CS3219 Group 22

Software Engineering Principles and Patterns

Project Report: PeerPrep

Members:
Eugene Ong Wei Xiang A0233338H
Lim Yu Young, Douglas A0233348E
Pang Kuang Wei A0233489U
Chong Chee Kai Clarence A0233313W
Eric Lee Ying Yao A0230337N

Introduction

Background

Increasingly, students are encountering more challenging technical interview questions when applying for internships or jobs. Issues encountered by students during the interview can range from the inability to comprehend and solve the problem to facing challenges when coding an optimal solution to the problem. There is currently a lack of platforms tailored for students to collaborate with each other on technical interview questions.

Purpose

We aim to develop a technical interview preparation platform, named PeerPrep, with a peer matching system that provides students with the ability to find peers to practise whiteboard-style interview questions together. Students will be able to collaborate with their peers to devise better solutions for the problems and to help each other when they experience difficulties in understanding the question.

Contributions

Subgroup	Member	Technical Contributions	Non-Technical Contributions	Nice-to-haves
1	Eugene Ong Wei Xiang	- Implemented Backend for Matching Service - Containerised Matching Service and Chat Service - Implemented search feature for Question Service - Implemented Frontend UI for Chat Service - Implemented Backend for Chat Service - Implemented serverless function for fetching questions from Leetcode	- Final Report - Assignment 5 Report - Assignment 6 Video	N1, N2, N4
	Lim Yu Young, Douglas	- Implemented Frontend UI for Matching Service - Containerised Frontend, User Service, Question Service, and History Service - Implemented tags for Question Service and search and filter feature for tagging - Implemented Frontend UI for History Service - Implemented Backend for History Service	- Final Report - Assignment 4 Video	
2	Pang Kuang Wei	- Implemented Frontend UI for Login, Register, Profile and Collaboration Pages - Implemented Search User and Edit Profile Modal - Implemented APIs in User Service - Implemented Authentication - Implemented Enhanced Code Editor - Containerised Collaboration Service	- Final Report - Assignment 1 Video	N5, N9, N11
	Chong Chee Kai Clarence	- Implemented Frontend UI for Adding/Editing Question	- Final Report - Assignment 2 Video	

		Modal - Setup User Service and User Service's PostgreSQL Database - Deployment of PeerPrep on Google Cloud Platform		
	Eric Lee Ying Yao	- Implemented Frontend UI for Question Bank, Question Description and Collaboration Pages - Setup Question Service and MongoDB NoSQL Database - Setup Collaboration Service and Collaboration Service's Redis - Implemented Concurrent Code Editing and Enhanced Code Editor - Implemented API gateway using NGINX ingress controller	- Final Report - Assignment 3 Video	

Requirement Specifications

Functional Requirements

These are the functional requirements (FR) we have selected for PeerPrep. The FRs are categorised by services, with one table per service. Each service is also tagged to the corresponding must-have or nice-to-have feature.

User Service

F1 User Service (tagged to M1)	
Functions	**Priority**
F1.1 The system should allow users to register a PeerPrep account	H
F1.1.1 The system should ensure username/email field is not duplicate with existing user account	H
F1.1.2 The system should ensure password field is sufficiently complex	M
F1.2 The system should allow users to login to PeerPrep	H
F1.3 The system should allow users to log out from PeerPrep	H
F1.4 The system should allow users to remain logged in after closing/refreshing the browser tab	M
F1.5 The system should allow users to view their profile	H
F1.6 The system should allow users to edit their profile	H
F1.7 The system should allow users to delete their profile (account)	H
F1.8 The system should allow users to view profiles of other users	L

Matching Service

F2 Matching Service (tagged to M2)	
Functions	**Priority**
F2.1 The system should allow users to specify the criteria for matching	H
F2.1.1 The system should allow question difficulty and category to be part of the criteria	H
F2.1.2 The system should allow question tag to be part of the criteria	L

F2.2 The system should inform users if no questions fit their criteria	H
F2.3 The system should match 2 users with the same criteria	H
F2.3.1 The system should select a question for matched users	H
F2.3.2 The system should select a random question for matched users	M
F2.4 The system should provide some form of visual indicator that users have to wait for the result of the matching	H
F2.4.1 The system should display a timer to show users how long they have been waiting	L
F2.5 The system should inform users that no match can be found if such a match cannot be found in 30 seconds	H
F2.6 The system should prevent users who are already matched from matching again	M
F2.6.1 The system should redirect users who are already matched to their collaborative space	L
F2.7 The system should prevent users from sending multiple match requests	M

Question Service

F3 Question Service (tagged to M3 and N4)	
Functions	**Priority**
F3.1 The system should allow maintainer users to create new questions	H
F3.1.1 The system should prevent new questions, with the same title as an existing question, from being created	H
F3.1.2 The system should allow questions to be created with tags	M
F3.2 The system should allow maintainer users to delete questions	H
F3.3 The system should allow all users to view all questions	H
F3.4 The system should allow maintainer users to modify existing questions	H
F3.4.1 The system should prevent changing a question’s title to the same title as another question	H
F3.4.2 The system should allow tags to be added to a question	M

F3.5 The system should display questions with their relevant tags, categories and difficulty.	M
F3.6 The system should allow users to view the description of each question.	M
F3.7 The system should allow users to sort the questions displayed	M
F3.8 The system should allow users to filter the questions displayed	M
F3.9 The system should allow users to search for a question by title	M
F3.9.1 The system should allow users to search for a question by a tag	M
F3.9.2 The system should allow users to search for a question by multiple tags	L

Collaboration Service

F4 Collaboration Service (tagged to M4 and N5)	
Functions	**Priority**
F4.1 The system should support a collaborative workspace for editing of code	H
F4.2 The system should synchronise changes made by 1 party with the other party's instance of the collaborative workspace	H
F4.3 The system should indicate changes in users connection status	L
F4.3.1 The system should alert users if their partner has disconnected from PeerPrep	M
F4.3.2 The system should alert users that their partner has reconnected to PeerPrep (after disconnecting)	M
F4.4 The system should indicate when one user has ended the session	M
F4.4.1 The system should alert users if their partner has ended the session (purposefully)	M
F4.4.2 The system should allow users to choose between ending the session and continuing the session on their own (after their partner has ended the session)	M

Chat Service

F5 Chat Service (tagged to N1)

Functions	Priority
F5.1 The system should allow users to chat (text-only) with their partners during collaboration	H
F5.1.1 The system should display the names of the partner that the user is chatting with	M
F5.2 The system should allow users to be notified when a new message is received from their partners	M
F5.2.1 The system should display an unread counter to the user that keeps track of the number of unread messages	L
F5.3 The system should differentiate the display of messages received and messages sent to users	M
F5.3.1 The system should use different colours to display messages received and messages sent	M
F5.3.2 The system should display messages sent and messages received in different positions	M
F5.4 The system should allow users to send images to their partners during collaboration	L

History Service

F6 History Service (tagged to N2)	
Functions	**Priority**
F6.1 The system should allow users to view the questions they have attempted in the past	H
F6.2 The system should allow users to view their attempt of the question	M
F6.3 The system should allow users to view the date and time they have attempted the question	M
F6.4 The system should allow users to view the partner user they attempted the question with	M
F6.5 The system should allow users to view the time they took to attempt the question	M
F6.6 The system should display history statistics to the user	M
F6.7 The system should allow users to view the suggested solutions to the questions they have attempted	L

Non-Functional Requirements

These are the non-functional requirements (NFRs) we have selected for PeerPrep. Most of them are external quality attributes that directly impact the user experience (UX) of PeerPrep. Hence, we ranked and prioritised the NFRs according to their importance for PeerPrep.

1) Security: This attribute is often overlooked in software planning and design. Therefore, we prioritised it as rank 1 to avoid overlooking it in PeerPrep.
2) Performance: This attribute is critical as we need to meet the user's expected performance to achieve a positive UX.
3) Usability: This attribute contributes to a user-friendly UX which can boost user adoption.
4) Capacity: This attribute contributes to the long-term UX of PeerPrep as a larger question bank can better satisfy frequent/long-time users.

Security

NFR 1 Security	
Functions	**Priority**
NFR 1.1 Users are unable to bypass Peerprep's authentication to access resources that they do not have permission for	H
NFR 1.2 Passwords should meet the minimum complexity requirements of at least 1 of uppercase, lowercase letter, number and special character.	M
NFR 1.3 Passwords should have a minimum length of 8.	M

For Security, we chose to focus on the security aspect of authentication, to deter unwanted access to PeerPrep.

First, we used JSON Web Tokens (JWT) for authentication. Upon logging in to PeerPrep, users are given a JWT which is stored in their browser's cookies. The JWT encodes the user's information including their role. In the frontend, access to sensitive pages is authenticated via the JWT of the user. In the backend, calls to sensitive REST APIs are also authenticated via the JWT of the user.

Secondly, we also paid attention to the user authentication of PeerPrep. We defined the necessary complexity requirements for user passwords to increase the entropy of user passwords and deter malicious users from brute-forcing other user's passwords. This NFR is related to F1.1.2. Where F1.1.2 checks that the password meets the complexity requirements, this NFR defines the complexity requirement such that it is sufficiently secure.

Performance

NFR 2 Performance	
Functions	**Priority**
NFR 2.1 All button presses except matching should have reasonable response time (i.e 1-5s)	H
NFR 2.2 Matching service should take a reasonable amount of time if a match is possible (i.e within 3s after a match is possible)	H
NFR 2.3 Chat service should allow messages between user to be sent quickly (within < 1s after message is sent)	M
NFR 2.4 Collaboration service should support fast updating of a user’s code when the other matched user stops typing (i.e ~1s).	H

For optimising performance, our focus was on ensuring the responsiveness of the frontend UI to guarantee an enhanced user experience.

For matching, we achieved our objective of displaying a success toast message in green within a defined timing threshold whenever a match was achieved.

For chat functionality, we achieved our goal of enabling messages to be exchanged between users almost instantaneously, facilitating smooth and cohesive communication.

For collaboration functionality, we achieved our goal of displaying the toast message promptly to the user if the other user has left the room, joined the room, changed the language or ended the collaboration. We also achieved fast updating of a user’s code when the other matched user stops typing.

Usability

NFR 3 Usability	
Functions	**Priority**
NFR 3.1 User interface should maintain a consistent look and feel across all modules and screens.	H
NFR 3.2 Interactive components (e.g., navigation menus, buttons) should be standardised throughout the system.	H
NFR 3.2 Clear error messages and feedback to be provided to user when mistakes are made	H

Our goal for Usability was to have a consistent UI as well as providing clear feedback messages to users. This is the reason why we used Tailwind CSS as it promotes a consistent UI design.

For feedback messages, we used toast messages to highlight certain information as well as erroneous usage to users. Toast messages can be both persistent as well as temporary, which provides flexibility for the various feedbacks we provide to users. Examples of such toasts can be found in Matching Frontend.

These decisions aim to ensure that PeerPrep is user-friendly and easy to adopt.

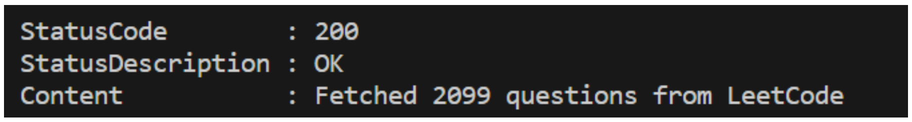
Capacity

NFR 4 Capacity	
Functions	**Priority**
NFR 4.1 Able to store at least 2000 questions	M

For Capacity, we chose to focus on the capacity of questions as we felt that the main attraction of PeerPrep would be the questions that users can practise. Therefore, the more questions we have, the more attractive PeerPrep would be to our target users.

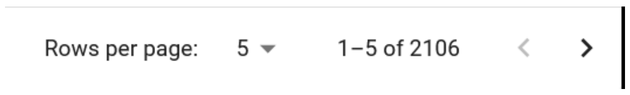
We selected the benchmark of 2000 questions for PeerPrep. Given that Leetcode, another technical interview preparation platform, has ~2354 non-premium questions, being able to store 2000 questions would make PeerPrep comparable in terms of number of questions.

To test this, we used a serverless function (shown in Figure 1) to fetch ~2000 questions from Leetcode into PeerPrep. As shown in Figure 2, PrepPeep can successfully contain at least 2000 questions.



```
StatusCode        : 200
StatusDescription : OK
Content           : Fetched 2099 questions from LeetCode
```

Figure 1: Using serverless function to fetch 2099 questions from Leetcode



Rows per page: 5 1–5 of 2106

Figure 2: PeerPrep with 2106 questions

Traceability

FR	Frontend	Backend
F1 User	Login, Register, Profile	User Service
F2 Matching	Matching	Matching Service
F3 Question	Collaboration	Question Service
F4 Collaboration	Question	Collaboration Service
F5 Chat	Chat	Chat Service
F6 History	History	History Service

Developer Documentation

Development Process

Scrum Framework

We made use of the Scrum framework for our development process during this project with each sprint lasting 2 weeks. At the beginning of each sprint, our group conducts sprint planning to establish the sprint goal, the sprint backlog, as well as the assignment of work. Typically, the sprint goal would involve the implementation of 1 to 2 services/nice-to-haves/must-haves. At the end of each sprint, we conduct a sprint review to discuss the progress that has been made in achieving the sprint goal as well as the challenges encountered. At the end of each sprint, a working prototype of PeerPrep is produced with the results of the sprint integrated into the application.

Branching Workflow

Each new feature of PeerPrep is implemented on a separate branch from the master branch. Code is never pushed directly to the master branch. Branching off to new features provides a form of encapsulation where we work independently on each of our features without affecting each other's code. Moreover, if there are bugs present in the code, they can be isolated and fixed in the branch before being merged into the main branch. This ensures that the master branch always has a tested and deployable working product.

Pull Requests

We also leveraged pull requests(PR) for code reviews on each branch that is to be merged with the master branch. For each PR, the code cannot be merged to the master branch until it has been approved. After the PR is ready, one team member has to review and test the code implemented to ensure that it works as intended and is able to integrate well with other features. Branch protection is used to enforce this behaviour. Once the PR has been approved by one of the members, it will be merged into the main branch.

