National University of Singapore

School of Computing

CS3219 Software Engineering Principles and Patterns

AY2022/2023 Semester 1

Project Report

Group 6

Team Members	**Student Number**	**Email**
Xie Yaoren	A0200116E	xie.yaoren@u.nus.edu
Yang Xiquan	A0211233B	yangx@u.nus.edu
Yu Shufan	A0211253X	shufan_yu@u.nus.edu
Zhang Sheng Yang	A0200498E	e0407479@u.nus.edu

Table of Contents

Overview

PeerPrep is a web application that helps students prepare for technical interviews. It allows users to match with one another and work on a coding question together.

There are multiple services built together to support this app, including user service, matching service, question service, editor service, and chat service. User service provides functionalities including user authentication, login, logout, signup, account deleting and password changing. Matching enables matching with another user based on certain criteria, such as question difficulty. Question service is a question bank in which it loads and stores questions of different difficulty levels. Editor service allows for concurrent editing. Chat service lets matched users communicate with each other.

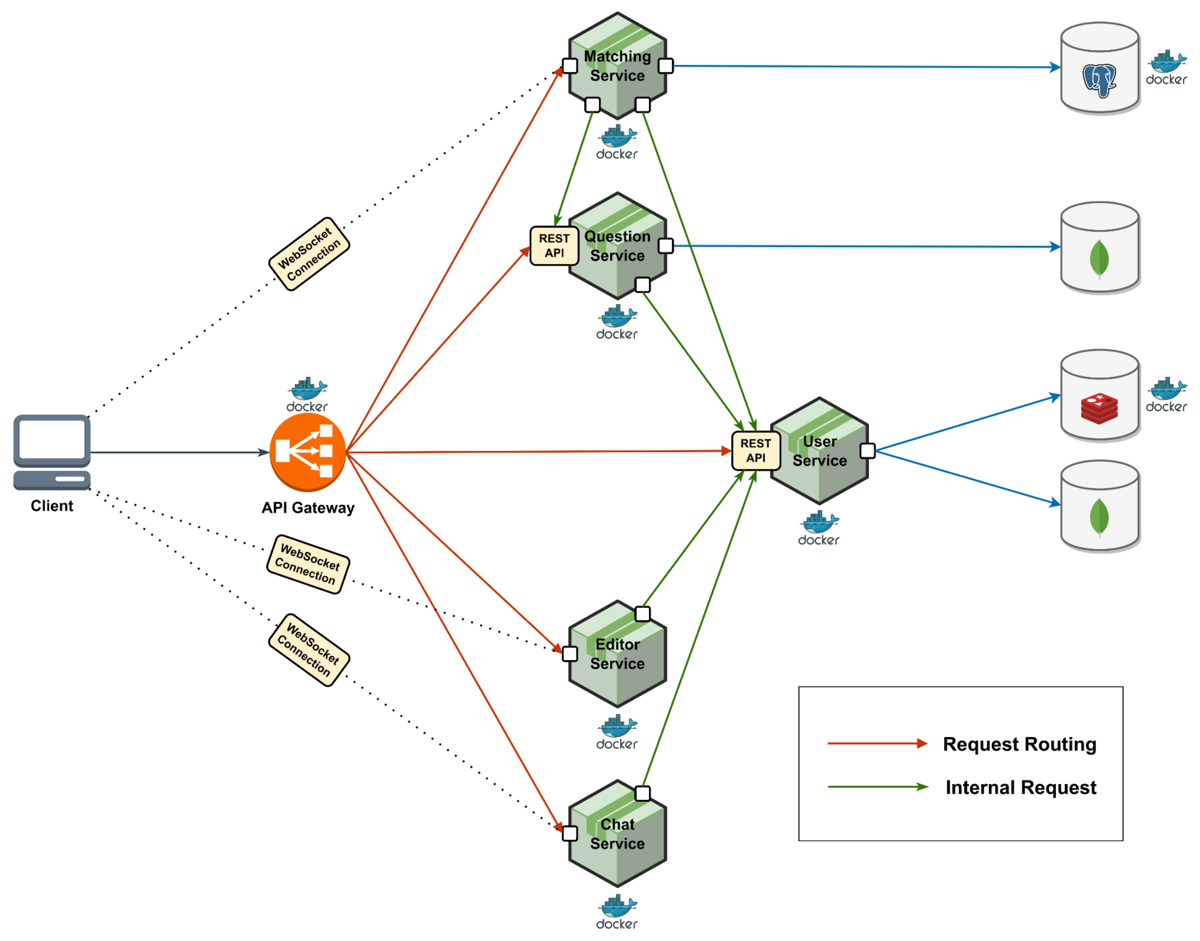
A typical flow of using the app would be as follows:

Users should log in before accessing other services. New users can sign up for a new account to access our service. Existing users have the option to change password if they decide to change passwords or forget their passwords. The button is also present at the login page. Upon login, users are directed to the matching page, where they can select a difficulty level and match with another user. Upon a successful match, the two users will enter the room page where the same coding question will be shown to both users and they can collaboratively work on the question in the textbox, and changes will be updated concurrently to both users. They can also chat with each other at the bottom of the page.

Besides, authentication is done before loading the pages. Users that have not logged in are not allowed to access functions such as matching and working on a question.

System Architecture and Overview

Architecture Diagram



Microservice Architecture

PeerPrep uses the microservice architecture instead of the traditional monolithic architecture. The system's backend is separated into five independently deployable services: user service, matching service, question service, editor service, and chat service.

User service: Responsible for maintaining information about users of PeerPrep.

Matching service: Responsible for matching two users, assigning them to a new session with a question, and ending a session.

Question service: Responsible for storing questions of different difficulty levels and returning a question upon request.

Editor service: Responsible for synchronising the code editor of both users in a session (i.e., providing concurrent code editing).

Chat service: Responsible for providing real-time text transmission between the two users in a session.

Using the microservice architecture brings a few benefits to the project:

1. Each microservice is concerned about only a small set of business concerns, which follows the Separation of Concerns principle.
2. It improves the development productivity when each microservice can use different tool stacks and be developed completely independently by 1-2 team members. For example, the developers of user service do not need to know or learn about Socket.IO, which is used by some other services.
3. When introducing a new microservice, there is no need to redeploy the entire system. Also, although not implemented, individual microservice can be scaled independently. For example, in the future, when the matching service is likely to require more capacity than the user service, we will not need to scale up the entire monolith but only the matching service. This will reduce the infrastructure cost a lot.

