CS3219 Software Engineering Principles and Patterns

Group 31: Final Project Report (Peerprep)

Team Members	Matriculation Number
Branson Lam Jian Tao	A0234559W
Ang Yuze	A0234609A
Samuel Pang Shao Herng	A0234648X
John Benedict Yan Yaoyi	A0245020Y
Chiang Ho Gee, Joshua	A0238502L

Content Page

1. Project Introduction

1.1 What is Peerprep?

Peerprep is an application designed to help students grow proficiency in solving technical interview questions through collaboration with other students online, thus helping to nurture their problem-solving skills, technical skills, and communication.

Users are able to chat with fellow users and work together on technical interview questions in real time while keeping track of their own personal progress.

1.2 Motivations

Many students apply for jobs and internships in roles where technical skills are required, such as in software development, data analytics, software testing, or systems engineering. These roles often require applicants to go through technical interviews.

Students who are new to coding, or are not yet proficient in solving technical interview questions may find such technical interviews challenging. They may not know how to approach the questions, or if they do, they may not know how to effectively communicate their thought process and intuitions.

Our application is designed to help students to resolve the above issues, by allowing students to collaborate on technical interview questions in real-time. The development of our application is deeply rooted in the idea of peer programming as an effective pedagogical approach, and we believe that students will be able to pick up core programming concepts quickly through learning from each other.

2. Individual Contributions

The table below lists the individual contributions of each member, which is split into technical and non-technical components. Non-technical contributions primarily refer to the contributions with regard to documentation.

Name	**Technical Contributions**	**Non-Technical Contributions**
John Benedict Yan Yaoyi	Frontend: User Service Question Service Collaboration Service Matching Service N3, N5, N8 N9 Backend: User Service Question Service Collaboration Service Matching Service N3, N5, N8, N9 Deployment: Deployment to Google Cloud	Documentation: Project Report
Branson Lam Jian Tao	Frontend: Question Service (partial) History Service N2, N4 Backend: Question Service History Service N2, N4	Documentation: Project Report
Ang Yu Ze	Frontend: User Service Chat Service N1 Backend: User Service Chat Service N1 Deployment: Dockerized Chat Service	Documentation: Project Report
Chiang Ho Gee Joshua	Frontend: Matching Service Backend: Matching Service	Documentation: Project Report
Samuel Pang Shao Herng	Frontend: Question Service, User Service, Authentication, Code Execution	Documentation: Project Report

	N3, N9 <u>Backend:</u> Question Service, User Service N8, N9 <u>Deployment:</u> Deployment to Google Cloud Run	

3. Functional Requirements

3.1 User Service

The user Service facilitates the creation or logging in to an account to access core features of the Peerprep application. The functional requirements below are categorised into two parts: Authentication or Profile, based on relevance.

S/N	Functional Requirement	Priority	Implementation
1.1 Authentication			
1.1.1	The system should allow users to create an account with an email and password.	High	Register button on register page, Firebase authentication and user-Service backend
1.1.2	The system should ensure that every account created has a unique email.	High	Firebase authentication
1.1.3	The system should allow users to log into their accounts by entering their username and password.	High	Login button on Login page, Firebase authentication and user-Service backend
1.1.4	The system should allow users to log out of their account.	High	Logout button on Navbar and Firebase authentication
1.2 Profile			
1.2.1	The system should allow users to delete their account.	High	Delete button on profile page, Firebase authentication and user-Service backend

1.2.2	The system should allow users to change their password.	Medium	Update password button on profile page and Firebase authentication
1.2.3	The system should allow users to change their username.	Medium	Update password button on profile page and user-Service backend
1.2.4	The system should allow users to view their account settings.	Medium	Profile page and user-Service backend

3.2 Question Service

S/N	Functional Requirement	Priority	Implementation
2.1 Question Management			
2.1.1	The system should allow admin users to create, update and delete questions.	High	Implemented CRUD endpoints in API. Add frontend controls to add/edit/delete questions by querying said endpoints.
2.1.3	The system should allow admin users to add suggested solutions for questions.	Medium	Attach solutions to Question schema in backend.
2.1.4	The system should support multiple suggested solutions per question.	Medium	Solutions identified by unique ID.
2.2 Question Retrieval/View			
2.2.1	The system should be able to provide a random question based on a specified difficulty.	High	Add difficulty field to Questions schema, backend fetches random question of specified difficulty.
2.2.2	The system should allow browsing of all questions to view their details.	High	Implemented a table to list all questions in the Frontend.
2.2.3	The system should allow sorting and filtering by tags such as difficulty or category when browsing.	Medium	Added Search, Filter and Sort by options to the Frontend and to the findAll questions endpoint.
2.2.4	The system should allow viewing of suggested solutions to questions.	Medium	Added a solutions page in the frontend for viewing solutions.

3.3 Matching Service

The matching service is a microservice designed to facilitate the pairing of users within the collaborative ecosystem of peer prep. Its primary function is to analyze user-provided criteria (the desired difficulty of a question) and match users based on these given criteria so that they are able to collaborate together in conjunction with the collaboration service.

S/N	Functional Requirement	Priority	Implementation
3.1 Matching Users			
3.1.1	The system should allow users to create matching requests to be matched with other users for collaboration.	High	
3.1.2	The system should allow users to choose the difficulty of questions to be matched with other users for collaboration.	High	
3.1.3	The system should allow users to delete matching requests should they stop matching or after 60 seconds of not having a match	High	
3.1.4	The system should allow admin users to view the current users requesting to be matched and those that have been matched before.	High	

3.4 Collaboration Service

This service is responsible for establishing a connection between matched users allowing them to code on a shared workspace where each of their individual changes are reflected in each others' coding space.

S/N	Functional Requirement	Priority	Implementation
Collaboration			
3.4.1	The system should allow users to collaborate on a given problem with their various inputs being transferred to their matched counterparty.	High	Collaboration Context
3.4.2	The system should allow users to change the coding language for the given collaboration	High	Collaboration Context

3.5 History Service

S/N	Functional Requirement	Priority	Implementation
5.1 History Creation			
5.1.1	The system should automatically create a history for an attempt to a question for each user when a collaboration session between 2 users ends.	High	History Service subscribes to “collaboration-ended” topic.
5.2 History View			
5.2.1	The system should allow users to view all attempts for each question.	High	Show a history component in the frontend.
5.3 History Deletion			
5.3.1	The system should allow users to delete any attempt history	Medium	
5.3.2	The system should delete the history of a specific question for all users when that question is deleted.	High	History Service subscribes to “question-deleted” topic

3.6 Chat Service

S/N	Functional Requirement	Priority	Implementation
6.1 Chat Room Creation			
6.1.1	The system should automatically create a chat room when a matching between 2 users is found	High	Socket.io emits topic “join_room” to backend,
6.1.2	The system should add both users in a successful matching into the same chat room	High	matchingId used as roomId, and passed as a prop from parent to child component in frontend
6.2 Live Chat			
6.2.1	The system should allow users to send and receive messages with each other in real-time	High	Socket.io emits “send_message” and “receive_message” topic. Frontend chat box with input
6.2.2	The system should allow users to view all chats concurrently during an ongoing collaboration session	High	Stores all messages locally in browser
6.2.3	Users should be able to view details such as author, time, and message content in the chat	High	UI display details such as username, time and message details in the chat box
6.3 Chat Room Disconnect			
6.3.1	The system should remove users from the chat room once collaboration session has ended.	High	Socket.io emits topic “disconnect”

4. Non-Functional Requirements

We selected which Quality Attributes to prioritise for our NFRs based on the scores calculated in the table below. From the results, we concluded that Usability, Security, Efficiency and Availability were the 4 most important Quality Attributes.

Attribute	Score	Availability	Integrity	Perform-ance	Reliability	Robust-ness	Security	Usability	Efficien-cy
Availability	4		<	<	<	<	^	^	^
Integrity	1			^	<	^	^	^	^
Performance	2				^	<	^	^	^
Reliability	2					<	^	^	^
Robustness	1						^	^	^
Security	6							^	<
Usability	7								<
Efficiency	5								

4.1 Usability

Our website aims to provide a good user experience through building a user-friendly UI with features that provide ease-of-use and customizability. This helps solidify an impression of a well-built website that users will want to come back to and use regularly.

S/N	Non-Functional Requirement	Priority	Implementation
1.1	The UI should be clean and interactable elements should be clearly visible.	High	Frontend styling and prescence of Navbar and footer, to allow for convenient pagination.
1.2	The UI should have changeable Light-mode/Dark-mode themes that users can pick to suit their preference	Medium	Frontend button to toggle between light and dark modes. Code editor also has a theme selector to cater to user's preferred code editor theme.
1.3	The UI should inform the user of any errors encountered that would affect their experience.	High	Error handling in Frontend and Backend Services, with error messages displayed to provide evidence of the particular error.

1.4	The UI should warn the user/admin when conducting any deletion operations.	Medium	Confirmation window on Frontend. For example, a confirmation password is required for deletion of user account.
1.5	The UI should be simple and easy to navigate around.	Medium	Design a smooth onboarding process for new users, guiding them through key features and functionality. Icons should meet standard practice
1.6	Users should have the capability to request assistance for any questions or issues they encounter.		Provide user-friendly help resources, FAQs, and documentation to assist users in using the website effectively.

4.2 Security

It is necessary to allow only registered users to access features in the application and restrict unauthorised users, thus both the frontend and backend endpoints should be protected.
User credentials should also be handled securely.

S/N	Non-Functional Requirement	Priority	Implementation
1.2	The system should not store passwords as plaintext but should be encrypted, whereby passwords should not be retrievable.	High	Firebase authentication stores encrypted passwords on Firebase cloud and provides no API calls to retrieve passwords
1.3	The system should ensure that users can only create accounts with passwords that meet a minimum length	High	Firebase authentication prevents accounts with passwords that do not meet the required minimum length from being created

4.3 Efficiency

Enabling users to collaborate on coding and communicate in real-time with minimal latency is essential. Therefore, the exchange of data between microservices and the transmission of messages from the frontend to the backend must be highly efficient, with minimal downtime. This is crucial to support effective collaboration among users.