Architecture: Microservices

We have split PeerPrep into the following microservices:

Service	**Description**
User Service	Responsible for user profiles
Matching Service	Responsible for matching users by a criteria
Question Service	Responsible for the question repository
Collaboration Service	Responsible for real-time collaboration between 2 matched users
Chat Service	Responsible for text-based communication between users in the collaborative space
History Service	Responsible for user collaboration history

Architecture Diagram

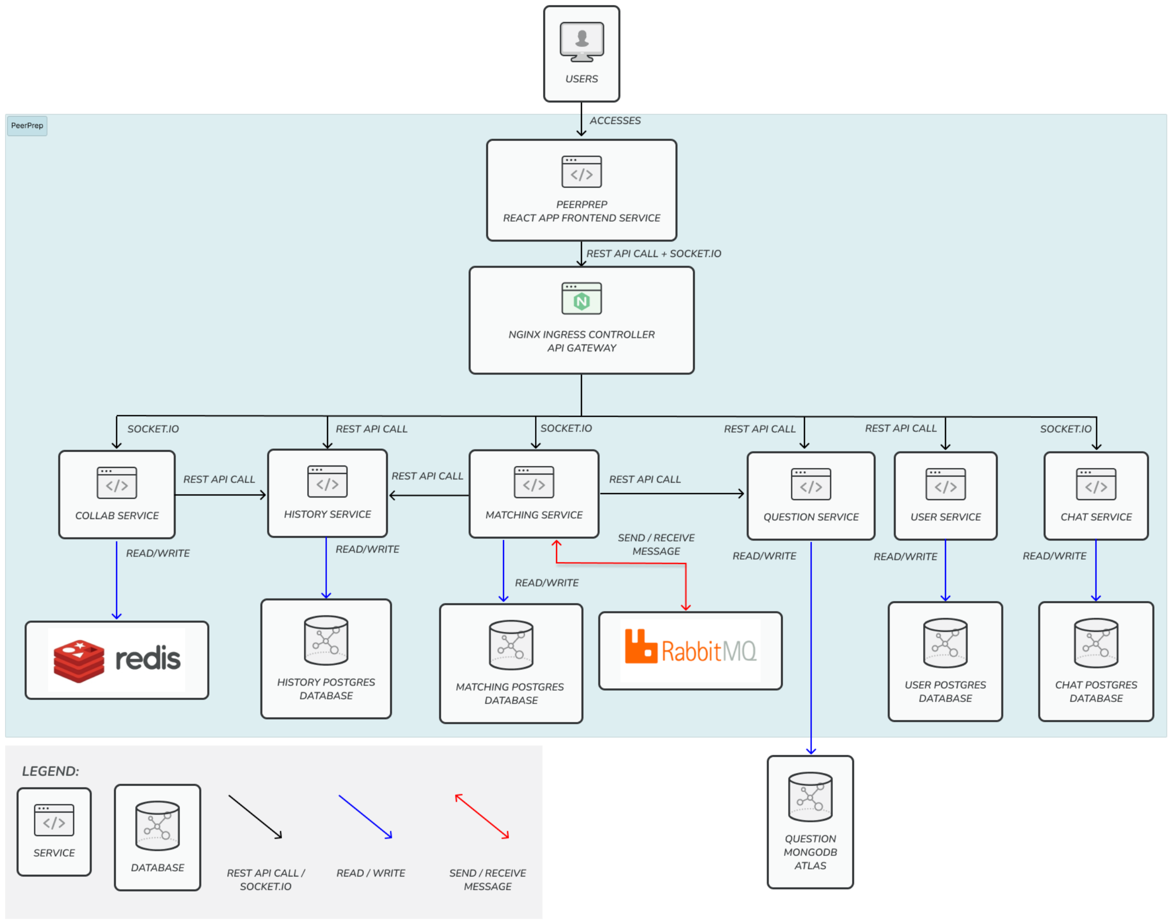


Figure 3: Architecture Diagram for PeerPrep

Design Decisions

Monolithic vs Microservice Architecture

Monolithic Architecture	Microservice Architecture
Strengths: ● Easy deployment. ● Increased familiarity for most programmers since it is the traditional way of building applications.	Strengths: ● Improves code readability and maintenance. ● Better scalability, and is more cost and time-effective to scale. ● Easier distribution of workload between group members. ● Easier implementation of separation of concerns.
Decision: Microservice Architecture We have decided to use the Microservices architecture so that each microservice can be segregated and owned by the 2 different subgroups in our group. Each subgroup can then be organised around different capabilities. The Microservices architecture also reduces implementation coupling between different sections of PeerPrep, so that not too much of the code base is dependent on any one part for easier debugging.	

Docker Compose vs Kubernetes

Docker Compose	Kubernetes
Strengths: ● Easy setup and coordination of microservices and their dependencies. ● Quick deployment of applications on a single host.	Strengths: ● Multicloud support. ● Distributed container orchestration tool with auto scalability and load balancing capabilities.
Decision: Docker Compose for local deployment and Kubernetes for deployment on the cloud via Google Kubernetes Engine (GKE). This project utilises both Docker Compose and Kubernetes based on different use cases. For testing on a local environment, docker compose will be used to quickly test and verify whether certain features are working as intended. Kubernetes can then be used in the form of GKE to scale the application on the cloud in a production build.	

Redis vs Memcached

Redis	Memcached
Strengths: ● Has growth path to a production-ready cache + database via Redis Enterprise. ● Out-of-the-box mapping libraries for popular web application frameworks.	Strengths: ● Less overhead memory
Decision: Redis for the intermittent code saving in collaboration service. Redis is used to save code that is intermittently written in the code editor in the collaboration room. It is chosen primarily due to its high availability and ease of use, given that we have selected ExpressJS as the framework used for our collaboration service.	

Use of Design Principles

Design Principles	Description
Abstraction	We abstracted out URL links, token expiry time, token refresh time, JWT secret token, etc. This allows us to save effort if these need to be changed.
High Cohesion	The Microservices architecture aids in achieving high cohesion by grouping all related behaviours into individual microservices.
Loose Coupling	Our choice of Microservice Architecture allows our PeerPrep to have loose coupling. Our services do not make many API calls to each other.

Use of Design Patterns

Design Patterns	Description
API Gateway	We applied the API gateway pattern to provide a single entry point for clients to access the various microservices. This also encapsulates the details of the different servers from the client.
Database per service	We applied the Database per service pattern to reduce the coupling between services. A service's database cannot be directly accessed by other services and can only be accessed via the service's API. This ensures that changing a service's database would not affect other services. One partial exception is the Collaboration service which does not have a database but uses Redis as a cache.

Access token	We implemented JWT access tokens to provide a way to secure and control access to protected resources in PeerPrep. Access tokens carry information about the client's role. This allows us to grant different permissions to different types of users. In this context, the role "maintainer" gets access to adding, editing, and deleting questions, while the default "user" role does not. Access tokens have a limited lifespan which enhances security by reducing the window of opportunity for unauthorised access. In our PeerPrep, each access token expires in 15 minutes. When a malicious individual manages to obtain a token from a PeerPrep user, it will likely expire and no longer be useful.
Externalized configuration	We applied the Externalized configuration pattern to abstract out the various configuration information from the app.
Service instance per container	We applied the Service instance per container pattern when deploying our application. Each service instance is being run in its own container and this allows for each service instance to be encapsulated within its own container, providing isolation. It also provides us with the ability to scale up or scale down each service easily by changing the number of containers being run simultaneously.
Remote Procedure Invocation	We applied the RPI pattern for inter-service communication and used REST API to send requests and responses between services.

Tech Stack

Tech	**Services using tech**	**Reason for Choice**
Nginx	Frontend, Collaboration, History, Matching, Question, User, Chat	- Popular reverse proxy and load balancer - Promotes easy scaling capabilities to multiple backend servers - Manage incoming traffic from the frontend - Removes convoluted port management on different microservices
React	Frontend	- Popular Javascript library for developing UI - One of the most useful web application development tools to learn and use - JSX allows us to write HTML elements in JavaScript and reduces code complexity.
Tailwind CSS	Frontend	- Popular CSS framework - Has a low learning curve - Speeds up the development process of UI - Removes the need for many .css files - Promotes a consistent UI design - Improves maintainability of the code as Tailwind is more understandable.
Node.js & Express.js	User, Question, Matching, Collaboration, Chat, History	- Javascript runtime environment - One of the more useful backend frameworks to learn and use - Has extensive documentation - Same language as React.js Frontend
MongoDB	Question	- MongoDB is a distributed database that supports automatic sharding. This gives us the ability to add horizontal scaling. - MongoDB has MongoDB Atlas, an integrated suite to host MongoDB databases in the cloud.
PostgreSQL	User, Matching, Chat, History	- PostgreSQL is SQL-compliant, so it is better for our learning if we have more experience using it. - PostgreSQL database which is a relational database provides flexibility in the types of queries that can be performed as compared to non-relational databases. - PostgreSQL gives us full control of our data as a relational database.
Socket.io	Matching, Collaboration, Chat	- Websockets are faster than HTTP calls so this allows for real-time communication and is ideal for the implementation of matching users, concurrent code editing, and real-time communication chats.
RabbitMQ	Matching	- Message broker that supports the Advanced

		Message Queuing Protocol. RabbitMQ is a good fit for implementing a queue for matching requests as well as for scalability of the matching service
Redis	Collaboration	- In-memory data store, which means it stores data in RAM for fast read/write operations. It has low latency and uses caching as well. These factors make it a good choice for storing code that is updated frequently in the concurrent code editing feature on the collaboration page.

Code Organisation

We organised the code into frontend code and backend code, separating the client-side code from the server-side code. Furthermore, we organised the backend code by services, separating the code for each service into its own directory. App-wide constant values were abstracted into the main directory while backend-wide constant values were abstracted into the backend directory. The aim of this organisation method was both to ease deployment of each service and also to encourage less coupling between the services.

Front End

Login:

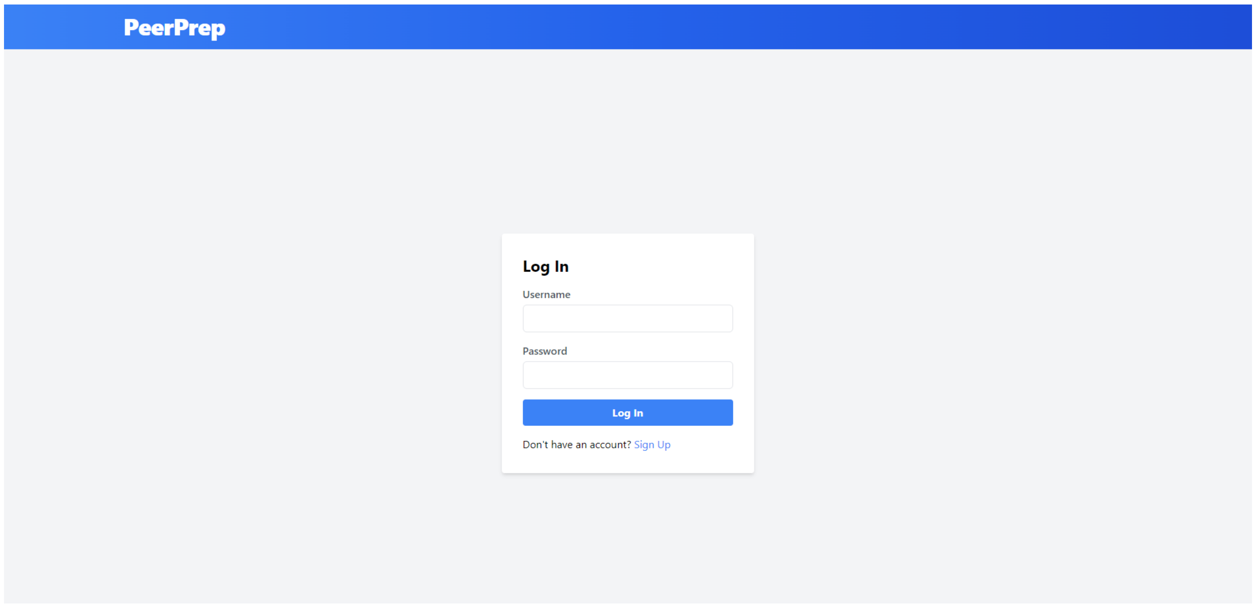


Figure 4: Login Page

When the user is at the Login or Register page, it means that they are not signed in. As shown in Figure 4, the “Home”, “Profile” and “Logout” buttons are hidden until the user logs in by entering their username and the correct password and pressing the “Sign In” button which calls the login API in User Service.

Register:

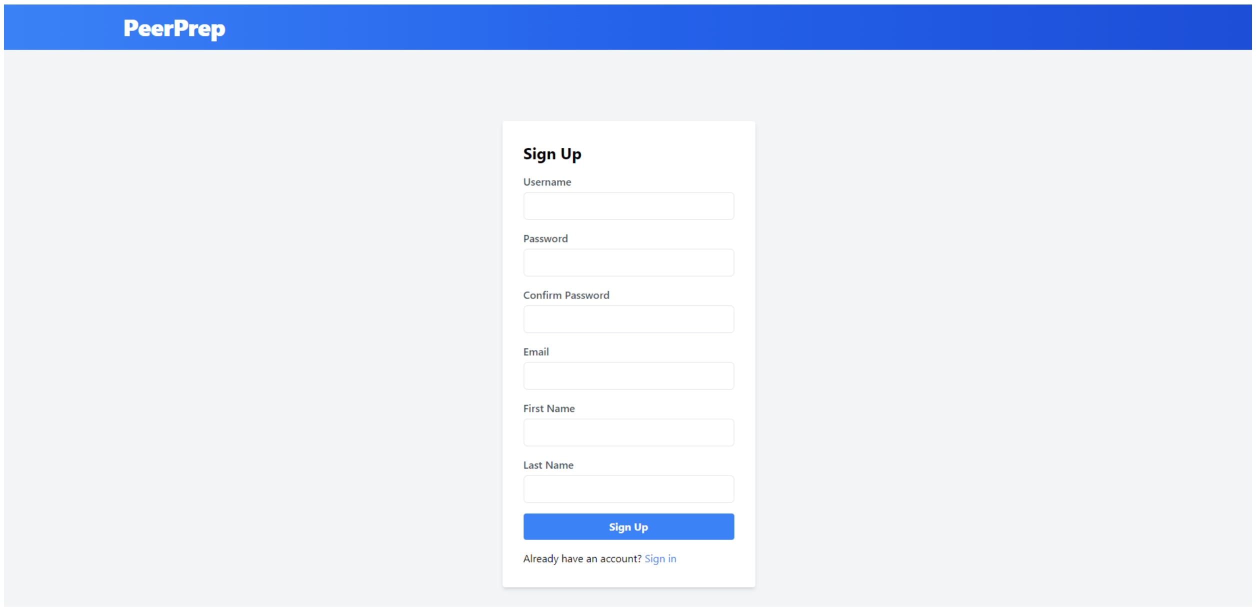


Figure 5: Register Page

When the user does not have an account, they can press the "Sign Up" button on the Login Page as shown in Figure 4. This navigates the user to the Register Page as shown in Figure 5. The Register Page gives the user the ability to create an account. They are required to specify the username, password, email, first name, and last name. When the user presses the "Sign Up" button, it will first check whether the "Confirm Password" field is the same as the "Password" field then it will call the register API in User Service. If the register API sends back an error code, an error toast will be shown. Otherwise, they will be shown a success toast and will be navigated back to the Login Page.

Homepage:

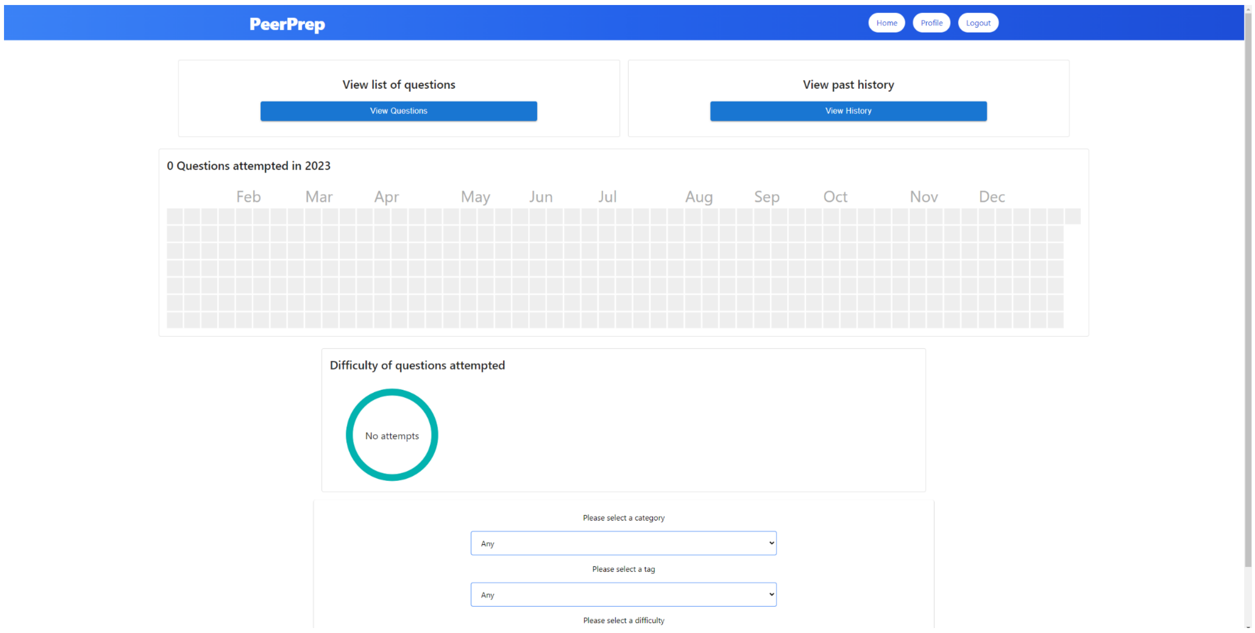


Figure 6: Home Page

As shown in Figure 6, this is the Homepage of PeerPrep. Users are navigated here after they log in. There are buttons to navigate to the Question Page and History Page. There are also statistics from the History Service, as shown in Figure 7, and there is the Matching component at the bottom of the Homepage.