Design decisions

We have in mind two main architecture designs, the monolithic and microservice architecture. After much consideration about their pros and cons as shown in the table below, we decided to choose the microservice architecture.

	Monolithic	Microservice
Ease of Deployment	Easier because it has one executable file or directory.	Harder as we need to deploy different services/directories separately.
Ease of Development	Easier in terms of only one code base. No need to handle multiple code bases.	Harder as we need to handle different code bases concurrently.
Speed of Development	Harder because it eventually evolves into a large and complex code base, which makes it harder to develop. Moreover, parallel development requires careful planning.	Easier as the code base for each microservice remains relatively small. Also, parallel development is easy by assigning 1-2 team members to each microservice. This makes the overall development faster.
Scalability	Impossible to scale individual components.	Able to scale each component independently.
Reliability	Not very reliable as if there’s an error in any component or module,	Reliable as each service would be relatively independent and if one

	it could affect the entire application’s availability.	service is down, we can ensure other services are still available.
Technology Adoption	Harder to adopt a new technology as any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.	Easier to adopt a new technology as each component is independent. It’s possible to change the technology of one component without affecting others.

Although the monolithic architecture has some benefits, we still prefer to use the microservice architecture because the ease of development and deployment brought by the monolithic architecture can be easily achieved by the parallel work split when using the microservice architecture. Moreover, the microservice architecture also brings many other benefits, such as the scalability, reliability, and flexibility of technology adoption.

Component Communication and Interaction

Communication between different components is an important aspect for the microservice architecture.

At a high level, the client communicates with the user service and question service via HTTP requests, and communicates with the matching service, editor service, and chat service using WebSockets. WebSockets are used to allow for full-duplex bidirectional communication between the client and the server. This is necessary for features such as matching users and concurrent code editing to work, where the server needs to send or distribute data to the client(s) asynchronously.

All HTTP requests and the initial WebSocket connection requests are firstly sent to the API gateway, which serves as a unified entry point to the backend’s internal APIs. The API gateway will then route a request to the appropriate microservice. By following the API Gateway pattern, the client does not need to be aware of all different microservices and their locations. This has simplified the frontend development a lot. The backend’s internal design is also hidden so that the application is more secure!

There are also internal communications between microservices, but those are minimised to reduce coupling. Currently, user authentication is performed inside each microservice by sending a request to the user service. Ideally, this should be handled by the API gateway before passing the requests to specific microservices.

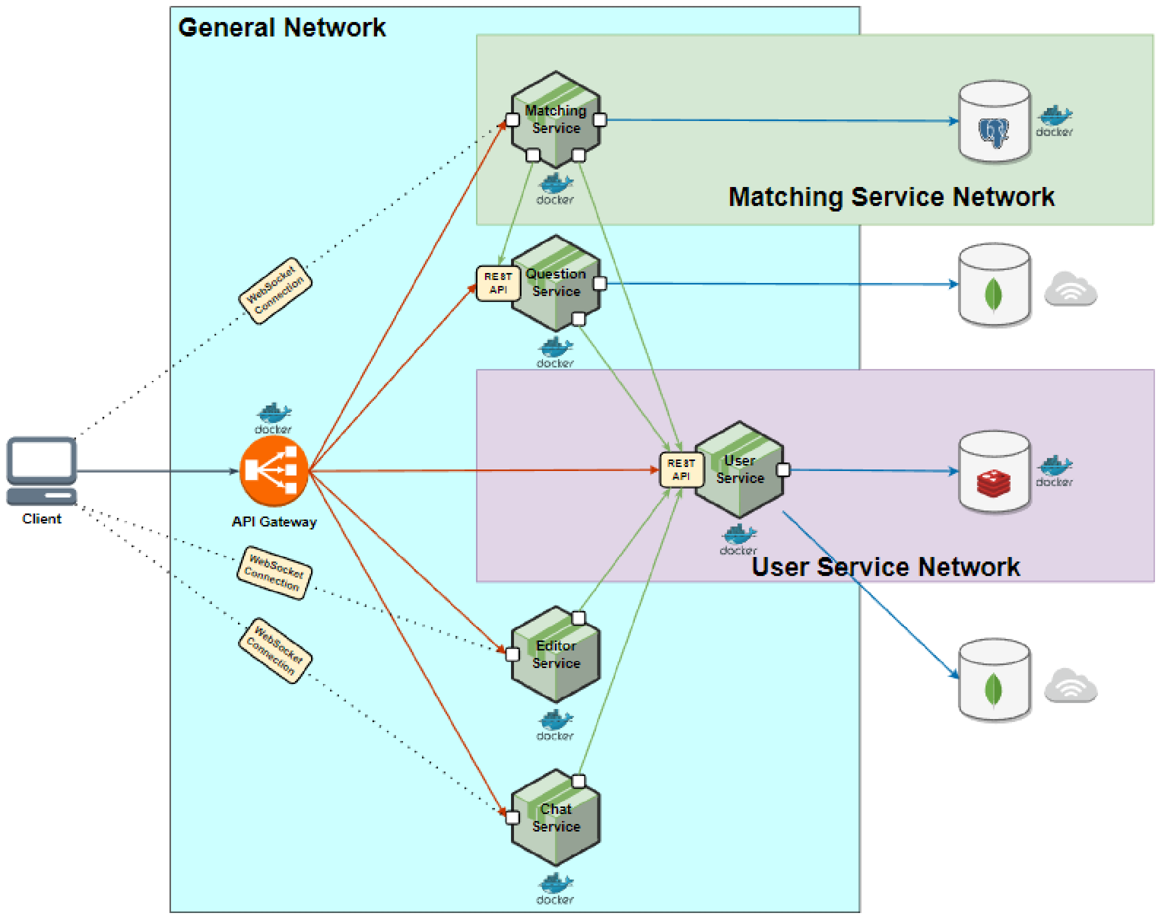
Deployment Method and Service Discovery/Registry

For our project deployment, we chose Docker + Docker-compose for our application deployment.