S/N	Non-Functional Requirement	Priority	Implementation

2.1	The system should be capable of handling a large number of concurrent users and messages with low latency. For example, it should be able to process and deliver messages within milliseconds.	Medium	Combine Kafka and Socket.IO for real-time collaboration. Kafka acts as a high-throughput, distributed event streaming platform, ensuring efficient and reliable communication between microservices. Messages from various sources are ingested into Kafka and Socket.IO topics, allowing for asynchronous and parallel processing.
2.2	The system must be scalable to accommodate an increasing number of users and messages over time.	Medium	Multiple replica containers from the same Docker image, each serving the same application or service. These containers can be deployed on different servers or nodes.

4.4 Availability

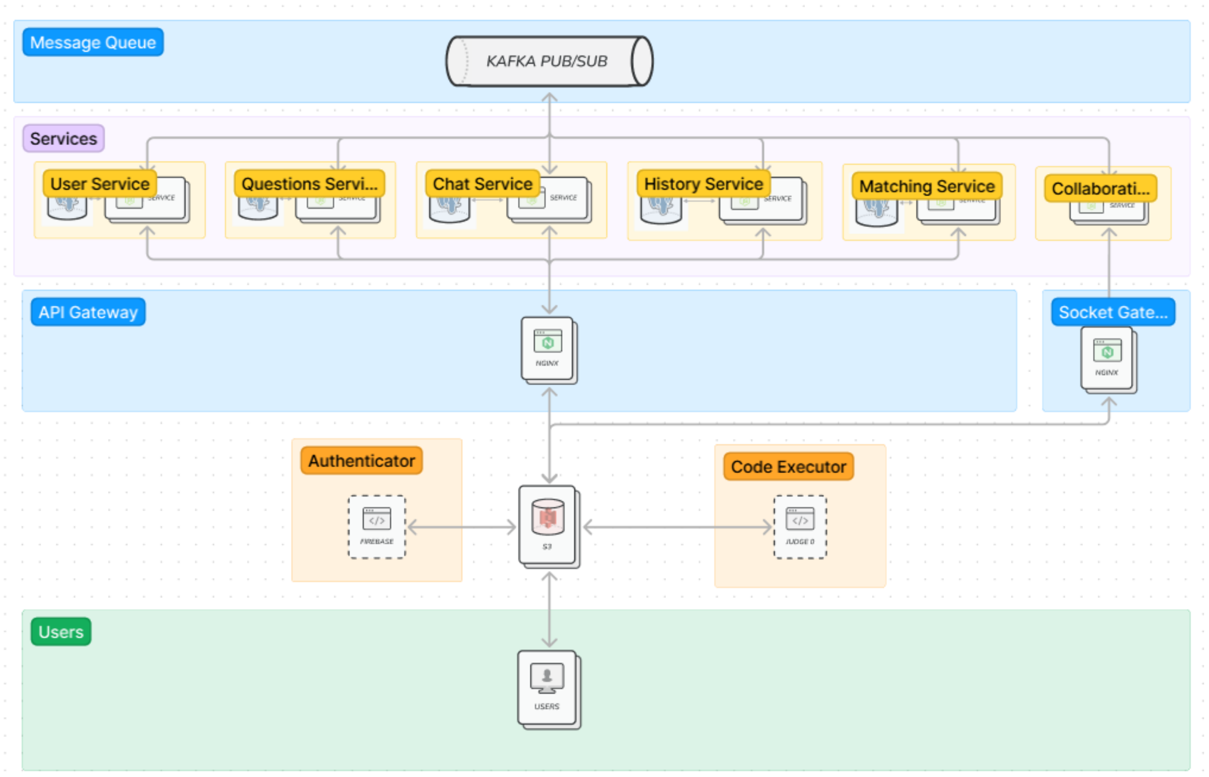
In a live collaboration application, maintaining high availability is essential to facilitate seamless collaboration. Therefore, it's imperative that the application minimises downtime for maintenance, ensuring that collaboration remains uninterrupted.

S/N	Non-Functional Requirement	Priority	Implementation
4.1	The system should be designed to operate without disruption even if one of its services experiences downtime. In other words, it should avoid having a single point of failure.	High	Continuous health checks will be conducted on each service.The microservices are designed to be loosely coupled, with well-defined APIs and clear boundaries. This isolation ensures that the failure of one service does not directly impact the functionality of other services.
4.2	The system should be designed to minimize downtime and quickly restore systems in case of failures	Medium	Implement disaster recovery tools such as Veeam or Zerto to ensure swift recovery of systems

5. Application Design

5.1 Architecture

5.1.1 Architecture overview



For the architecture of our application, we decided to use Microservice Architecture. To enforce strict decoupling, all services do not communicate directly with one another. For services that require persistent data storage, we chose to use PostgreSQL database. Each of these services will have their own instance of database. We utilised a Public-Subscribe pattern with Kafka as our Event Broker for any services' functionality that requires passing or receiving information from other services. These services would produce or consume events to communicate with one another through the Event Broker.

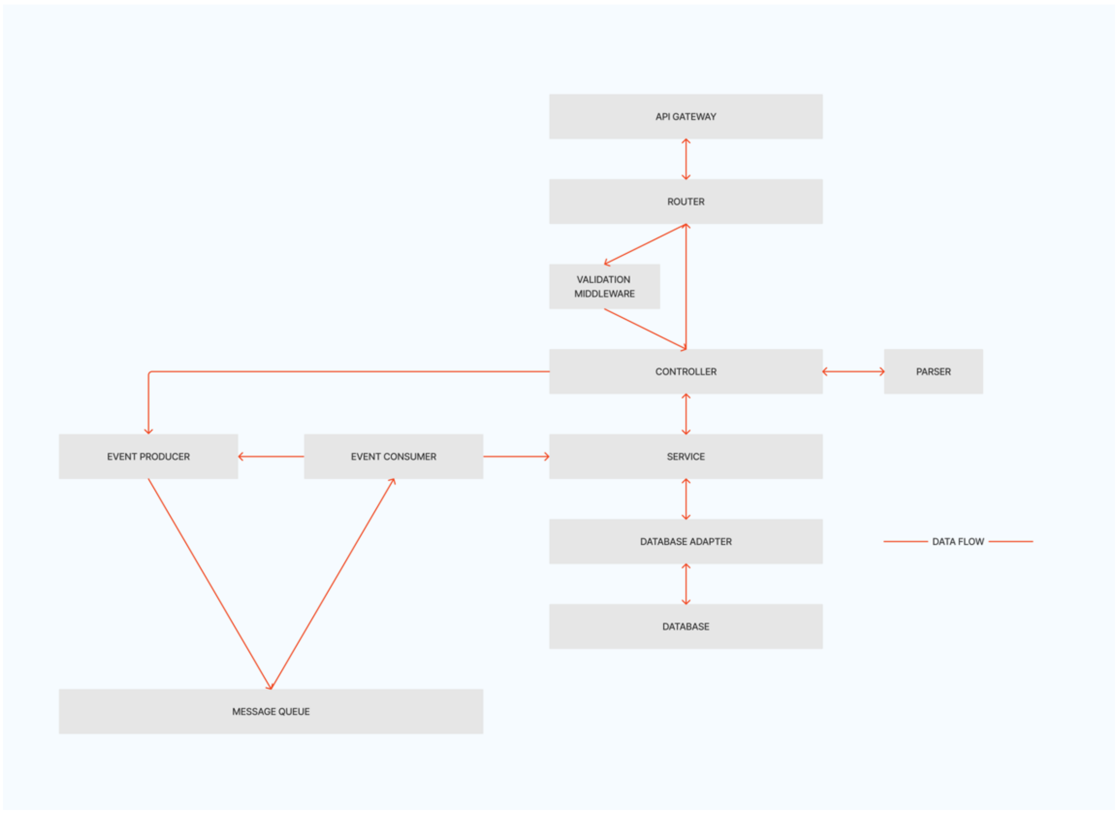
Based on the user's interaction with the Frontend, the Frontend will execute the user's instructions and communicate with each service via querying its REST api or through Socket connections.

We chose this Architecture and design because of the low dependency and separation of concerns between services. This allows each member of our team to work on different services and features in parallel, simplifying and accelerating our development process.

5.1.2 Services

Our application consists of the following Services:

- Front end Service
- Back end
    - User Service
    - Question Service
    - Matching Service
    - Collaboration Service
    - History Service
    - Chat Service



Each microservice within our architecture follows a structured and modular design pattern, consisting of three core components: Parser, Service, and Controller. This design approach provides a systematic and organised way to manage the flow of data and functionality within each microservice.

- Parser: The Parser component is responsible for handling the input and output data for the microservice. It serves as the entry point for data ingestion, processing incoming requests, and formatting outgoing responses. The Parser ensures that data is received in the correct format and is prepared for further processing within the microservice.

- Service: The Service component contains the core business logic and functionality of the microservice. It processes the data received from the Parser, performs necessary operations, and interacts with internal dependencies and external services or databases. This is where the main processing tasks of the microservice are carried out.
- Controller: The Controller component acts as an interface between the microservice and external systems or clients. It handles incoming requests, routes them to the appropriate Service functions, and manages the overall communication flow. The Controller ensures that the microservice responds to external requests efficiently and effectively.

Each microservice operates as an independent and self-contained unit, encapsulating its own set of dependencies, libraries, and resources. This isolation allows microservices to be containerized using Docker, enabling them to run autonomously without relying on the environment of other services.

Moreover, this microservice architecture inherently prevents a single point of failure. Since each microservice runs independently, the failure of one microservice does not impact the entire system.

Additionally, this design approach significantly reduces coupling among microservices. Each microservice has well-defined boundaries and communicates with other services through Kafka, an event broker. This loose coupling allows for flexibility in development, scalability, and maintenance. Changes to one microservice do not necessitate changes to others, making it easier to evolve and enhance individual components of the system.

5.2 Tech stack

The following table describes the technologies used for the respective Services of our application:

Technology	**Used in**	**Rationale**
React with **Vite**	Frontend	● Frontend library that team members are most experienced with, allowing a faster and smoother development process ● Vite is utilised due to the seamless development experience due to responsive file updates and efficient build process features
Express	All Microservices	● Popular framework for Node.js with a huge community developing many different libraries that can suit any need, giving us a lot of versatility. ● Team members are most familiar with Express.