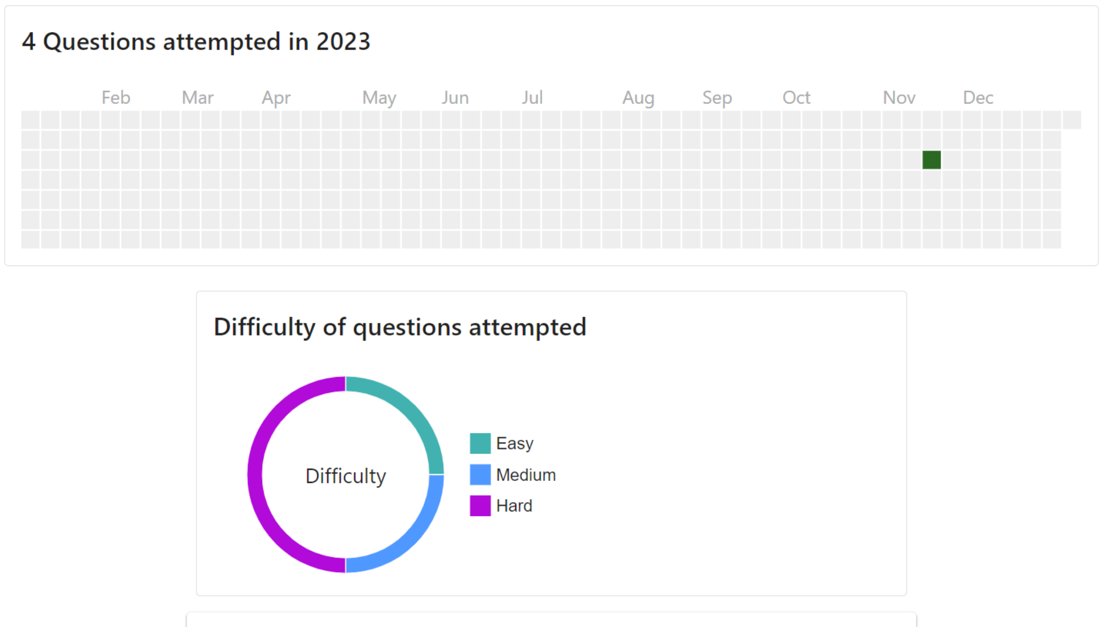
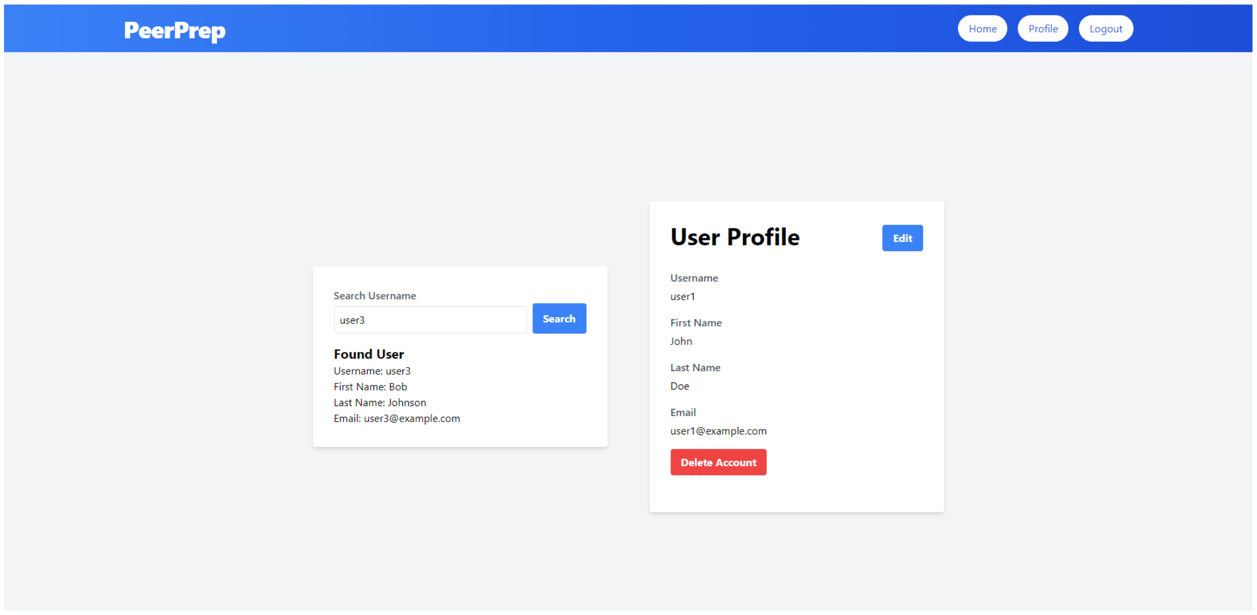

4 Questions attempted in 2023
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Difficulty of questions attempted
Difficulty
Easy
Medium
Hard

Figure 7: Statistics of collaboration in Home Page

Profile:



Found User
Username: user3
First Name: Bob
Last Name: Johnson
Email: user3@example.com

User Profile
Username
user1
First Name
John
Last Name
Doe
Email
user1@example.com

Figure 8: Profile Page (Searching for user3)

When the user is logged in, they can click the “Profile” button on the top right and it navigates the user to the Profile Page. They can view their own user profile here and even search for other users, as shown in Figure 8. They can also edit their profiles or delete their account.

Question:

On the Homepage, the user may choose to view all the questions available on PeerPrep by clicking the button shown in Figure 9.

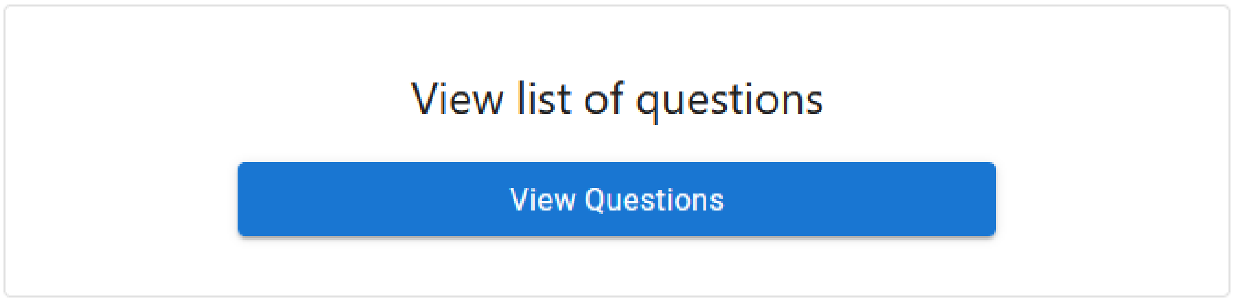
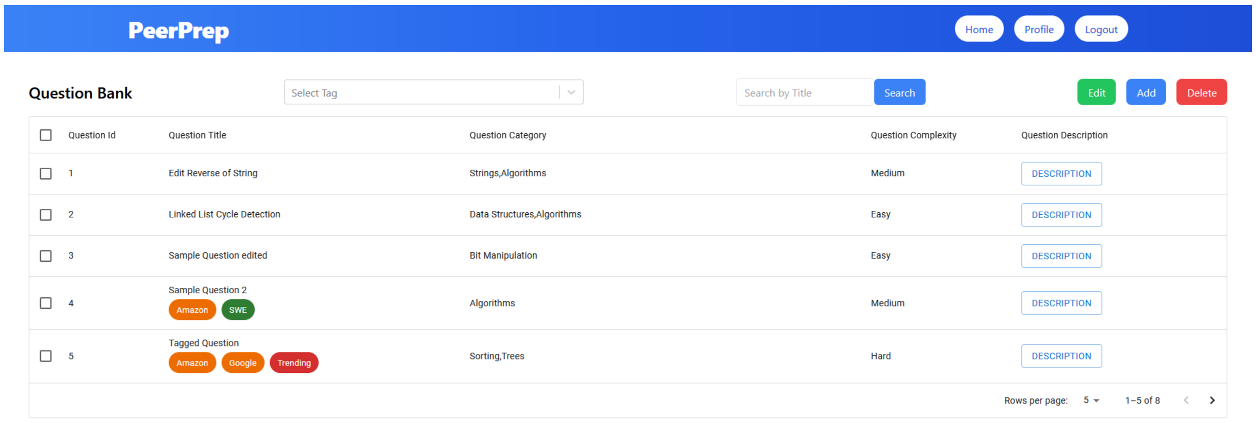


Figure 9: UI for viewing the Question Bank

This will bring the user to the Question Bank page:



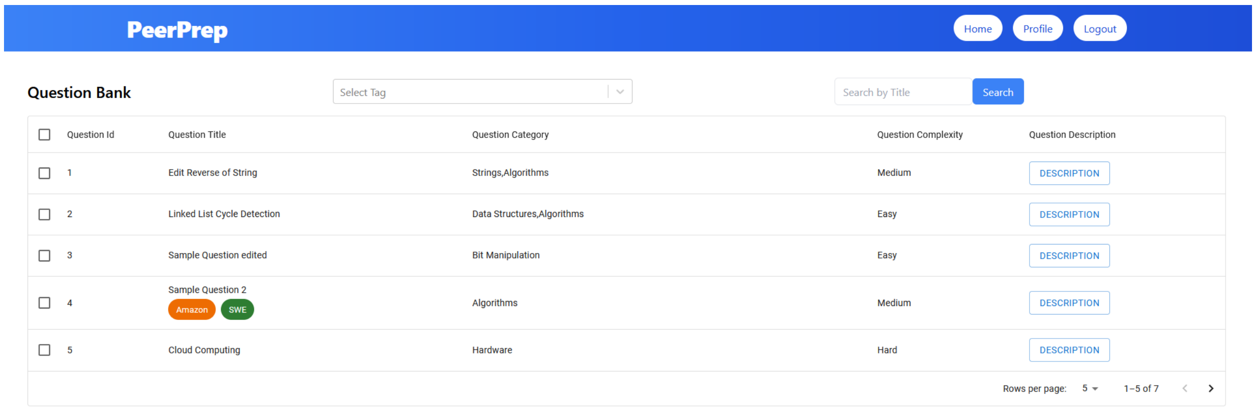
Question Bank

	Question Id	Question Title	Question Category	Question Complexity	Question Description
☐	1	Edit Reverse of String	Strings,Algorithms	Medium	DESCRIPTION
☐	2	Linked List Cycle Detection	Data Structures,Algorithms	Easy	DESCRIPTION
☐	3	Sample Question edited	Bit Manipulation	Easy	DESCRIPTION
☐	4	Sample Question 2 Amazon SWE	Algorithms	Medium	DESCRIPTION
☐	5	Tagged Question Amazon Google Trending	Sorting,Trees	Hard	DESCRIPTION

Rows per page: 5 1–5 of 8

Figure 10: Question Bank page as a maintainer

Figure 10 shows the Question Bank page that a maintainer user would see. This is as opposed to the Question Bank page seen by a default user, as shown in Figure 11.



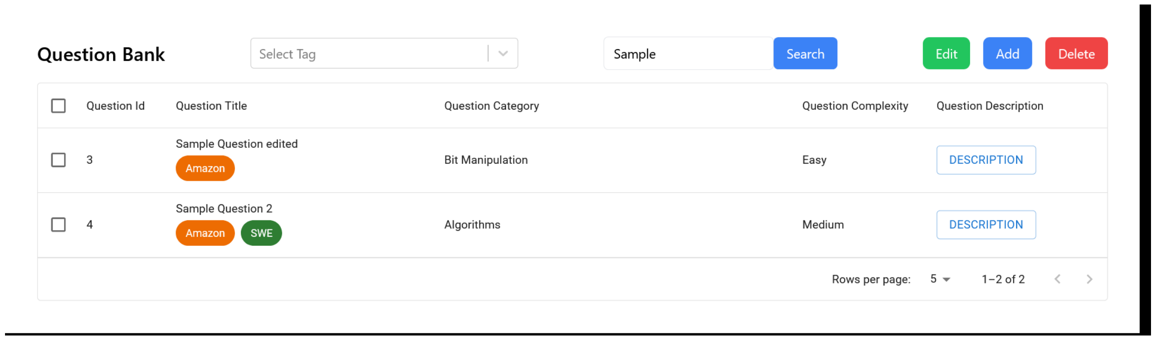
Question Bank

	Question Id	Question Title	Question Category	Question Complexity	Question Description
☐	1	Edit Reverse of String	Strings,Algorithms	Medium	DESCRIPTION
☐	2	Linked List Cycle Detection	Data Structures,Algorithms	Easy	DESCRIPTION
☐	3	Sample Question edited	Bit Manipulation	Easy	DESCRIPTION
☐	4	Sample Question 2 Amazon SWE	Algorithms	Medium	DESCRIPTION
☐	5	Cloud Computing	Hardware	Hard	DESCRIPTION

Rows per page: 5 1–5 of 7

Figure 11: Question Bank page as a default user

The difference is that a maintainer has access to the edit, add, and delete buttons which allow the maintainer to perform CRUD operations on the question bank. The question bank details are maintained by the Question Service. The coloured ovals that appear next to some questions are tags that are added to a question when adding or editing a question. The headers of the question bank table allow users to sort and filter the question bank.



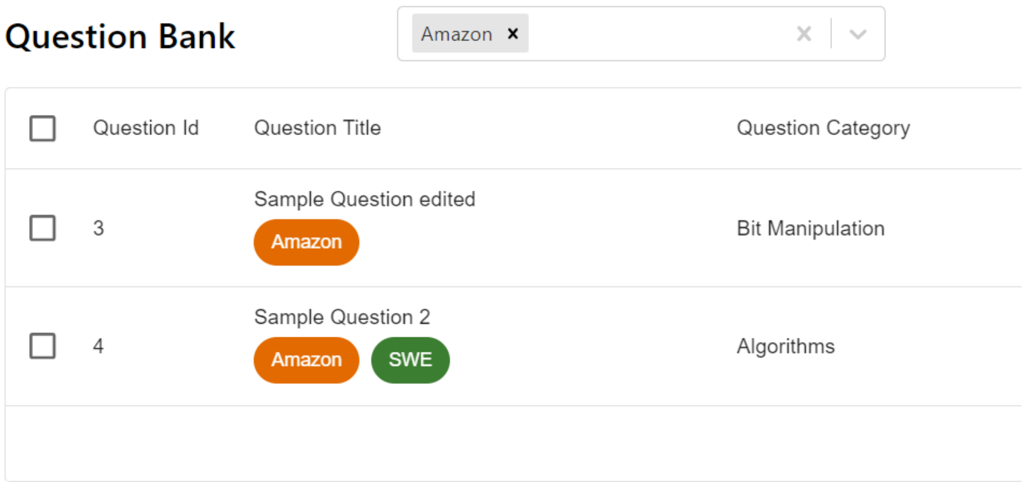
Question Bank

	Question Id	Question Title	Question Category	Question Complexity	Question Description
☐	3	Sample Question edited Amazon	Bit Manipulation	Easy	DESCRIPTION
☐	4	Sample Question 2 Amazon SWE	Algorithms	Medium	DESCRIPTION

Rows per page: 5 1–2 of 2

Figure 12: Searching for questions by title

Figure 12 shows how a user can quickly search the question bank for questions of a particular name by typing in part of the question title and pressing the Search button.