A few reason why we came to this decision:

- Containerizing each microservice with docker allows us to deploy the services without worrying too much about the environment and configurations. For instance, Redis requires Linux environment while most people use Windows, if anyone wants to deploy and try out our wonderful application, he/she may have to download a WSL and install Redis in order to run it)
- If we want to have higher scalability in the future using kubernetes, we can use these docker images for deployments with the microservices already configured.
- Docker containers are very easy to manage (e.g., using Docker Desktop) and we can easily monitor the status of the services and reboot whenever necessary.
- Ease of deployment. With docker-compose, we can deploy the entire application with all the microservices and databases required in less than a minute.
- Usage of docker network allows us to group services together in a network and exclude those that are theoretically not accessible or irrelevant.

The following figure illustrates how our docker networks are configured.



Networks:

- General Network(pp-g6): A network that includes all backend microservices and an API gateway.
- User Service Network(pp-user-service-network): A network that includes the User Service and its Redis instance.
- Matching Service Network(pp-matching-service-network): A network that includes the Matching Service and its postgres database.

Due to the nature of the docker network, when a container joins a network, the container's assigned IP address and hostname will be registered in the docker network table for this particular network. Then, the container will be able to discover other containers that are registered in the network with the help of the embedded DNS server of docker. Furthermore, as long as the containers are created with valid names and have joined the network, they are able to communicate with other containers in the network from the inside.

As the IP addresses for the containers created are dynamically allocated to them upon creation, services are basically inaccessible without knowing the hostname. With this in mind, we can isolate containers or services from other containers or services that should not have access to them. For example, the User Service is in the same network as the Redis instance so it can access Redis. On the other hand, other services such as Matching Service will not be able to discover the Redis instance though it may know the hostname of the Redis instance. This can also be considered as a security measure to ensure that data storages are safe.

Apart from above mentioned advantages, this feature also allows one to configure an API gateway without hardcoding the proxy passes in the configuration file. The services can simply send the requests to API gateway and it will automatically forward the requests to the corresponding target services based on the given hostname.

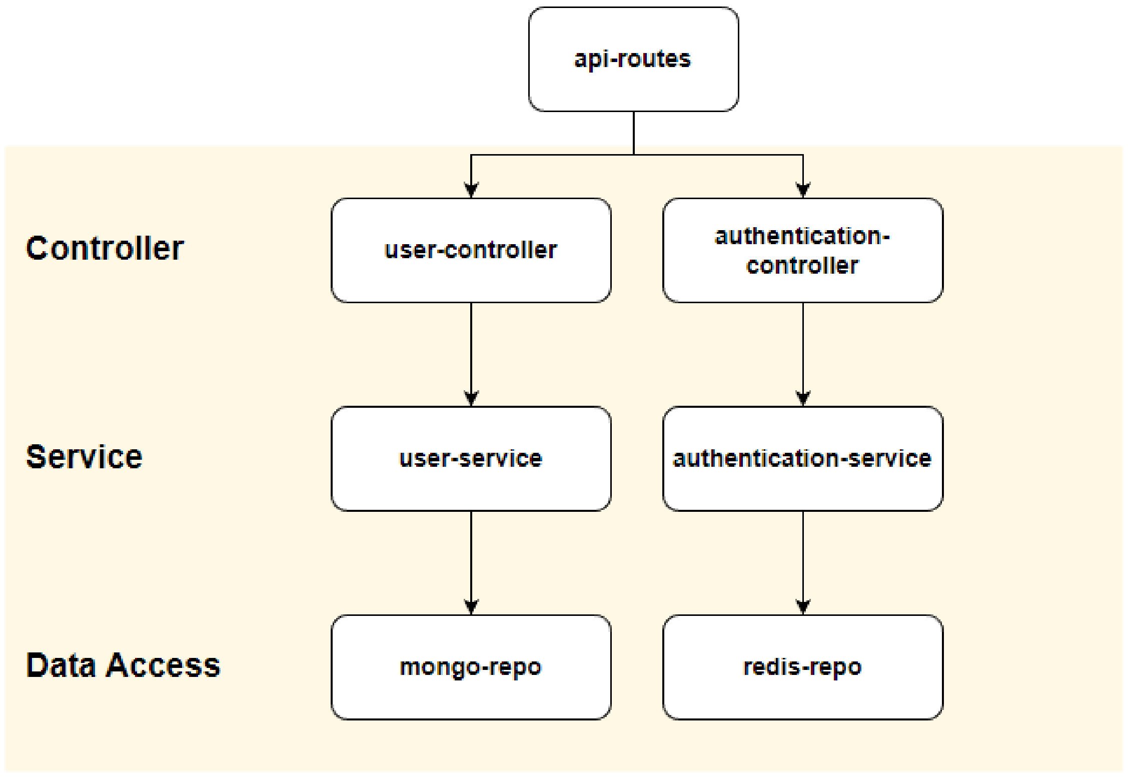
Microservices

User Service

User service component is a component that manages user information and it also provides authentication services to other services that require the user to be logged in before accessing.

Component design

The user service follows a 3-layer architecture:



- **Controller layer:** A layer which is defined by a set of controllers. Each controller will have the definition and logic of handling incoming requests and returning responses to the client. It can also act as middlewares for handling requests and get “chained” together with other middlewares from controllers to form a sequence of logic based on the requirements.

- **Service layer:** A layer that defines and handles business logic needed by the controller layer. Different services have clearly stated boundaries and they will not overlap. For instance, user-service will only contain business logic specific to user services. Even though user services usually come with authentication services, the user service does not do anything authentication related in this case.

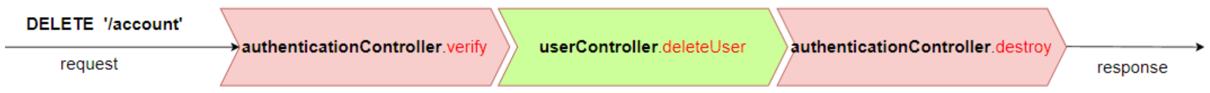
- **Data Access layer:** A layer that is dedicated to interact with the databases. It provides APIs for the service layer which allow services to retrieve data from the database without knowing the implementation details. In our case, the data access object provides "atomic" operations on the database(they are basic actions performed on the database such that they can be easily reused) and the services can be implemented based on a combination of these "atomic" operations.

