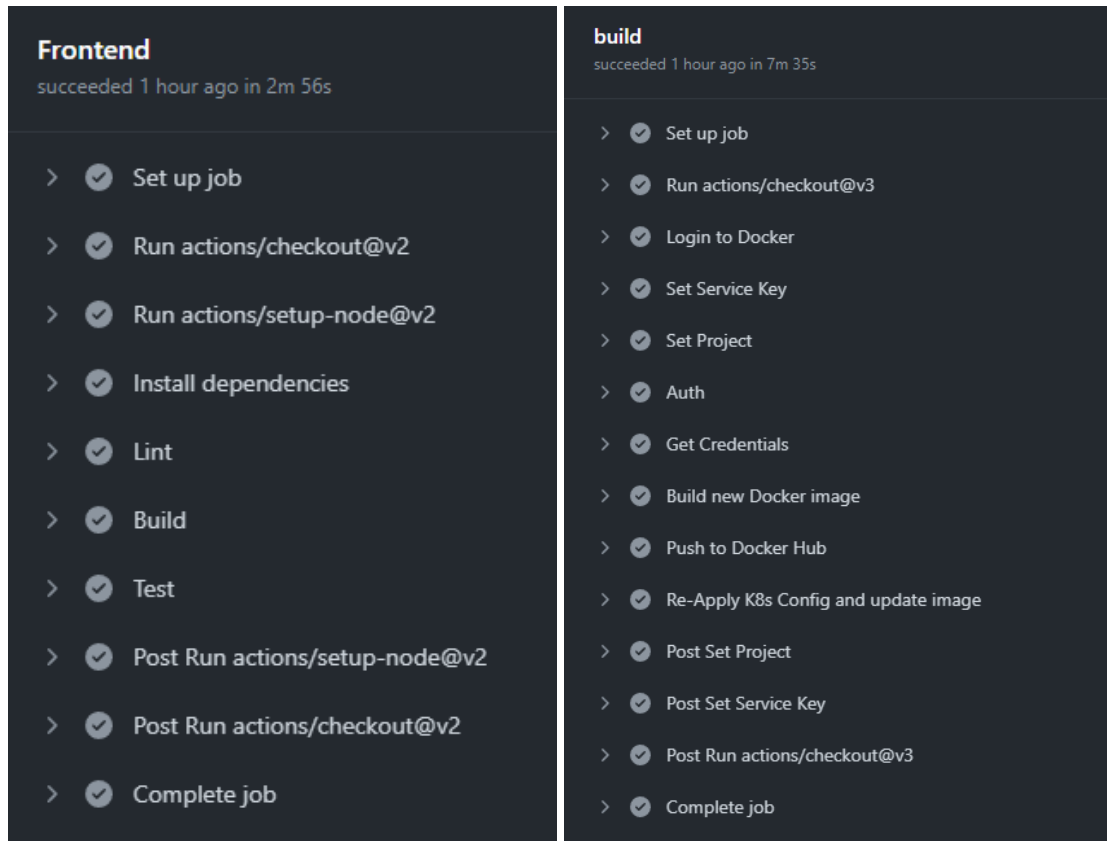


Diagram 24 Example workflow runs for frontend and pushing to production

For each microservice, we decided to write tests to ensure that the services work as expected, when functioning in isolation. This is mostly achieved via Mocha and Chai to create endpoint related tests.

With the tests in place, we verify on each PR the correctness of the code by automatically running the following via GitHub Actions:

1. Linting and formatting via ESLint & Prettier
2. Building and testing via Docker-compose

When the main branch is stable, we push the changes to the production branch and trigger the following events:

- Rebuilding the Docker images for each service: as the code change may affect application functionality, docker images for each component will be rebuilt such that the latest changes are applied.
- Tag and push the images to Docker Hub: after we have the images, we will be tagging them with the git commit hash (git SHA) before publishing them on Docker Hub. This is because in the later steps we will configure the cluster to fetch and apply these new images.
- Reapply all kubernetes configurations: this is done to ensure that any changes to the configuration files are applied on the cluster.
- Assert that the deployments use the latest tagged images: with some imperative kubectl commands, we ensure that the cluster fetches the latest images defined by the commit hash and apply them accordingly. This is done because kubernetes would not apply the deployment configurations if the image tag is not changed.