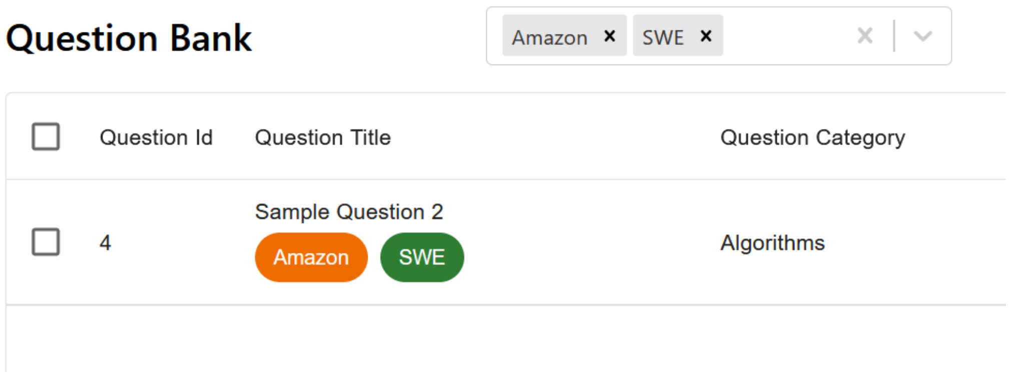
Question Bank

Amazon ×

	Question Id	Question Title	Question Category
☐	3	Sample Question edited Amazon	Bit Manipulation
☐	4	Sample Question 2 Amazon SWE	Algorithms

Figure 13: Questions filtered by a tag

Figure 13 shows how a user can also search the question bank for questions with a particular tag by choosing a tag from the dropdown box. If multiple tags are selected, only questions that have all the tags selected will be filtered as shown in Figure 14.



Question Bank

Amazon × SWE ×

	Question Id	Question Title	Question Category
☐	4	Sample Question 2 Amazon SWE	Algorithms

Figure 14: Questions filtered by multiple tags

Matching:

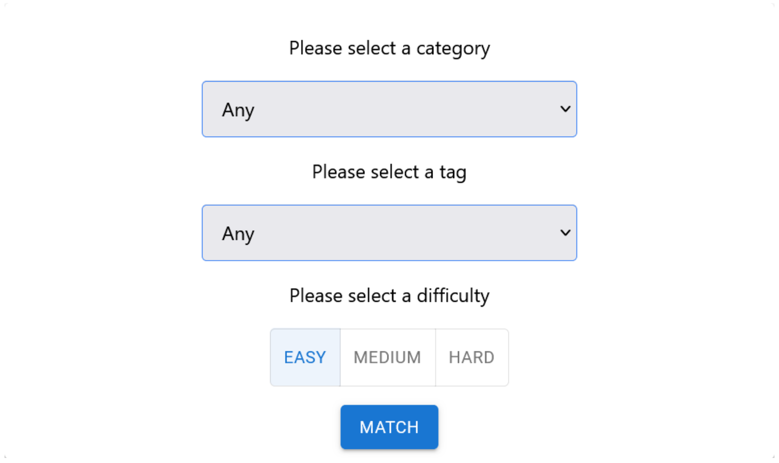
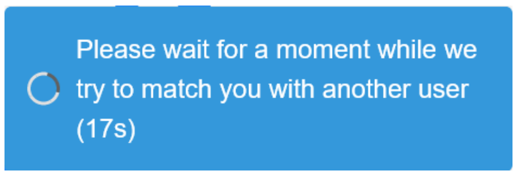


Figure 15: Matching UI component

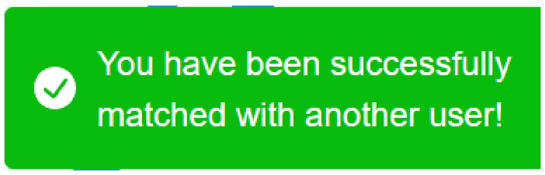
Figure 15 shows the Matching UI component found in the Homepage. Users can select their desired question category and question tag from the dropdown boxes. The list of categories and tags is obtained via an API call to the Question Service. The "Any" option is available for users who do not have a specific category and/or tag in mind. Users have to select a question difficulty. Pressing the Match button sends a match request to the Matching Service, following which, the Match button will be greyed out and disabled.



Please wait for a moment while we try to match you with another user (17s)

Figure 16: Toast with timer while waiting for response

While waiting for a response from the Matching Service, the persistent notification that contains a timer, shown in Figure 16, will appear at the top of the webpage.



You have been successfully matched with another user!

Figure 17: Toast for a successful match

If the Matching Service responds with a successful match, the success notification shown in Figure 17 will appear, following which the user will be redirected to the Collaboration room.

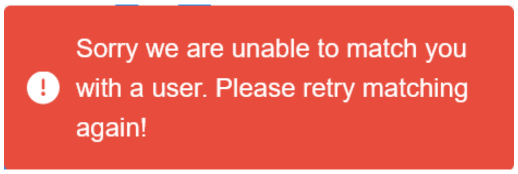

Sorry we are unable to match you with a user. Please retry matching again!

Figure 18: Toast for timeout after 30 seconds

If the Matching Service responds with no match found, the error notification shown in Figure 18 will appear, following which the Match button will be re-enabled for the user to send another match request.

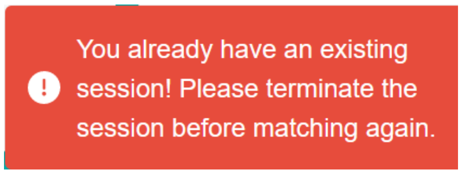

You already have an existing session! Please terminate the session before matching again.

Figure 19: Toast for a matched user who makes another request

If the Matching Service responds that the user already has an active collaboration, the error notification shown in Figure 19 will appear, following which the user will be redirected to the Collaboration room.

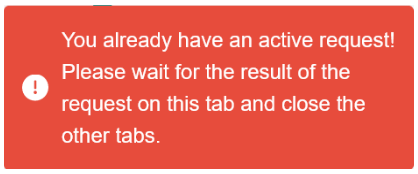

You already have an active request! Please wait for the result of the request on this tab and close the other tabs.

Figure 20: Toast for a user who sends multiple requests

If the Matching Service responds that the user already has a match request being processed, the error notification shown in Figure 20 will appear, informing the user to wait for the result of his request. The persistent notification will appear again.

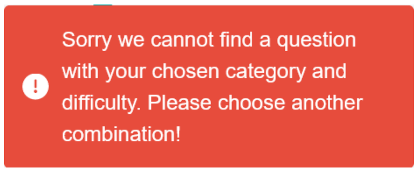

Sorry we cannot find a question with your chosen category and difficulty. Please choose another combination!

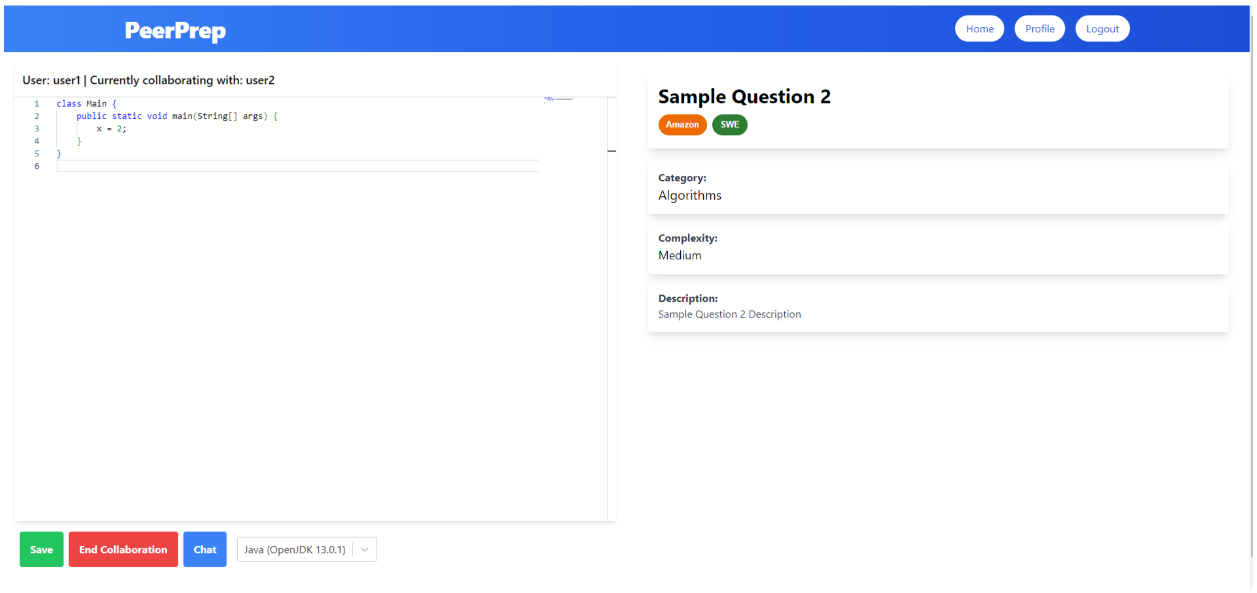
Figure 21: Toast for question not found

If the Matching Service responds with no question found, the error notification shown in Figure 21 will appear, following which the Match button will be re-enabled for the user to send another match request with a different combination of criteria.

Collaboration:

As shown in Figure 22, on the collaboration page, we have the code editor on the left and the question on the right. The “Save” button causes the code to be saved in the History Service PostgreSQL Database and the “End Collaboration” button redirects both users to the Home Page and ends the collaboration session.

We used Microsoft’s Monaco Editor to implement a code editor with code formatting and syntax highlighting for multiple languages such as Java, C++, JavaScript, etc. The 2 screenshots below show the code editor having syntax highlighting for Java (Figure 22) and C++ code (Figure 23).



User: user1 | Currently collaborating with: user2

```java
class Main {
    public static void main(String[] args) {
        x = 2;
    }
}

```

Sample Question 2

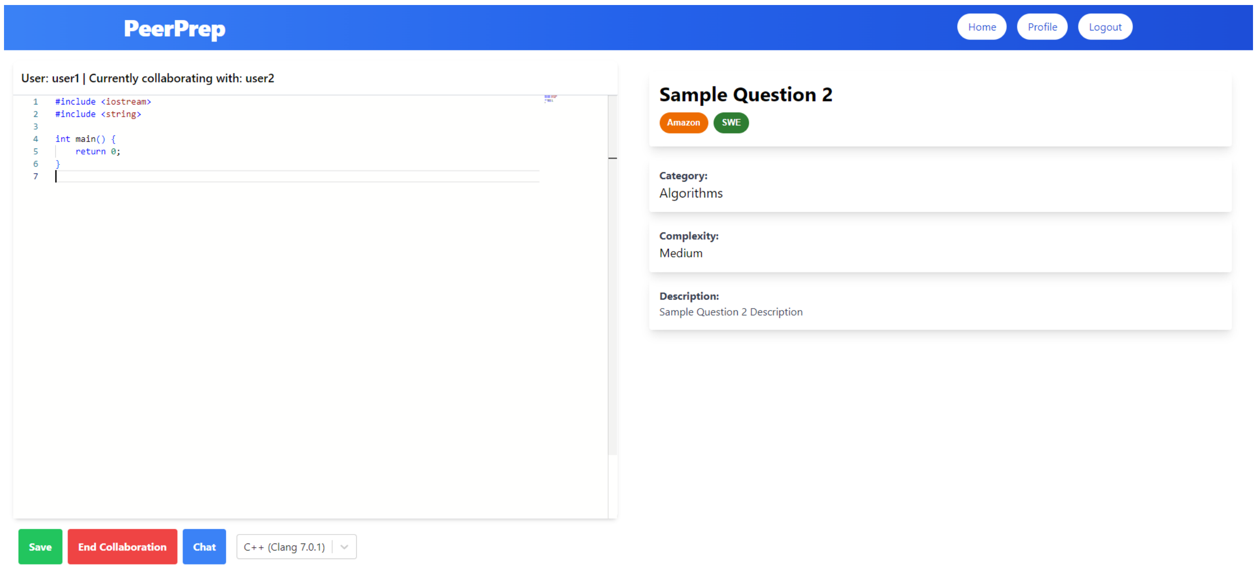
Category:
Algorithms

Complexity:
Medium

Description:
Sample Question 2 Description

Java (OpenJDK 13.0.1)

Figure 22: Collaboration Page with the programming language set to Java



User: user1 | Currently collaborating with: user2

```cpp
#include <iostream>
#include <string>

int main() {
    return 0;
}

```

Sample Question 2

Category:
Algorithms

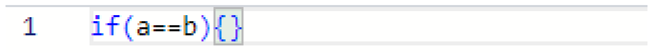
Complexity:
Medium

Description:
Sample Question 2 Description

C++ (Clang 7.0.1)

Figure 23: Collaboration Page with the programming language set to C++

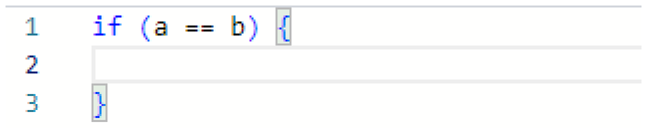
There is also code formatting. For example when we type if(a==b){ in Figure 24, there will be a } instantly added to the back.



```
if(a==b){}
```

Figure 24: Sample “if” code without code formatting yet

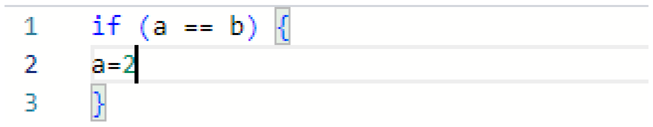
When we press enter, there will be automatic code formatting and the code would look like this (Figure 25).



```
if (a == b) {

}
```

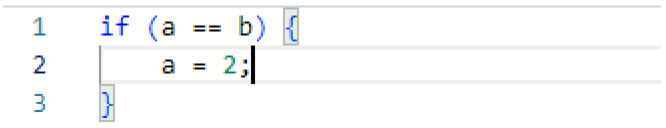
Figure 25: Sample “if” code after automatic code formatting

Regarding Figure 26 and Figure 27, another example of code formatting would be even though it auto indents and we purposely delete the indent and input a line of code, the moment we type the semi-colon, it automatically does code formatting and adds back the correct indentation.



```
if (a == b) {
a=2
}
```

Figure 26: Sample “if” code with wrong code indentation



```
if (a == b) {
    a = 2;
}
```

Figure 27: Sample “if” code with auto indentation

This feature and the syntax highlighting for multiple languages feature is part of the improved code editor feature.

Chat:

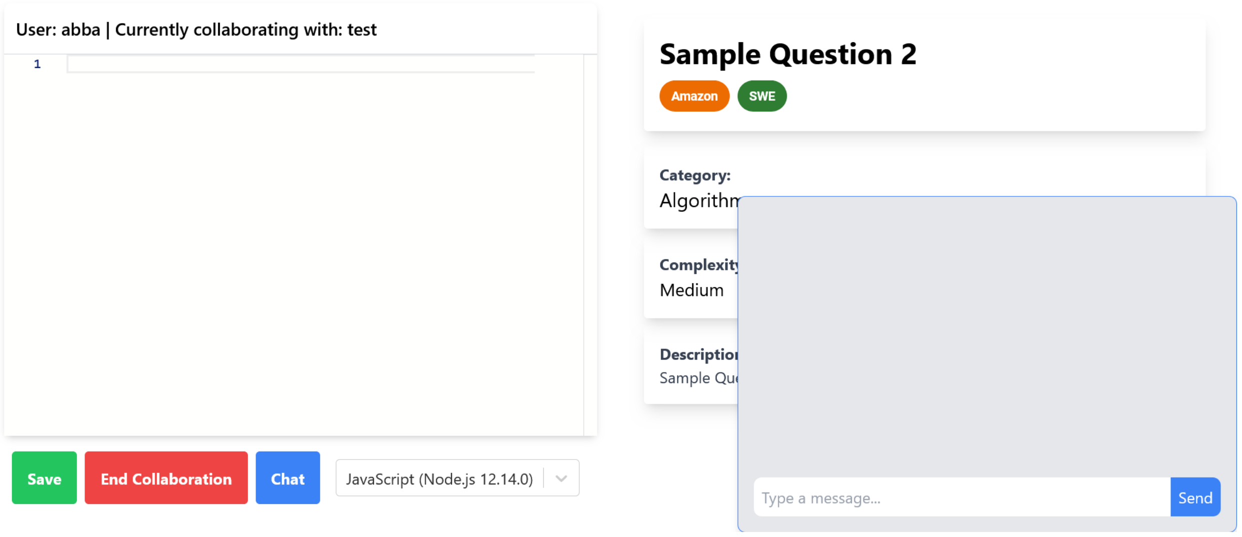
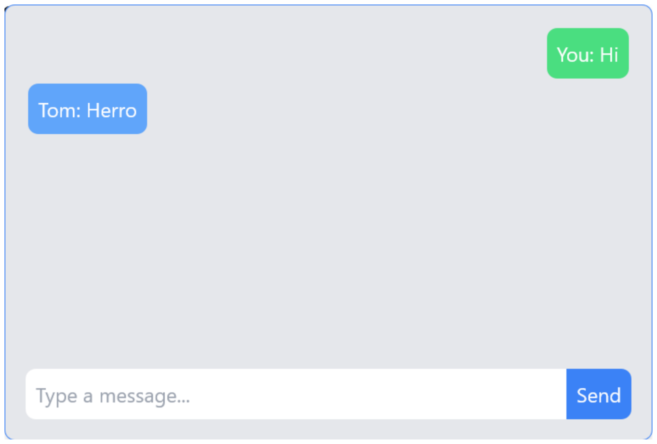


Figure 28: Collaboration Page with the Chat Box open

Figure 28 shows the Chat Box that appears when the Chat button is clicked on the Collaboration Page. The chat history of the collaboration is retrieved from the Chat Service. When no messages have been sent by either users during the session, the Chat Box will be empty as shown in Figure 28. Users can send messages using the message bar either by clicking the Send button or the Enter key on the keyboard. The Chat Box closes when either the Chat button is clicked on or when the user clicks outside of the Chat Box.


