



CS3219 AY23/24 S1

Final Report

Group 13

Team Members	Student No.	Email
Lim Hong Wei, Jovon	A0234684X	e0726684@u.nus.edu
Yeo Yiheng	A0217605J	e0543641@u.nus.edu
Wu Changjun	A0234631M	e0726631@u.nus.edu
Ho Lok Wah	A0233843E	holokwah@u.nus.edu
Ong Han Wei	A0234074L	e0726074@u.nus.edu

Content Page

1. Introduction.....	6
1.1 Background.....	6
1.2 Purpose.....	6
2. Contributions.....	7
2.1 Subgroup Contributions.....	7
2.2 Individual Contributions.....	7
3. Application Requirements.....	10
3.1 Functional Requirements.....	10
F1 User Service.....	10
F1.1 Authentication / Authorization.....	10
F1.2 User Profile.....	10
F1.3 Attempted Questions History.....	10
F2 Matching Service.....	11
F2.1 Matching Algorithm.....	11
F2.2 User Actions.....	11
F3 Question Service.....	11
F3.1 Question Repository.....	11
F4 Collaboration Service.....	12
F4.1 Environment.....	12
F4.2 Communication.....	12
3.2 Non-Functional Requirements.....	12
NF1 Deployment.....	12
NF1.1 Cloud Deployment.....	12
NF1.2 Service Discovery.....	12
NF1.3 Manageability.....	13
NF2 Usability.....	13
NF3 Reliability.....	13
NF4 Security.....	13
NF5 Scalability.....	13
4. Tech Stack & Tools.....	14
4.1 Frontend.....	14
4.2 Backend.....	15
4.3 Database.....	15
4.4 Deployment.....	15
5. Main Architecture.....	16
5.1 Microservices vs Monolithic.....	16
5.2 Overall Application Architecture.....	17
6. Frontend.....	18
6.1 Frontend Middleware.....	18

6.2 Frontend Design.....	18
6.2.1 Landing Page.....	18
6.2.2 Signup Page.....	19
6.2.3 Login Page.....	20
6.2.4 Question Page.....	21
6.2.5 Interviews Page.....	24
6.2.6 Collaboration Page.....	25
6.2.6.1 Chat Room.....	26
6.2.6.2 Code Editor.....	27
6.2.7 Profile Page.....	28
6.2.8 History Page.....	30
6.2.9 Manage Users Page.....	31
7. Individual Service Documentation.....	32
7.1 API Gateway.....	32
7.1.1 Background.....	32
7.1.2 Service Architecture.....	32
7.1.3 Endpoints.....	33
7.1.4 Auth User Schema.....	33
7.1.5 Query Sequence.....	34
7.1.5.1 Create Auth User Flow.....	34
7.1.5.2 Login Flow.....	35
7.1.5.3 Logout Flow.....	36
7.1.5.4 Refresh Flow.....	36
7.1.5.5 Delete Auth User.....	37
7.1.5.6 Upgrade to Super Admin Role Flow.....	38
7.1.5.7 Upgrade Auth User Role Flow.....	39
7.1.5.8 Downgrade Auth User Role Flow.....	40
7.1.5.9 Get Auth User Info Flow.....	40
7.1.5.10 Get Auth Users Info Flow.....	41
7.2 User Service.....	42
7.2.1 Background.....	42
7.2.2 Endpoints.....	42
7.2.3 Components.....	42
7.2.3.1 User Schema.....	42
7.2.3.2 History Schema.....	43
7.2.4 Query Sequence.....	43
7.2.4.1 Create User.....	43
7.2.4.2 Get User.....	44
7.2.4.3 Delete User.....	45
7.2.4.4 Update User.....	45
7.2.4.5 Create History.....	46

7.2.4.6 Get Histories.....	47
7.3 Question Service.....	48
7.3.1 Background.....	48
7.3.2 Endpoints.....	48
7.3.3 Components.....	48
7.3.3.1 Questions Collection Schema.....	48
7.3.3.2 Categories Collection Schema.....	48
7.3.4 Query Sequence.....	49
7.3.4.1 Create Question.....	49
7.3.4.2 Delete Question.....	50
7.3.4.3 Get Question.....	50
7.3.4.4 Get Questions.....	50
7.3.4.5 Get Random Question Id.....	51
7.3.4.6 Get Categories.....	51
7.3.5 Leetcode Serverless Function.....	51
7.3.6 Database Seeding.....	52
7.4 Matching Service.....	53
7.4.1 Background.....	53
7.4.2 Service Architecture.....	53
7.4.3 API Endpoints.....	54
7.4.4 Components.....	55
7.4.4.1 Queueing Mechanism.....	55
7.4.4.2 Cancel Mechanism.....	56
7.4.4.3 Length Tracking Mechanism.....	57
7.5 Collaboration Service.....	59
7.5.1 Background.....	59
7.5.2 Service Architecture.....	59
7.5.3 API Endpoints.....	59
7.5.4 Components.....	59
7.5.4.1 Hub Schema.....	59
7.5.4.2 Room Schema.....	60
7.5.4.3 User Schema.....	60
7.5.5 Query Sequence.....	61
7.5.5.1 Create Room.....	61
7.5.5.2 Join Room.....	62
7.5.5.3 Get Question Id.....	62
7.5.6 Message Mechanism.....	62
7.5.6.1 Message Schema.....	62
7.5.8.2 Message Types.....	63
7.5.8.3 Reading Messages.....	63
7.5.8.4 Writing Messages.....	63

7.5.8.5 Disconnection.....	64
8. Deployment.....	65
8.1 Summary.....	65
8.1.1 Replica Sets.....	65
8.1.2 Horizontal pod auto-scaler & Service discovery.....	66
8.1.3 Ingress.....	66
8.2 Significant Decisions.....	67
8.2.1 Cloud Service Choice.....	67
8.2.2 Cluster Mode.....	68
8.2.3 Database Choice.....	68
9. Project Management.....	69
9.1 Progress Tracking.....	69
9.2 Technical Communication.....	70
10. Suggestions for improvements.....	71
11. Reflections and learning points.....	73

1. Introduction

1.1 Background

In the current technical landscape, it is imperative for students to be adequately prepared for countless technical interviews that would stand in their way of landing a job. Oftentimes, simply practicing problem solving on their own is insufficient. In most technical interviews, the articulation and communication of thoughts are just as important as being able to solve the problem itself. Hence, the optimal way to prepare for these interviews would be through pair programming, where a student can practice communication and problem solving alongside another like-minded peer. This paves the way for PeerPrep to be introduced.

1.2 Purpose

PeerPrep is a technical interview preparation platform and peer matching system, where students can find peers to practice whiteboard-style interview questions together. PeerPrep comes equipped with a wide array of commonly tested interview questions, a multitude of language configurations, and extremely useful collaborative functionalities like real-time live coding and a messaging system. These features ensure that PeerPrep provides adequate technical interview preparation for students.

2. Contributions

2.1 Subgroup Contributions

	Members	N2H's
Subgroup 1	Changjun Han Wei Jovon	N1 N2 N5
Subgroup 2	Ho Lok Wah Yeo Yiheng	N9 N10 N12

2.2 Individual Contributions

Contributor	Contributions
Lim Hong Wei, Jovon	<ul style="list-style-type: none">• Developed and integrated the question repository for Assignment 2• Developed the frontend for the matching service for Assignment 5• Developed the frontend of the collaboration service• Integrated retrieval of random question for the collaboration service• Handled notifications between 2 users whenever one joins/leaves/refreshes during collaboration• Developed the chat service between 2 users in the collaboration room
Ho Lok Wah, Thomas	<ul style="list-style-type: none">• Developed and integrated the user service for Assignment 2• Developed and integrated the real time collaboration feature for the project• Designed the deployment architecture for PeerPrep• Deployed PeerPrep onto GKE by setting up deployment, service, configmap, secret and ingress files. Also set up various resources such as IP address, GKE cluster, GCP Postgres instances and MongoDB Atlas• Kept track of the project's progress by adding Github issues and setting internal deadlines

	<ul style="list-style-type: none"> • Initiated the use of development Dockerfiles and docker compose files to minimize environmental errors and create a smoother developing experience • Created a linting workflow file and deploy it to Github Actions • Main point of contact with assignment TAs and mentor
Yeo Yiheng	<ul style="list-style-type: none"> • Developed assignment 5 backend matching algorithm for matching users based on question difficulty • Reworked assignment 5 to accommodate k8s distributed architecture which allows matching-service to still function when there is > 1 matching pod instance • Developed and integrated assignment 4 containerisation of backend and frontend environments. Segregated docker environments into development with hot reloading and production with minimal and lean image size via multi-stage docker builds • Deployment and setup of PeerPrep onto a GCP VM (Compute Engine) for milestone evaluation • Deployment and set up of PeerPrep onto GKE by automating image building and pushing onto Google Container Registry using Makefile, and also automating service deployment into GKE cluster. Additionally, helped set up GKE VPC to interact with MongoDB Atlas network via network peering and cloud NAT • Set up Github issues with meaningful statuses
Ong Han Wei	<ul style="list-style-type: none"> • Developed authentication and authorization service for Assignment 3 • Set up NextJs, TailwindCSS and NextUI for the frontend • Developed CRUD operation for managing questions and make the page usable in displaying information in an organized manner for Assignment 1 • Developed frontend login and sign out page • Developed frontend for profile page • Implemented syntax highlighting, code formatting and IntelliSense(for some languages) on the editor for the collaboration page • Developed serverless function to retrieve LeetCode questions for Assignment 6

Wu Changjun	<ul style="list-style-type: none">• Developed CRUD operation for managing questions for Assignment 1• Set up Redux Persist to manage persistent data through local storage for Assignment 1• Developed landing page and navigation bar for Assignment 1• Developed error handling for adding questions in Assignment 1• Developed CRUD operation for managing user profile for Assignment 2• Developed login and sign out functionality in Assignment 2• Developed frontend home page and profile page in Assignment 2• Developed a history feature to track and record the questions attempted by the user
-------------	--

3. Application Requirements

3.1 Functional Requirements

F1 User Service	Priority
F1.1 Authentication / Authorization	
F1.1.1 Users must be able to log in via GitHub OAuth	High
F1.1.2 API Gateway must validate user's cookie before forwarding requests to other microservices	High
F1.1.3 User must be able to log out of their accounts through the logout button at the profile page	High
F1.1.4 User's access cookie must be valid for only 1 minute	High
F1.1.5 User's access cookie must be able to be re-generated by a refresh cookie	High
F1.1.6 User's access cookie from refresh must be valid for only 1 day	High
F1.1.7 There should be 3 different roles for users, which are normal user, admin and super admin. Only a super admin can upgrade a normal user to an admin. Only an admin can add or delete the questions	High
F1.2 User Profile	
F1.2.1 Users must be able to view their details at the profile page, such as: username, preferred language, and profile picture	High
F1.2.2 Users must be able to update their details at the profile page, such as: username, preferred language, and profile picture	High
F1.2.3 Users must be able to delete their account at the profile page	High
F1.2.4 Users must be able to create an account via GitHub OAuth	High
F1.3 Attempted Questions History	

F1.3.1 Users must be able to view his / her history of the questions attempted during collaboration	Medium
F1.3.2 History of the questions attempted must be recorded when a user leaves a room	Medium
F1.3.3 History of the questions should be displayed in order of descending time	Medium
F2 Matching Service	
F2.1 Matching Algorithm	
F2.1.1 Users must be able to match by the same question difficulty level, which are easy, medium, and hard	High
F2.1.2 The server should time out the user after 30 seconds if a match is not found. The frontend should inform the user that the match is unsuccessful	High
F2.1.3 Users should be redirected to the collaboration room after match is successful	High
F2.2 User Actions	
F2.2.1 Users should be allowed to cancel matching midway	High
F2.2.2 Users should be able to change the difficulty to be matched on	High
F3 Question Service	
F3.1 Question Repository	
F3.1.1 Admin must be able to perform repository fetching of questions to populate the question table	High
F3.1.2 Admin must be able to create new questions. There will be a button above the question table. Once the button is pressed, there will be a modal to add the necessary fields	High
F3.1.3 Admin must be able to delete existing questions. There will be a delete button next to each question. Once the button is pressed, a modal is needed to confirm the delete action	High

F3.1.4 Users must be able to view questions in a sorted manner by the question index	Medium
F4 Collaboration Service	
F4.1 Environment	
F4.1.1 Users must be able to edit in the same code editor and see real time changes	High
F4.1.2 Users must be able to view the question whilst coding	High
F4.1.3 Only a maximum of 2 users can be in the same room	Medium
F4.1.4 Users must be able to switch and toggle between multiple programming languages	Medium
F4.2 Communication	
F4.2.1 Users must be able to communicate with one another via text chat in real time and be notified when a text is sent	Medium
F4.2.2 Users must be notified whenever the other user leaves / joins the room	Medium
F4.2.3 Users must be able to see the name of whom they are matched with	Low

3.2 Non-Functional Requirements

NF1 Deployment	Priority
NF1.1 Cloud Deployment	
NF1.1.1 Application should be deployed on GCP GKE	Medium
NF1.1.2 Application should have an Ingress to redirect frontend requests to the frontend pages and API requests to API Gateway	Medium
NF1.2 Service Discovery	
NF1.2.1 Application should be able to automatically locate the other microservices on the network	High

NF1.3 Manageability	
NF1.3.1 Application should provide useful timestamped error logs	Low
NF2 Usability	
NF2.1 Provide clear error messages for the users to understand what went wrong	High
NF2.2 Able to load up any page in the application under 5 seconds	Medium
NF2.3 Optimize the number of interactions the user needs when using the application	Low
NF2.4 Clear labeling and tool tip for components in pages	Low
NF3 Reliability	
NF3.1 The application should be able to support at least 50 questions in the database	Medium
NF3.2 The application should achieve a 99.9% availability	Medium
NF4 Security	
NF4.1 Non-authenticated users should not be able to access any page other than the landing page, signup page and login page	High
NF4.2 Non-authenticated users should be redirected back to the landing page	High
NF5 Scalability	
NF5.1 Application should support horizontal scaling of up to a maximum of 3 pods if CPU usage is more than 80% for a single instance	Medium
NF5.2 Application should support vertical scaling of cloud resources if database capacity hits the 75% threshold	Low