Currently, the deployment will still have to be manually triggered by making a PR/Push from our stable main branch to the production branch. We chose continuous delivery over continuous deployment as we feel that the application will benefit from some manual testing before pushing a new version out, to compensate for the lack of end to end tests.

4. Future Improvements & Enhancements

4.1 User Service

For a full fledged user management system, we could consider implementing some of the additional functionalities such as:

- Forget password
- Custom user profile upload
- Etc

To increase the ease of adoption, we could also implement Oauth integrations for social accounts such as Google/Facebook/GitHub login. This allows users to use our services without the need to create a new account

4.2 Matching Service

A more powerful matching system, leveraging on user attributes instead of just question attributes, such as by school or by username (to allow guaranteed matching to another user of your choice).

4.3 Deployment

To help secure the production system from accidental changes by developers, we could consider creating a staging deployment of all the microservices, such that it creates an offline application that can be used for testing, without touching the production data.

5. Project Management

5.1 Timeline

We use GitHub milestones and the project board to keep track of our TODOs and project backlog. Please refer to the [dashboard view](#) for a full record. (Note: you will need to be part of the GitHub organisation for the module CS3219-AY2223S1 in order to view it. We are

unable to change the visibility of the board)

G58 Project Board									
Backlog		By priority		Timeline		+ New view			
Milestone		Title	Assignees	Status	Linked pull requests	Labels			
1 v1.0	-	⌚ [User-Service] Setup Linting and prettier format #5	👤 tlylt	🟢 Done					
2 v1.0	-	⌚ Setup Login/Signup/Home/404 pages on the frontend #6		🟢 Done	🔗 #4				
3 v1.0	-	🔗 Setup frontend for user service & refactor JWT auth #4		🟢 Done					
4 v1.0	-	⌚ Refactor JWT handling logic #7		🟢 Done	🔗 #4	Frontend User Service			
5 v1.0	-	⌚ Setup socket.io #13	👤 MarcusTXX	🟢 Done	🔗 #11	Matching Service			
6 v1.0	-	⌚ Create Match Model and ORM #12	👤 MarcusTXX	🟢 Done	🔗 #11	Matching Service			
7 v1.0	-	⌚ Set up code linting #14	👤 MarcusTXX	🟢 Done	🔗 #11	Matching Service			
8 v1.0	-	⌚ Dockerize matching service #15	👤 MarcusTXX	🟢 Done	🔗 #11	Matching Service			
9 v1.0	-	🔗 Add core components for matching service #11	👤 MarcusTXX	🟢 Done		Matching Service			
10 v1.1	-	⌚ Improve frontend login/signup page design #8	👤 tlylt	🟢 Done	🔗 #42	Frontend			
11 v1.1	-	⌚ Dockerize Frontend #9		🟢 Done	🔗 #10	Frontend			
12 v1.1	-	🔗 Configure CI and docker for frontend #10		🟢 Done					
13 v1.1	-	⌚ Add matching logic based on difficulty #16	👤 MarcusTXX	🟢 Done	🔗 #11	Matching Service			
14 v1.1	-	⌚ Add validations for payload #19	👤 MarcusTXX	🟢 Done	🔗 #49	Matching Service			

Diagram 25 Project Board Timeline

5.2 Development Process

The team will follow a modified version of the iterative Scrum workflow.

We strive to make incremental updates to each service such that the product can be iteratively improved upon. One example of such a process is how our question service was first created to supply a fixed list of questions when queried and later on incorporated the required functionalities of filtering and getting randomised questions.

We will have weekly meetings every Thursday to plan for the goals for the upcoming weekly sprint (From Thursday to the next Thursday) and will draft them in a GitHub issue. During the meeting, meeting minutes are to be taken so that relevant discussions can be duly noted. The minutes are available for viewing at [58-MeetingMinutes](#).

GitHub issues/PRs will be used and tagged under weekly GitHub milestones, serving as a TODO list. The issues can also be viewed via the kanban board in the GitHub project management tab.