You: Hi

Tom: Herro

Figure 29: Chat Box with a message sent and a message received

Figure 29 shows how messages are displayed in the Chat Box. Each message has the first name of the user who sent it appended in front (or “You” for messages sent by the user). The position and colour of the messages are differentiated to differentiate between messages sent by the user and messages received from the other user. Incoming messages are received from the Chat Service while outgoing messages are sent to the Chat Service, all via Socket.io.

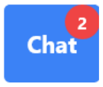
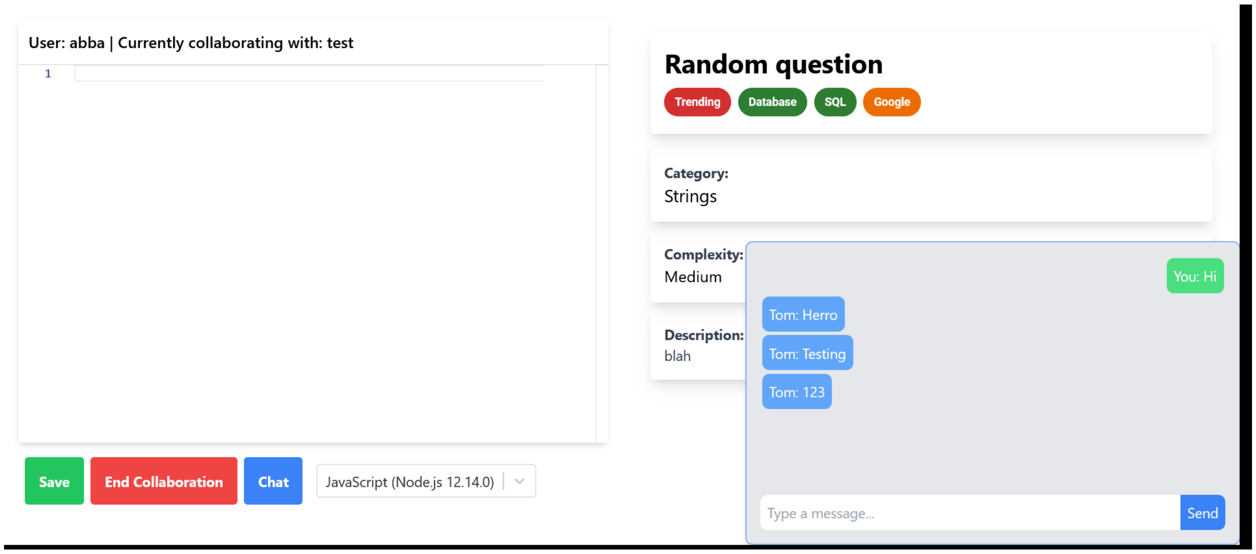


Figure 30: Chat Button with 2 unread messages

When the Chat Box is closed, incoming messages are tracked via an unread counter displayed on the Chat button as shown in Figure 30. The unread counter only appears when there is at least 1 unread message.


User: abba | Currently collaborating with: test

Random question

Category:
Strings

Complexity:
Medium

Description:
blah

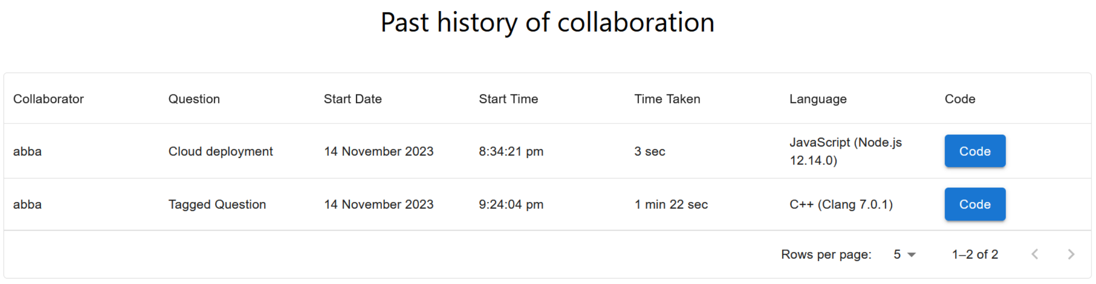
You: Hi
Tom: Herro
Tom: Testing
Tom: 123

Figure 31: When viewing unread messages, the unread counter disappears

Figure 31 shows how the unread counter resets and disappears when the Chat Box is opened to view the unread messages. Messages sent while the Chat Box is opened will not be registered as unread.

History:

On the history page, the user will be able to view the history of their collaboration with other users. The user will be able to see the collaborator, the question they attempted, the start date of the collaboration, the start time of the collaboration, the time spent during the collaboration, the language used for the code and the code implemented during collaboration.

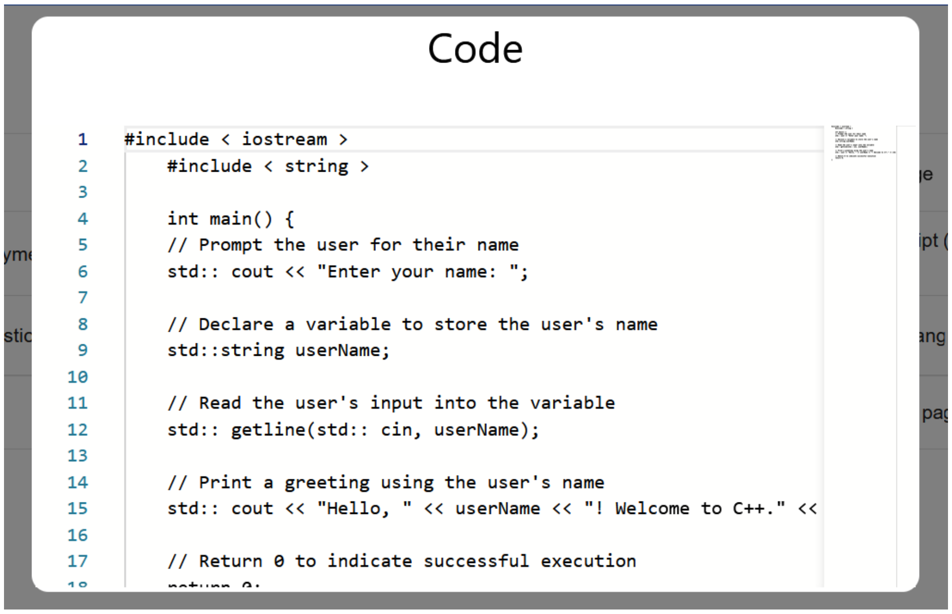

Past history of collaboration

Collaborator	Question	Start Date	Start Time	Time Taken	Language	Code
abba	Cloud deployment	14 November 2023	8:34:21 pm	3 sec	JavaScript (Node.js 12.14.0)	Code
abba	Tagged Question	14 November 2023	9:24:04 pm	1 min 22 sec	C++ (Clang 7.0.1)	Code

Rows per page: 5 1–2 of 2

Figure 32: History page showing past collaboration

The user can click on the code button to view the code completed during the past collaboration


Code

```
#include < iostream >
    #include < string >

    int main() {
    // Prompt the user for their name
    std:: cout << "Enter your name: ";

    // Declare a variable to store the user's name
    std::string userName;

    // Read the user's input into the variable
    std:: getline(std:: cin, userName);

    // Print a greeting using the user's name
    std:: cout << "Hello, " << userName << "! Welcome to C++." <<

    // Return 0 to indicate successful execution
```

Figure 33: Viewing code implemented during collaboration

Microservices

User Service:

Responsibilities

1) User Account Registration:
   When the register API is called, the User Service will check whether the username or email already in use by another account and whether the password provided is at least 8 characters long and has at least one lowercase letter, uppercase letter, number, and special character. If the criteria is met, an account is created and saved in the User Service PostgreSQL Database. Otherwise, an error status code will be sent to the user client.

2) Login and Authentication with JWT Tokens:
   When the login API is called, the User Service verifies whether the username and password combination is correct. If it is wrong, the user service sends a status code of 401, and an error toast `Incorrect username or Password` is shown to the user. If it is correct, the user service signs and sends a JWT access token to the user client.

   JWT access tokens provide a way to secure and control access to protected resources. They carry information about the client's role. This allows us to grant different permissions to different types of users. In this context, the role "maintainer" gets access to adding, editing, and deleting questions, while the default "user" role does not.

   Access tokens have a limited lifespan which enhances security by reducing the window of opportunity for unauthorised access. In our PeerPrep, each access token expires in 15 minutes. When a malicious individual manages to obtain a token from a PeerPrep user, it will likely be expired and no longer useful.

   When PeerPrep first loads or the page is refreshed, the access token will be refreshed after 10 seconds. After the first refresh, the access token will be refreshed every 5 minutes. This ensures that the user can continue to use PeerPrep beyond the 15-minute access token expiration time when they are on PeerPrep.

3) User Profile Management:
   There are APIs in the User Service that support the user profile page. They allow the user to search for other users, edit their profiles, and delete their own account. For the PUT and DELETE API calls for the user profile, the User Service verifies whether the owner of the JWT access token is the same as the account being edited or deleted.

Architectural overview

The User Service reads and writes from the User Service PostgreSQL Database. The frontend makes REST API calls to the user service through the API gateway.

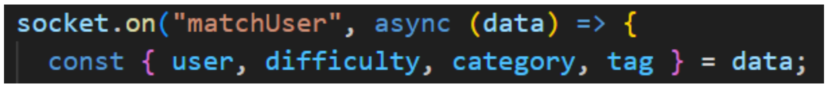
Design considerations

The User Service is rather decoupled from the rest of the other services because the JWT access token provides enough context for the other services. This keeps the user data more secure from potential attacks.

Matching Service:

Responsibilities

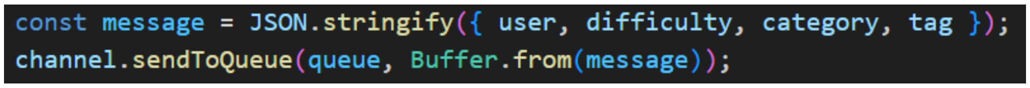
1) Receiving matching requests:
   The Matching Service receives matching requests via Socket.io. Figure 34 shows the logical representation of a matching request received via Socket.io.



```
socket.on("matchUser", async (data) => {
  const { user, difficulty, category, tag } = data;
```

Figure 34: Matching request emitted from the client

   The matching request is then sent to a RabbitMQ queue as shown in Figure 35:



```
const message = JSON.stringify({ user, difficulty, category, tag });
channel.sendToQueue(queue, Buffer.from(message));
```

Figure 35: Sending matching requests to RabbitMQ

2) Matching 2 users:
   The Matching Service processes matching requests from RabbitMQ on a first-come-first-serve basis. The Matching Service maintains a Map of requests received from RabbitMQ. For each request received from RabbitMQ, the Map is checked for a request with the same criteria. If no requests are found, the request is added to the Map. Otherwise, the found request will be removed from the Map and the event "matched" will be emitted to the clients via socket.io.

3) Timing out a user:
   When a request is added to the Map, a timeout is set for 30 seconds, upon which the request will be removed from the Map and the event "timeout" will be emitted to the client via socket.io.

4) Preventing matched users from matching again:
   The Matching Service does not allow matched users (users who are currently in an active collaboration with another user) to send a match request. Therefore, each successful match is stored in a PostgreSQL database, and only removed when the

collaboration has ended. The Matching Service checks each new request against the database and emits the event "already_matched" to matched clients via socket.io.

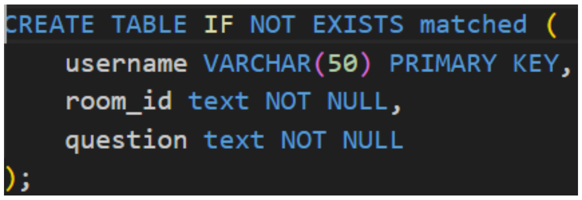
5) Preventing multiple match requests:
   The Matching Service only allows users to have one match request at any given time. Therefore, the Matching Service maintains a Map of active requests, and new requests whose users are found in the Map are rejected. The Matching Service emits the event "already_requested" message to the client via socket.io.

6) Selecting random questions:
   There are 2 steps to the question selection performed in the Matching Service. First, when a request is received by the server, an API call is made to the Question Service to check the database if a question exists that matches the criteria of the request. If no such question exists, The Matching Service emits the event "not_found" message to the client via socket.io.

   The second round of question selection occurs when the Matching Service matches 2 users. Then, another API call is made to the Question Service to get the list of questions that match the criteria of the two users. A random question is selected from the list and assigned to the 2 users.

Architectural overview

The Matching Service uses a PostgreSQL database to store the list of active matched users (users who have been matched and have yet to end their collaboration session). Shown below is the schema used.



```
CREATE TABLE IF NOT EXISTS matched (
    username VARCHAR(50) PRIMARY KEY,
    room_id text NOT NULL,
    question text NOT NULL
);
```

Figure 36: Schema for PostgreSQL database for Matching Service

The Matching Service reads and writes to the PostgreSQL database. The Matching Service also sends and receives matching requests from RabbitMQ. Communication between the frontend and the Matching Service goes through the API gateway and is facilitated by socket.io events. The Matching Service makes REST API calls to the Question Service to get questions. The Matching Service also makes REST API calls to the History Service to update the collaboration history of matched users.

Design considerations

Socket.io was selected for communication between the frontend and the Matching Service as opposed to REST API due to the time-sensitive (timer, first-come-first-serve, etc) nature of the service. Socket.io is based on the WebSockets framework, making it suitable for real-time communications. Additionally, this was also our first encounter with socket.io so it served as a good opportunity to test out the suitability of socket.io for use in implementing the Collaboration Service.

RabbitMQ was used to implement the matching queue as opposed to a normal queue to allow for the scalability of the service. The intention was for multiple matching servers to receive match requests from RabbitMQ and split the request processing workload.