Using a 3-layer architecture for User Service has the benefit of each layer decoupling from other layers, allowing them to have better maintainability as the size of the project grows. It also allows each layer to be developed and tested independently if there are multiple teams working on this component in the future.

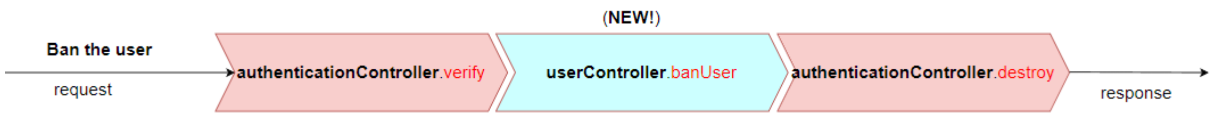
Extensibility and reusability

The User Service can also be extended since middlewares are used for the routes. Each middleware can be seen as a functional module which can be combined with other existing modules to handle newly defined routes.

The following figure illustrates how these middlewares from different controllers can be grouped together to handle a request in the current implementation **[FR1.5]**:


DELETE '/account'
request
authenticationController.verify
userController.deleteUser
authenticationController.destroy
response

Since these middlewares are modularised, we can reuse them to accommodate future requirements:


(NEW!)
Ban the user
request
authenticationController.verify
userController.banUser
authenticationController.destroy
response

The above combination can be implemented simply with one line of code while defining the routes in the router.

Design decisions

1. We are using Redis to cache the generated JSON web tokens with a predefined Time-to-live such that the token can be automatically removed once it expires. We did not choose to, as suggested in the development guide, blacklist tokens once a user logs out and denies logging in with the same token. If we choose to blacklist the

token, the tokens will be exhaustive and records will accumulate in the database simply due to login and logout actions. And as the number of users grows, this can potentially be an issue where users can no longer log in. If any malicious users know about this mechanism, they can log in and log out at a massive scale and invalidate the tokens.

2. We choose to use MongoDB + mongoose with the following considerations:
    a. MongoDB has cloud servers (Mongo Atlas) which are free to use and can be shared among other services as well.
    b. High performance: Due to the NoSQL nature of MongoDB, data retrieval, manipulation and storage can be done at a high speed.
    c. Flexibility: Document-oriented structure of MongoDB allows it to accommodate future changes to the structure of the models.

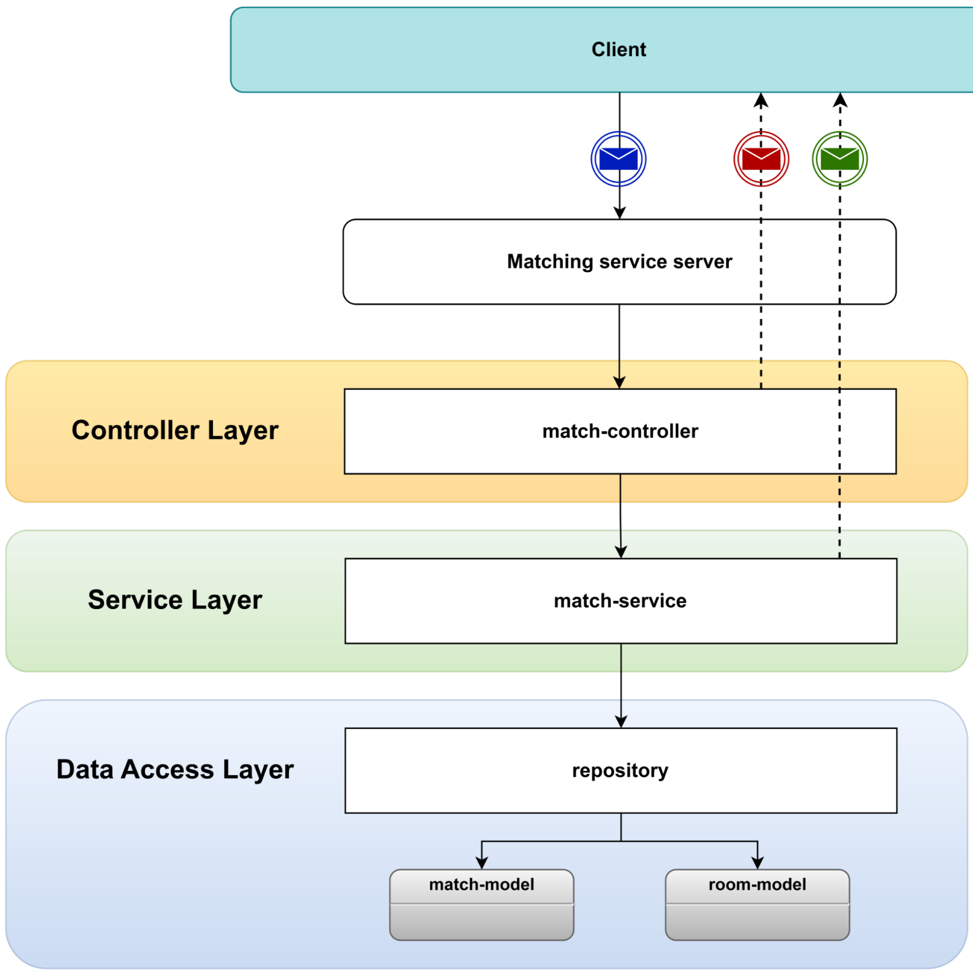
Matching Service

Matching service handles the logic of matching two users, assigning them to a new session with a question, and ending a session.

Component design

Matching service uses the Socket.IO library to have real-time bi-directional communication with the clients. Events with data payload are transmitted between the client and the server.

The code base of the matching service is organised into 3 layers.