4. Tech Stack & Tools

4.1 Frontend

Technology	Reasons
Next.js	<ul style="list-style-type: none">• Automatic code splitting, so it has small bundle sizes and thus quicker loading time• Built-in Routing, with each file in the <code>app</code> directory corresponding to a specific route
Tailwind CSS	<ul style="list-style-type: none">• Highly flexible and customizable options for styling• Has a comprehensive set of predefined classes, enables rapid and efficient styling without CSS files
NextUI	<ul style="list-style-type: none">• Pre-built UI components• Reduce the need for us to design everything from scratch
Redux-Persist	<ul style="list-style-type: none">• Manage persistent states• Global states to minimize coupling and props passing• Ensures user preferences and data persist beyond individual sessions
Linting	<ul style="list-style-type: none">• Enforce code standards and style guidelines across the codebase• Help to identify potential errors and bugs in the code• Enhance collaboration as it ensures consistent coding standards

4.2 Backend

Technology	Reasons
Golang	<ul style="list-style-type: none">• Fast performance speed• Lean concurrency model with goroutines and channels• Strong standard library

4.3 Database

Technology	Reasons
MongoDB Atlas (Question Service)	<ul style="list-style-type: none">• Efficient storage and retrieval of questions• Support horizontal scalability to handle the growing volume of interview questions effortlessly• Clean GUI to view database schema
PostgreSQL (User service & API Gateway)	<ul style="list-style-type: none">• Handles structured and interconnected data, making it suitable for storing user profiles, authentication details and history records• ACID Compliance - crucial for maintaining accurate and reliable user information and history records

4.4 Deployment

Technology	Reasons
Docker (Containerization)	<ul style="list-style-type: none">• Fast deployment• Speedy set up for development environment• Revision control
Kubernetes (Orchestration)	<ul style="list-style-type: none">• High scalability and availability• Revision control

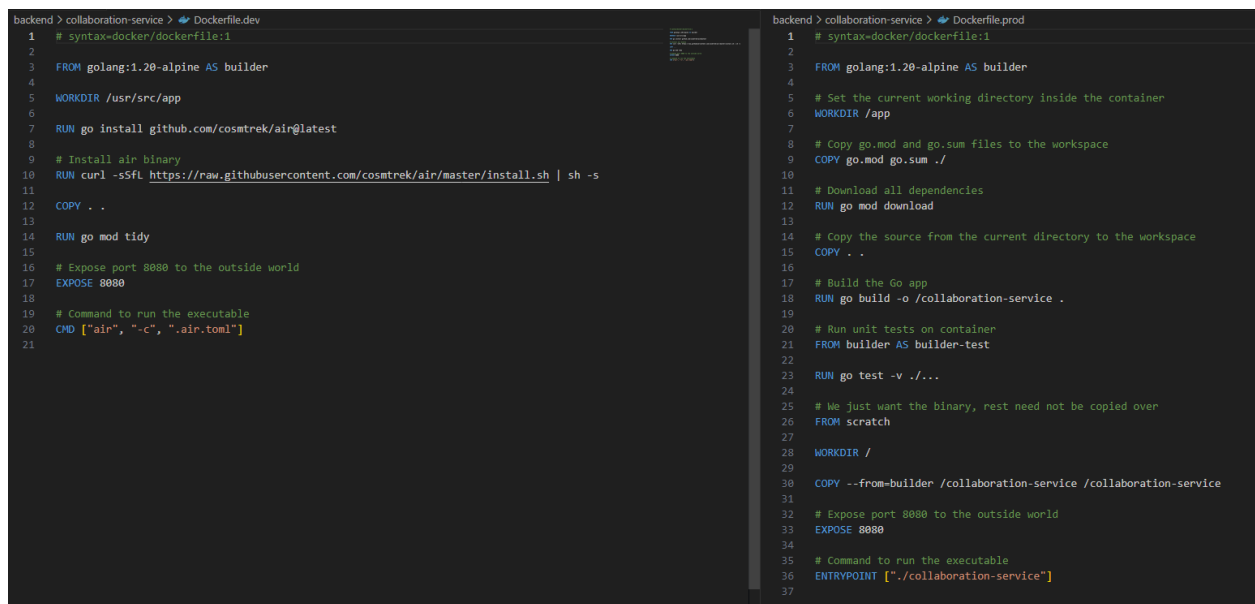
5. Main Architecture

5.1 Microservices vs Monolithic

When choosing the architectural style of our application, we decided between microservices and monolithic.

While a monolithic application is easier to develop and debug during testing, it lacks modularity and scalability. Different components in the application are also strongly coupled, which makes it hard to coordinate between us since changes must be made throughout the application.

Instead, we decided on the microservices architecture style since each microservice is designed to handle a specific function, which aligns with the Single Responsibility Principle. This makes it easier for us to maintain and develop each service without affecting the implementation of other services since they are loosely coupled. To save the trouble of setting up the development environment across different devices, we have made one version of Dockerfile.dev for each service that is dedicated to the development environment. Dockerfile.dev has hot reload functionality that allows developers to review any changes instantaneously without having to rebuild the containers again, while Dockerfile.prod makes use of multi-stage build and has a leaner image size that is suitable for production environments.



```
backend > collaboration-service > Dockerfile.dev
1 # syntax=docker/dockerfile:1
2
3 FROM golang:1.20-alpine AS builder
4
5 WORKDIR /usr/src/app
6
7 RUN go install github.com/cosmtrek/air@latest
8
9 # Install air binary
10 RUN curl -sSfL https://raw.githubusercontent.com/cosmtrek/air/master/install.sh | sh -s
11
12 COPY . .
13
14 RUN go mod tidy
15
16 # Expose port 8080 to the outside world
17 EXPOSE 8080
18
19 # Command to run the executable
20 CMD ["air", "-c", ".air.toml"]
21

backend > collaboration-service > Dockerfile.prod
1 # syntax=docker/dockerfile:1
2
3 FROM golang:1.20-alpine AS builder
4
5 # Set the current working directory inside the container
6 WORKDIR /app
7
8 # Copy go.mod and go.sum files to the workspace
9 COPY go.mod go.sum ./
10
11 # Download all dependencies
12 RUN go mod download
13
14 # Copy the source from the current directory to the workspace
15 COPY . .
16
17 # Build the Go app
18 RUN go build -o /collaboration-service .
19
20 # Run unit tests on container
21 FROM builder AS builder-test
22
23 RUN go test -v ./...
24
25 # We just want the binary, rest need not be copied over
26 FROM scratch
27
28 WORKDIR /
29
30 COPY --from=builder /collaboration-service /collaboration-service
31
32 # Expose port 8080 to the outside world
33 EXPOSE 8080
34
35 # Command to run the executable
36 ENTRYPOINT ["/collaboration-service"]
37
```

Figure 1: Difference between Dockerfile.dev and Dockerfile.prod

5.2 Overall Application Architecture

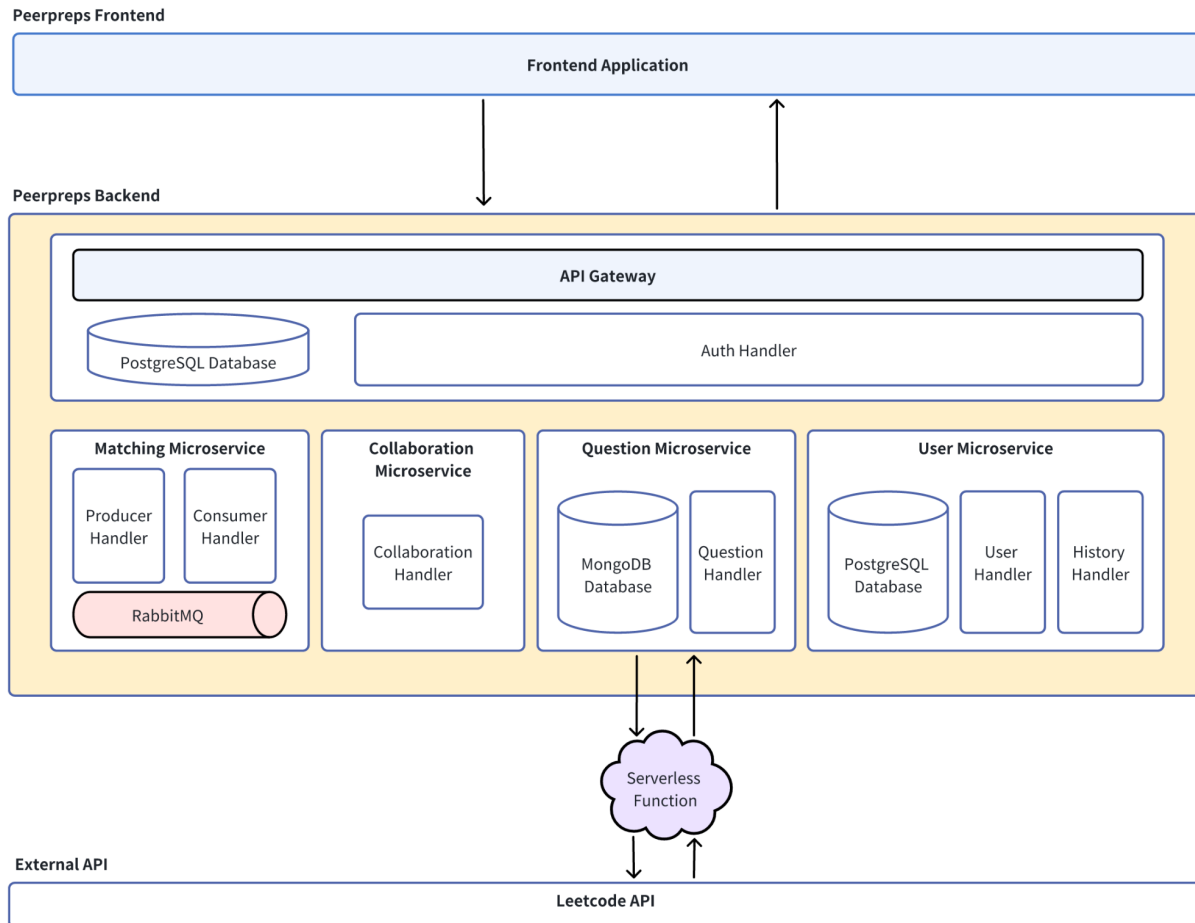


Figure 2: Overall application architecture diagram

6. Frontend

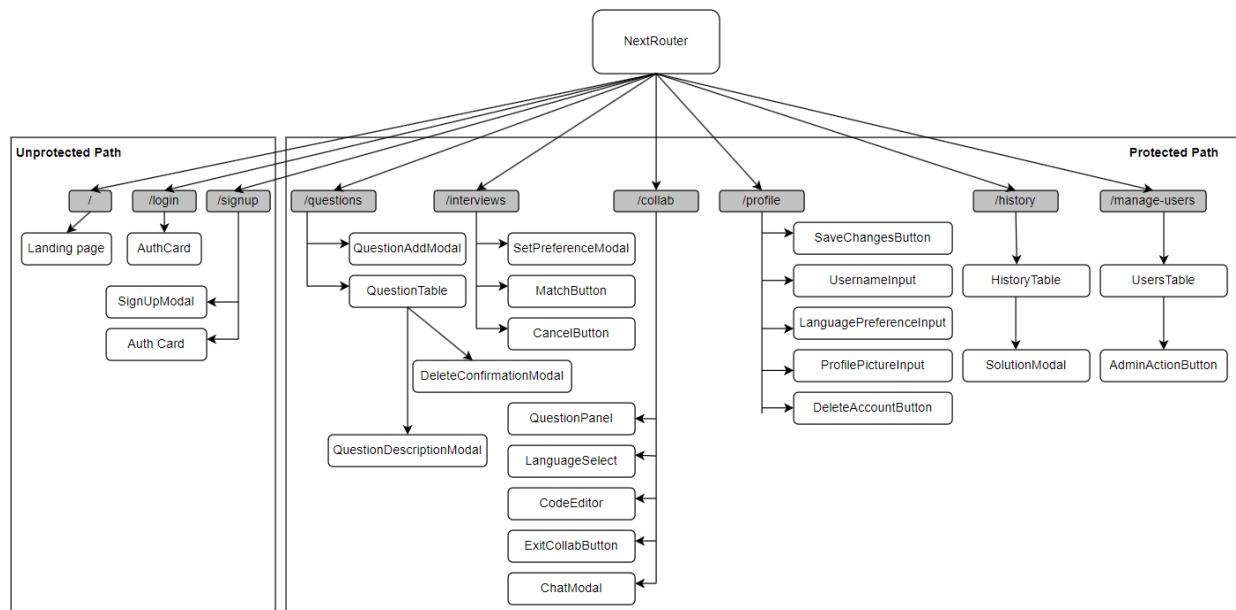


Figure 3: Frontend Component Tree

6.1 Frontend Middleware

We leverage Next.js Middleware to handle the routes in the route end. This middleware would be executed before the routes are matched and redirects the user in the event where the user tries to access a page they are not supposed to access ([NF4](#)).

When an unauthenticated user tries to access any routes other than the landing, login or signup pages, they would be redirected back to the landing page. Conversely, when an authenticated user tries to access the landing, login and signup page, they would be redirected back to the questions page.

6.2 Frontend Design

6.2.1 Landing Page

Upon visiting our platform, users will see a landing page to encourage them to sign up and embark on their PeerPrep journey.