PostgreSQL with **Prisma** ORM	User, Question, Matching, History services	● Most team members are more familiar with using relational databases than NoSQL databases. ● Using Prisma ORM allows us to interact with the database in a Typesafe manner, simplifying the development process. ● Prisma auto-generates SQL migrations from the schema, giving us the flexibility to make changes to our database design easily during the development process.
Socket.IO	Frontend, Socket, Chat services	● Socket.IO library provides a simple and straightforward interface to facilitate real-time communication. ● This library allows endpoints to produce and consume events that are subscribed to that socket instance.
Kafka	User, Question, Matching, Socket History services	● Even though most team members aren't familiar with using event brokers for cross-service communication, Kafka offers a straightforward and efficient interface to adopt. ● The team aimed to implement an event-driven architectural style to minimize the interdependencies among individual services. Kafka serves as an effective means for services to communicate without requiring them to have in-depth knowledge or understanding of the specifics of each service.
Docker	All Microservices	● Isolation: Docker containers provide a high degree of isolation for each microservice. This means that each service runs in its own container with its own dependencies, ensuring that changes or issues in one service do not impact others. ● Simplified Deployment: Docker simplifies the deployment process. The team can easily define the environment and dependencies for a microservice in the Dockerfile, allowing the team to facilitate and organize the individual containers in a consistent manner.
ESlint	All Microservices	● Preventing Common Errors: ESLint helps identify and prevent common programming errors and pitfalls ● Fixing Issues Automatically: ESLint can automatically fix many common issues, such as code formatting problems, using the --fix option. This help us save time in manually editing errors

		● Code Consistency: ESLint enforces coding standards and style guidelines, ensuring that code within our project is consistent and follows a predefined set of rules. This consistency improves code readability and maintainability.
Tailwind CSS and Post CSS	Frontend	● Tailwind CSS streamlines the process of styling web applications, offering a balance between utility and flexibility. It allows the team to create well-designed and responsive user interfaces efficiently while providing extensive customization options. ● PostCSS provides the team with the flexibility to choose and customize the processing we need while improving code maintainability and performance.
Playwright	Frontend	● Cross-Browser Testing: Playwright supports multiple web browsers, including Chromium, Firefox, and WebKit, allowing the team to perform cross-browser testing and ensure compatibility with different browsers.
Firebase	User Service	● Firebase provides robust auto-authentication features for login and registration, coupled with its strong security measures. The choice was driven by the seamless user authentication experience it offers, while also ensuring data security and protection.
JEST	All Microservices	● Zero Configuration: Jest requires minimal setup and configuration, making it easy to test our JavaScript code. ● Fast Execution: Jest is known for its speed. It optimizes test execution by running only the necessary tests and parallelizing them when possible. This results in quick feedback during development. ● Mocking Capabilities: Jest provides built-in mocking capabilities, which allows us to easily mock dependencies and functions, making it simpler to isolate and test specific parts of our code.

6. Microservices

6.1 User Service

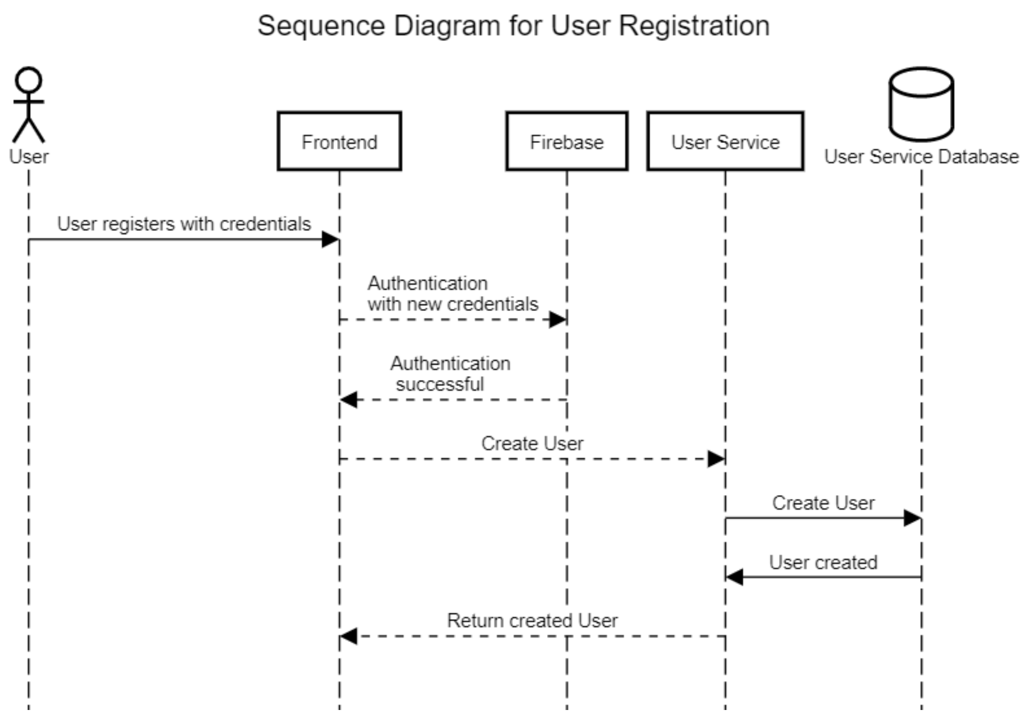
6.1.1 Overview

The user service is responsible for handling the creation of user accounts, storing of user data, as well as editing and deleting respective user accounts. Both the user service, and frontend authentication functionality (implemented with Firebase) works together to ensure successful user registration. As seen in the summarised diagram below, Firebase handles the authentication portion and stores the user email and password. When the authentication is successful on registration, a User is then created in the user service backend, which stores relevant profile data. We designed the user creation and authentication process in this way due to the secure way that Firebase handles user credentials, namely:

1. Reauthentication is required for resetting user-sensitive information. Changing of passwords and deletion of user strictly requires reauthentication, which ensures no important user information is changed without the proper user credential.

2. Passwords are kept strictly confidential, whereby existing Firebase API calls do not reveal the plaintext password.

Other less sensitive information that are not necessary for User login, such as user information other than user email and password can then be stored in our user service database backend. We believe that the credential security provided by Firebase would help in building a robust and scalable application with many user credentials.


Sequence Diagram for User Registration
User
Frontend
Firebase
User Service
User Service Database
User registers with credentials
Authentication with new credentials
Authentication successful
Create User
Create User
User created
Return created User

6.1.2 Schema

Type	Field	Description	Type
User	id	Primary key for a User	string
	username	String denoting a User’s registered username	string
	createdAt	Ttime of creation of the User	Date
	updatedAt	Llatest time of update of the User profile	Date
	questionsAuthored	Number of questions created by the User	number
	roles	Roles that the user can take	string[]

6.2 Question Service

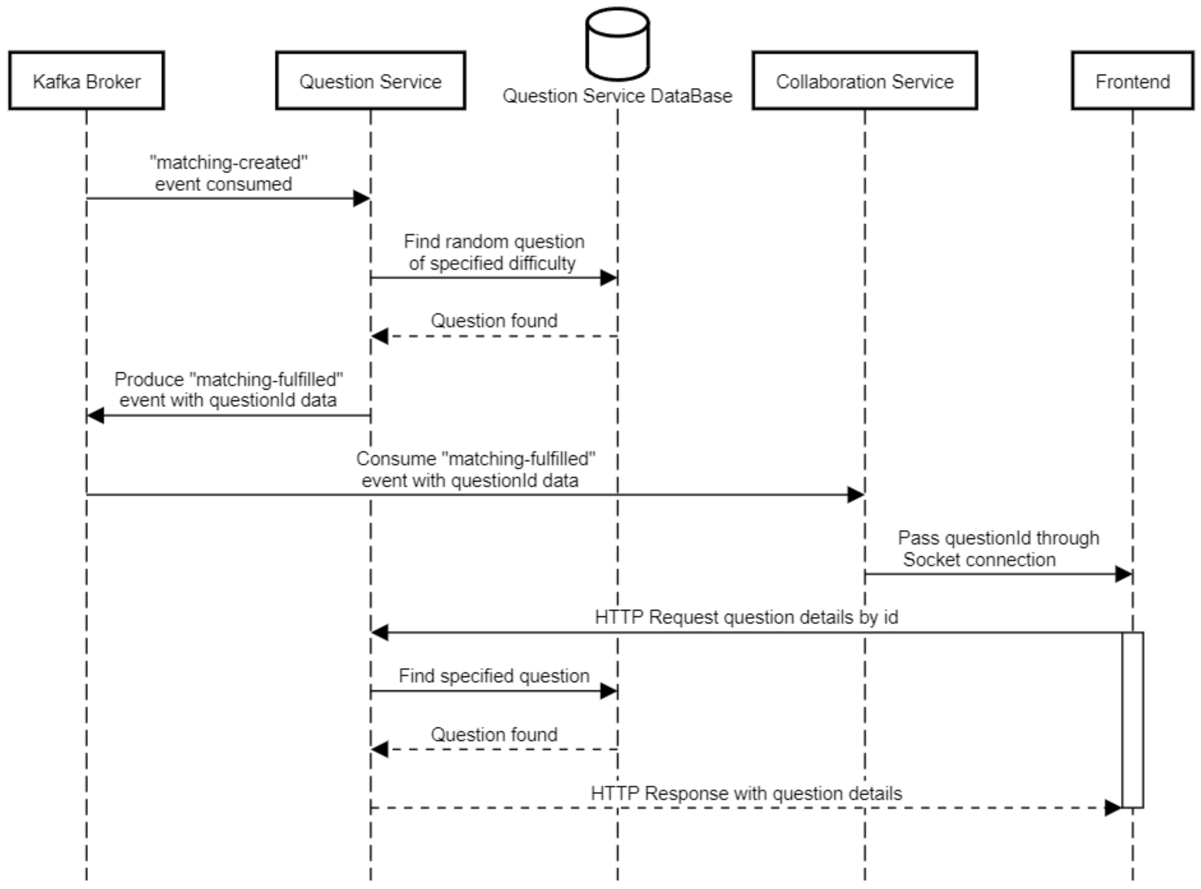
6.2.1 Overview

The Question Service is mainly in charge of storing the questions for users to attempt. It also stores extra information like suggested solutions and test cases for a question.

Besides allowing the user to browse through questions, the Question Service is also responsible for selecting and providing a Question for a collaboration session between 2 users.

When a match between 2 users is found, a “matching-created” event is created by the Matching Service, the sequence diagram below describes how Question service will provide a Question to the Frontend for the collaboration.