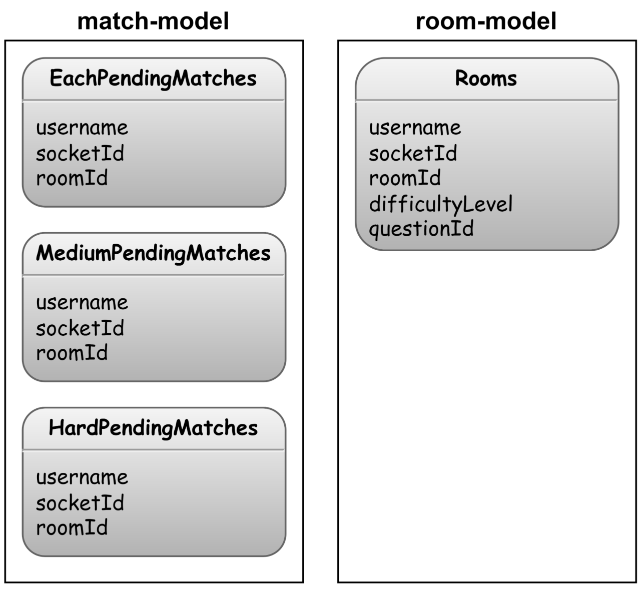


- **Controller layer:** match-controller handles the incoming events and performs basic data validation as well as user authentication.
- **Service layer:** match-service handles the business logic of matching-related events. These include the logic of deciding match success/failure, cleanup when a session ends or when any user disconnects, etc.

- **Data access layer:** This layer interacts with the persistent database and provides match-service with simple APIs to store and retrieve matching-related data (e.g., pending matches, active session info).

Data models

The database of the matching service contains two parts: match-model and room-model.

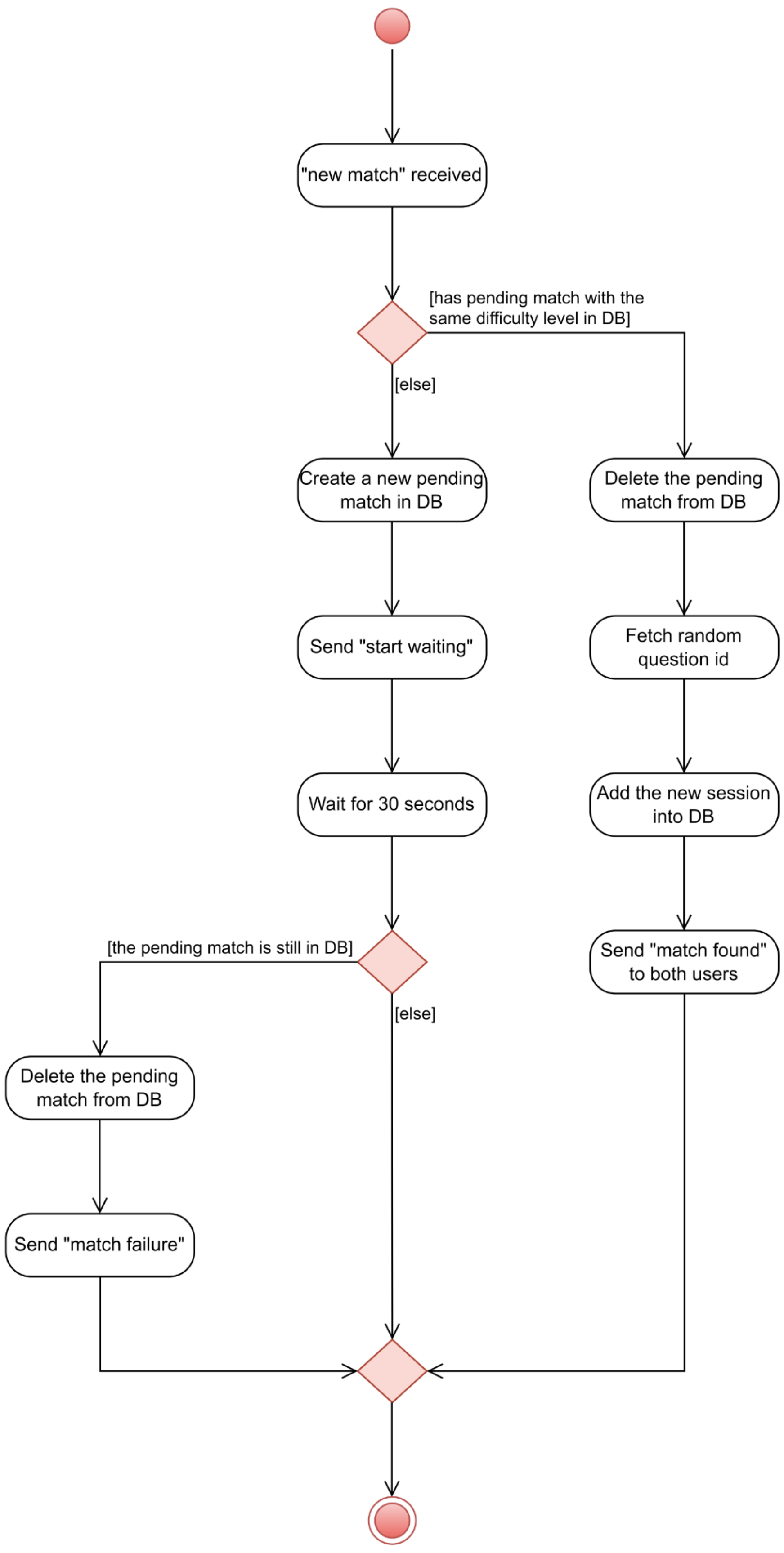


- **match-model:** It consists of three separate tables that store information about all pending matching clients of the corresponding difficulty level. All entries only exist for a short period of time. If the user is successfully matched with another user within the wait time limit, the entry will be removed immediately. Otherwise the entry will be removed once the wait time limit is reached.

- **room-model:** The “Rooms” table stores the information of all currently active sessions. For each session, two entries are expected in the table, corresponding to the two users participating in the session. The two entries are removed when the session ends.

New match handling

The activity diagram below illustrates the logic for handling a new match request.


"new match" received
[has pending match with the same difficulty level in DB]
[else]
Create a new pending match in DB
Delete the pending match from DB
Send "start waiting"
Fetch random question id
Wait for 30 seconds
Add the new session into DB
[the pending match is still in DB]
[else]
Send "match found" to both users
Delete the pending match from DB
Send "match failure"

Design decisions

1. Matching service does not strictly follow the traditional layered architecture, in which only the controller layer is responsible for sending responses to the client. Instead, we decided to pass the socket object from the controller layer to the service layer so that both layers are able to emit events to the client.