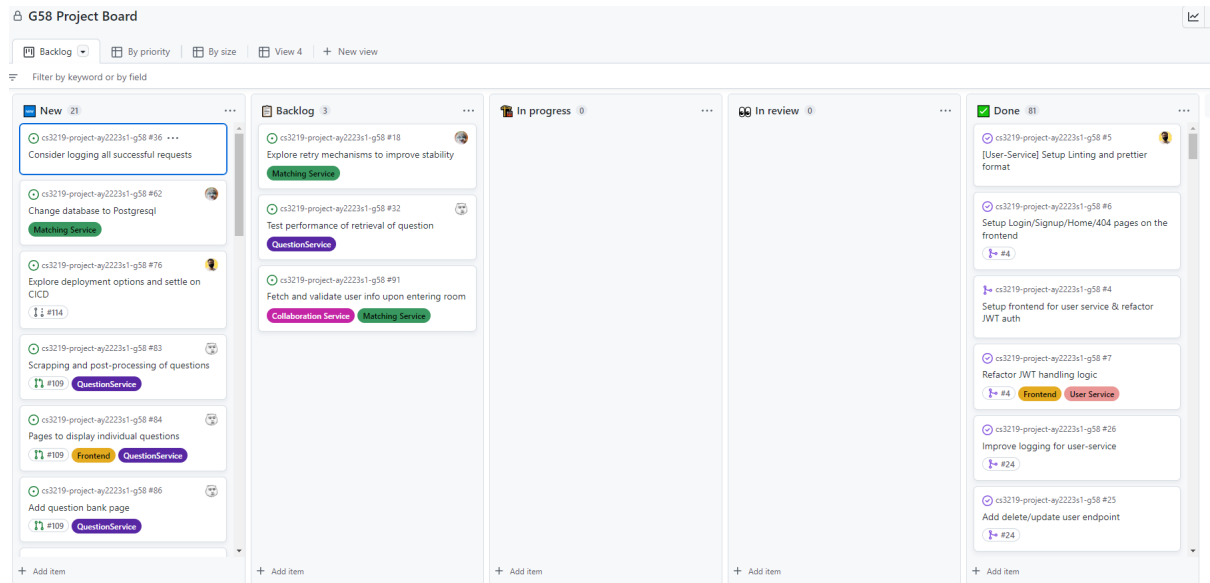


Diagram 26 Screenshot of Project Board

As we feel that the Scrum daily standup is suited to teams which are working on the tasks on a daily basis, our team decided to have a less frequent status update on Wednesday and Saturday morning, to get everyone up-to-date with the progress and any risks to task completion.

- What did I work on yesterday?
- What am I working on today?
- What issues are blocking me?

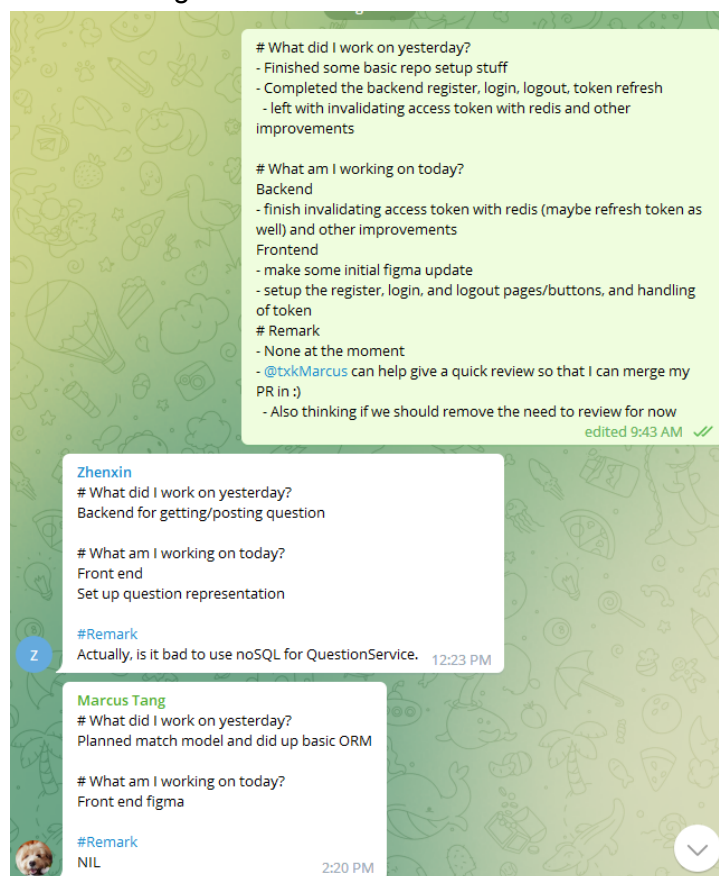


Diagram 27 Screenshot of some of the telegram updates

5.3 Version control

The team will adopt the Feature branch workflow. In general, before working on a new issue, an individual will first update their local main branch with the main branch of the remote repository. Then a feature branch will be created and later on pushed to the remote repository for PR into the main branch.