Getting Collaboration Question


Kafka Broker
Question Service
Question Service DataBase
Collaboration Service
Frontend
"matching-created" event consumed
Find random question of specified difficulty
Question found
Produce "matching-fulfilled" event with questionId data
Consume "matching-fulfilled" event with questionId data
Pass questionId through Socket connection
HTTP Request question details by id
Find specified question
Question found
HTTP Response with question details

6.2.2 Schema

Type	Field	Description	Type
Question	id	Primary key for a question	number
	title	Title of the question	string
	difficulty	“Easy”/”Medium”/”Hard”	string
	description	Description of the question	string
	examples	Example expected inputs and outputs	string[ ]
	constraints	Constraints for the question	string[ ]
	popularity	Number of times question was attempted by ALL users	number
	authorId	User id of question author	string
	createdAt	Date and time question was created	Date
	updatedAt	Date and time question was updated most recently	Date
	solutions	Suggested solutions to the question	Question Solution[ ]
	categories	Categories question belongs to	Question Category[ ]
	testCases	Set of test cases for the question	QuestionTest Case[ ]
	initialCodes		QuestionInitial Code[ ]
	runnerCodes		Question RunnerCode[ ]
Solution	id	Primary key for a solution	string

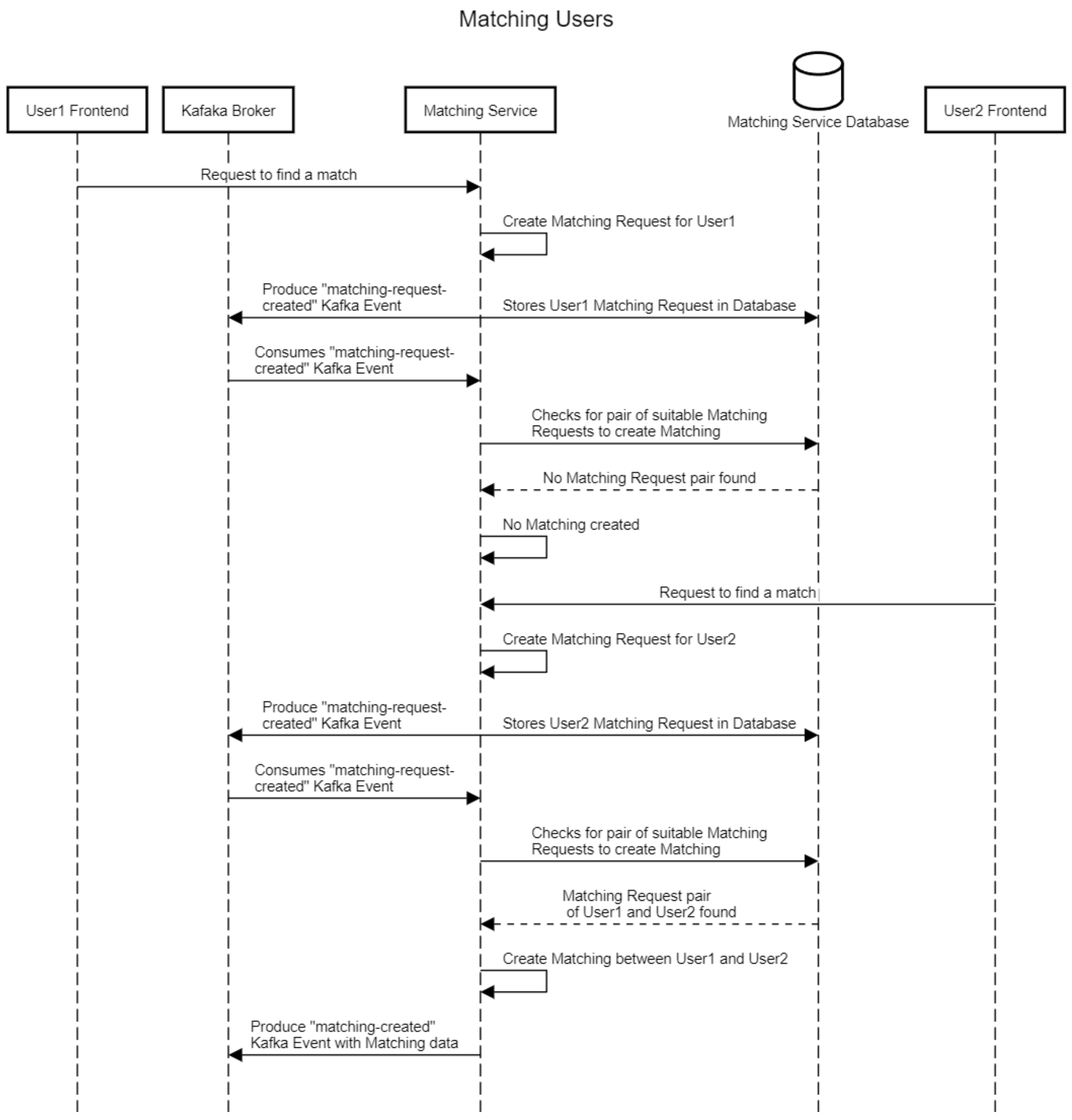
	questionId	Id of question this solution is for	number
	title	Title of the solution	string
	description	Description of the solution	string
	code	Code for this solution	string
	language	Language used for the code	string
Question Category	name	Name of the category	string
	questionId	Id of question this category belongs to	number
Question TestCase	testCaseNumber	Index of test case within the question	number
	input	Input of test case	string
	expectedOutput	Expected outputs of test case	string[ ]
	questionId	Id of question this test case belongs to	number
Question Initial Code	language	Language used for the code	string
	code	Code	string
	questionId	Id of question this code belongs to	number
Question Runner Code	language	Language used for the code	string
	code	Code	string
	questionId	Id of question this code belongs to	number

6.3 Matching Service

6.3.1 Overview

The Matching Service is responsible for matching 2 users to collaborate on a question together. The Matching Service creates and stores matching requests from the Frontend clients. Once matching requests from 2 compatible users are received, it pairs them up to create a Matching.

The sequence diagram below shows how 2 users will be matched with each other.


Matching Users
User1 Frontend
Kafaka Broker
Matching Service
Matching Service Database
User2 Frontend
Request to find a match
Create Matching Request for User1
Produce "matching-request-created" Kafka Event
Stores User1 Matching Request in Database
Consumes "matching-request-created" Kafka Event
Checks for pair of suitable Matching Requests to create Matching
No Matching Request pair found
No Matching created
Request to find a match
Create Matching Request for User2
Produce "matching-request-created" Kafka Event
Stores User2 Matching Request in Database
Consumes "matching-request-created" Kafka Event
Checks for pair of suitable Matching Requests to create Matching
Matching Request pair of User1 and User2 found
Create Matching between User1 and User2
Produce "matching-created" Kafka Event with Matching data

6.3.2 Schema

Type	Field	Description	Type

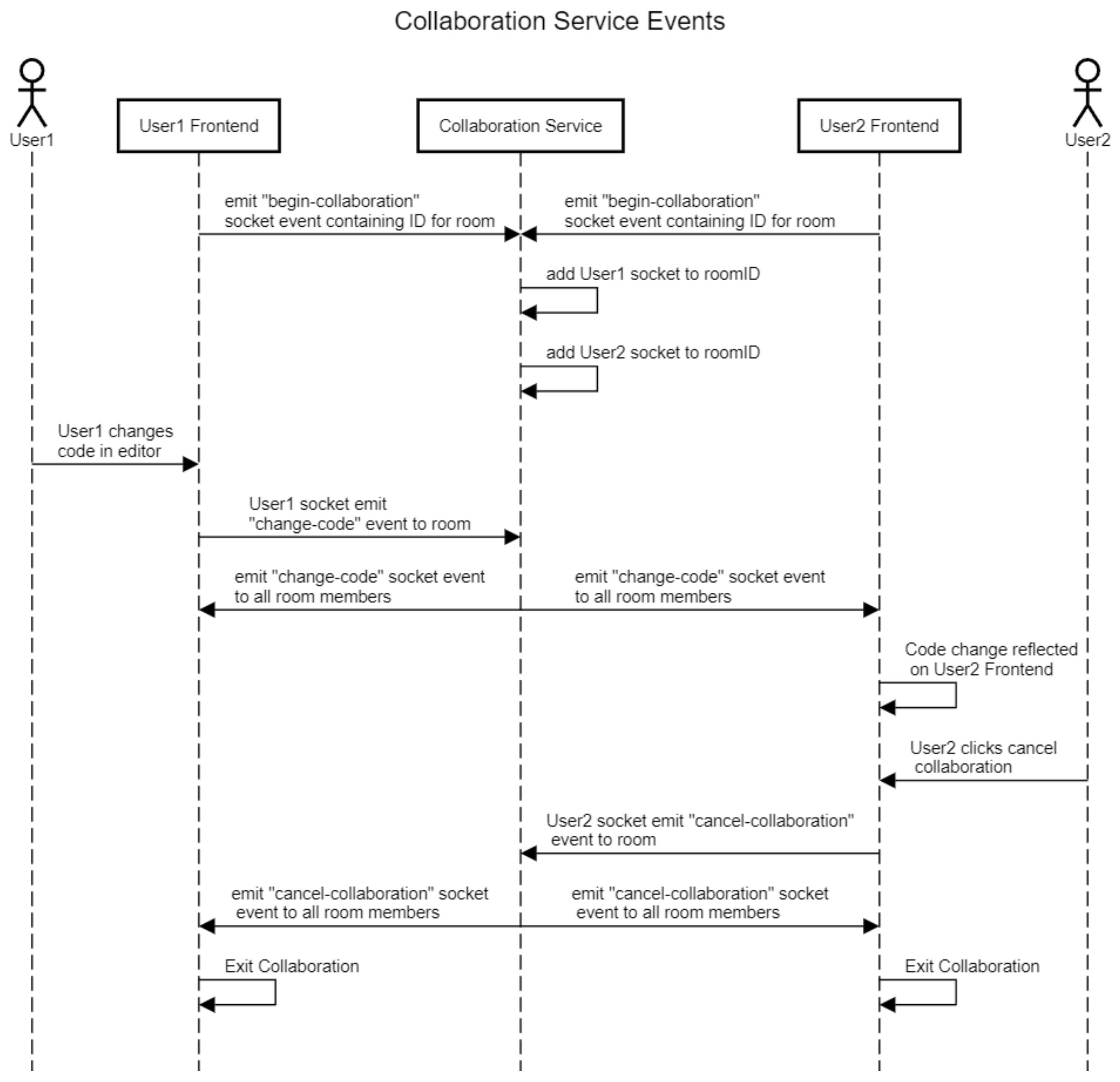
Matching	id	Primary key for a matching	number
	user1Id	Id of the first user of the match	string
	user2Id	Id of the second user of the match	string
	requestId	Id of the Matching Request	number
	difficulty	Requested question difficulty by users of the match: “Easy”/”Medium”/”Hard”	string
	questionIdRequested	Id of question if requested	number?
	dateTimeMatched	Date and time matching was created	Date
Matching Request	id	Primary key for a matching	number
	userId	Id of the user making the request	string
	difficulty	Requested question difficulty by user “Easy”/”Medium”/”Hard”	string
	questionIdRequested	Id of question if requested	number?
	dateRequested	Date and time matching request was created	Date
	success	Whether match request was successfully matched	boolean

6.4 Collaboration Service

6.4.1 Overview

The Collaboration Service is responsible for the real-time communication between frontend clients through socket-connections. The sockets of 2 clients that are matched will join the same room in the Collaboration Service server. From there, they will be able to exchange important events in real-time like changing the code in the collaborative code-editor, changing the programming language used or cancelling the collaboration session.