   The main reason is that events such as “start waiting” and “match failure” (when no match is found) are asynchronous and should be emitted as the matching algorithm runs. Since the matching algorithm is inside the service layer as part of the business logic, it is difficult to return such messages to the controller layer to emit.

   Although the boundaries of responsibilities are not as clear as the traditional layered architecture, we enforced the distinction between the types of events sent in the controller layer and the service layer. All events sent in the controller layer are error/failure messages (e.g., “match failure” due to invalid request or authentication failure), while all events sent in the service layer are status/success messages (e.g., “match found”, “room closing”).

2. Matching service is responsible for retrieving a random question id from the question service after a successful match, and returns it to both clients (joining the new session).

   The main reason for this decision is to reduce the implementation complexity. An alternative approach here is to shift the responsibility above to the question service, so that the matching service can be totally unaware of the question service. However, a problem to tackle is how to ensure the same random question is sent to two users in the same session. In order to achieve this, the question service will need to request the session info from the matching service and store it in its own database. Then when a GET question request comes in, it needs to determine whether a new random question id should be generated. These lead to duplicate data being stored and extra complexity.

   It is also noteworthy that by making such a decision, essentially the definition of “matching” is extended from “matching two users” to “matching two users with a question”. Also, now the question service can be a pure question bank independent of all other microservices.

Question Service

Question service handles the logic of loading, storing, and retrieving the questions from our storage system. The questions stored contain a variety of tags such as frequency of occurrence and difficulty level

Component design

Matching service uses MongoDB as the storage system.

The code base of the question service is organised into 3 layers.

Controller layer: match-controller handles the incoming events and performs basic data validation before returning the output.

Service layer: match-service handles the business logic of question-related events. These include the algorithm of loading questions and deciding retrieving success/failure etc.

Data access layer: This layer interacts with the persistent database and provides question-service with simple APIs to store and retrieve question-related data (e.g., new question).

Question schema

```
_id: {
  type: String,
  required: true
},
question_title: {
  type: String,
  required: true
},
question_description: {
  type: String,
  required: true
},
question_examples: {
  example_input: {
    type: String,
    required: true
  },
  example_output: {
    type: String,
    required: true
  },
  example_explanation: String
},
question_constrains: {
  question_constrains: {
    type: String,
    required: true
  }
},
has_solution: {
  type: String,
  required: true
},
acceptance: {
  type: String,
  required: true
},
difficulty: {
  type: String,
  required: true
},
frequency: String,
create_date: {
  type: Date,
  default: Date.now
}
```

Design decisions

1. We decided to use MongoDB to store our system because we do not need and may not have fixed the schema required by traditional SQL databases such as MySQL. This is because we might need to store extra fields such as number of likes in future and if we choose SQL, it may requires a huge change in the overall schema and this is definitely costly.

2. Moreover, MongoDB can use BASE properties instead of ACID properties ensured by MySQL. Even though ACID properties can ensure consistency, this will affect performance as well due to the locking mechanism it provides. In this case, we require availability and throughput and there is not much consistency we need to handle in our scenario.

Editor Service & Chat Service

Editor service enables concurrent code editing, and chat service enables real-time messaging between two users in the same session.

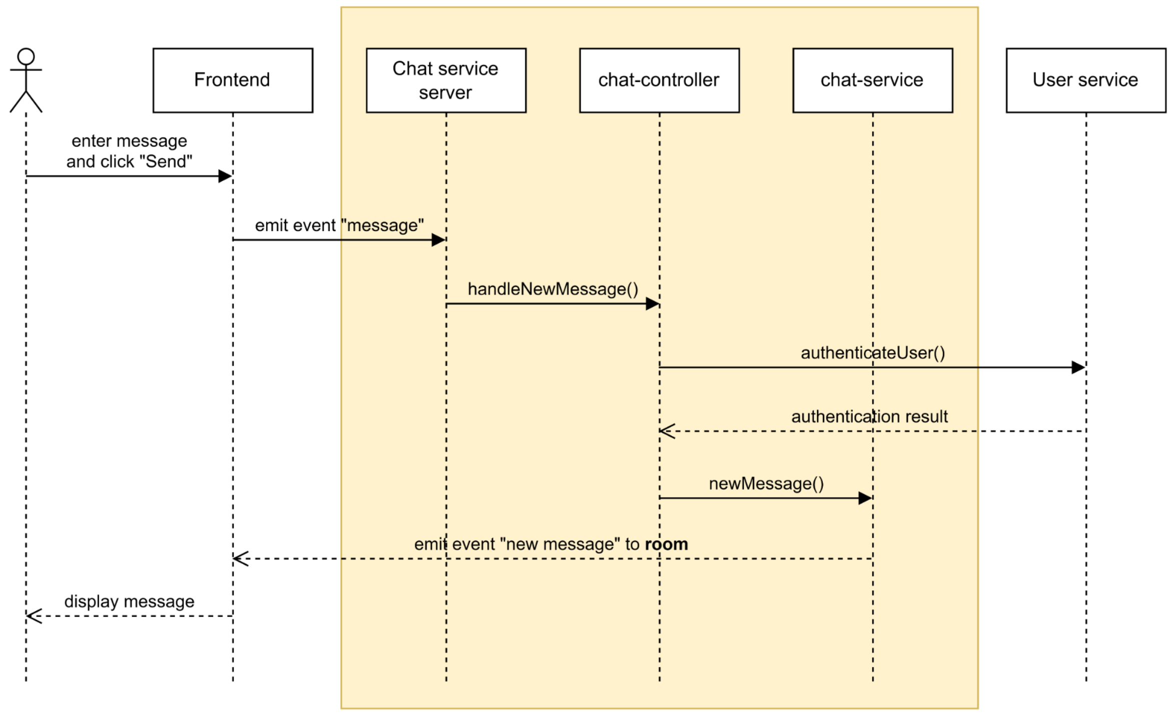
Component design

Similar to the matching service, both services use WebSockets to communicate with the clients. Their structure is also very similar to the matching service but without the data access layer, as they are only responsible for receiving an update from a client and then sending the update to the other client (more precisely, both clients), and do not have any data to persist.