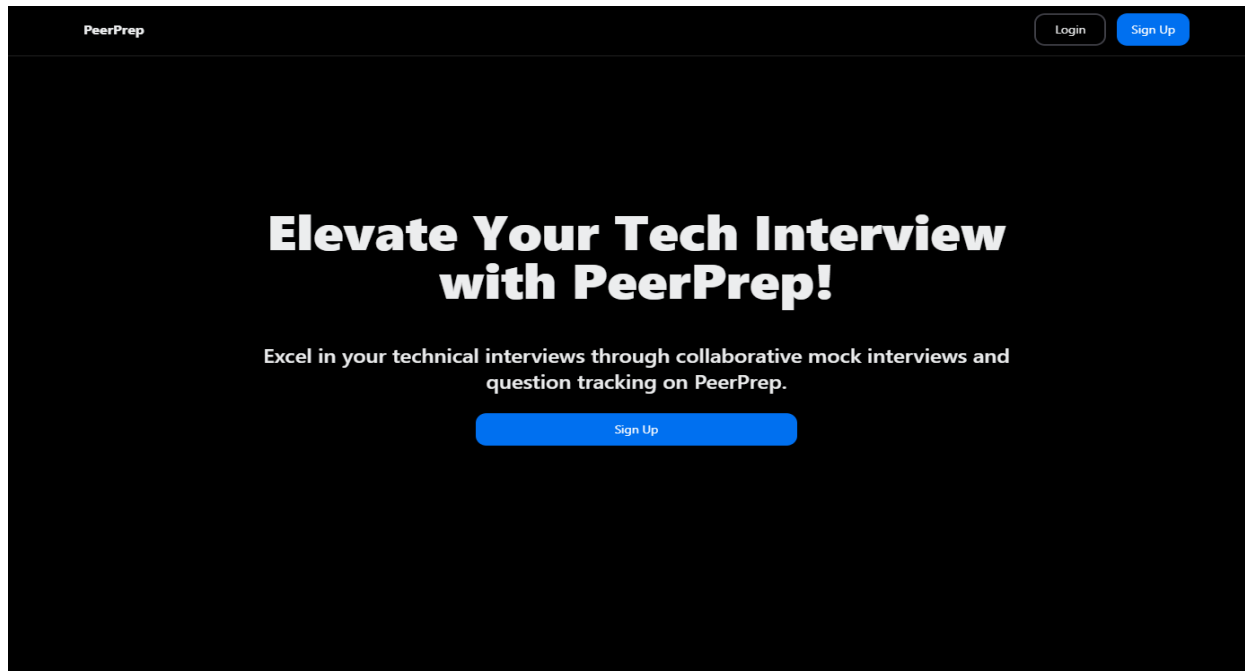


Figure 4: Landing Page

6.2.2 Signup Page

Upon clicking the sign up button on the landing page, users will navigate to our signup page where it prompts them to sign up via their GitHub account.

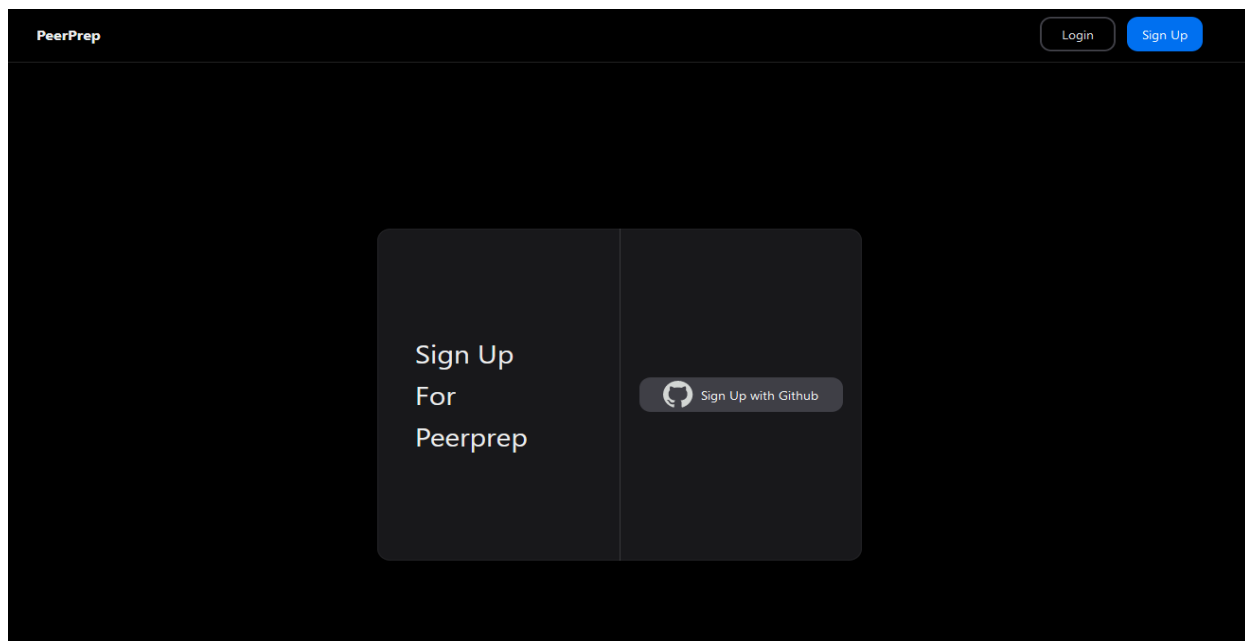


Figure 5: Signup Page

The user will have to fill up the Sign Up Form to complete the signup process.

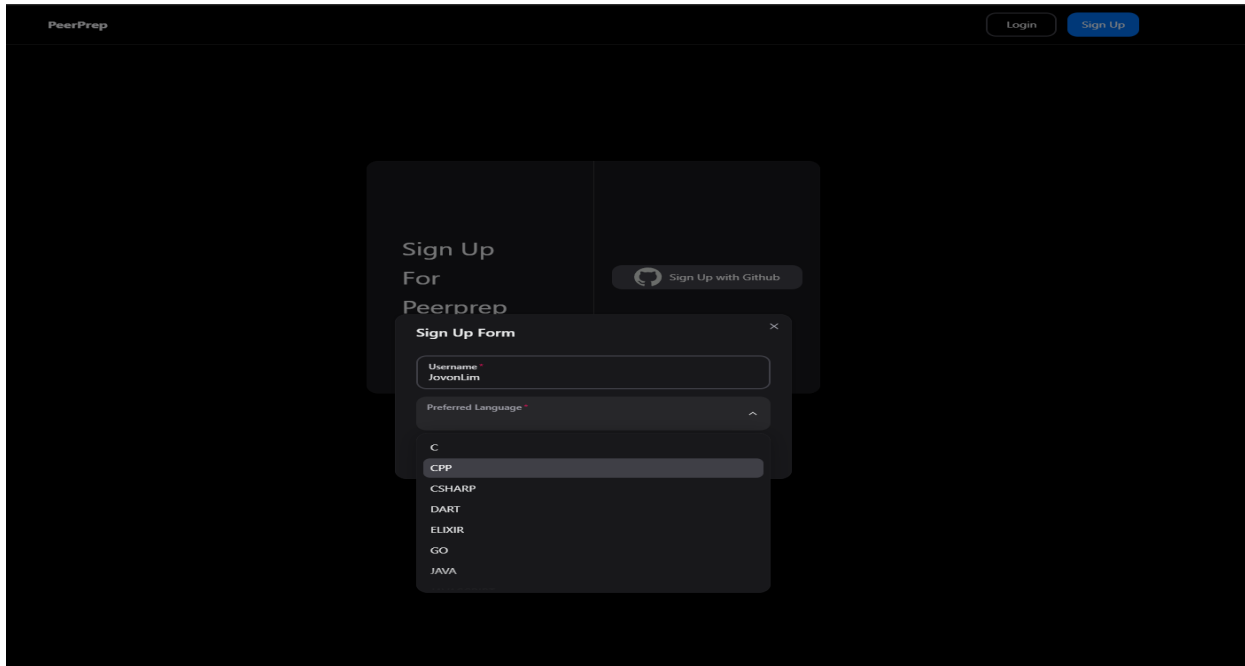
The screenshot shows the PeerPrep website's sign-up interface. At the top, the 'PeerPrep' logo is on the left, and 'Login' and 'Sign Up' buttons are on the right. The main content area features a dark background with a light gray box containing the text 'Sign Up For Peerprep'. To the right of this text is a button labeled 'Sign Up with Github'. Overlaid on this is a 'Sign Up Form' modal. The modal has a title bar with a close button. It contains two input fields: 'Username' with the value 'JovonLim' and 'Preferred Language' with a dropdown menu. The dropdown menu is open, showing a list of programming languages: C, CPP (which is highlighted), CSHARP, DART, ELIXIR, GO, and JAVA.

Figure 6: Sign Up Form

6.2.3 Login Page

After signing up, the user will be redirected to the login page.

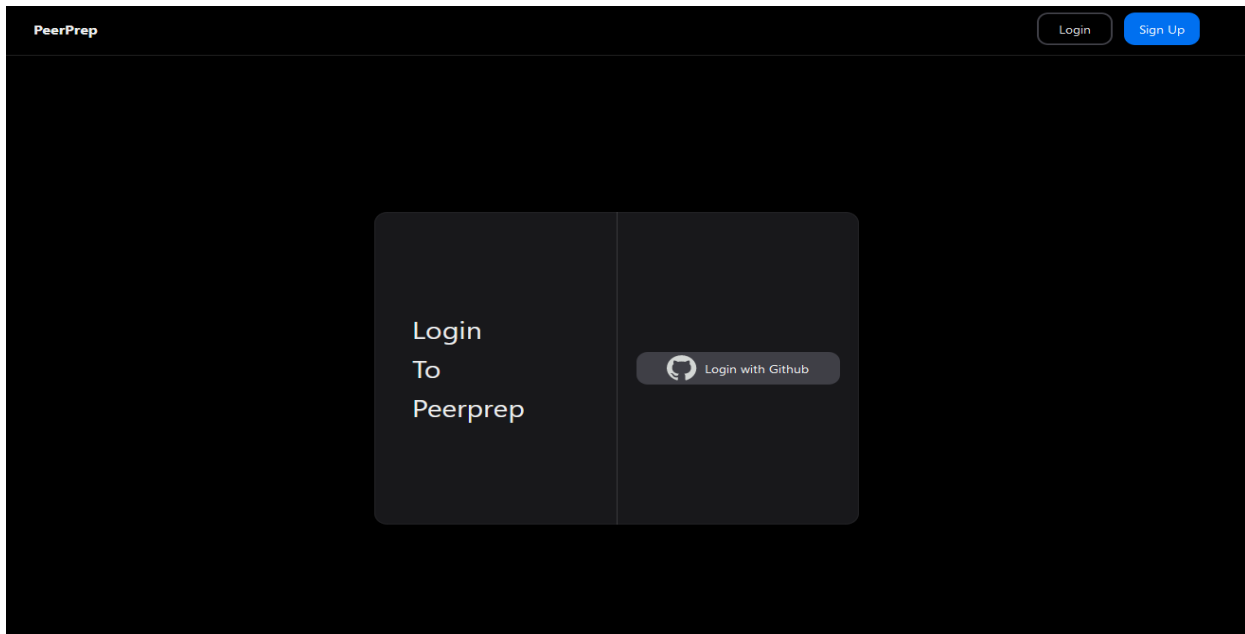
The screenshot shows the PeerPrep website's login interface. At the top, the 'PeerPrep' logo is on the left, and 'Login' and 'Sign Up' buttons are on the right. The main content area features a dark background with a light gray box containing the text 'Login To Peerprep'. To the right of this text is a button labeled 'Login with Github'.

Figure 7: Login Page

6.2.4 Question Page

Once logged in, users gain access to the question table.

ID	TITLE	CATEGORY	COMPLEXITY	ACTIONS
1	Reverse a String	Strings Algorithms	Easy	👁
2	Course Schedule	Data Structures Algorithms	Medium	👁
3	Add Binary	Bit Manipulation Algorithms	Easy	👁
4	Longest Common Subsequence	Strings Algorithms	Medium	👁
5	Chalkboard XOR Game	Brain Teaser	Hard	👁
6	Two Sum	Array Hash Table	Easy	👁
7	Add Two Numbers	Linked List Math Recursion	Medium	👁
8	Longest Substring Without Repeating Characters	Hash Table String Sliding Window	Medium	👁
9	Median of Two Sorted Arrays	Array Binary Search Divide and Conquer	Hard	👁
10	Longest Palindromic Substring	String Dynamic Programming	Medium	👁

Figure 8: Question Page

Normal users can engage with the questions by simply clicking on the eye icon, and a question description modal will open up for them to attempt the questions in their free time.

Longest Substring Without Repeating Characters

Given a string s , find the length of the **longest substring** without repeating characters.

Example 1:
Input: $s = \text{"abcabcbb"}$
Output: 3
Explanation: The answer is "abc", with the length of 3.

Example 2:
Input: $s = \text{"bbbbbb"}$
Output: 1
Explanation: The answer is "b", with the length of 1.

Example 3:
Input: $s = \text{"pwwkew"}$
Output: 3
Explanation: The answer is "wke", with the length of 3. Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

- $0 \leq s.length \leq 5 * 10^4$
- s consists of English letters, digits, symbols and spaces.

Figure 9: Question Description Modal

Admin user's view on the question table, they have the ability to add and delete questions.

Question Bank

Add Question

ID	TITLE	CATEGORY	COMPLEXITY	ACTIONS
1	Reverse a String	Strings Algorithms	Easy	
2	Course Schedule	Data Structures Algorithms	Medium	
3	Add Binary	Bit Manipulation Algorithms	Easy	
4	Longest Common Subsequence	Strings Algorithms	Medium	
5	Chalkboard XOR Game	Brain Teaser	Hard	
6	Two Sum	Array Hash Table	Easy	
7	Add Two Numbers	Linked List Math Recursion	Medium	
8	Longest Substring Without Repeating Characters	Hash Table String Sliding Window	Medium	
9	Median of Two Sorted Arrays	Array Binary Search Divide and Conquer	Hard	
10	Longest Palindromic Substring	String Dynamic Programming	Medium	

<

1

2

3

4

5

...

11

>

Figure 10: Admin's Question Page

To add new questions, admin users have to fill up all the required fields in the form below.

Question Bank

Add Question

ID	TITLE
1	Reverse a String
2	Course Schedule
3	Add Binary
4	Longest Common Subsequence
5	Chalkboard XOR Game
6	Two Sum
7	Add Two Numbers
8	Longest Substring Without Repeating Characters
9	Median of Two Sorted Arrays
10	Longest Palindromic Substring

Add Question

Title *

Reverse a String

Complexity *

Select Complexity

Complexity is required

Category *

Select Categories

Category is required

Description *

Enter Question Description (Markdown Syntax)

Description is required

Close

Add





















COMPLEXITY	ACTIONS
Easy	 
Medium	 
Easy	 
Medium	 
Hard	 
Easy	 
Medium	 
Medium	 
Hard	 
Medium	 

Figure 11: Add Question Modal - Error Checks

Admin users are not allowed to add questions with the same title to ensure the uniqueness of the question database, an error will be thrown with the toast.