The sequence diagram below shows the process of a collaboration session, from the start of a session, making changes to code and to the end.


Collaboration Service Events
User1
User1 Frontend
Collaboration Service
User2 Frontend
User2
emit "begin-collaboration" socket event containing ID for room
emit "begin-collaboration" socket event containing ID for room
add User1 socket to roomID
add User2 socket to roomID
User1 changes code in editor
User1 socket emit "change-code" event to room
emit "change-code" socket event to all room members
emit "change-code" socket event to all room members
Code change reflected on User2 Frontend
User2 clicks cancel collaboration
User2 socket emit "cancel-collaboration" event to room
emit "cancel-collaboration" socket event to all room members
emit "cancel-collaboration" socket event to all room members
Exit Collaboration
Exit Collaboration

6.4.2 Socket Events

Socket events are emitted from both frontend and backend to facilitate real-time communication.

Event	**Description**	**Body**
join	Client joins a holding room	userId: string;

begin_collaboration	Client joins a collaboration room with another user client.	{
		  requestId: string;
		  userId: string;
		}

change-code	Client made a change to the code.	{
		  requestId: string;
		  userId: string;
		  code: string;
		}

change-language	Client made a change to the language.	{
		  requestId: string;
		  userId: string;
		  language: string;
		}

cancel-collaboration	Client ended collaboration session.	{
		  questionId: number;
		  requestId: string;
		  userId: string;
		  matchedUserId: string;
		  language: string;
		  code: string;
		}

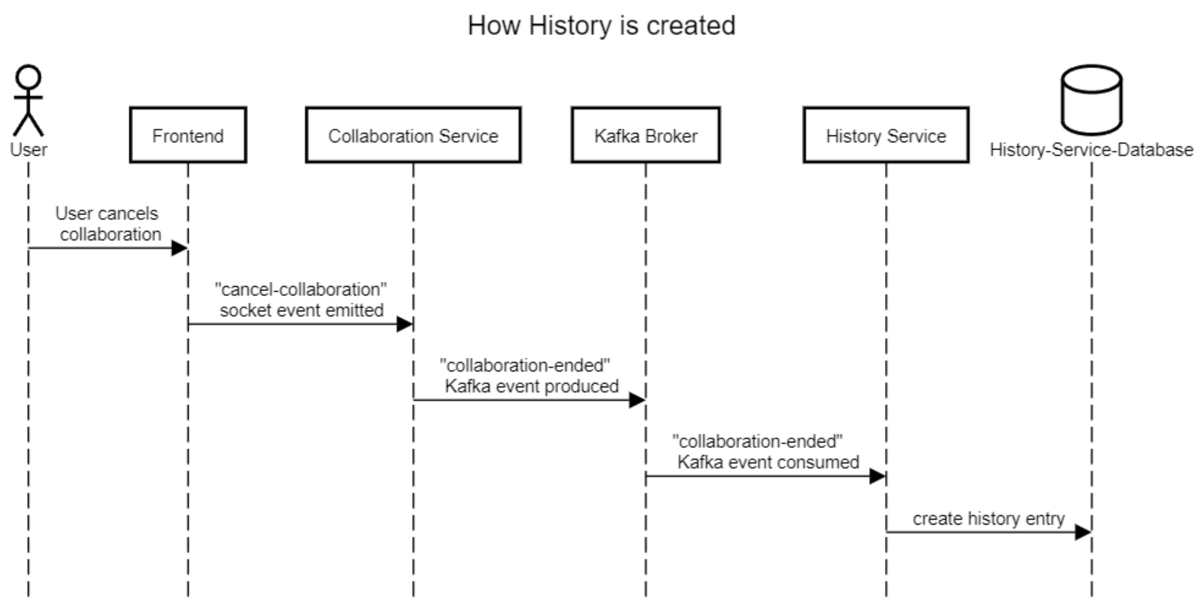
6.5 History Service

6.5.1 Overview

The History Service is in charge of saving the attempts of questions by users after a collaboration. It stores the final code written in the code editor and the final language used when the collaboration session ends. This is stored alongside the id of the question attempted during the session, and the id of the 2 collaboration users.

Each user may only see their own history, they may not see the history or attempts of other users. They can view their history for any question via the “History” panel when viewing a question.

The details of how History is created is shown in the following sequence diagram.


How History is created
User
Frontend
Collaboration Service
Kafka Broker
History Service
History-Service-Database
User cancels collaboration
"cancel-collaboration" socket event emitted
"collaboration-ended" Kafka event produced
"collaboration-ended" Kafka event consumed
create history entry

6.5.2 Schema

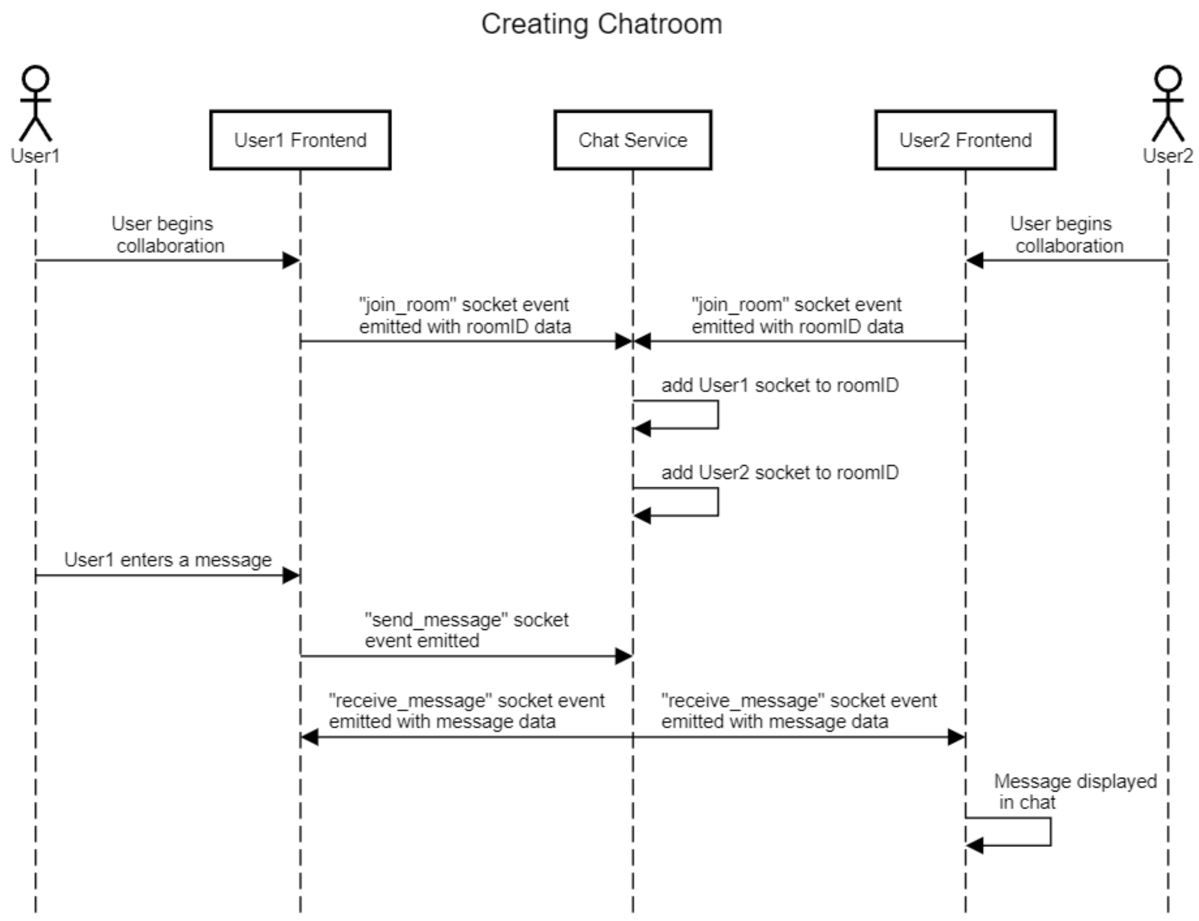
Type	Field	Description	Type
History	id	Primary key for a history	string
	questionId	Id of question this history belongs to	number
	user1Id	Id of user1 of the collaboration session this history was created for	string
	user2Id	Id of user2 of the collaboration session that created this history.	string
	createdAt	Date and time this history was created	Date
	code	Code for the attempt of the question	string
	language	Language used for the code	string

6.6 Chat Service

6.6.1 Overview

The Chat Service is in charge of passing messages between 2 users during a collaboration. It sends and receives messages in real time, allowing users to communicate via a chat box.

It will first add both users during a collaboration into the same chat room, with the room id received from the frontend. Then, it facilitates communication between the 2 users by emitting "send_message" and "receive_message" events, and passes the message body back and forth among users.


Creating Chatroom
User1
User1 Frontend
Chat Service
User2 Frontend
User2
User begins collaboration
User begins collaboration
"join_room" socket event emitted with roomID data
"join_room" socket event emitted with roomID data
add User1 socket to roomID
add User2 socket to roomID
User1 enters a message
"send_message" socket event emitted
"receive_message" socket event emitted with message data
"receive_message" socket event emitted with message data
Message displayed in chat

6.6.2 Socket Events

Socket events are emitted from both frontend and backend to facilitate real-time communication.