Just like the matching service, both the controller layer and the service layer send events to the clients. Although for these two services, it is not difficult to pass the message back to the controller layer to send or even completely remove the service layer, we want to follow a consistent design and also keep the separation of concerns. This means that the controller layer will only be responsible for validating requests and sending failure messages, and the service layer sends successful status update messages according to the business logic.

Sequence diagram - send a new chat message

The sequence diagram below shows the flow and interaction when sending a new chat message (assuming user authentication succeeds). Making changes in the code editor follows an almost identical process.



FRs and NFRs

Term definition

- Session: When two users are matched, they are assigned to a `session` where they can view the same question, edit answers collaboratively, and chat to each other.

User Service

S/N	Requirement	Priority	Timeline
FR1.1	The system should allow users to create an account with username and password	High	Week 4
FR1.2	The system should ensure that every account created has a unique username	Medium	Week 4
FR1.3	The system should allow users to log into their accounts by entering their username and password	High	Week 5
FR1.4	The system should allow users to log out of their account	High	Week 5
FR1.5	The system should allow users to delete their account	High	Week 5
FR1.6	The system should allow users to change their password	Medium	Week 5
FR1.7	The system should allow users to maintain their login status after login	High	Week 7
NFR1.1	[Security] Users' passwords should be hashed and salted before storing in the DB	Medium	Week 5
NFR1.2	[Security] The system should assign a JWT token to each user upon login	High	Week 7
NFR1.3	[Security] Only logged in users should be able to delete their account or change their password	High	Week 8
NFR1.4	[Security] The system should log out users if they are inactive for 20 minutes	Medium	Week 6

Matching Service

S/N	Requirement	Priority	Timeline
FR2.1	The system should allow users to select the difficulty level of the questions they wish to attempt	High	Week 6
FR2.2	The system should be able to match two waiting users with the same difficulty level and put them in the same room	High	Week 7
FR2.3	If there is a valid match, the system should match users within 30s	High	Week 7
FR2.4	The system should inform users that no match is available if a match cannot be found within 30s	High	Week 7

FR2.5	The system should provide a means for users to leave a room once matched	High	Week 7
FR2.6	If there is a successful match, the system should retrieve a question from the question bank to be shown to both users	High	Week 8
FR2.7	The system should ensure that a user is in most one session at any one time	Medium	Week 10
NFR2.1	[Security] Only logged in users should be able to start a new session	High	Week 8

Question Service

S/N	Requirement	Priority	Timeline
FR3.1	The system should provide an API to add a question into the database	High	Week 8
FR3.2	The system should provide an API to get a random question ID of any specified difficulty level	High	Week 8
FR3.3	The system should display questions to users in the correct format	High	Week 8
FR3.4	The system should provide an API to get the details of a question by ID	High	Week 9
FR3.5	The system should provide an API to delete a question by ID	Medium	Week 9
FR3.6	The system should provide an API to update the details of a question by ID	Medium	Week 9
NFR3.1	The system should have a good variety of questions for each difficulty level	Low	Week 9

Editor Service

S/N	Requirement	Priority	Timeline
FR4.1	The system should allow users to make changes to the answers in the textbox	High	Week 10
FR4.2	The system should update users changes of content in the textbox made by the other user	High	Week 11
NFR4.1	[Performance] Changes made by one user should be seen by both users within 3s	High	Week 11

Chat Service

S/N	Requirement	Priority	Timeline
FR5.1	The system should allow users to key in and send messages	High	Week 11
FR5.2	The system should allow users to receive messages	High	Week 12
FR5.3	The system should display all messages in a session in the sequence of sending time	Medium	Week 12
NFR5.1	[Performance] Messages sent should be received within 3s	High	Week 12

System Level

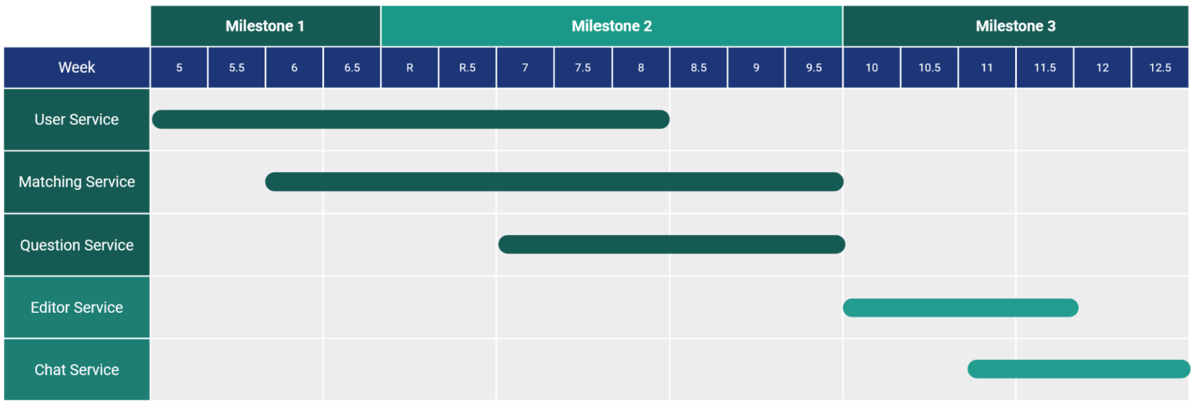
S/N	Requirement	Priority	Timeline
NFR6.1	[Portability] The system should be compatible with popular browsers: Google Chrome, Firefox, Microsoft Edge, Safari	High	NA
NFR6.2	[Maintainability] The system should be easily maintainable - open to changes	Medium	NA
NFR6.3	[Usability] UI should be simply to learn and understand for new users. Users should be able to familiarise themselves with the system functionality within 10 minutes without any tutorials.	Medium	NA