In addition to the main branch, which will be for staging/latest code changes, we will dedicate a production branch for production release once the production setup is completed.

Before a PR is merged into the main branch, we require that the continuous integration (linting & testing) is green and minimally one review approval is added.

For the milestone declaration, we loosely followed semantic versioning. Our milestone versions are of the form 'vX.Y' to denote major & minor version updates. In line with the project requirement from the module, we dedicated v2.0 and v3.0 to the milestone at week 6 and week 9. In other weeks, we will increment the minor version by 1.





v2.0 Closed 6 days ago ⌚ Last updated 6 days ago	 100% complete 0 open 11 closed Edit Reopen Delete
v1.2 Closed 6 days ago ⌚ Last updated 6 days ago	 100% complete 0 open 22 closed Edit Reopen Delete
v1.1 Closed on Sep 15 ⌚ Last updated about 1 month ago	 100% complete 0 open 21 closed Edit Reopen Delete
v1.0 Closed on Sep 14 ⌚ Last updated about 1 month ago	 100% complete 0 open 11 closed Edit Reopen Delete

Diagram 28 Screenshot of the versions under each milestone

5.4 Role Allocation & Challenges

The roles of each developer is determined by interest and familiarity with the upcoming services to be developed. Typically, we will first discuss and allocate a microservice or an independent area to avoid friction and merge conflicts. Later on, the developers of related microservices will communicate closely to integrate the services together. The details of the individual contributions will be listed in the next section.

While the above approach works well to split the workload fairly and promotes efficiency as everyone can usually make progress independently, there are challenges in integration and

in code-review. This is because the person who is reviewing the code for a service that he/she has not worked on will require extra effort to understand the context. In the same vein, connecting services together may require further tweaking on both ends to ensure compatibility.

5.5 Individual Contributions

Contribution Overview Description	Type	Team Member Involved
Drafting meeting minutes Review PRs and provide comments Documentation	non-technical	all
Local and cloud deployment configurations, CI/CD User service Frontend: User registration/login/profile update flow	technical	Liu Yongliang
Frontend: Matching Page, Room, Fancy UI Backend: Matching Service, Room Service, Video Communication	technical	Marcus Tang Xin Kye
Collaboration Service History Service Frontend: Collaborative editor, Learning history	technical	Ngo Ngoc Phuong Uyen
Scrapping and processing of questions Question Service Matching by types Frontend: Question Pages, Question Banks.	technical	Huang Zhenxin

Table 7 Contribution summary

The details of our code contributions are available on this [dashboard](#):



Diagram 29 Code Contribution Chart

6. Reflections & Learning Points

6.1 Ensuring Deployment Success

While we wanted to keep the configurations of the application the same across local and production environments, we ended up utilising the Kubernetes cluster for more reliability and scalability in the cloud. For convenience and due to system resource constraints, we developed the application locally with docker-compose, and deployed it in the cloud with Kubernetes configurations. Even though both use docker to specify the container details, the change in routing strategy as well as the multi-pod setup in the cluster resulted in production issues such as CORS errors and failure to ensure sticky sessions. These implications called for last-minute bug-fixes and refactoring which were difficult to debug and implement. In the future, it would be better to consider the potential differences between different environments, and handle them in a timely manner.

6.2 Importance of Clear Communication

In the planning and designing phase of our project, for certain APIs, we did not clearly communicate the input and output data of our service API. This resulted in different data being returned than expected, which caused unexpected behaviours. One such case was for the next question API, when both the difficulty and the type of the question was specified, the expected output is a question with the same difficulty and one overlapping type. However, the actual question being returned was a question of the same difficulty but with the exact type. For eg. Given type "Array", the next question will return a question with only type "Array" but not a question with both "Array" and "Dynamic Programming". Due to miscommunication, we only discovered this problem at a very late stage when we were system testing and had to make a last minute fix. In the future it would be better to communicate the interface for the APIs clearly in writing so that no misunderstanding or mis-rememberings can occur and all parties are clear on what to expect.

END