Event	**Description**	**Body**
join_room	Joins a specific chat room	matchingId: string;
send_message	Sends a message in a specific chat room	messageData: string;
receive_message	Receives a message in a specific chat room	messageData: string;
disconnect	Disconnect users in a chat room	-

7. API Documentation

Kafka Events

Topic	**Service Source**	**Produced When**	**Data**
collaboration-ended	Collaboration	A Collaboration session ended	{ questionId: number user1Id: string; user2Id: string; code: string; language: string; }
matching-created	Matching	A Matching Object is created in the Database	{ id: number; user1Id: string; user2Id: string; requestId: number; difficulty: string; questionIdRequested: number | null; dateTimeMatched: Date; }
matching-updated	Matching	A Matching Object is updated in the Database	{ id: number; user1Id: string; user2Id: string; requestId: number; difficulty: string; questionIdRequested: number | null;

			dateTimeMatched: Date;
			}

matching-deleted	Matching	A Matching Object is deleted from the Database	{
			id: number;
			user1Id: string;
			user2Id: string;
			requestId: number;
			difficulty: string;
			questionIdRequested: number | null;
			dateTimeMatched: Date;
			}

matching-request-created	Matching	A Matching Request Object is created in the Database	{
			id: number;
			userId: string;
			questionId: number | null;
			difficulty: string;
			dateRequested: Date;
			success: boolean;
			}

matching-request-updated	Matching	A Matching Request Object is updated in the Database	{
			id: number;
			userId: string;
			questionId: number | null;
			difficulty: string;
			dateRequested: Date;
			success: boolean;
			}

matching-request-failed	Matching	There is no Matching Request counterparty available to match with a Matching Request.	{
			id: number;
			userId: string;
			questionId: number | null;
			difficulty: string;
			dateRequested: Date;
			success: boolean;
			}

matching-request-deleted	Matching	A Matching Request Object is deleted from the Database	{
			id: number;
			userId: string;
			questionId: number | null;
			difficulty: string;
			dateRequested: Date;
			success: boolean;
			}

matching-fulfilled	Question		{
			id: number;
			user1Id: string;
			user2Id: string;

			requestId: number;
			difficulty: string;
			questionIdRequested: number;
			dateTimeMatched: Date;
			}

question-deleted	Question	A Question is deleted	{
			questionId: number;
			}

7.1 User Service

Health Check- GET

Endpoint: http://localhost:5001/api/users/healthCheck
Description: Endpoint to verify User Service is accessible.

Response		
Status	Description	Data
200 OK	User Service Working	{ message: “OK” }

Get User by ID - GET

Endpoint: http://localhost:5001/api/users/:id
Description: Endpoint to retrieve user matching provided id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	User retrieved successfully.	User
400 BAD REQUEST	Invalid params.	Error Message
500 INTERNAL SERVER ERROR	Internal configuration error.	Error Message

Get all Users - GET

Endpoint: http://localhost:5001/api/users
Description: Endpoint to retrieve all users, if optional id is passed, it retrieves user by id instead.

Request		
Property	Optional/Required	Data
Query	Optional	id

Response		
Status	Description	Data
200 OK	Users retrieved successfully.	List of Users
400 BAD REQUEST	Invalid params.	Error Message

Create User - POST

Endpoint: http://localhost:5001/api/users
Description: Endpoint to create a User.

Request		
Property	Optional/Required	Data
Body	Required	Id, username, roles

Response		
Status	Description	Data
200 OK	User created successfully.	User
400 BAD REQUEST	Invalid params.	Error Message

Update User - PUT

Endpoint: http://localhost:5001/api/users/:id
Description: Endpoint to update a User matching the given id.

Request		
Property	Optional/Required	Data
Param	Required	id
Body	Optional	Username, roles, questionsAuthored

Response		
Status	Description	Data
200 OK	User updated successfully.	User
400 BAD REQUEST	Invalid params.	Error message
500 INTERNAL SERVER ERROR	Internal configuration error.	Error Message

Delete User - DELETE

Endpoint: http://localhost:5001/api/users/id
Description: Endpoint to delete a history matching the given id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	User deleted successfully.	User
400 BAD REQUEST	Invalid params.	Error message

7.2 Question Service

Health Check- GET

Endpoint: http://localhost:5003/api/questions/healthCheck
Description: Endpoint to verify the Question Service is accessible.

Response		
Status	Description	Data
200 OK	Matching Backend Working	{ message: "OK" }

Get Question - GET

Endpoint: http://localhost:5003/questions/:id
Description: Endpoint to retrieve a specific question of given id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	Question retrieved successfully.	Question
400 BAD REQUEST	Invalid params.	-

Get all Questions - GET

Endpoint: http://localhost:5003/api/questions?
Description: Endpoint to retrieve questions matching provided query.

Request		
Property	Optional/Required	Data
Query	Optional	id, title, difficulty, description, examples, constraints, authorId, solutions, categories, testCases, initialCodes, runnerCodes, offset, limit

Response

Status	Description	Data
200 OK	Questions retrieved successfully.	Question[ ]
400 BAD REQUEST	Invalid params.	-

Create Question - POST

Endpoint: http://localhost:5003/api/questions
Description: Endpoint to create a question.

Request		
Property	Optional/Required	Data
Body	Required	title, difficulty, description, examples, constraints, authorId, solutions, categories, testCases, initialCodes, runnerCodes

Response		
Status	Description	Data
200 OK	Question created successfully.	Question
400 BAD REQUEST	Invalid params.	-

Update Question - PUT

Endpoint: http://localhost:5003/api/questions/:id
Description: Endpoint to update a question matching the given id.

Request		
Property	Optional/Required	Data
Param	Required	id
Body	Optional	title, difficulty, description, examples, constraints, authorId, solutions, categories, testCases, initialCodes, runnerCodes

Response		
Status	Description	Data

200 OK	Question updated successfully.	Question
400 BAD REQUEST	Invalid params.	-

Delete Question - DELETE

Endpoint: http://localhost:5003/api/question/:id
Description: Endpoint to delete a question matching the given id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	Question deleted successfully.	Question
400 BAD REQUEST	Invalid params.	-

7.3.1 Matching Service (Matching)

Health Check- GET

Endpoint: http://localhost:5002/api/matching/healthCheck
Description: Endpoint to verify the Matching backend of the Matching Service is accessible.

Response		
Status	Description	Data
200 OK	Matching Backend Working	{ message: "OK" }

Get Matching by ID - GET

Endpoint: http://localhost:5002/api/matching/:id
Description: Endpoint to retrieve Matching corresponding to provided id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	Matching retrieved successfully.	Matching
400 BAD REQUEST	Invalid params.	Error Message

Get all Matchings - GET

Endpoint: http://localhost:5002/api/matching
Description: Endpoint to retrieve all Matchings, if optional id is passed, it retrieves Matching by id instead.

Request		
Property	Optional/Required	Data
Query	Optional	id

Response

Status	Description	Data
200 OK	Matchings retrieved successfully.	List of Matchings
400 BAD REQUEST	Invalid params.	Error message

Create Matching - POST

Endpoint: http://localhost:5002/api/matching
Description: Endpoint to create a Matching.

Request		
Property	Optional/Required	Data
Body	Required	user1Id, user2Id, requestId, difficulty, questionIdRequested

Response		
Status	Description	Data
200 OK	Matching created successfully.	Matching
400 BAD REQUEST	Invalid params.	Error Message

Update Matching - PUT

Endpoint: http://localhost:5002/api/matching/:id
Description: Endpoint to update a Matching having the corresponding id.

Request		
Property	Optional/Required	Data
Param	Required	id
Body	Optional	user1Id, user2Id, requestId, difficulty, questionIdRequested

Response		
Status	Description	Data

200 OK	Matching updated successfully.	Matching
400 BAD REQUEST	Invalid params.	Error message

7.3.2 Matching Service (MatchingRequest)

Health Check- GET

Endpoint: http://localhost:5002/api/matchingRequest/healthCheck
Description: Endpoint to verify MatchingRequest backend of the Matching Service is accessible.

Response		
Status	Description	Data
200 OK	MatchingRequest Backend Working	{ message: “OK” }

Get MatchingRequest by ID - GET

Endpoint: http://localhost:5002/api/matchingRequest/:id
Description: Endpoint to retrieve MatchingRequest corresponding to provided id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	MatchingRequest retrieved successfully.	MatchingRequest
400 BAD REQUEST	Invalid params.	Error Message

Get all MatchingRequests - GET

Endpoint: http://localhost:5002/api/matchingRequest

Description: Endpoint to retrieve all MatchingRequests, if optional id is passed, it retrieves MatchingRequest by id instead.

Request		
Property	Optional/Required	Data
Query	Optional	id

Response		
Status	Description	Data
200 OK	MatchingRequests retrieved successfully.	List of MatchingRequests
400 BAD REQUEST	Invalid params.	Error message

Create MatchingRequest - POST

Endpoint: http://localhost:5002/api/matchingRequest
Description: Endpoint to create a MatchingRequest.

Request		
Property	Optional/Required	Data
Body	Required	userId, difficulty
	Optional	questionId

Response		
Status	Description	Data
200 OK	MatchingRequest created successfully.	MatchingRequest
400 BAD REQUEST	Invalid params.	Error Message

Update MatchingRequest - PUT

Endpoint: http://localhost:5002/api/matchingRequest/:id
Description: Endpoint to update a MatchingRequest having the corresponding id.

Request		
Property	Optional/Required	Data
Param	Required	id
Body	Optional	userId, questionId, difficulty, success

Response		
Status	Description	Data
200 OK	MatchingRequest updated successfully.	MatchingRequest
400 BAD REQUEST	Invalid params.	Error message

Delete MatchingRequest - DELETE

Endpoint: http://localhost:5002/api/matchingRequest/:id
Description: Endpoint to delete a MatchingRequest having the given id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	MatchingRequest deleted successfully.	MatchingRequest
400 BAD REQUEST	Invalid params.	Error message

7.4 History Service

Health Check- GET

Endpoint: http://localhost:5007/api/history/healthCheck
Description: Endpoint to verify History Service is accessible.

Response		
Status	Description	Data
200 OK	History Service Working	{ message: “OK” }

Get History by ID - GET

Endpoint: http://localhost:5007/api/history/:id
Description: Endpoint to retrieve history matching provided id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	History retrieved successfully.	History
400 BAD REQUEST	Invalid params.	-

Get all History - GET

Endpoint: http://localhost:5007/api/history?
Description: Endpoint to retrieve history matching provided query.