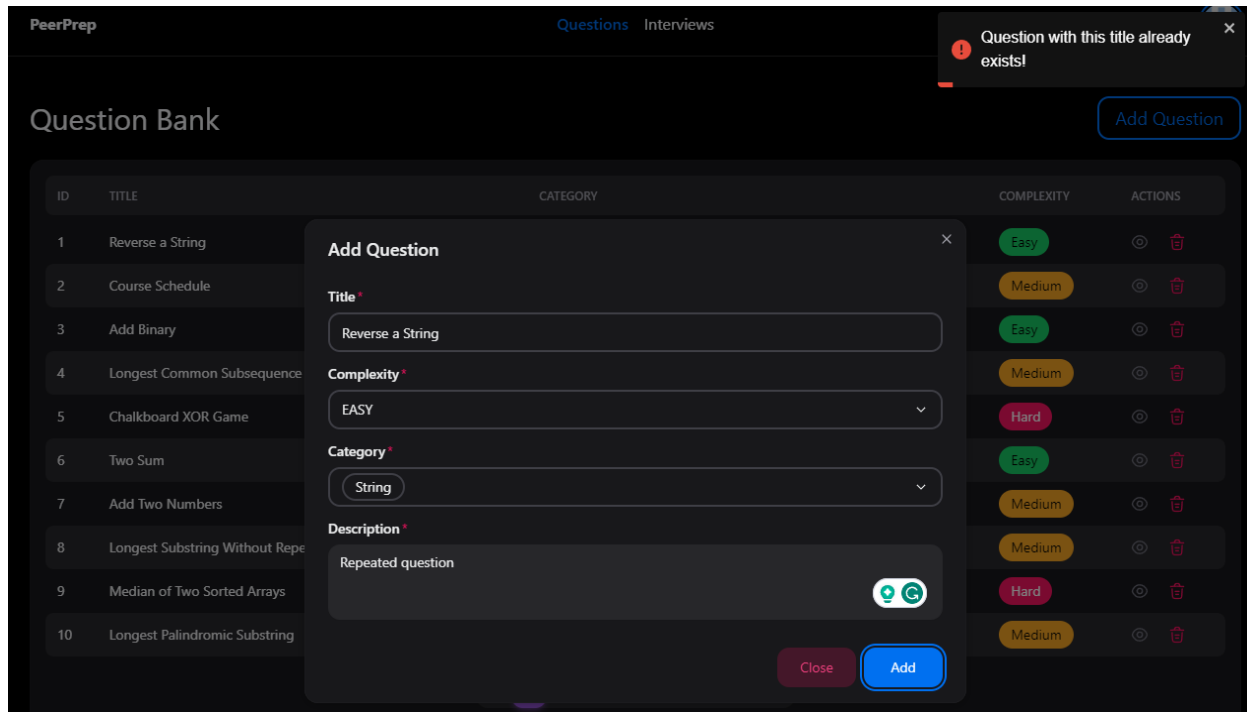


Figure 12: Add Question Error - Title Already Exists

Admin users can also delete questions

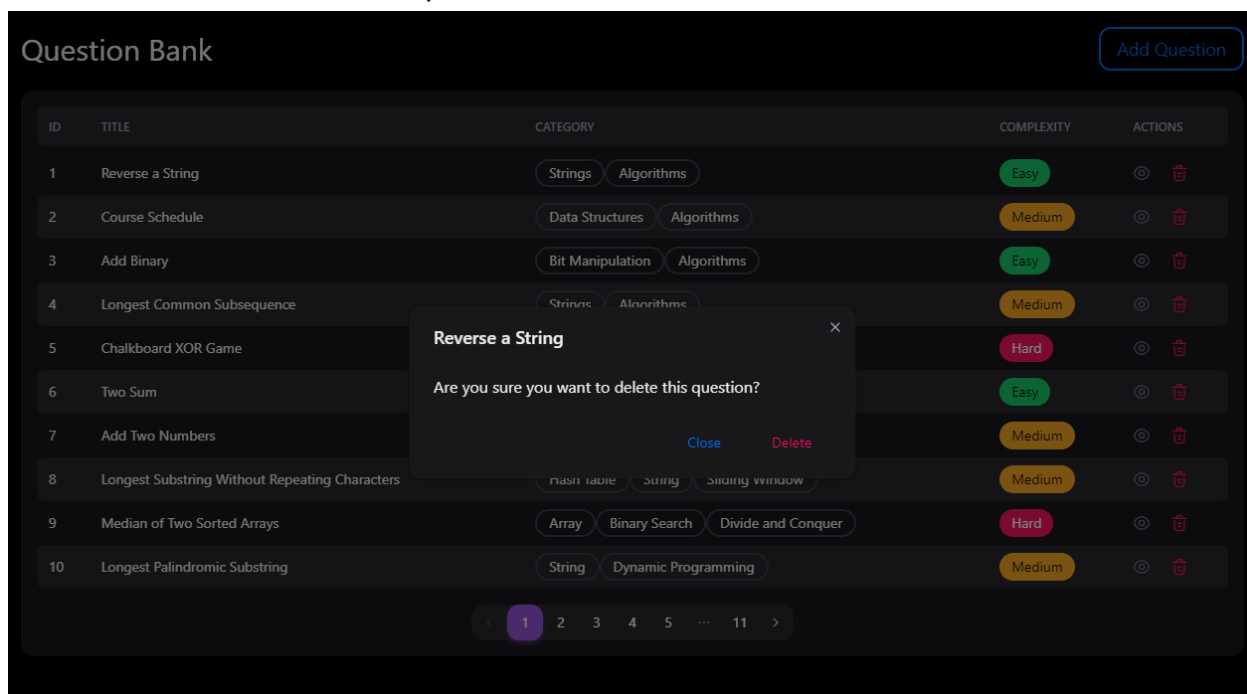


Figure 13: Delete Question Modal

6.2.5 Interviews Page

This page allows users to specify their preferred complexity level for matchmaking.

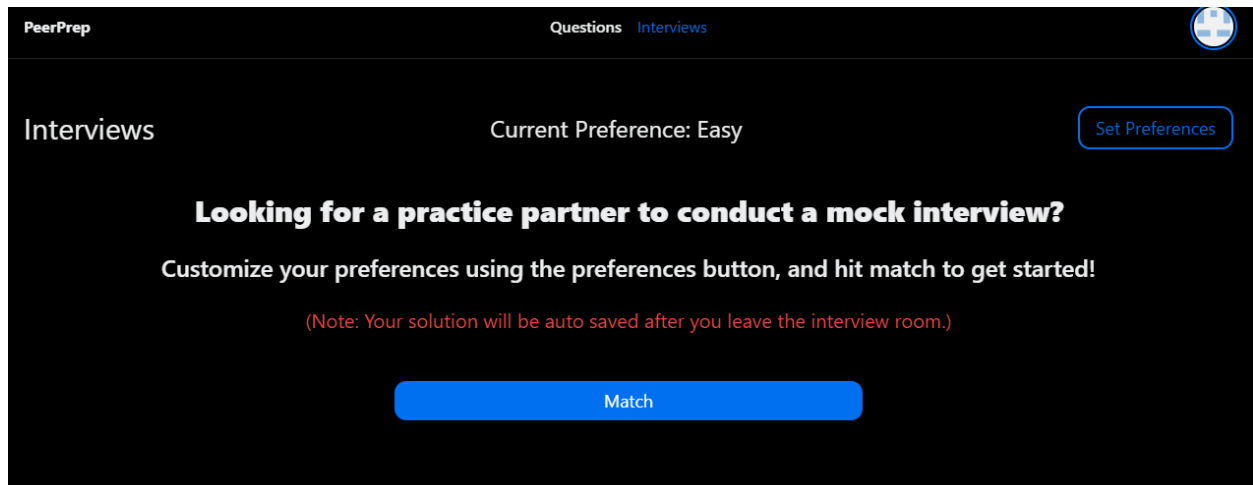


Figure 14: Interviews Page

Upon pressing the **Match** button, it initiates a search for users who share the same complexity level preference.

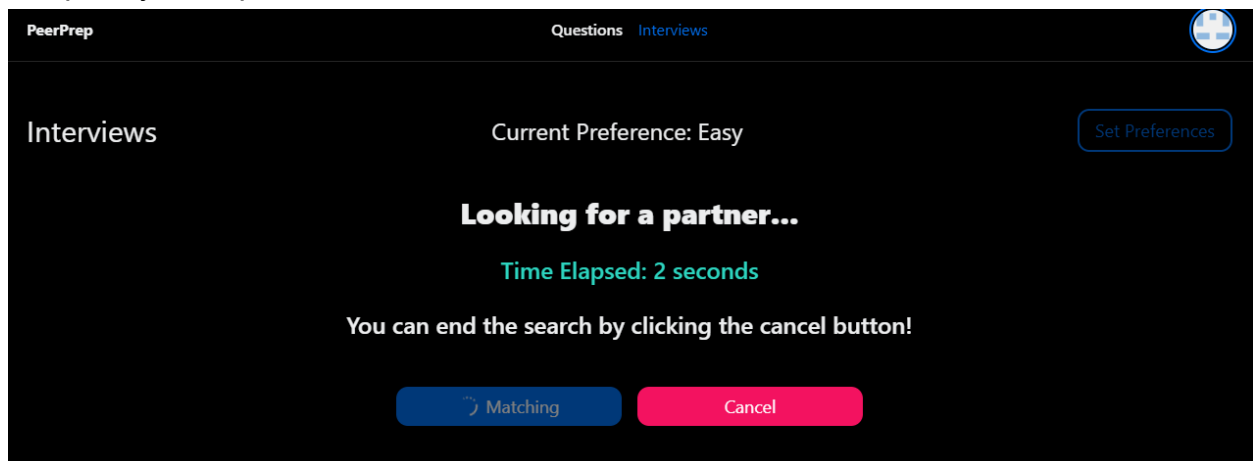


Figure 15: Finding Match

In the event of no matching users within a 30-second period, the process gracefully times out.

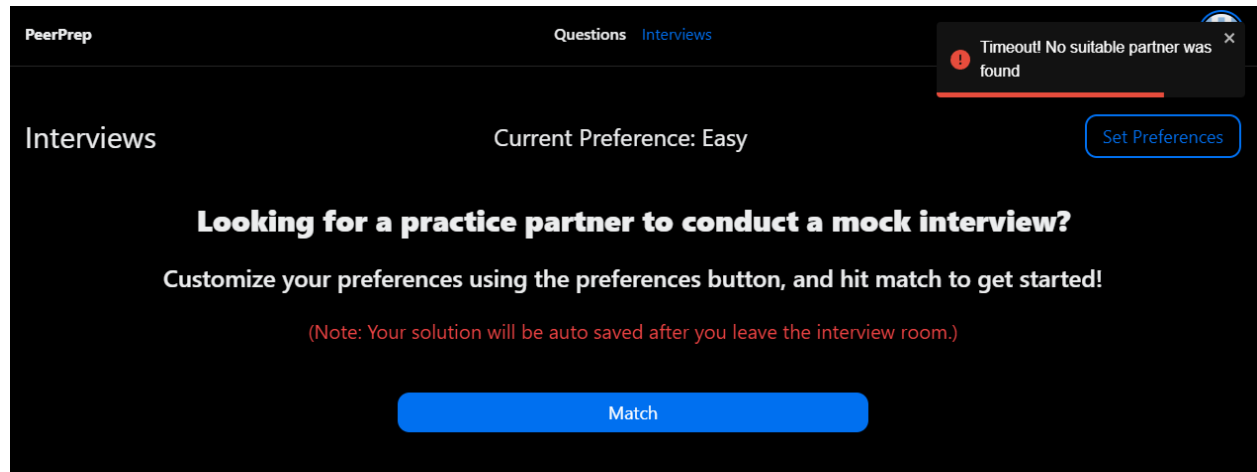


Figure 16: Match Timeout

6.2.6 Collaboration Page

Two users will be redirected to this collaboration page after they are matched by the matchmaking system. The question will be selected randomly from the question table based on the complexity set by the users. They are able to edit and work on the question together in the code editor.

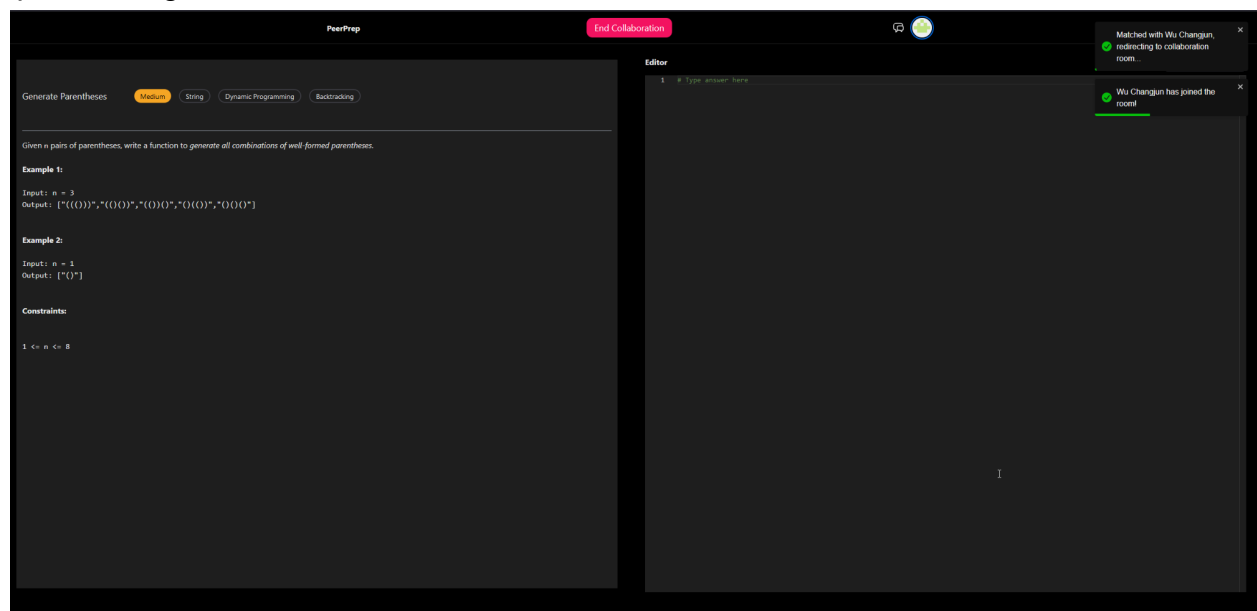


Figure 17: Collaboration Page

6.2.6.1 Chat Room

Users are able to chat with each other by clicking on the chat icon beside their profile picture.