Question Service:

Responsibilities

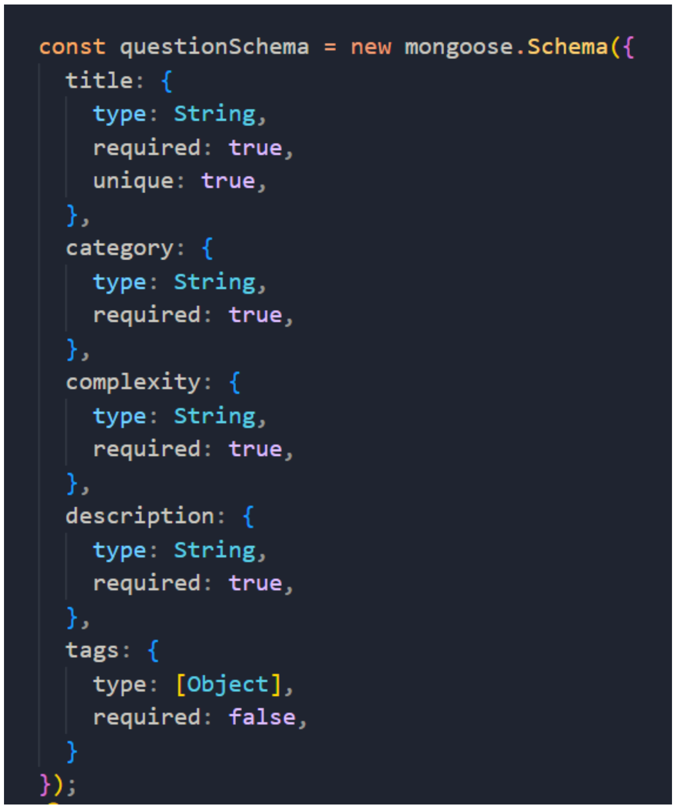
1) Management of PeerPrep questions:
   The question service manages the question table seen in the view question tab and enables the persistence of question data by storing the question information in a MongoDB database.

2) Easy access to recognized programming questions:
   It shows users and maintainers all questions recognized by PeerPrep which may be selected in the collaboration room, managed by the Collaboration Service. Users may also search for certain questions in the question bank.

Architectural overview

The question service utilises MongooseJS for REST API calls to the MongoDB server, hosted on MongoDB Atlas. The question service allows for the read, write, update, and delete operations on all questions through GET, POST, PUT, and DELETE HTTP methods respectively.

The details for each question are stored in MongoDB in the following format:



```
const questionSchema = new mongoose.Schema({
  title: {
    type: String,
    required: true,
    unique: true,
  },
  category: {
    type: String,
    required: true,
  },
  complexity: {
    type: String,
    required: true,
  },
  description: {
    type: String,
    required: true,
  },
  tags: {
    type: [Object],
    required: false,
  }
});
```

Figure 37: Schema used in MongoDB

Here, we do not allow a missing title, category, complexity, or description in the database. This is to facilitate completeness in the question when retrieving it during collaboration.

Below is a sequence diagram for a POST call to add a question to the question repository:

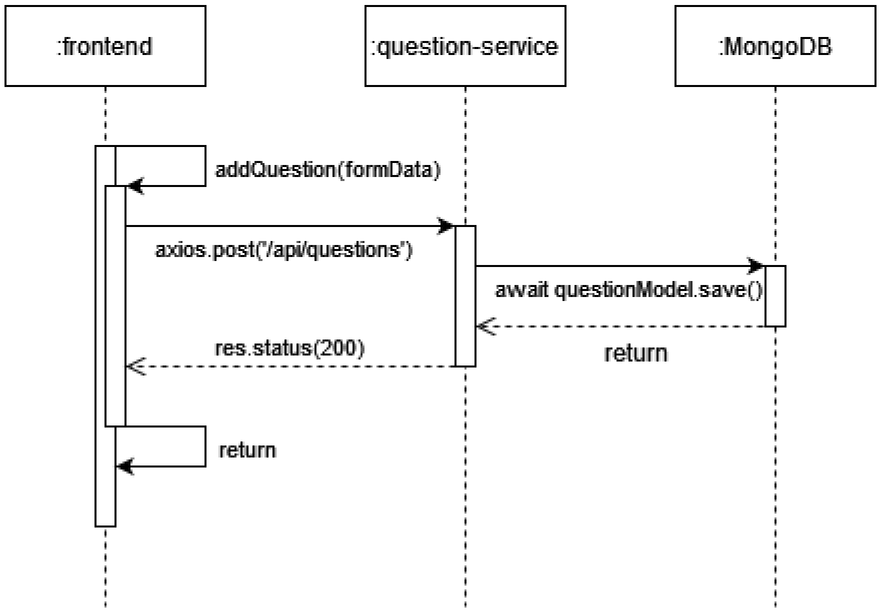
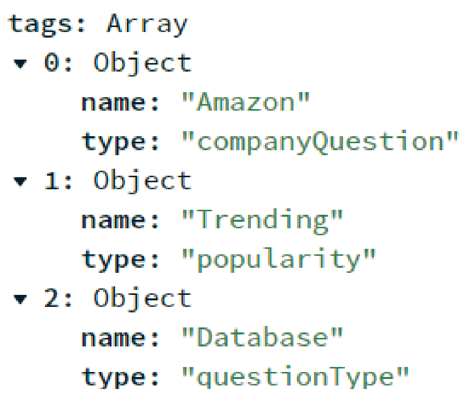


Figure 38: Sequence diagram of adding a question to PeerPrep through a form

Each tag is stored as an array of objects with each object having the properties name and type in the MongoDB database. The type of the tag can be either "`companyQuestion`", "`popularity`", or "`questionType`".



```
tags: Array
▾ 0: Object
    name: "Amazon"
    type: "companyQuestion"
▾ 1: Object
    name: "Trending"
    type: "popularity"
▾ 2: Object
    name: "Database"
    type: "questionType"
```

Figure 39: How tag is stored in MongoDB database

Design considerations

We decided to use a NoSQL database since it was easier to manage. We also chose to use MongoDB since it has MongoDB Atlas, a multi-cloud database service for MongoDB applications. This means that our database can be easily deployed onto the cloud and connected via a given URL.

REST API calls were also used to encourage decoupling between the question service and other services in the microservices architecture.

The tag is stored as an object rather than as a string as fewer operations will need to be performed to obtain the colour of the tag in the frontend. Moreover, if the tag is stored as a string with some value as the separator, it means that this value cannot be used in the name of the tag. To prevent restrictions on the choice of the name of the tag, we have decided to store it as an object.

Collaboration Service:

Responsibilities

1) Real-time collaboration:
   Collaboration Service provides the mechanism for real-time collaboration (e.g., concurrent code editing) between the authenticated and matched users in the collaborative space. This is implemented using Socket.io. Whenever a user stops typing for 300ms, the code is sent through socket.io to Collaboration Service, where the code is

then sent through Socket.io to the other user. The 300ms delay is to reduce the stress on the server via debouncing the function.

2) Automatic saving of code to Redis:
Collaboration service saves the code to Redis every time the code is sent through Socket.io. This allows the code to be saved and reloaded even if the user refreshes the page or leaves and rejoins.

3) Manual saving of code to History Service:
However, Redis loses its data if it goes down. Therefore, for the persistence of data, we implemented a manual save button which makes an API call to the History Service, saving the code in the History Service PostgreSQL Database. If Redis is down, the Collaboration Service gets the code by making an API call to the History Service, retrieving the code from the History Service PostgreSQL Database instead.

4) Synchronisation of selected programming language:
The Collaboration Service uses socket.io to ensure the synchronisation of the languages used by the 2 users. When the language is changed by one of the users, the Collaboration Service makes an API call to the History Service, saving the language used in the History Service PostgreSQL Database.

Concurrent Code Editing:

Concurrent code editing that only sends changes to selected sockets can be done through Socket.io rooms. This follows a publish/subscribe messaging pattern where 2 users in the same collaboration room subscribe to a room made on Socket.io. Note that only 2 users can be in a room at any one time, and any other users that try to connect to the room will not be allowed. When 1 user makes a change in the code editor, it will publish the code changes to all subscribed sockets in the room except his socket. Therefore, the other user can dynamically view the changes made by this user.

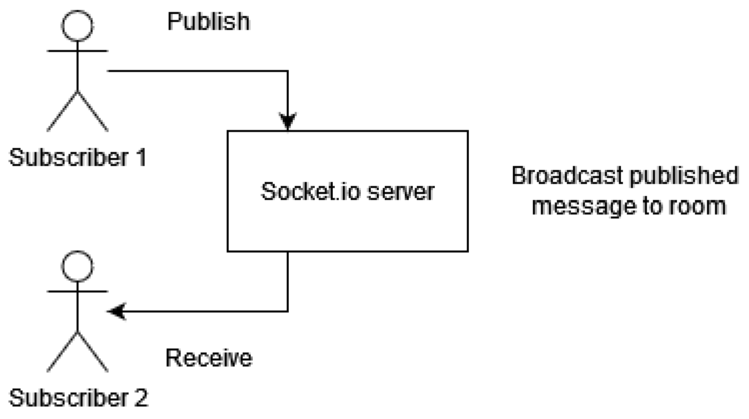


Figure 40: How 2 users in the same collaboration room receive code updates

When all users disconnect and leave the room, the room will automatically be closed by the Socket.io server instance.

Architectural overview

The Collaboration Service reads and writes from Redis. The frontend makes REST API calls to the Collaboration Service through the API gateway. The Collaboration Service makes REST API calls to the History Service to save and get code and language.

Design considerations

We used Redis instead of another PostgreSQL Database to store code as Redis is an in-memory data store, which means it stores data in RAM for fast read and write operations. It is also because it has low latency and uses caching as well, which is ideal for intermittent saving of code. However, Redis loses its data if it goes down. Therefore, for the persistence of data, we implemented manual saving of the code in the History Service PostgreSQL Database.

Chat Service:

Responsibilities

1) Sending conversation history to clients:
   The Chat Service receives requests for conversation history via socket.io. The conversation history is retrieved from the Chat PostgreSQL database. The conversation history is emitted back to the client with a “get_chat” event via socket.io

2) Passing new messages between 2 clients:
   The Chat Service receives new messages via socket.io and writes it to the database. The message is then emitted to the other user with a “new_message” event via socket.io.

3) Deleting conversation history:
   When the Chat Service is notified by the Collaboration Page that the session has ended, the conversation history is deleted from the database.

Architectural overview

The Chat Service uses a PostgreSQL database to store the messages. Shown below is the schema used.

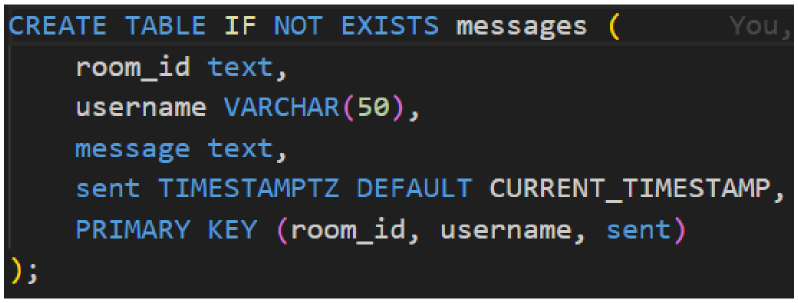

```sql
CREATE TABLE IF NOT EXISTS messages (        You,
    room_id text,
    username VARCHAR(50),
    message text,
    sent TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (room_id, username, sent)
);
```

Figure 41: Schema for PostgreSQL database for Chat Service

The Chat Service reads and writes to the PostgreSQL database. Communication between the frontend and the Chat Service goes through the API gateway and is facilitated by socket.io events.

Design considerations

Socket.io was selected for communication between the frontend and the Chat Service as opposed to REST API for real-time communication. Having had a good experience with socket.io from the Matching Service and the Collaboration Service, we were confident that this choice would both make the implementation easier as well as achieve good results for real-time communication.

History Service:

Responsibilities

1) Save the history of past collaboration
   The History Service will create a new record for the history of collaboration when it receives an API call from the Matching Service

   When the History Service receives an API call from the Collaboration Service to save the code or language, the history service will then query the PostgreSQL database to update the code or language for the two users.

   When the History Service receives an API call from the Collaboration Service to save the end time of collaboration, the History service will then query the PostgreSQL database to update the time ended for the two users.

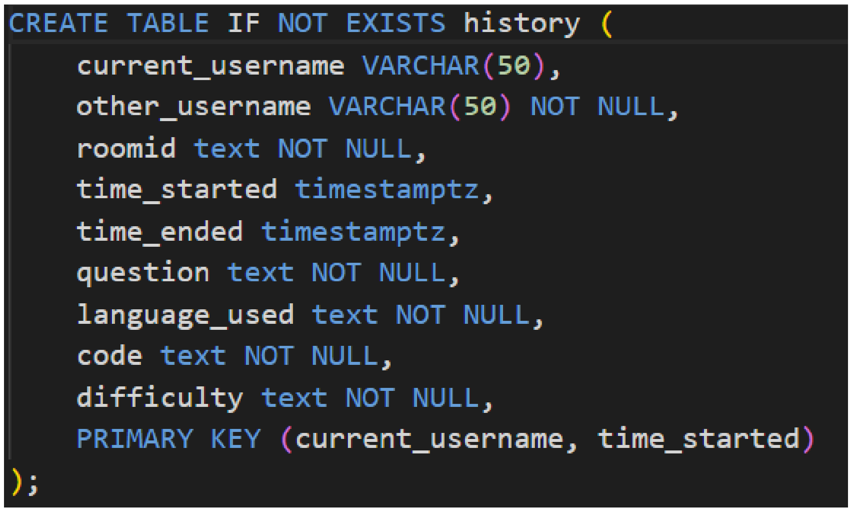
2) Sending history of past collaboration
   The History Service will send the history of collaboration for the user to the frontend when the frontend makes a GET request for the history. The History Service can also only send the difficulty of the questions and time started for the collaboration when requested by the frontend.

3) Delete the history of collaboration
   When the History Service receives an API call from the frontend to delete the of history collaboration for the user, the History Service will query the PostgreSQL database to delete the history of collaboration for that user. The records will only be deleted for the current user and any other users who have collaborated with the current user will still have their history of collaboration preserved.

Architectural overview

The History Service uses a PostgreSQL database to save the data for the history of collaboration. Communication between the History Service with the other services such as Matching Service, Collaboration Service, and Frontend is done through the API Gateway.

Shown below is the schema used for the history PostgreSQL database.



```sql
CREATE TABLE IF NOT EXISTS history (
    current_username VARCHAR(50),
    other_username VARCHAR(50) NOT NULL,
    roomid text NOT NULL,
    time_started timestamptz,
    time_ended timestamptz,
    question text NOT NULL,
    language_used text NOT NULL,
    code text NOT NULL,
    difficulty text NOT NULL,
    PRIMARY KEY (current_username, time_started)
);
```

Figure 42: Scheme for PostgreSQL database for History Service

The `roomid` is also stored in the schema as the `roomid` is sent by the Collaboration Service and is used in the query to update the code of both users who are collaborating.

For the API calls from the frontend to the History Service, there is a verification of JWT to ensure that only authorised users have access to the history of the collaboration.

Design Considerations

We decided to choose a relational database for History Service as more complicated queries were needed for the updating of the database. Since we had more experience in writing PostgreSQL queries, we decided to use a PostgreSQL database.

The API call to insert the new record for the history of collaboration was called in Matching Service rather than in the Collaboration Page on the frontend as it helped to reduce the amount of unnecessary POST requests being made to the History Service since duplicate POST requests would be made every time the user joins or refreshes the Collaboration Page.