Request		
Property	Optional/Required	Data
Query	Optional	id, questionId, user1Id, user2Id, language, code

Response		
Status	Description	Data
200 OK	History retrieved successfully.	History
400 BAD	Invalid params.	-

REQUEST		

Create History - POST

Endpoint: http://localhost:5007/api/history
Description: Endpoint to create a history.

Request		
Property	Optional/Required	Data
Body	Required	questionId, user1Id, user2Id, language, code

Response		
Status	Description	Data
200 OK	History created successfully.	History
400 BAD REQUEST	Invalid params.	-

Update History - PUT

Endpoint: http://localhost:5007/api/history/:id
Description: Endpoint to update a history matching the given id.

Request		
Property	Optional/Required	Data
Param	Required	id
Body	Optional	language, code

Response		
Status	Description	Data
200 OK	History updated successfully.	History

400 BAD REQUEST	Invalid params.	-

Delete History - DELETE

Endpoint: http://localhost:5007/api/history/:id
Description: Endpoint to delete a history matching the given id.

Request		
Property	Optional/Required	Data
Param	Required	id

Response		
Status	Description	Data
200 OK	History deleted successfully.	History
400 BAD REQUEST	Invalid params.	-

8. Nice to haves

N1 - Communication

We have implemented a chat function that allows for real-time communication between users working on a question. We achieved this by creating the microservice Chat Service.

N2 - History

We have implemented a history functionality that allows users to view their post work. We achieved this by creating the microservice History Service.

N3 - Code execution

Our application allows our users to run their code directly on the browser. This was achieved by implementing Judge0 in our frontend. Judge0 is a widely used code execution platform with a free subscription service that allows free API calls up to a certain daily threshold.

Our rationale for using Judge0 is due to the fact that it can not only support different languages, it also allows us to implement security functions that allow the code to run separately from the host. Although this has not been implemented, we believe it to be necessary for future development.

N4 - Question Enhancement

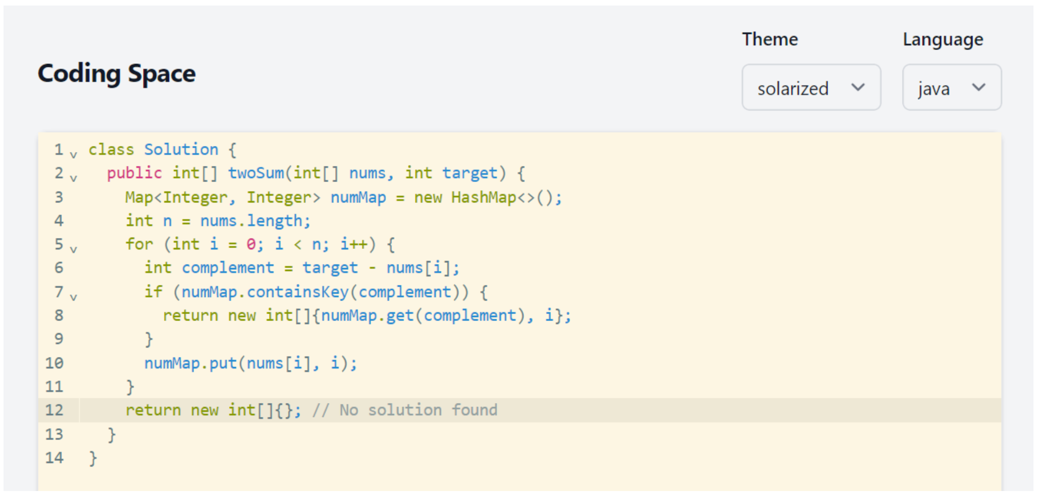
To enhance our question service, firstly we added options for Searching, Filtering and Sorting through the question repository which will make it easier for users to view the question they want. Users can search for a specific question or use keywords to find multiple questions. Users can add question categories like “Strings” or “Algorithms” to the filter, or filter by difficulties: “Easy”, “Medium” or “Hard”. Lastly Users can sort the list of questions by parameters like Title or Popularity.

We added the Popularity Tag to questions which reflect how many times the question has been attempted by all users. Lastly, we added Solutions and Test Cases for all questions, which will allow users to verify if their answers are correct and check the suggested solutions if they are stuck on a question.

N5 - Syntax Highlighting

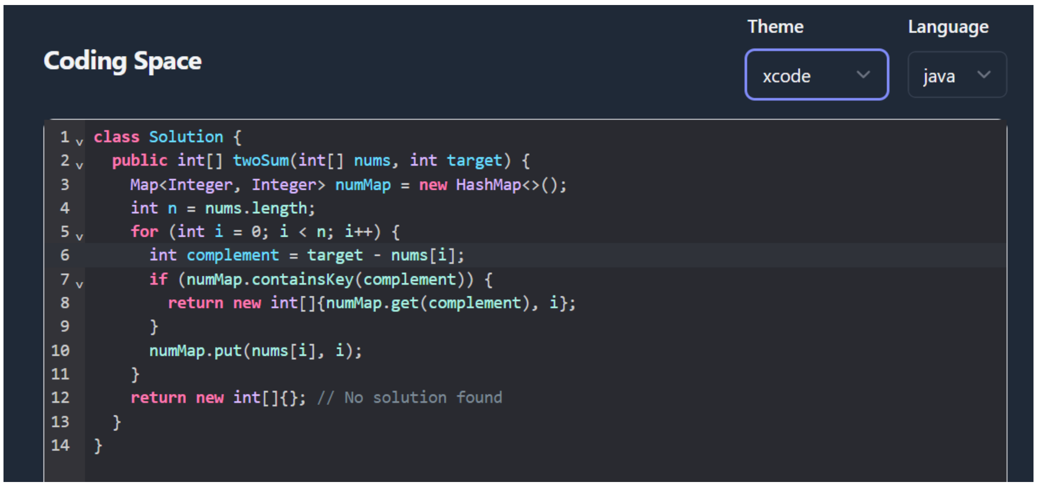
We have implemented syntax highlighting in the user’s code environment by implementing CodeMirror as our code editor. We chose code mirror as it provides additional functionalities on top of providing syntax highlighting, such as code folding and auto-complete which we believe would improve the user’s coding experience on our application.

In addition, we have added multiple themes that users can choose from to suit their editor preferences. Uses can also toggle between dark and light mode on the application on top of the theme selector dropdown which further changes the look of the current selected theme, leading to more editor options for users.



```java
class Solution {
  public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> numMap = new HashMap<>();
    int n = nums.length;
    for (int i = 0; i < n; i++) {
      int complement = target - nums[i];
      if (numMap.containsKey(complement)) {
        return new int[]{numMap.get(complement), i};
      }
      numMap.put(nums[i], i);
    }
    return new int[]{}; // No solution found
  }
}
```

Light mode editor theme example



```java
class Solution {
  public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> numMap = new HashMap<>();
    int n = nums.length;
    for (int i = 0; i < n; i++) {
      int complement = target - nums[i];
      if (numMap.containsKey(complement)) {
        return new int[]{numMap.get(complement), i};
      }
      numMap.put(nums[i], i);
    }
    return new int[]{}; // No solution found
  }
}
```

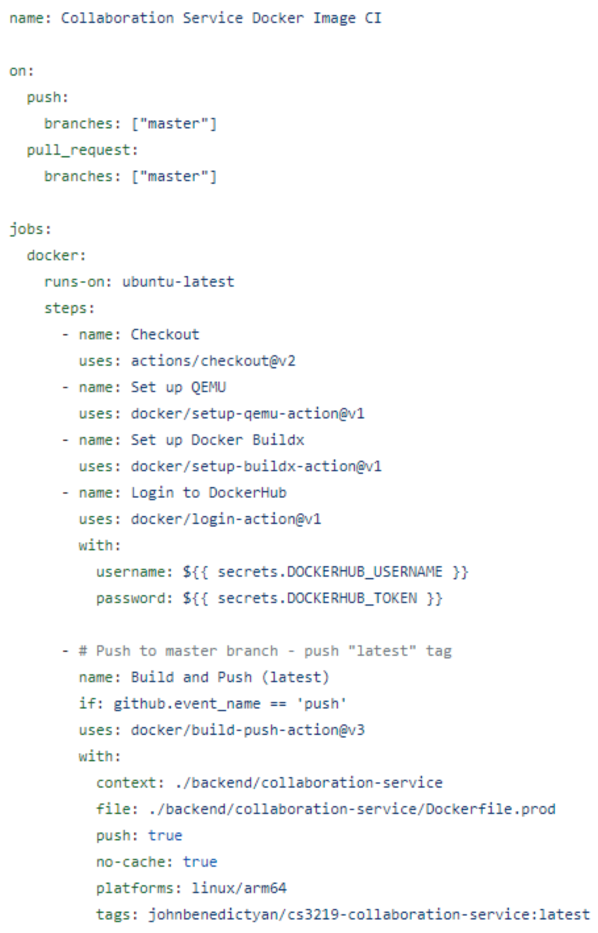
Dark mode editor theme example

N8 - Testing

For testing, we have implemented jest unit testing for all our microservices API, we have achieved good code coverage and have mocked various 3rd party libraries when needed. We have integrated these testing workflows with a CI/CD Pipeline using GitHub Actions, when code is pushed or merged to the master brush, test are run automatically to ensure that code quality remains high and no newly written code causes existing functionality to fail.