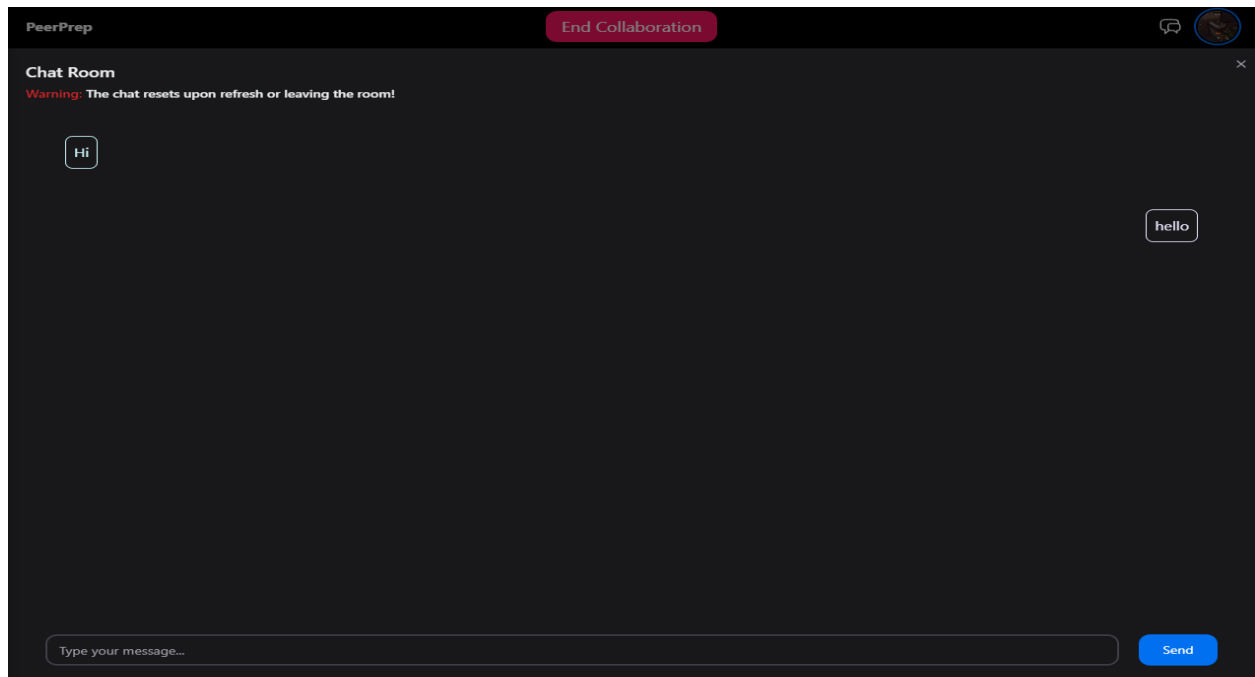


Figure 18: Chat Modal

When the user has an unread message, the user will be notified by a toast.

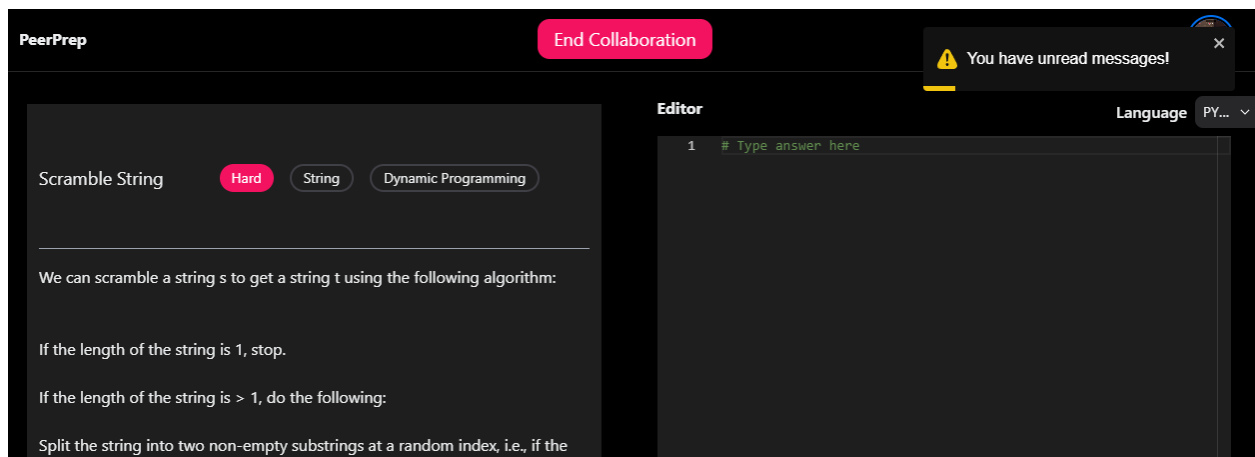


Figure 19: Unread Message Toast

6.2.6.2 Code Editor

For the code editor, we used monaco-editor/react, which includes substantial built-in features, like syntax highlighting, code formatting and IntelliSense.

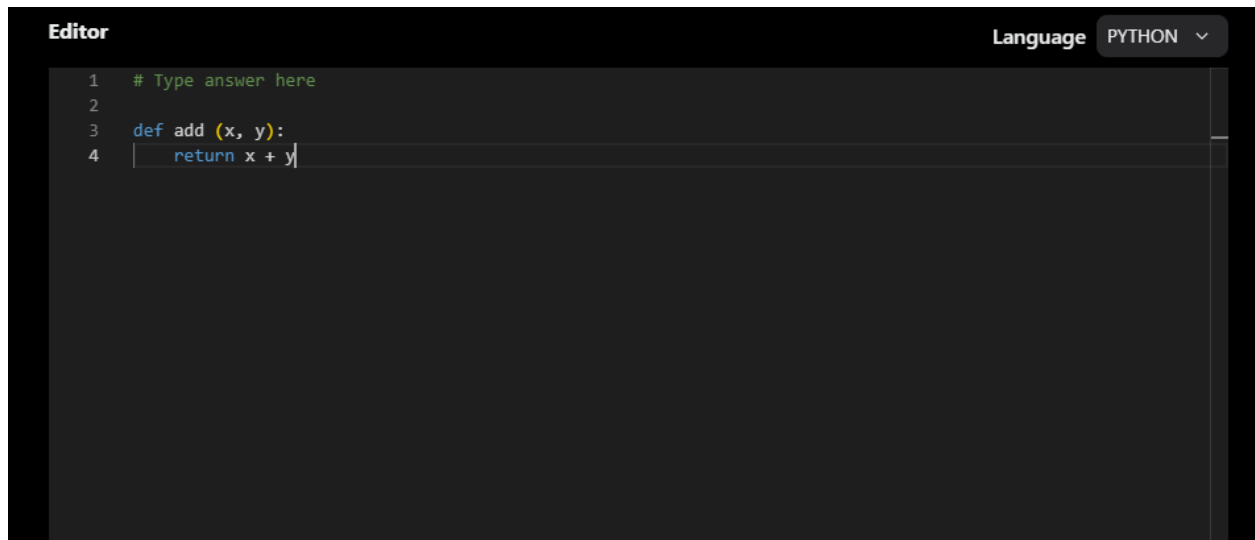


Figure 20: Code Editor

By default, when both users enter the collaboration room, the language is set to Python. However, a dropdown box is provided for the users to change to their preferred languages from the 14 languages choices that we have provided.

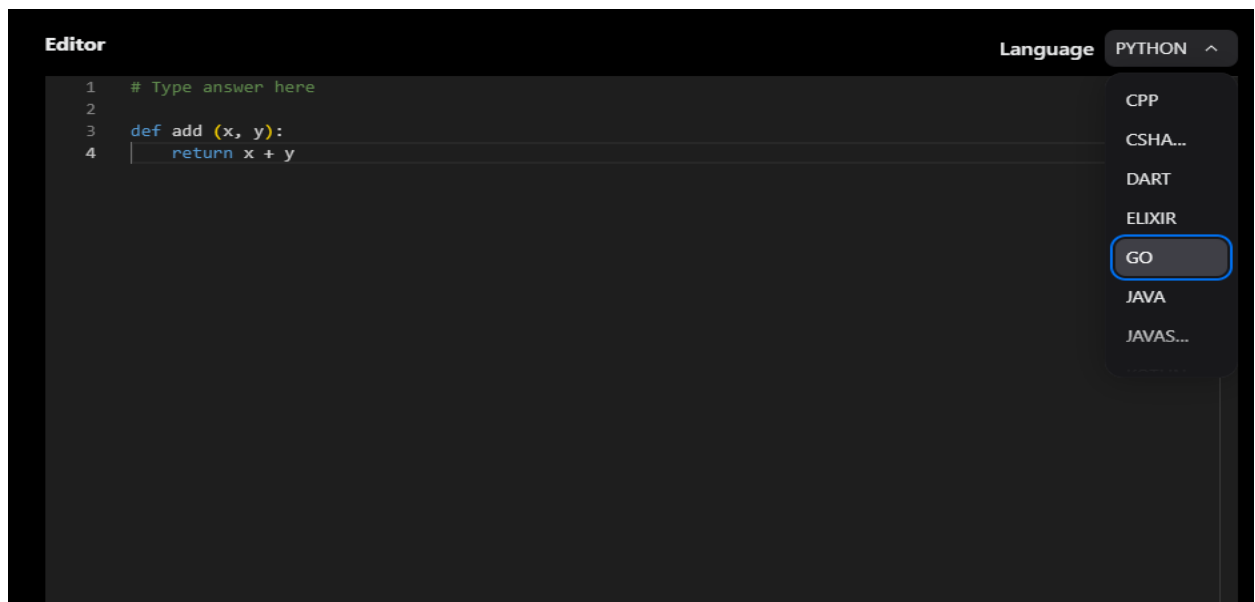


Figure 21: Language Selection in Code Editor

The 14 languages come with syntax highlighting and code formatting provided by the monaco-editor/react package. In addition, the editor also has IntelliSense for TypeScript and JavaScript.

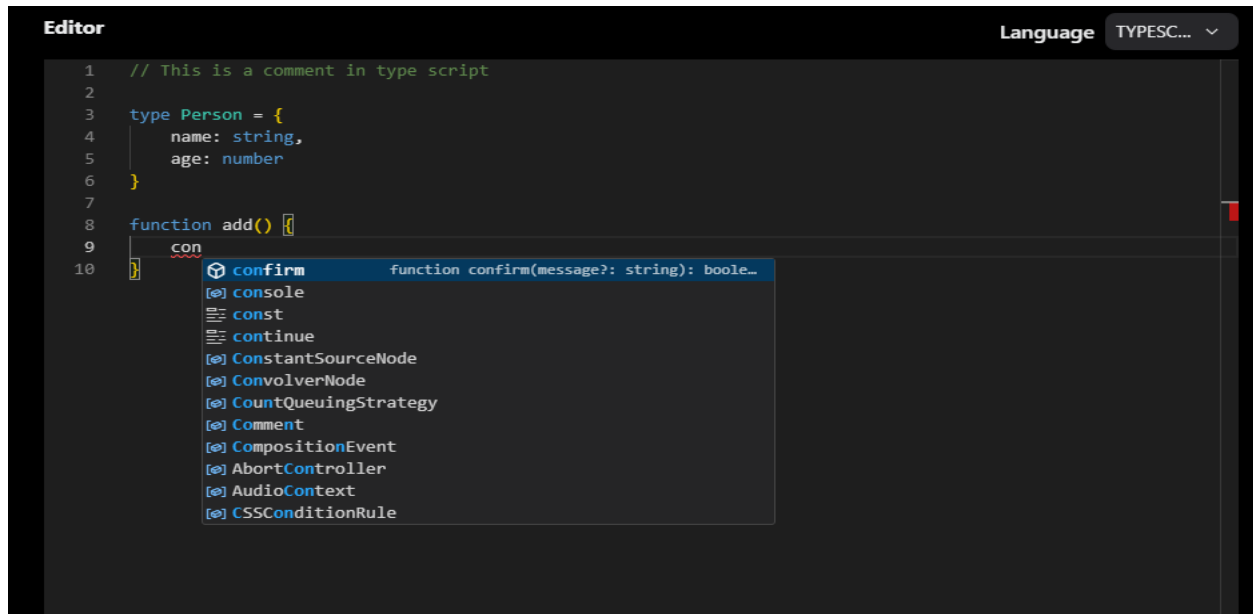


Figure 22: IntelliSense in Code Editor

6.2.7 Profile Page

This page allows users to update their user preferences and settings.

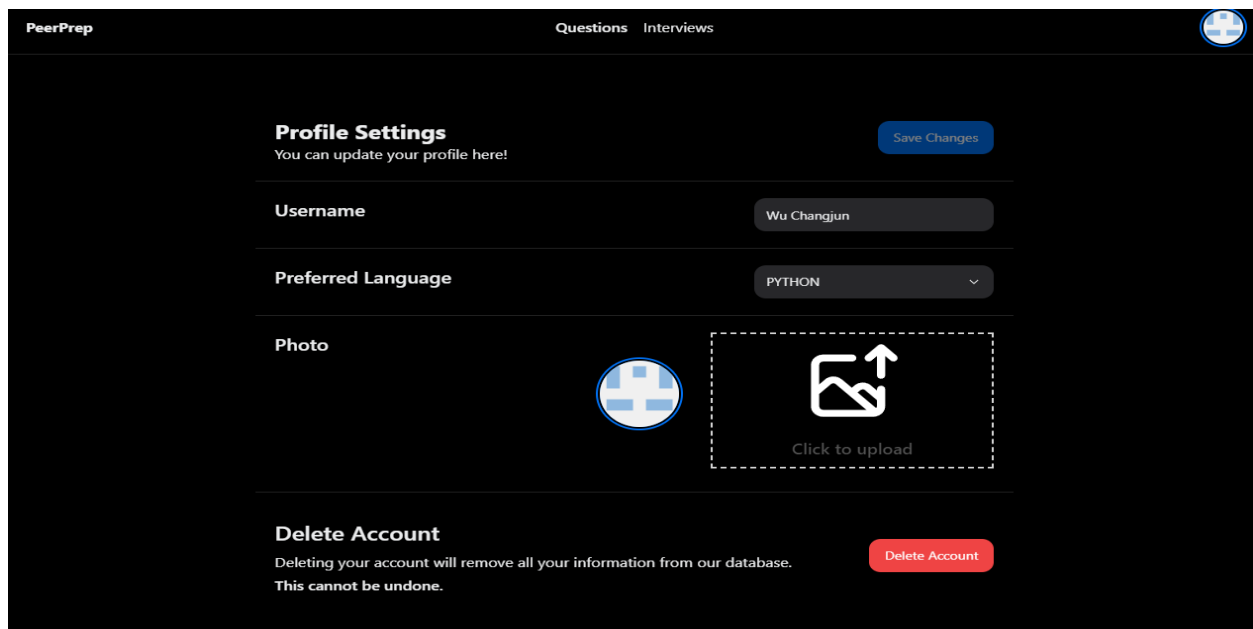


Figure 23: Profile Page

The user is able to change their username, preferred language and profile picture.