DevOps

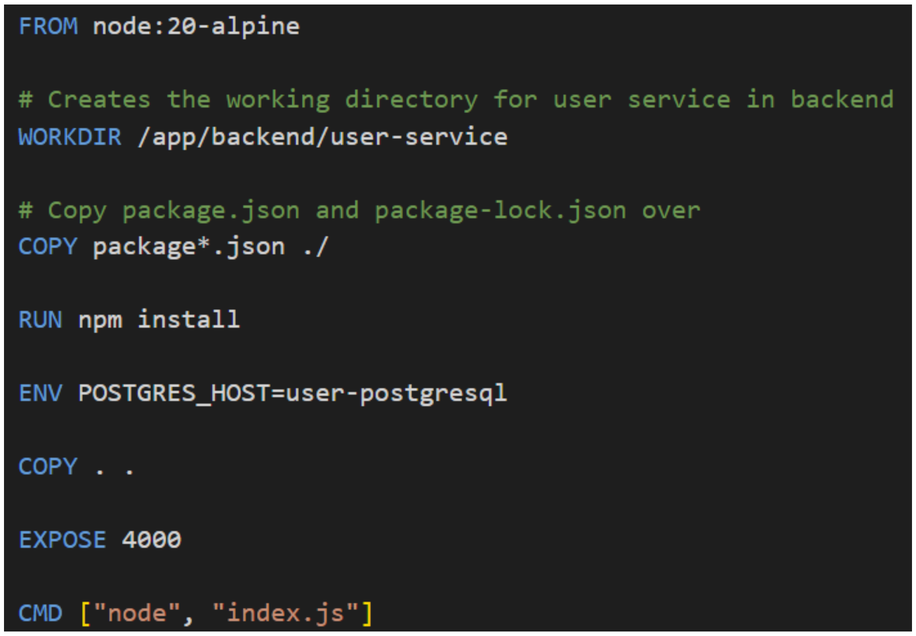
Local Deployment using Docker Compose

Responsibilities

1) As there are multiple microservices present in the application, starting up the application would be cumbersome without Docker Compose as the user would have to manually start up each of the microservices to run the application. By utilising Docker along with Docker Compose, the user will just need to run a single command “**`docker compose up --build -d`**” to build and start all the microservices at the same time.

Architectural overview

For each microservice, we have a dockerfile which contains the set of instructions to build the docker image for each service. Inside the dockerfile, we configured the environment that the service will run in, installed the necessary dependencies, specified the ports exposed by the container, and specified the command that will be executed when the container is running.



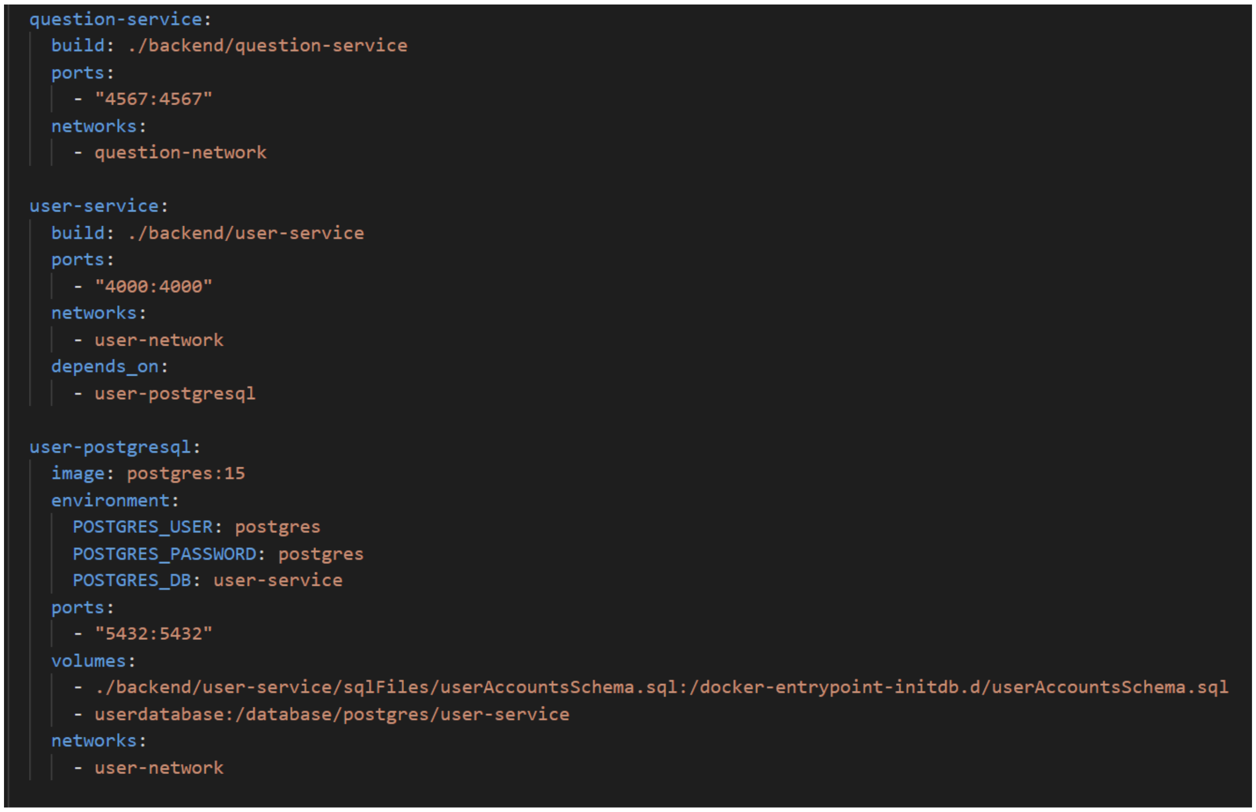
```
FROM node:20-alpine

# Creates the working directory for user service in backend
WORKDIR /app/backend/user-service

# Copy package.json and package-lock.json over
COPY package*.json ./

RUN npm install

ENV POSTGRES_HOST=user-postgresql

COPY . .

EXPOSE 4000

CMD ["node", "index.js"]
```

Figure 43: Dockerfile for User Service

We also have a `docker-compose.yml` file which specifies the configuration of the services that will be built by Docker Compose. This includes the path to the image to build, the mapping of the host port to the container ports, the dependency between services, the environment variables of some containers, the volumes that serve as a means of persisting data generated by the Docker containers and the networks that the services are connected to.

Volumes provide a way for the data to persist even after the containers stop running. This enables users to stop the containers and resume using them without any loss of data.

For the RabbitMQ service, there is also a health-check label where there will be a check to determine if the RabbitMQ service is healthy and able to receive requests. For the collab-redis service, we also have the restart label where the service will restart if it encounters an on-failure error.



```
question-service:
  build: ./backend/question-service
  ports:
    - "4567:4567"
  networks:
    - question-network

user-service:
  build: ./backend/user-service
  ports:
    - "4000:4000"
  networks:
    - user-network
  depends_on:
    - user-postgresql

user-postgresql:
  image: postgres:15
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: user-service
  ports:
    - "5432:5432"
  volumes:
    - ./backend/user-service/sqlFiles/userAccountsSchema.sql:/docker-entrypoint-initdb.d/userAccountsSchema.sql
    - userdatabase:/database/postgres/user-service
  networks:
    - user-network
```

Figure 44: Snippet of the docker-compose.yml file

Design Considerations

As we decided to use the MongoDB Atlas database which is hosted on the cloud for the database of Question Service, we did not dockerise the database for question service.

We made use of the `depends_on` in some of the services to ensure that the service waits for its dependencies to start up first before it starts up.

Nginx API Gateway

Responsibilities

Nginx as a backend server holds multiple responsibilities:

1) API gateway for backend services

   It acts as an API gateway and sits in front of all the backend servers for this project. All HTTP requests and responses between the frontend and backend, as well as between backend services will be rerouted through the Nginx web server.

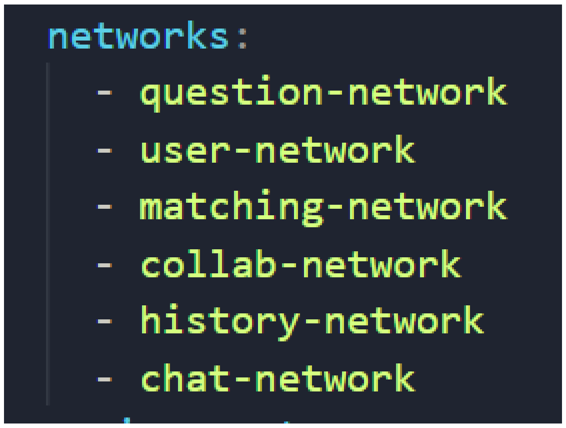
2) Aids in the scalability of PeerPrep

   It enables horizontal duplication of microservices and load balance across the duplicated backend servers concerning the X-axis of the scale cube from The Art of Scalability. If one of the backend services starts to get congested with traffic, Nginx can distribute new incoming traffic to less congested backend servers.

Note that the Nginx server does not scale the databases for this project.

Architectural overview

Due to the API gateway needing to communicate with all backend services, it needs to be in all Docker networks, as shown below:

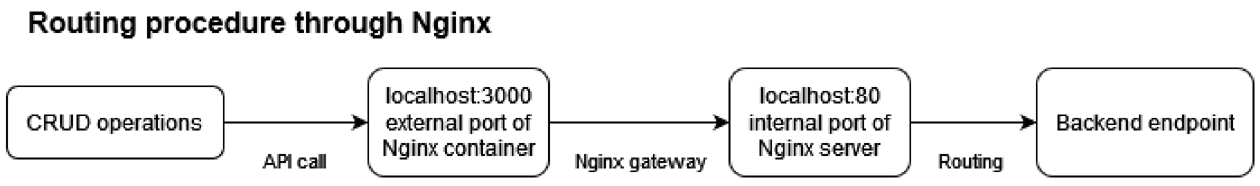


```
networks:
  - question-network
  - user-network
  - matching-network
  - collab-network
  - history-network
  - chat-network
```

Figure 45: List of Docker networks

Communication and routing for each service depends on the URI specified for each backend service. Typically, the backend services will route with the URI of `/api/` prefix to signify communication and routing of HTTP requests or responses to backend servers. This is agreed upon on the side of the backend services as well. Nginx will appropriately listen to all API calls on port 3000 (on localhost) which are routed to port 80 inside the container. Therefore, communication to a backend server can be done on port 3000, provided that there is appropriate authentication between both services.

Below is an example of the routing procedure through the Nginx container:


Routing procedure through Nginx
CRUD operations
API call
localhost:3000 external port of Nginx container
Nginx gateway
localhost:80 internal port of Nginx server
Routing
Backend endpoint
Figure 46: Nginx routing procedure

The SocketIO HTTP requests require further parameters to enable switching to different protocols, such as WebSockets. As such, each API call to a backend service for SocketIO requests and responses has `proxy_set_header Upgrade $http_upgrade;` and `proxy_set_header Connection "upgrade";` to enable communication between WebSockets through Nginx.

Design considerations

Nginx also routes HTTP requests to and from the frontend service. However, it is noted that the Nginx service and the frontend service will run in the same Docker container, and this is possible due to multi-stage builds from Docker 17.06 CE. The frontend service is now compiled into static files beforehand and passed to an Nginx image. Therefore, any API calls to the frontend will now route through Nginx, which then routes to the static files on its container.

Cloud deployment on Google Cloud Platform (GCP)

Responsibilities

1) Availability of application outside of localhost

   To ensure availability of the application, deployment of the application is done on GCP using Google Kubernetes Engine (GKE). GKE is a managed Kubernetes service for deploying, managing, and scaling containerized applications.

2) Scaling the application

   GKE automates the provisioning and scaling of resources, allowing users to create clusters that host their applications' containers.

Developers interact with GKE using `kubectl` or the Google Cloud Console, enabling them to deploy, monitor, and manage their applications effectively within the Kubernetes ecosystem provided by GKE.

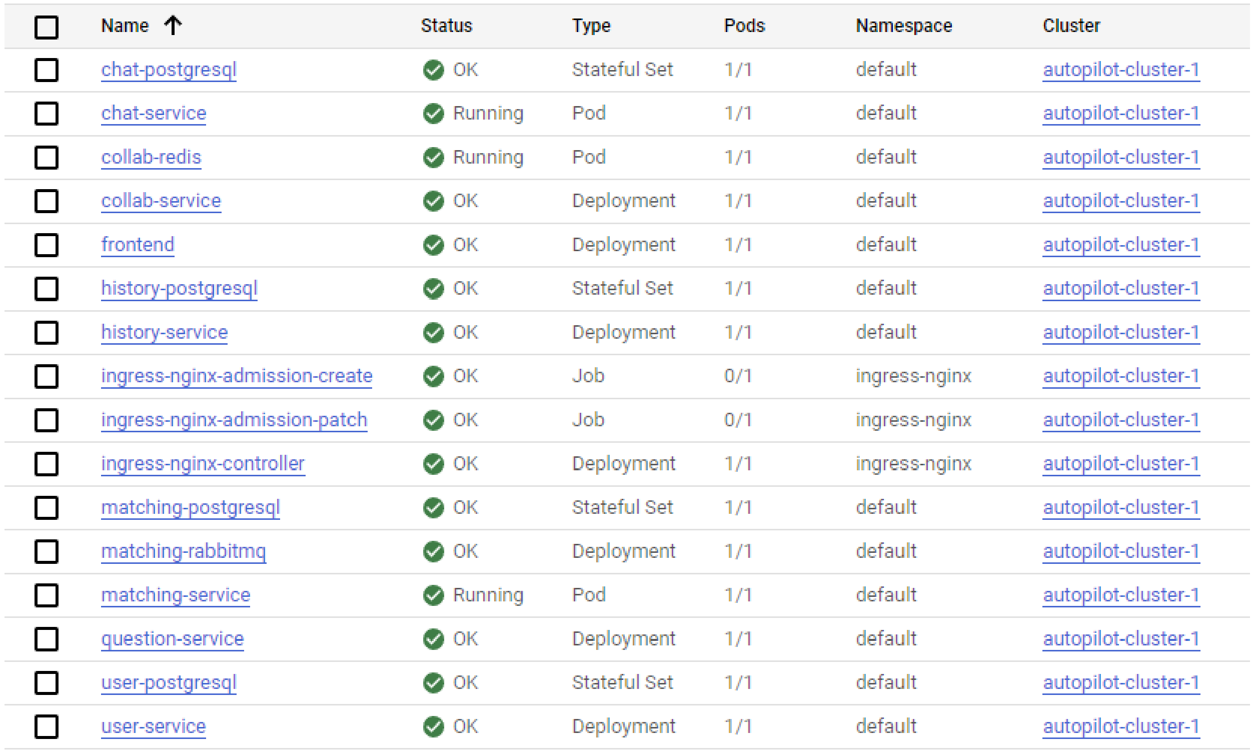
Architectural overview

An autopilot cluster is configured for streamlined scaling purposes, utilising YAML files within Google Cloud Shell for setup and application.

Each individual component such as microservices, frontend website, databases (excluding the question database), RabbitMQ, and Redis is allocated its own workload and service YAML files. Within each workload, a label is included to establish a connection with its respective service. Additionally, specifications entail the Docker image to be hosted within the container and the assignment of a port number. Service YAML files contain labels to link them to their respective workloads, set port numbers for container connectivity, and define external port numbers. Application of these files is carried out through **kubectl apply -f <yaml file>**.

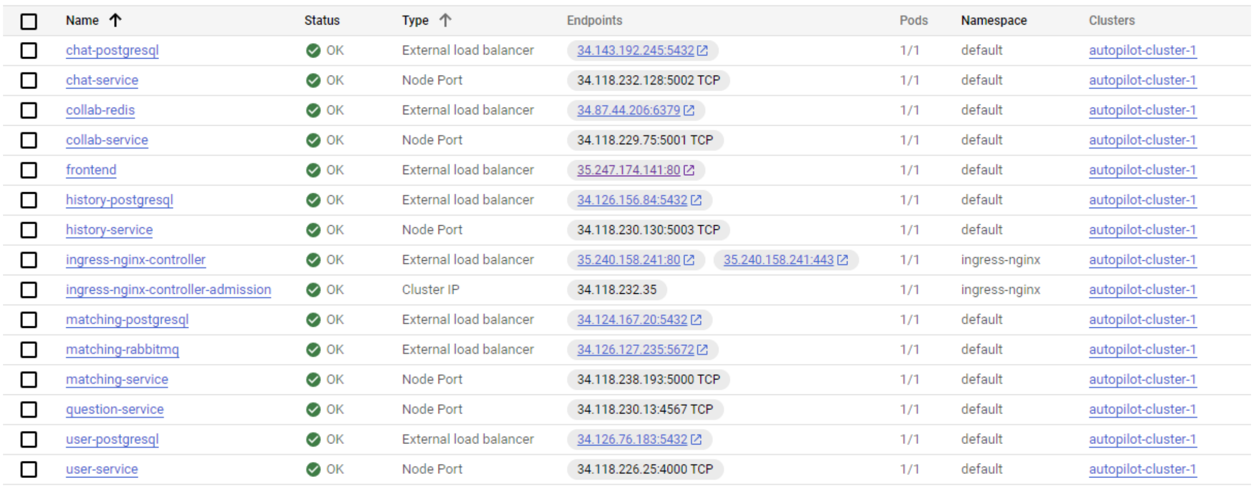
Databases, config maps and persistent volumes are created to initialise databases and ensure data retention.