Development Process

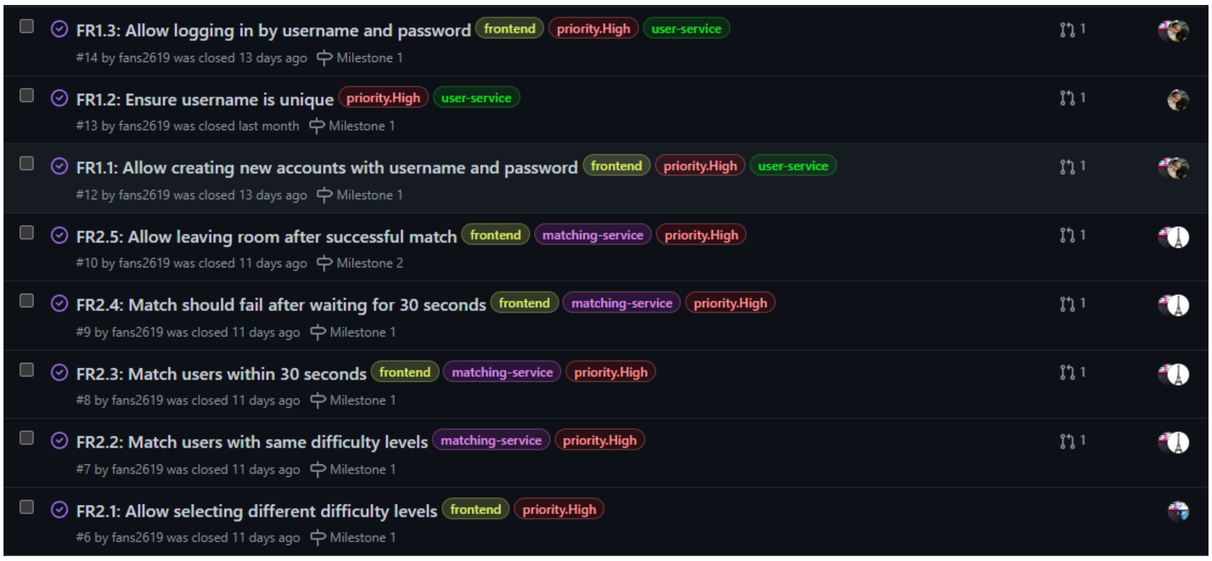
Team meetings were held twice every week to check on each other's progress, make plans for the next week, and discuss design decisions. A group chat was also set up for timely communication between members.

Each service and frontend are assigned to one of the team members so that we make a clear separation of work.

Timeline



During each development phase, we analysed the requirements of the new services and came up with FRs and NFRs. These requirements will be recorded and updated in our Github repository in the form of Github Issues.



FR1.3: Allow logging in by username and password frontend priority.High user-service 1
#14 by fans2619 was closed 13 days ago Milestone 1

FR1.2: Ensure username is unique priority.High user-service 1
#13 by fans2619 was closed last month Milestone 1

FR1.1: Allow creating new accounts with username and password frontend priority.High user-service 1
#12 by fans2619 was closed 13 days ago Milestone 1

FR2.5: Allow leaving room after successful match frontend matching-service priority.High 1
#10 by fans2619 was closed 11 days ago Milestone 2

FR2.4: Match should fail after waiting for 30 seconds frontend matching-service priority.High 1
#9 by fans2619 was closed 11 days ago Milestone 1

FR2.3: Match users within 30 seconds frontend matching-service priority.High 1
#8 by fans2619 was closed 11 days ago Milestone 1

FR2.2: Match users with same difficulty levels matching-service priority.High 1
#7 by fans2619 was closed 11 days ago Milestone 1

FR2.1: Allow selecting different difficulty levels frontend priority.High
#6 by fans2619 was closed 11 days ago Milestone 1

When there are pull requests being created, team members will link these github issues to the pull requests. The issues will be closed on merge after review.

Tools and Tech Stack

We used the following tools and technologies for our project.

Purpose	**Tools and Technologies**
Source Control	Git/Github
Frontend	React, Material UI
User Service	MongoDB, Redis, Nodejs, Express
Matching Service	PostgresQL, Socket.io, Nodejs, Express
Question Service	MongoDB, Nodejs, Express
Editor Service & Char Service	Socket.io, Nodejs, Express
API Gateway	Nginx
Deployment	Docker, Docker-compose

Commonly used throughout the project:

- Axios
- Mongoose
- dotenv
- JSON Web Token
- bcrypt

Areas for Improvements and Potential Enhancement

1. Currently, API gateway simply redirects received requests. Services such as Matching Service requires authentication with accessing it, and it will have to send an authentication request to the User Service through the API gateway. An enhancement will be extracting the authentication logic, as a service by itself, from User Service. The new authentication service can then be a service provider or broker for the API gateway.

2. A better-designed frontend as the current front-end is quite simple with no proper header, navigation bar and footer and more interaction can be developed for users.

3. In future, we can consider supporting more functionalities. For example, for matching services, we can support cooperation among more people. instead of just two people. Moreover, we can also provide a variety of matching criteria based on the question type, question category and frequency. In terms of user service, we can also include more features such as showing their cooperation records and feedback. In terms of chat service, we can support more variety of chat such as video conferencing. In terms of collaboration, we can support more instead of just editing, for example, we can have a whiteboard for users to draw and share their ideas.

Reflections

In terms of product features, we have achieved and implemented all if not most of the required functionalities. However, there is still room for improvement and certainly more features can be added to better enhance the user experience. However, due to time constraints, we cannot achieve all of them.

In terms of development planning, it is slightly unbalanced in which the first few weeks we have lesser development and have more progress at the later stage. This is because we are spending more time picking up the necessary skills in the initial stage. Probably we can split the development more evenly and balance the learning and development phase better.

Individual Contributions

Xie Yaoren	Technical: - Implement matching service - Implement question service Non-technical: - Report: Microservices > Question Service - Report: Areas of improvement - Report: Reflection - Report overall editing
Yang Xiquan	Technical: - Implement frontend Non-technical: - Report: Background - Report: FRs and NFRs - Report: Develop Process
Yu Shufan	Technical: - Implement matching service, editor service, chat service backend - Implement part of frontend Non-technical: - Report: System Architecture and Overview > Architecture Diagram - Report: System Architecture and Overview > Microservice Architecture (excluding Design decision) - Report: System Architecture and Overview > Component Communication and Interaction - Report: Microservices > Matching Service - Report: Microservices > Editor Service & Chat Service
Zhang Sheng Yang	Technical: - Implement User service - Implement API gateway - Docker and Docker-compose deployment Non-technical: - Report: System Architecture and Overview > Deployment Method and Service Discovery/Registry - Report: Microservices > User Service

References

1. https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith#:~:text=A%20monolithic%20application%20is%20built,of%20smaller%2C%20independently%20deployable%20services