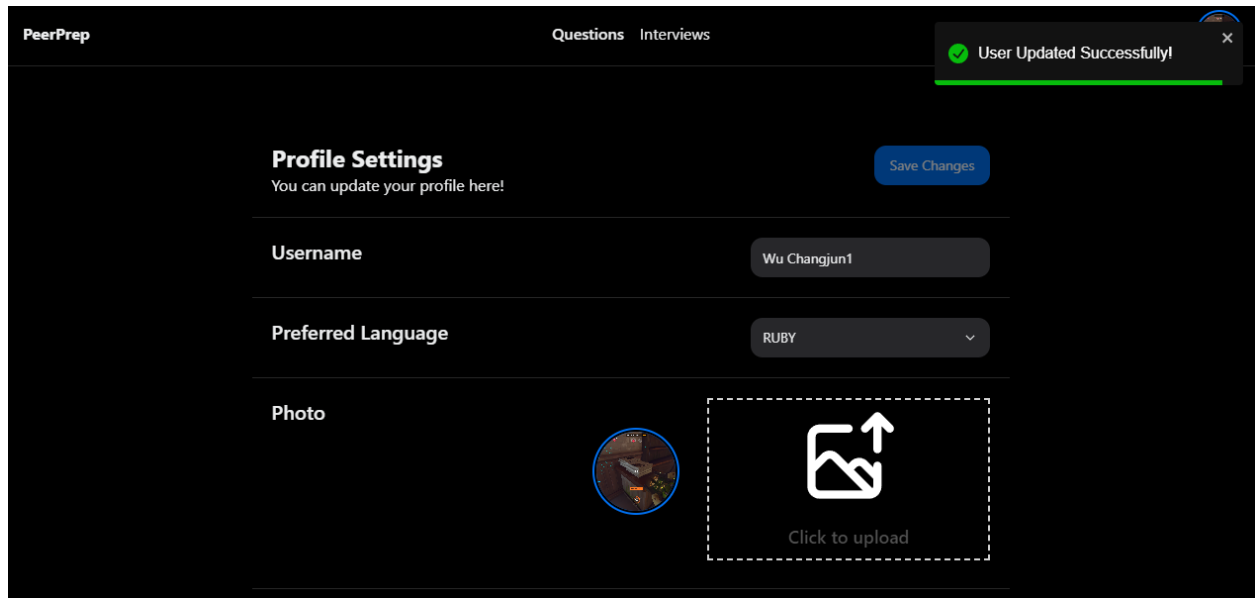


Figure 24: User Update Profile - Success Message

The user is unable to change their username to one that already exists in the database

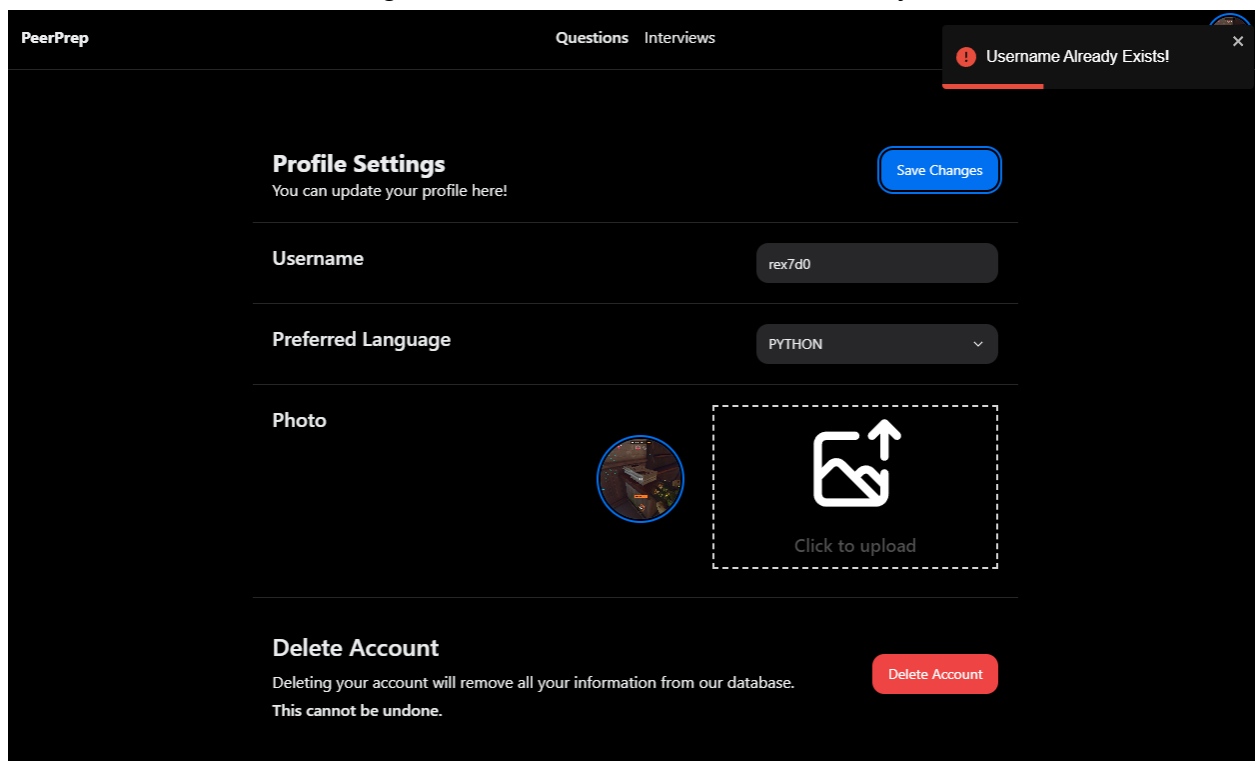


Figure 25: Error message when updating user's username to one already in database

Users have the ability to delete their accounts. A modal will pop up and prompt them to confirm their decision. This additional step ensures intentional and secure account deletions.

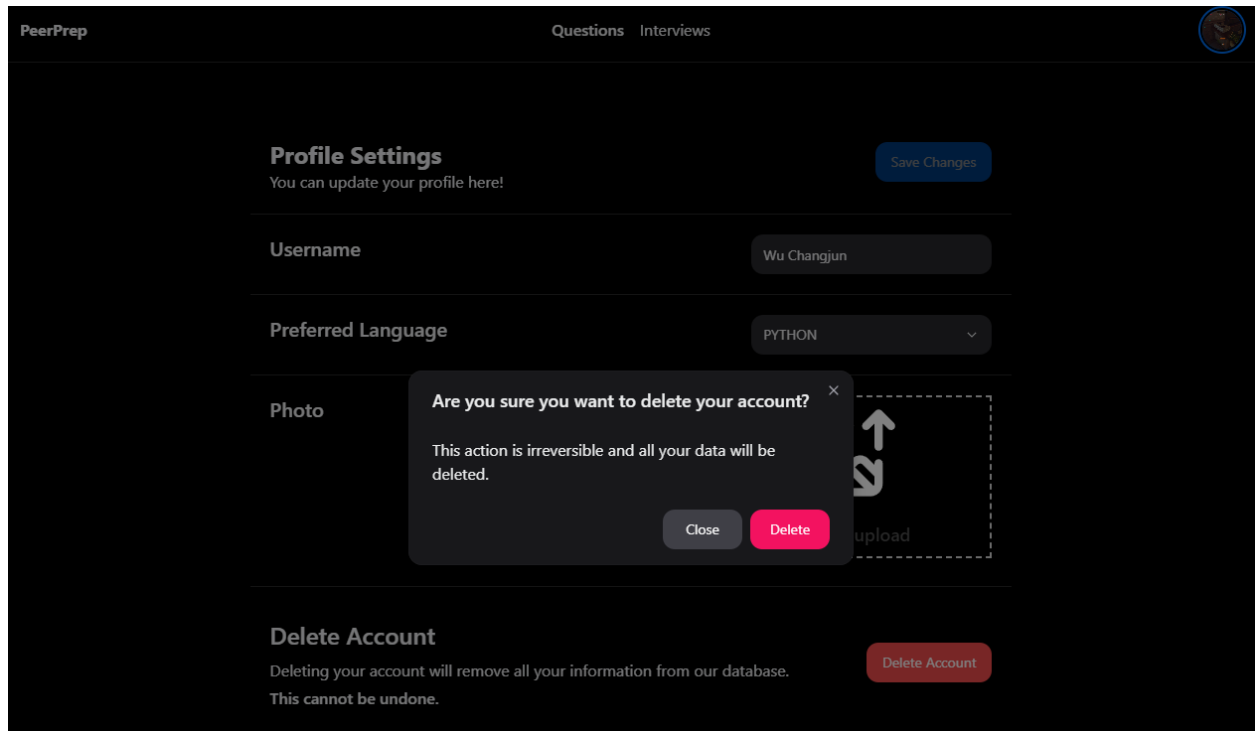


Figure 26: Delete Account Modal

6.2.8 History Page

This page allows users to view the questions attempted by them in the collaboration room. The history table is sorted based on the time they completed their collaborative sessions in descending order.

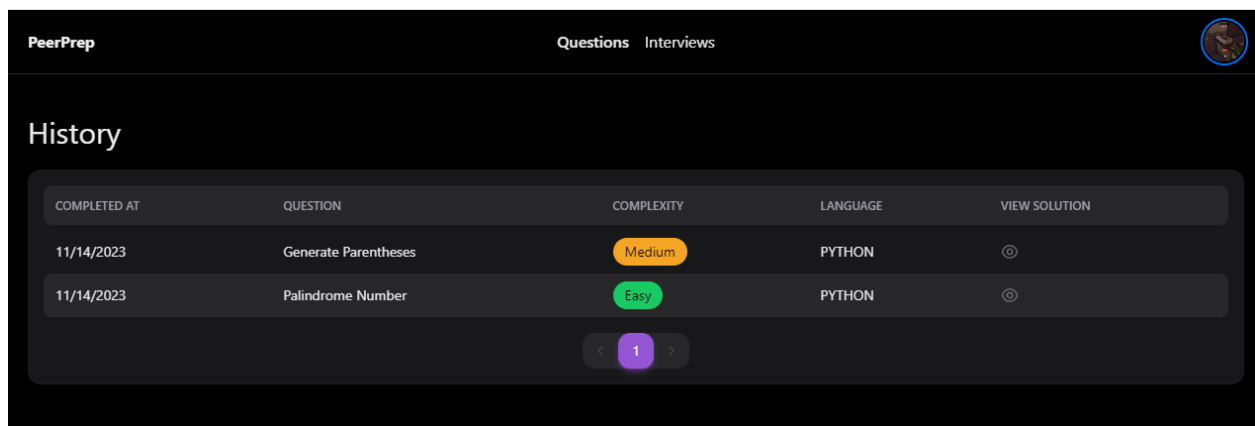


Figure 27: History Page

They can view the solutions written by them during the collaboration.

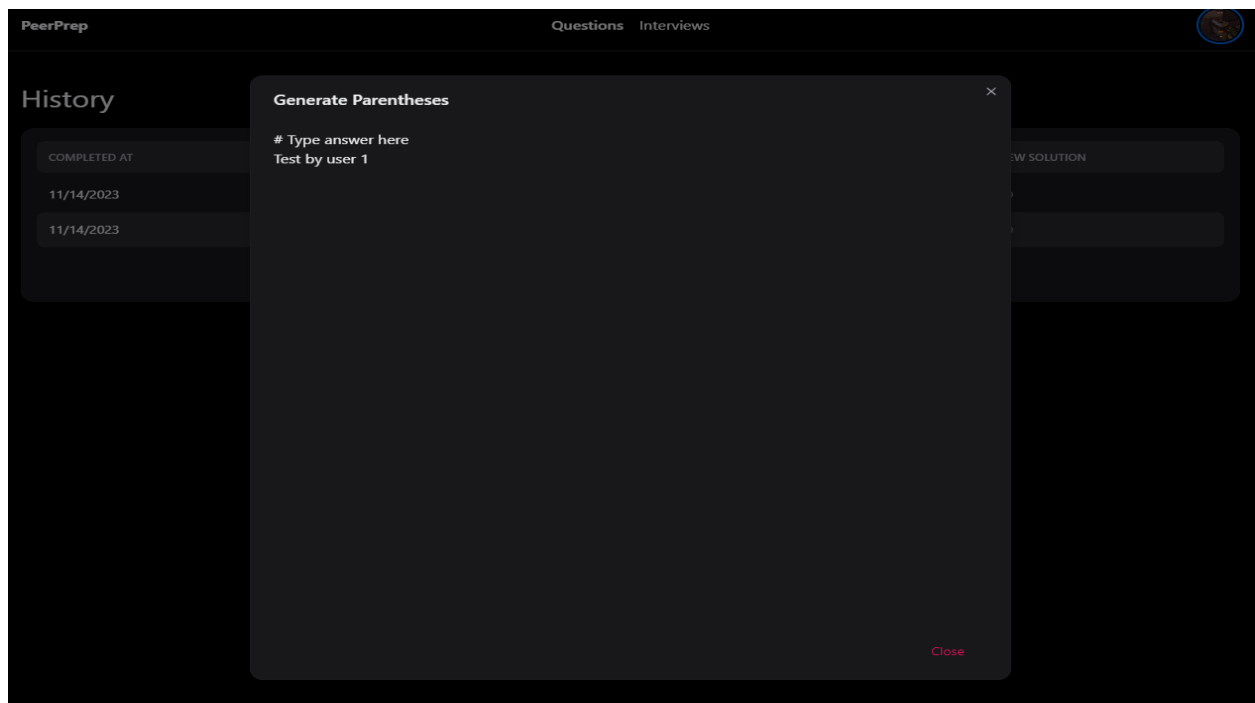


Figure 28: Solution Modal

6.2.9 Manage Users Page

This page can only be accessed by super admin users. Super Admins are able to view all the users that use the PeerPrep. They would also be able to upgrade and downgrade users' roles between admin and user.