Furthermore, an API gateway is configured using NGINX Ingress, effectively linking all microservices to it. Other components utilise a load balancer and have individual external IP addresses for their functionalities.



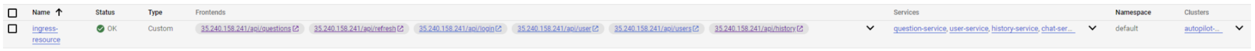
☐	Name ↑	Status	Type	Pods	Namespace	Cluster
☐	chat-postgresql	✓ OK	Stateful Set	1/1	default	autopilot-cluster-1
☐	chat-service	✓ Running	Pod	1/1	default	autopilot-cluster-1
☐	collab-redis	✓ Running	Pod	1/1	default	autopilot-cluster-1
☐	collab-service	✓ OK	Deployment	1/1	default	autopilot-cluster-1
☐	frontend	✓ OK	Deployment	1/1	default	autopilot-cluster-1
☐	history-postgresql	✓ OK	Stateful Set	1/1	default	autopilot-cluster-1
☐	history-service	✓ OK	Deployment	1/1	default	autopilot-cluster-1
☐	ingress-nginx-admission-create	✓ OK	Job	0/1	ingress-nginx	autopilot-cluster-1
☐	ingress-nginx-admission-patch	✓ OK	Job	0/1	ingress-nginx	autopilot-cluster-1
☐	ingress-nginx-controller	✓ OK	Deployment	1/1	ingress-nginx	autopilot-cluster-1
☐	matching-postgresql	✓ OK	Stateful Set	1/1	default	autopilot-cluster-1
☐	matching-rabbitmq	✓ OK	Deployment	1/1	default	autopilot-cluster-1
☐	matching-service	✓ Running	Pod	1/1	default	autopilot-cluster-1
☐	question-service	✓ OK	Deployment	1/1	default	autopilot-cluster-1
☐	user-postgresql	✓ OK	Stateful Set	1/1	default	autopilot-cluster-1
☐	user-service	✓ OK	Deployment	1/1	default	autopilot-cluster-1

Figure 47: Screenshot of workloads



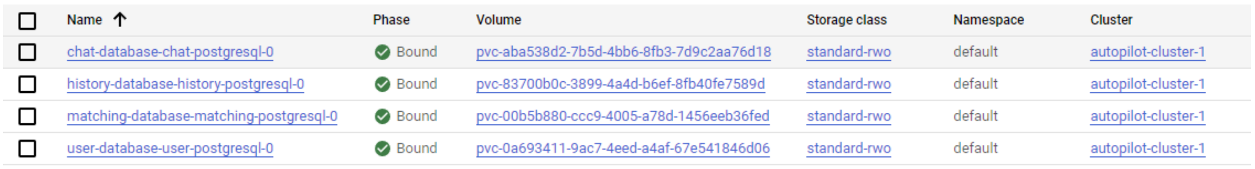
Name	Status	Type	Endpoints	Pods	Namespace	Clusters
chat-postgresql	OK	External load balancer	34.143.192.245:5432	1/1	default	autopilot-cluster-1
chat-service	OK	Node Port	34.118.232.128:5002 TCP	1/1	default	autopilot-cluster-1
collab-redis	OK	External load balancer	34.87.44.206:6379	1/1	default	autopilot-cluster-1
collab-service	OK	Node Port	34.118.229.75:5001 TCP	1/1	default	autopilot-cluster-1
frontend	OK	External load balancer	35.247.174.141:80	1/1	default	autopilot-cluster-1
history-postgresql	OK	External load balancer	34.126.156.84:5432	1/1	default	autopilot-cluster-1
history-service	OK	Node Port	34.118.230.130:5003 TCP	1/1	default	autopilot-cluster-1
ingress-nginx-controller	OK	External load balancer	35.240.158.241:80 35.240.158.241:443	1/1	ingress-nginx	autopilot-cluster-1
ingress-nginx-controller-admission	OK	Cluster IP	34.118.232.35	1/1	ingress-nginx	autopilot-cluster-1
matching-postgresql	OK	External load balancer	34.124.167.20:5432	1/1	default	autopilot-cluster-1
matching-rabbitmq	OK	External load balancer	34.126.127.235:5672	1/1	default	autopilot-cluster-1
matching-service	OK	Node Port	34.118.238.193:5000 TCP	1/1	default	autopilot-cluster-1
question-service	OK	Node Port	34.118.230.13:4567 TCP	1/1	default	autopilot-cluster-1
user-postgresql	OK	External load balancer	34.126.76.183:5432	1/1	default	autopilot-cluster-1
user-service	OK	Node Port	34.118.226.25:4000 TCP	1/1	default	autopilot-cluster-1

Figure 48: Screenshot of services



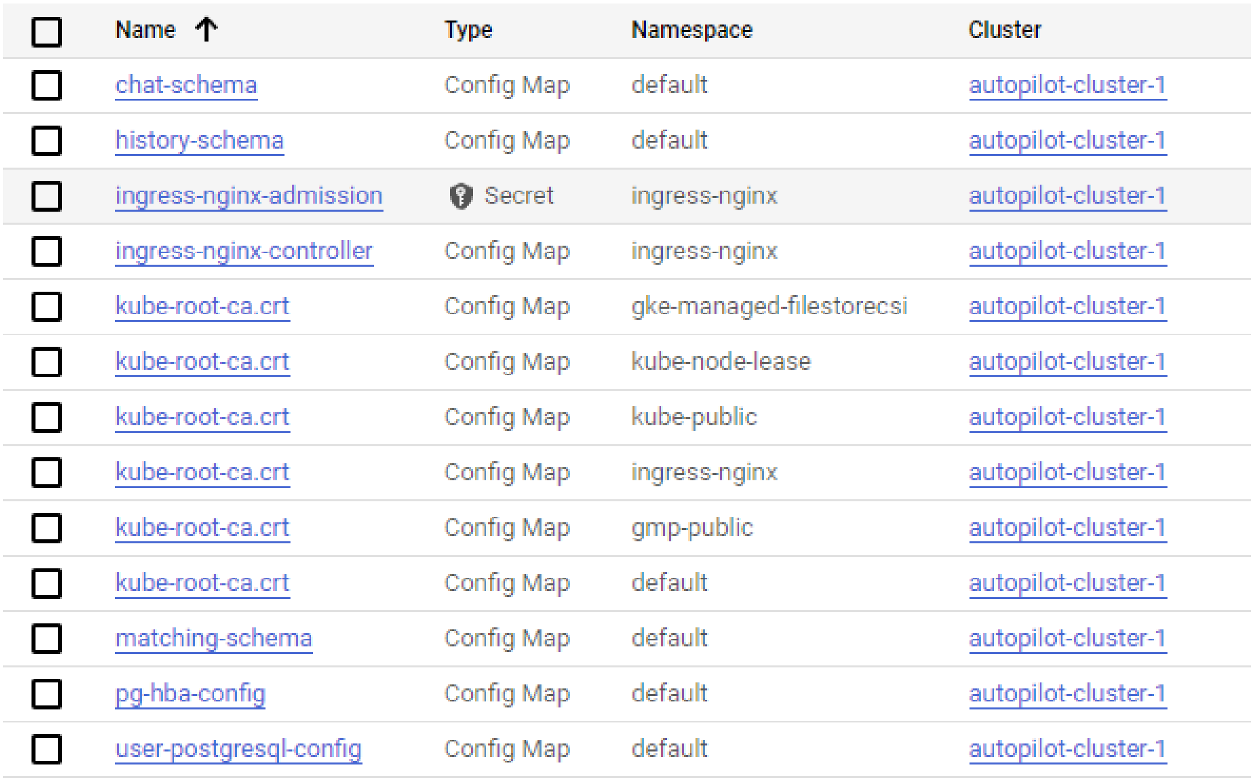
Name	Status	Type	Frontends	Services	Namespace	Clusters
ingress-resource	OK	Custom	35.240.158.241/api/questions 35.240.158.241/api/refresh 35.240.158.241/api/login 35.240.158.241/api/user 35.240.158.241/api/users 35.240.158.241/api/history	question-service, user-service, history-service, chat-ser...	default	autopilot-...

Figure 49: Screenshot of ingress



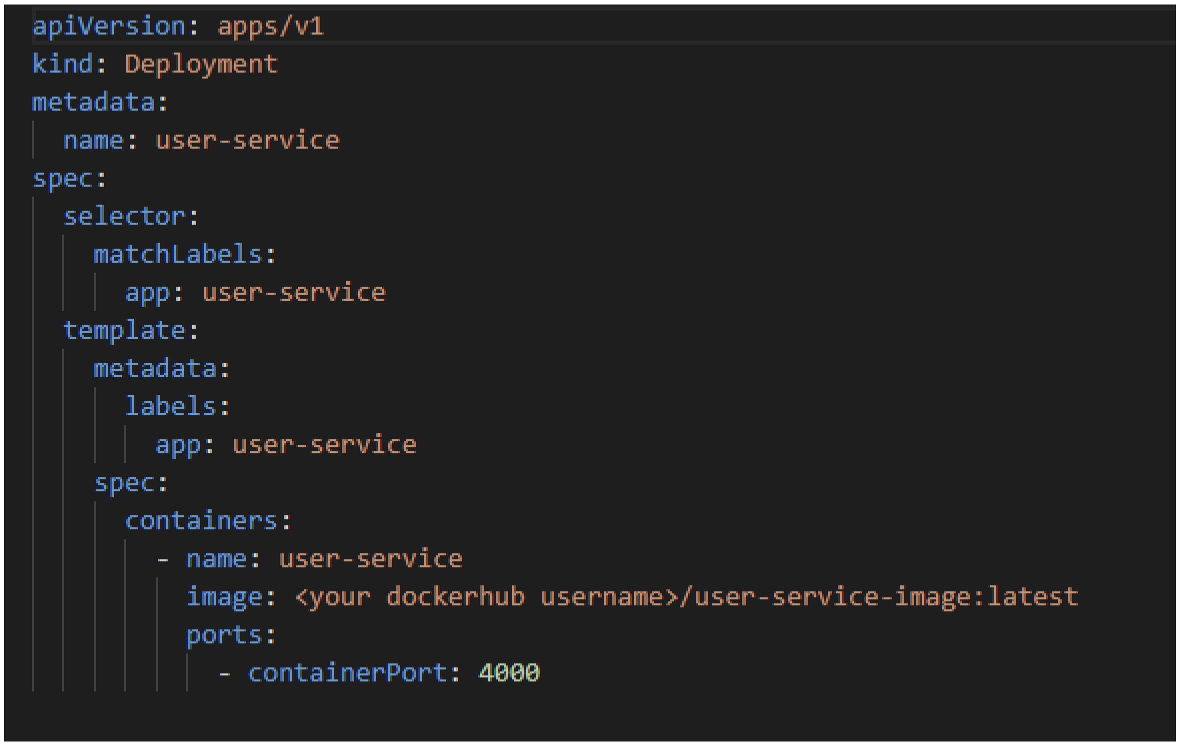
Name	Phase	Volume	Storage class	Namespace	Cluster
chat-database-chat-postgresql-0	Bound	pvc-aba538d2-7b5d-4bb6-8fb3-7d9c2aa76d18	standard-rwo	default	autopilot-cluster-1
history-database-history-postgresql-0	Bound	pvc-83700b0c-3899-4a4d-b6ef-8fb40fe7589d	standard-rwo	default	autopilot-cluster-1
matching-database-matching-postgresql-0	Bound	pvc-00b5b880-ccc9-4005-a78d-1456eeb36fed	standard-rwo	default	autopilot-cluster-1
user-database-user-postgresql-0	Bound	pvc-0a693411-9ac7-4eed-a4af-67e541846d06	standard-rwo	default	autopilot-cluster-1

Figure 50: Screenshot of persistent volumes



Name	Type	Namespace	Cluster
chat-schema	Config Map	default	autopilot-cluster-1
history-schema	Config Map	default	autopilot-cluster-1
ingress-nginx-admission	Secret	ingress-nginx	autopilot-cluster-1
ingress-nginx-controller	Config Map	ingress-nginx	autopilot-cluster-1
kube-root-ca.crt	Config Map	gke-managed-filestorecsi	autopilot-cluster-1
kube-root-ca.crt	Config Map	kube-node-lease	autopilot-cluster-1
kube-root-ca.crt	Config Map	kube-public	autopilot-cluster-1
kube-root-ca.crt	Config Map	ingress-nginx	autopilot-cluster-1
kube-root-ca.crt	Config Map	gmp-public	autopilot-cluster-1
kube-root-ca.crt	Config Map	default	autopilot-cluster-1
matching-schema	Config Map	default	autopilot-cluster-1
pg-hba-config	Config Map	default	autopilot-cluster-1
user-postgresql-config	Config Map	default	autopilot-cluster-1

Figure 51: Screenshot of config maps



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
  selector:
    matchLabels:
      app: user-service
  template:
    metadata:
      labels:
        app: user-service
    spec:
      containers:
        - name: user-service
          image: <your dockerhub username>/user-service-image:latest
          ports:
            - containerPort: 4000
```

Figure 52: Snippet of a deployment yaml file

Design Considerations

Service Decoupling: Each microservice should be designed to operate independently with minimal dependencies on other services. This decoupling ensures that changes or updates to one service don't adversely affect others, promoting system resilience.

Containerization: Utilise Docker containers to encapsulate each microservice. This approach ensures consistency across different environments and simplifies deployment on GKE, where containers are a fundamental unit of deployment.

Granular Service Boundaries: Clearly define service boundaries to ensure that each microservice handles a specific business domain. This clarity simplifies maintenance, scaling, and troubleshooting within GKE.

Resource Optimization: Optimise resource utilisation within GKE by properly configuring resource requests and limits for each microservice. Regularly monitor and adjust resource allocations based on usage patterns to achieve cost-effectiveness.

Challenges

Technical challenges

1. Google Kubernetes Engine (GKE) isn't widely adopted, making it challenging to find support or assistance when needed.
2. MIcroservice application is relatively more challenging to deploy compared to monolithic applications because of the large amount of services.
3. Unfamiliarity with the current tech stack before this project. A lot of documentation reading and trial and error had to be done to learn how to use each of the techs to develop the project.
4. Debugging JavaScript code is arguably more challenging than languages such as Java

Non-technical challenges

1. GKE is expensive to host.
2. Google's artifact repository does not have enough space to host all the docker images, resorted to using docker hub.

Suggestions

Potential Improvements	**Description**
Add suggested solutions	To allow users to view sample solutions.
Add voice/video communication	To allow users to better communicate and collaborate with each other.
Add a Queue Viewer for Matching	To allow users to see what Category, Tag, and Difficulty other users on PeerPrep are queuing for.
Allow skipping of questions while collaborating	To allow users in the collaboration room, upon approval by both parties, to skip and try another question.
Enhance scalability	To allow the app to maximise resources and adjust to varying traffic levels (such as Kubernetes horizontal pod auto-scaler)

Reflections

Learning points from the project process and the group work

When working in an Agile or Scrum environment and especially without in-depth experience in certain areas, it is sometimes hard to estimate the amount of effort required for each task. This affects planning and we had to adapt along the way.

We learned that by having a microservice architecture, applying software engineering principles such as separation of concerns, and having a good coding style, we encountered only a few merge conflicts along the way. This speeds up development and shows us that following software engineering principles is beneficial for software development.