N9 - Deploy on Cloud

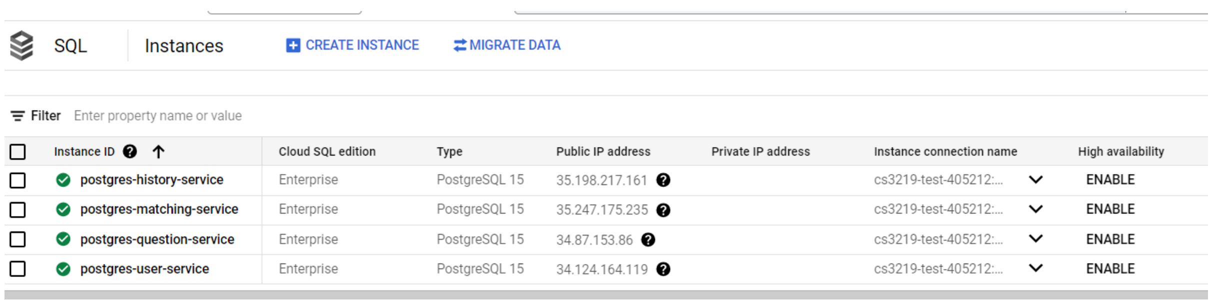
We have implemented a pipeline for deployment, utilising both Github actions and Google Cloud Run. We first implemented Github workflows for each microservice, by which a change in the repository in that microservice will result in an automatic update of the corresponding microservice image on DockerHub. This ensures that images on DockerHub would reflect the latest updates made by the latest push or pull on the master branch.



```
name: Collaboration Service Docker Image CI

on:
  push:
    branches: ["master"]
  pull_request:
    branches: ["master"]

jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Set up QEMU
        uses: docker/setup-qemu-action@v1
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - # Push to master branch - push "latest" tag
        name: Build and Push (latest)
        if: github.event_name == 'push'
        uses: docker/build-push-action@v3
        with:
          context: ./backend/collaboration-service
          file: ./backend/collaboration-service/Dockerfile.prod
          push: true
          no-cache: true
          platforms: linux/arm64
          tags: johnbenedictyan/cs3219-collaboration-service:latest
```

Example workflow for the Collaboration Service

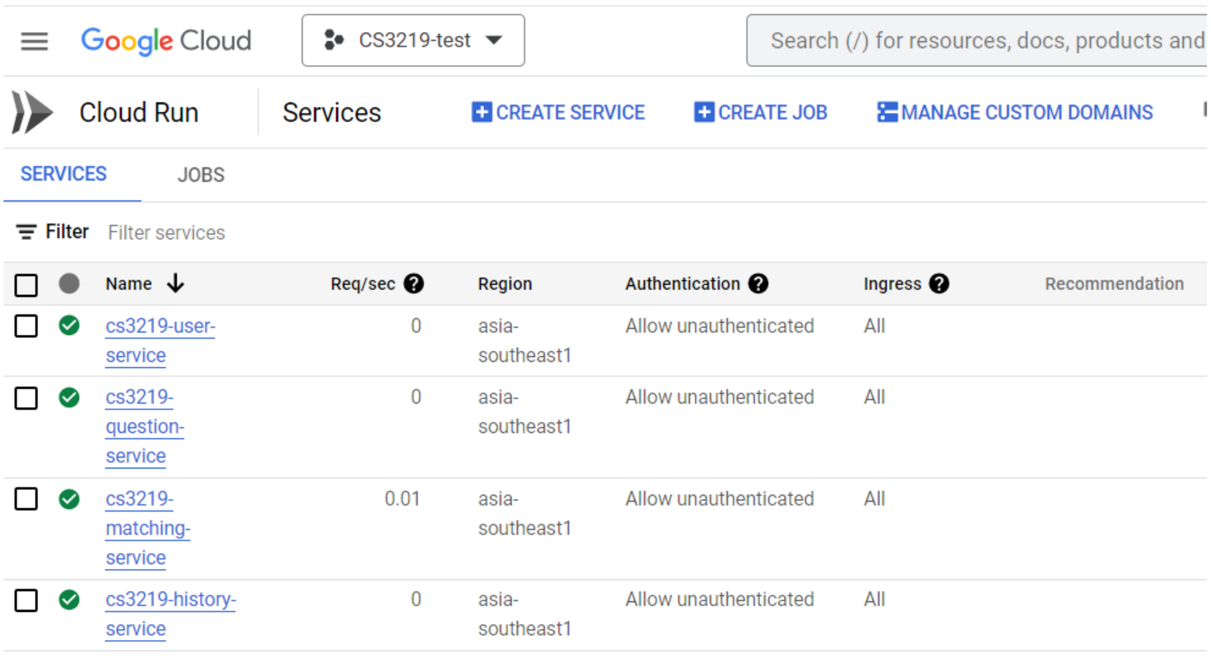
Next, an ecosystem of Google services such as Google Cloud Run and Google Cloud SQL are utilised to deploy the application. We first set up Google Cloud SQL instance for each of the microservice database, which would aid in deploying the service instance later on. Google Cloud SQL is chosen as our preferred Cloud-based storage system due to its ability to ensure high-availability and resilience due to features such as automated backups and failover mechanisms, which is thus a scalable and robust option.



Instance ID	Cloud SQL edition	Type	Public IP address	Private IP address	Instance connection name	High availability
postgres-history-service	Enterprise	PostgreSQL 15	35.198.217.161		cs3219-test-405212:...	ENABLE
postgres-matching-service	Enterprise	PostgreSQL 15	35.247.175.235		cs3219-test-405212:...	ENABLE
postgres-question-service	Enterprise	PostgreSQL 15	34.87.153.86		cs3219-test-405212:...	ENABLE
postgres-user-service	Enterprise	PostgreSQL 15	34.124.164.119		cs3219-test-405212:...	ENABLE

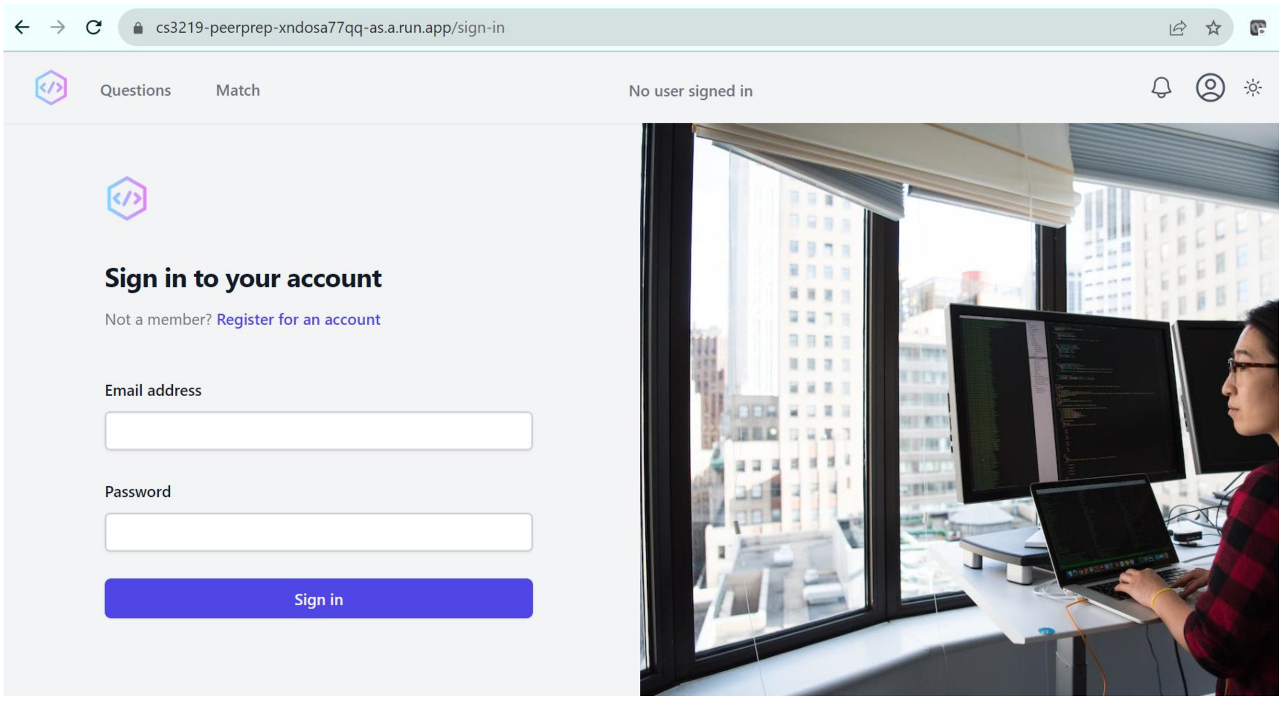
Multiple Google Cloud SQL instances

After the setting up of the Google Cloud SQL instances, each service instance corresponding to a microservice is then created. Each service is configured to pull the latest DockerHub image corresponding to the related microservice and is also configured with a related Cloud SQL instance if the service depends on the instance. On top of this, service instances are also configured with their relevant secrets, port numbers as well as liveness checks before deployment.



		Name	Req/sec	Region	Authentication	Ingress	Recommendation
☐	✓	cs3219-user-service	0	asia-southeast1	Allow unauthenticated	All	
☐	✓	cs3219-question-service	0	asia-southeast1	Allow unauthenticated	All	
☐	✓	cs3219-matching-service	0.01	asia-southeast1	Allow unauthenticated	All	
☐	✓	cs3219-history-service	0	asia-southeast1	Allow unauthenticated	All	

Deployed service instances


cs3219-peerprep-xndosa77qq-as.a.run.app/sign-in
Questions
Match
No user signed in
Sign in to your account
Not a member? Register for an account
Email address
Password
Sign in

Deployed Application

9. Future Improvements & Enhancements

9.1 Chat Service

Instead of having the frontend trigger events to communicate with the backend chat services, the frontend should instead initiate a room joining request through API endpoints. The backend will subsequently establish a client socket based on this request and send back the response.

This approach reduces the workload on the frontend and simplifies the communication process. During the chat phase, this approach will minimise the coupling between the frontend and backend, as communication no longer relies on direct end-to-end communication between their sockets.

9.2 User Service

9.2.1 User Login

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

- Forget password
- Custom user profile upload
- Remember me

Additionally, to encourage adoption, we could implement Firebase integrations for social accounts such as Gmail, GitHub, or LinkedIn login. This will help users create accounts more easily.

8.2.2 User reward system

To encourage repeat users, we can implement the following to give users a sense of progress and achievement:

- Achievement Badges
- Points
- Leaderboard
- Peer review system (to encourage healthy behaviour and punish toxic behaviour)

9.3 Matching Service

9.3.1 Optimised matching

Currently, the matching service only matches users based on their questions. In the future, we hope to be able to match users based on their skill level.

9.3.2 Direct matching

Additionally we want to allow users to collaborate with specific users (such as their friends). Users can generate a link and send their friends the link. Other users who go to the link address will enter the same room as the user who created the link and can immediately start working on the question.

9.4 Question Service

9.4.1 Select Question Collaboration

Currently, users can only collaborate by selecting random questions of a specified difficulty.
In the future, we hope that we can allow users to select specific questions to collaborate with others.

10. Reflections & Learning Points

10.1 Challenges faced

The nature of this project required us to learn new technologies and to implement them to create a viable product over the course of a semester, where we had to also juggle our schoolwork and commitments. This posed some challenges as some of us were not familiar with certain technologies.

Project management

The management of the project workflow could be improved by using tools such as Github Issues and Github milestones. By pairing the Functional requirements with the milestones, we could improve the pacing of the project.