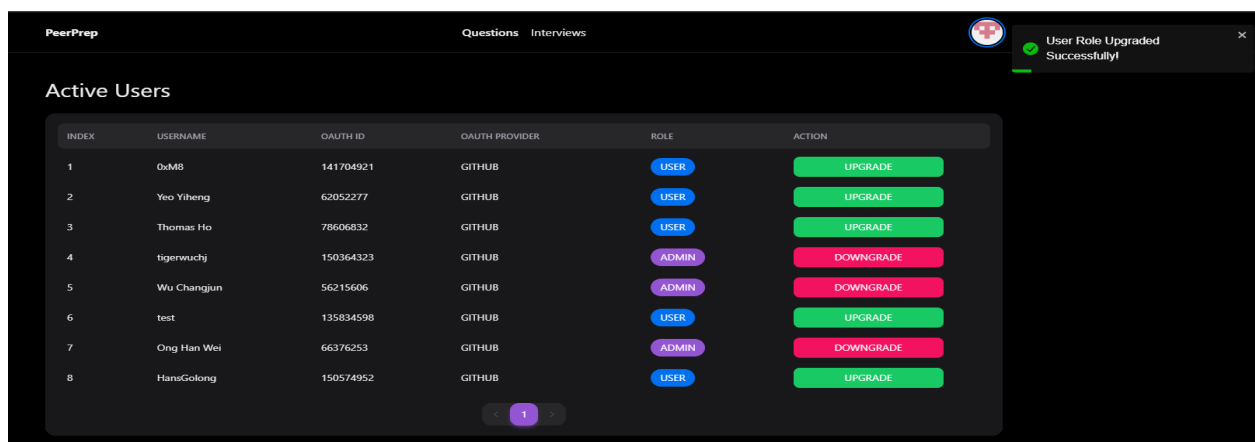


Figure 29: Manage Users Page

When the user role is successfully updated, there will be a success toast to notify the user. There would also be a success toast then the user is downgraded successfully.

7. Individual Service Documentation

7.1 API Gateway

7.1.1 Background

This service will handle the authentication and authorization logic. This is the first entry point to any other microservices, so it will first verify whether a user is authorized to the service by validating the token present in the request headers. It also handles the routing of requests to the other microservices.

7.1.2 Service Architecture

API Gateway Middleware

Before any endpoints reach their services (Question Service, User Service, etc), it would have to go through the API Gateway Middleware. The only endpoints that do not have to go through this middleware are /auth/signup, /auth/login, /auth/logout, /auth/refresh.

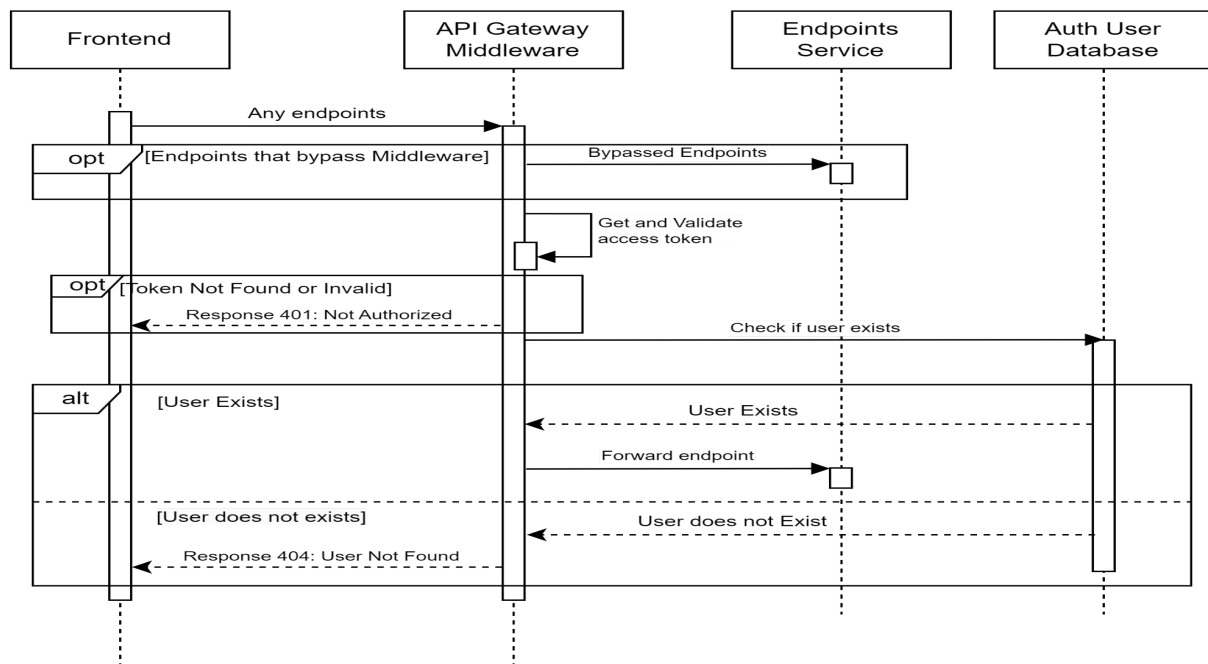


Figure 30: API Gateway Middleware Sequence Diagram

The above illustrates the flow of the middleware.

Suppose the user is trying to get the questions from the question service using the endpoint /questions.

1. It will first check if the user has a valid access token.
 1. If the user does not have a valid access token, the middleware would return a Not Authorized error.
 2. If the user has a valid access token, the middleware checks if the user exists.
2. If the user exists, the middleware would forward the endpoint /questions to the question service.

7.1.3 Endpoints

- POST /auth/signup
- POST /auth/login
- GET /auth/logout
- GET /auth/refresh
- GET /auth/users
- GET /auth/user
- DELETE /auth/user
- GET /auth/user/upgrade-super-admin
- POST /auth/user/upgrade
- POST /auth/user/downgrade

7.1.4 Auth User Schema

```
type User struct {  
    gorm.Model  
    OAuthID      int    `json:"oauth_id"`  
    OAuthProvider string `json:"oauth_provider"`  
    Role          string `json:"role"`  
}
```

7.1.5 Query Sequence

7.1.5.1 Create Auth User Flow

Endpoint	/auth/signup
Method	POST
Content Type	application/json

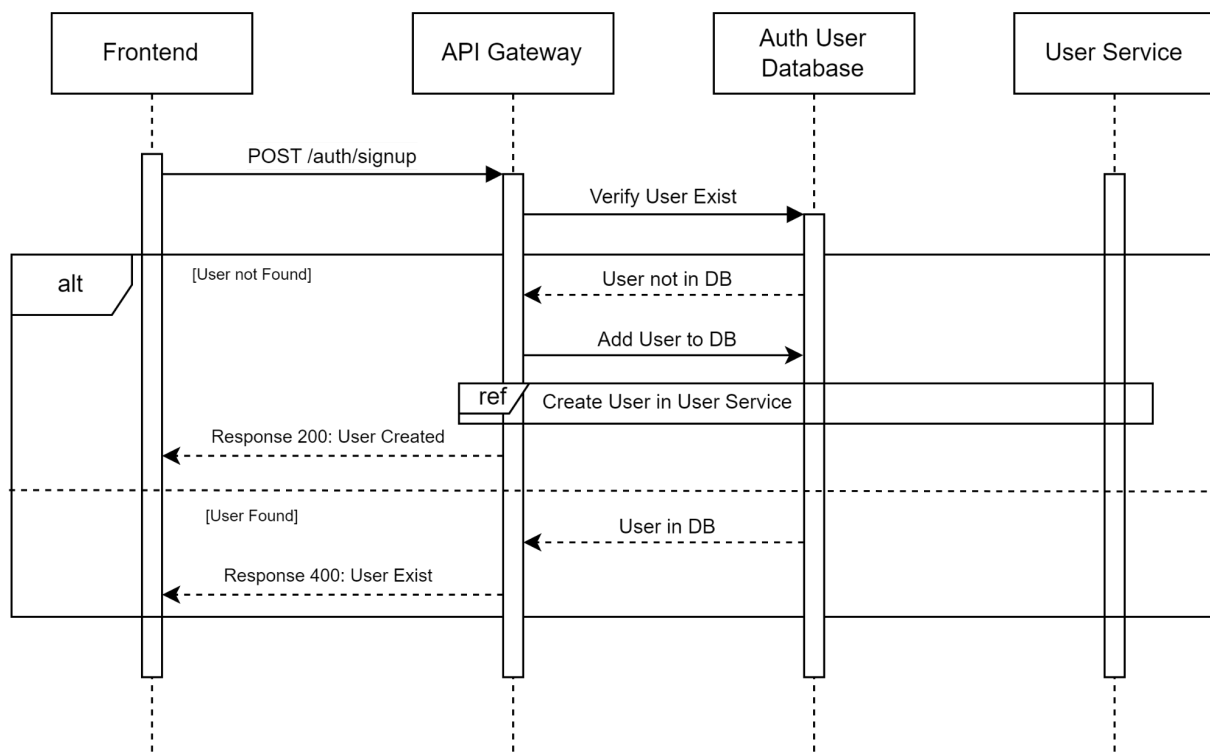


Figure 31: Create Auth User Sequence Diagram

This flow illustrates how a user is created in the API Gateway. When a user is not found in the database, API Gateway will add the user into its database by storing its `oauth_id` and `oauth_provider`. It will also [create the user](#) in User Service.

7.1.5.2 Login Flow

Endpoint	/auth/login
Method	POST
Content Type	application/json

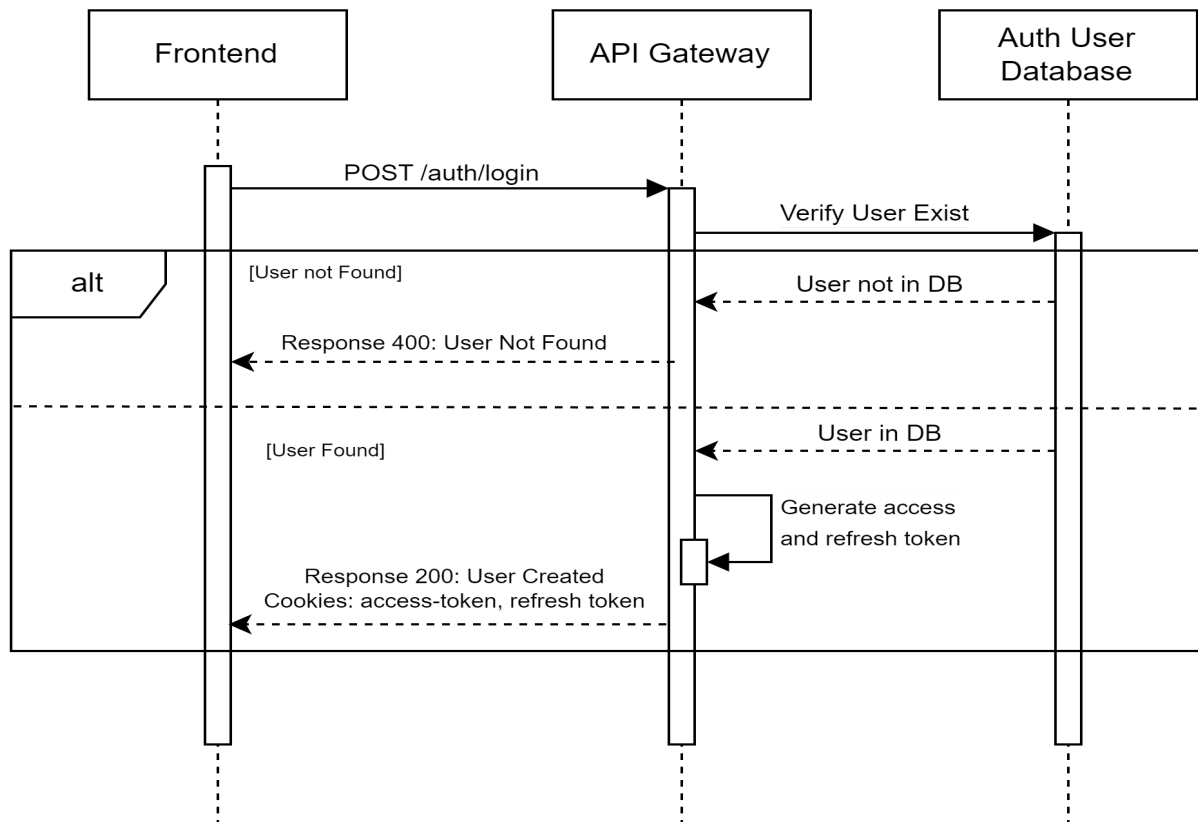


Figure 32: Login Sequence Diagram

This flow illustrates the login process in PeerPrep. When the user is found, the access token and refresh token would be set as cookies. These tokens are jwt tokens, whereby the access token is used to access the services and has an expiration time of 1 minute while the refresh token refreshes the access token and has an expiration time of 1 day.

7.1.5.3 Logout Flow

Endpoint	/auth/logout
Method	GET

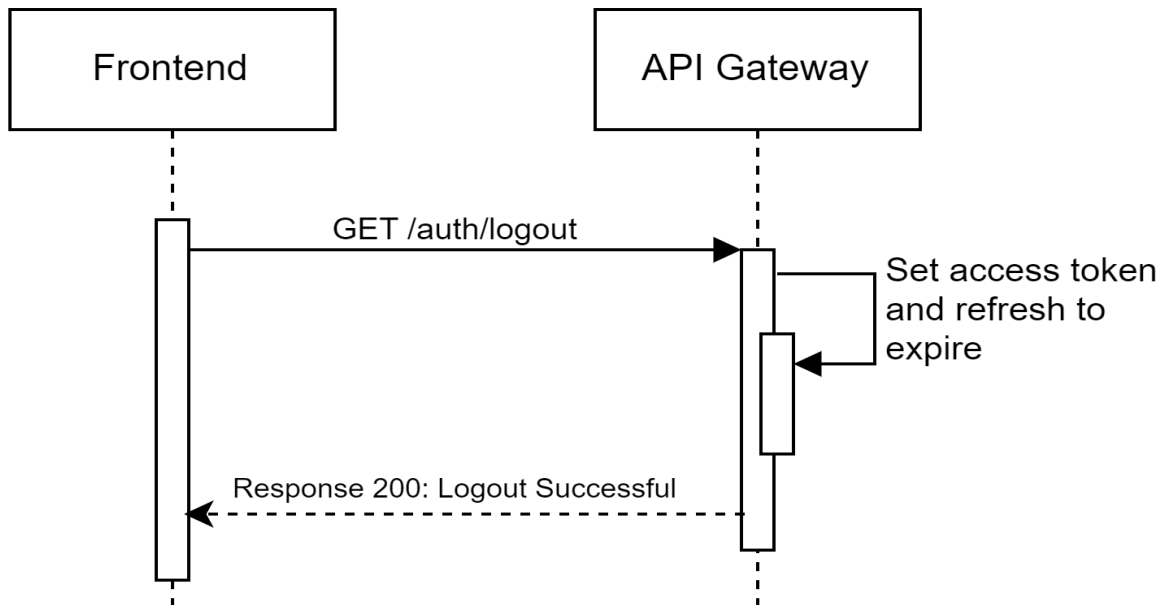


Figure 33: Logout Sequence Diagram

This diagram shows the flow of user logging out. When the endpoint is called, both access and refresh token cookies would be set to expire immediately.

7.1.5.4 Refresh Flow

Endpoint	/auth/refresh
Method	GET

The refresh flow has a similar flow as the logout flow. However, instead of /auth/logout, it is /auth/refresh. Also, instead of setting the tokens to expire, it refreshes both tokens and sets the new tokens as cookies.

7.1.5.5 Delete Auth User

Endpoint	/auth/user
Method	DELETE

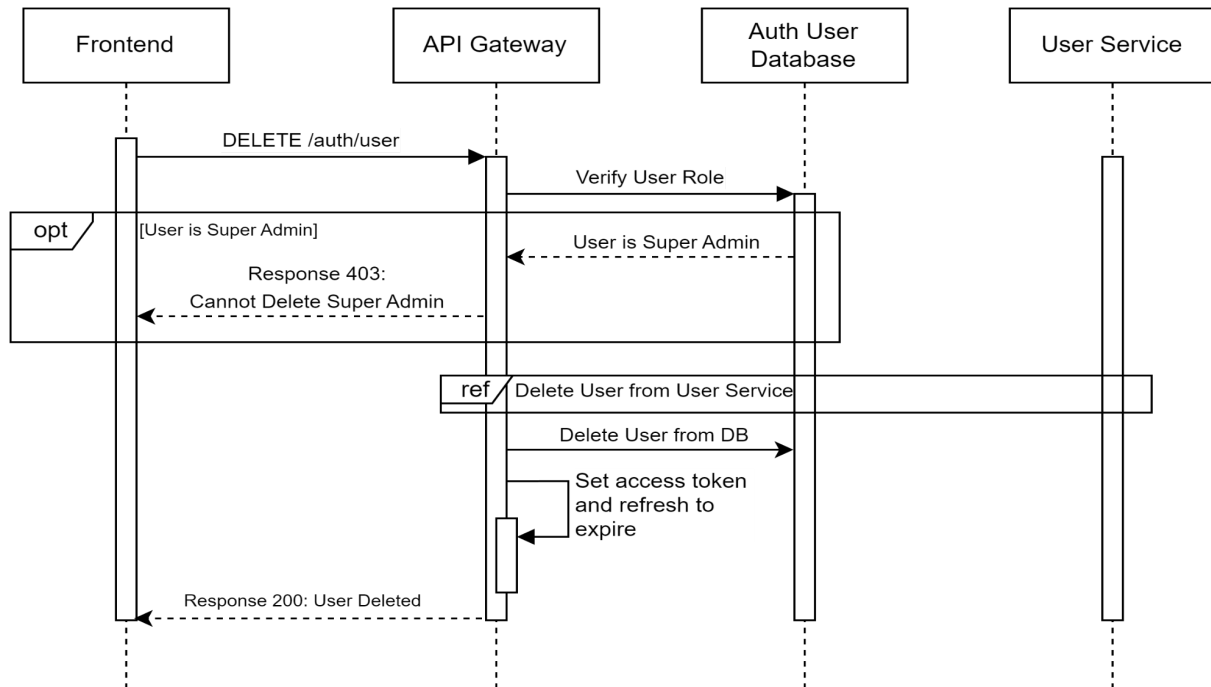


Figure 34: Delete Auth User Sequence Diagram

This illustrates the flow to delete a user. Note that if the user is a Super Admin, the user cannot be deleted. When deleting the user, just like creating a user, API Gateway will [delete the user](#) from user service as well.

7.1.5.6 Upgrade to Super Admin Role Flow

Endpoint	/auth/user/upgrade-super-admin
Query Parameter	key: string
Method	GET

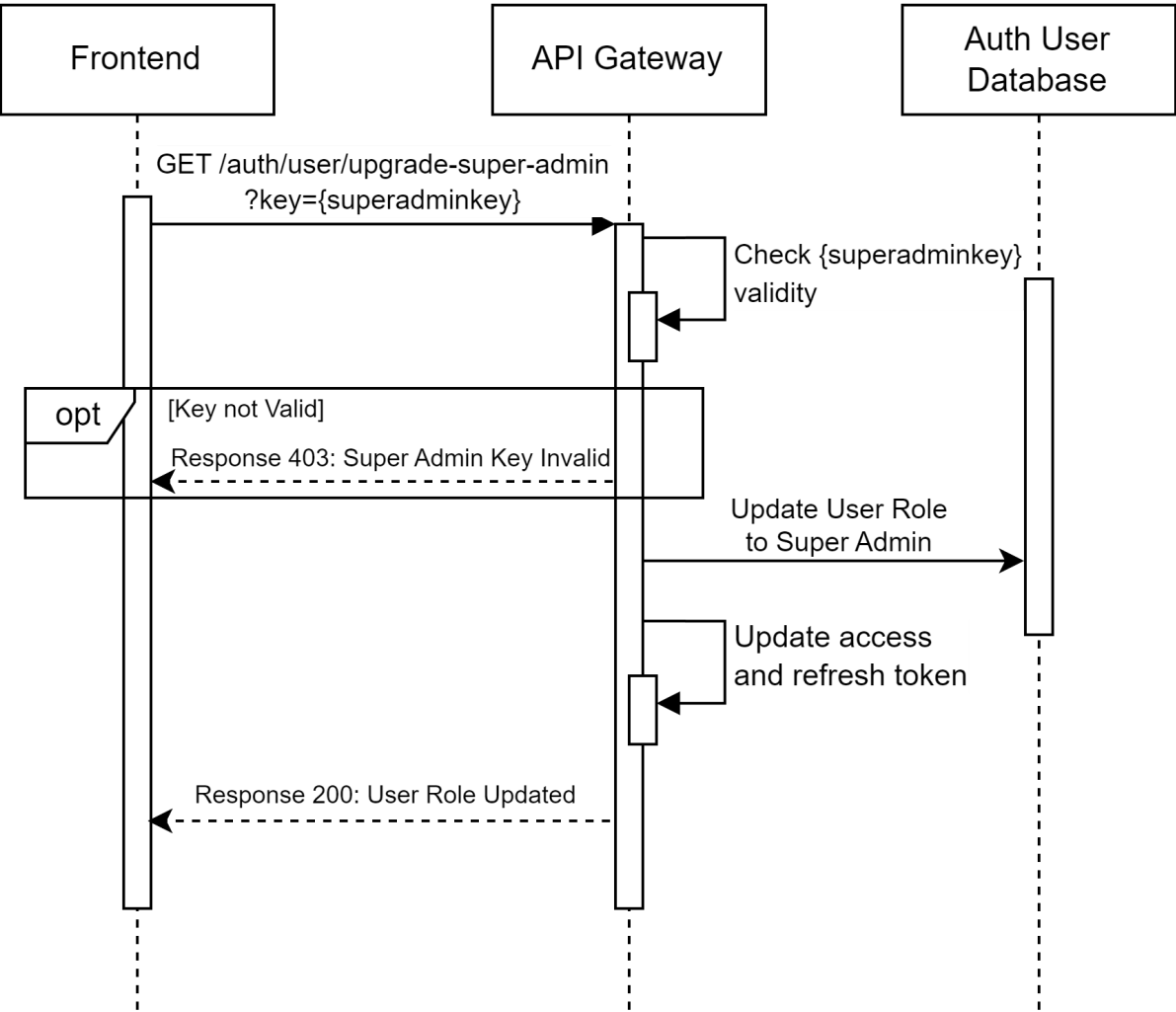


Figure 35: Upgrade Super Admin Role Sequence Diagram

This is the flow for upgrading users to a super admin role. To upgrade to super admin, the user would need to know the superadminkey which would only be passed around by word of mouth.

Super Admin cannot be upgraded just by using the frontend application. In order for the user to be upgraded to super admin,

1. Open Browser and Login to PeerPrep
2. Enter

<http://34.143.144.123.nip.io/api/auth/user/upgrade-super-admin?key={superadminkey}>

7.1.5.7 Upgrade Auth User Role Flow

Endpoint	/auth/user/upgrade
Method	POST
Content Type	application/json

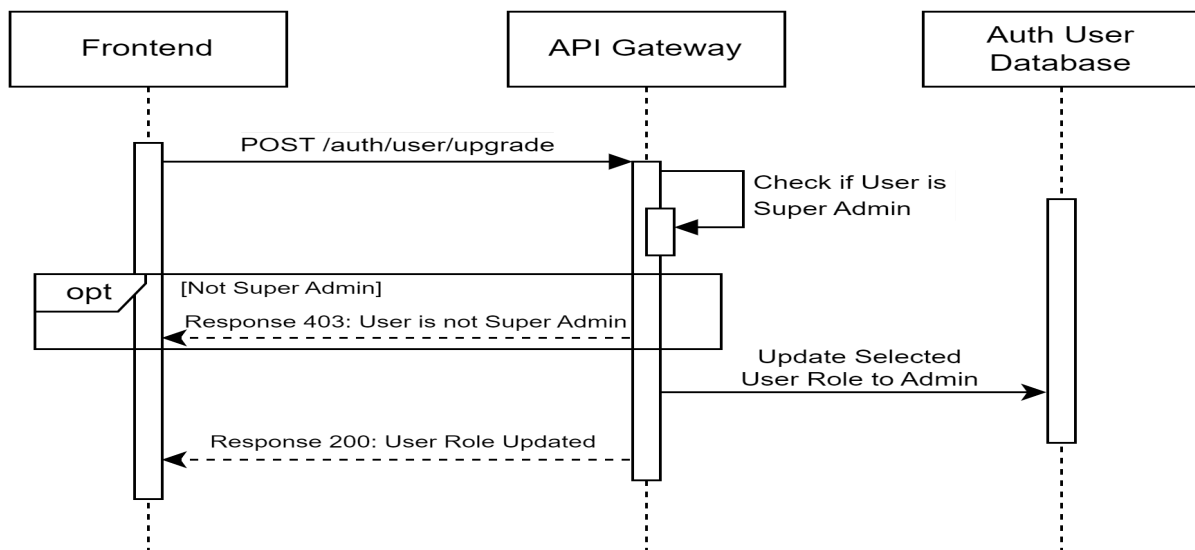


Figure 36: Upgrade Auth User Sequence Diagram

This illustrates the flow of upgrading a user. Only Super Admin is able to upgrade the roles of selected users. The selected user information would be passed over as a request payload.

7.1.5.8 Downgrade Auth User Role Flow

Endpoint	/auth/user/downgrade
Method	POST
Content Type	application/json

The process to downgrade a user is similar to upgrading a user. It can only be done by Super Admin and the selected user who is being downgraded would have its information passed over as request payload.

7.1.5.9 Get Auth User Info Flow

Endpoint	/auth/user
Method	GET

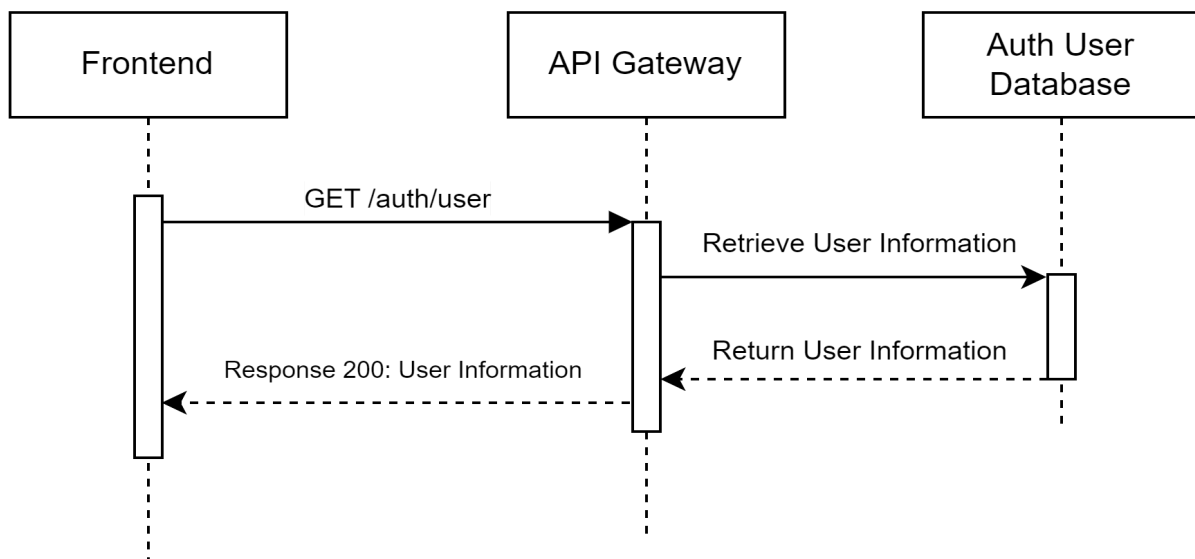


Figure 37: Get Auth User Info Sequence Diagram

The API Gateway would access the database to retrieve the information regarding the user.

7.1.5.10 Get Auth Users Info Flow

Endpoint	/auth/users
Method	GET

The process of getting all users' information is similar to retrieving information about a single user. The API Gateway would access the database to retrieve information of all the users and return a list of users.

7.2 User Service

7.2.1 Background

This service is responsible for maintaining users and their history of attempted questions. The users can manage their profile information and view their history on the frontend.

Difference Between authentication in API Gateway and User Service

API Gateway should only handle the Authentication and Authorization logic. This includes creating users (for authentication purposes), generating and validating JWT tokens, and user roles.

User Service will have any other user information that is related to the user. This includes username, user photo and user's matching preference etc, which are not the responsibilities of authentication and authorization.

7.2.2 Endpoints

- POST /users
- GET /users/:authId
- DELETE /users/:authId
- PUT /users/:authId
- POST /history
- GET /histories/authId

7.2.3 Components

7.2.3.1 User Schema

```
type User struct {  
    gorm.Model  
    AuthUserID      uint    `json:"auth_user_id"`  
    Username         string  `json:"username"`  
    PhotoUrl         string  `json:"photo_url"`  
    PreferredLanguage string  `json:"preferred_language"`  
}
```

7.2.3.2 History Schema

```
type History struct {  
    gorm.Model  
    RoomId string `json:"room_id"`  
    QuestionId string `json:"question_id"`  
    Title string `json:"title"`  
    Solution string `json:"solution"`  
    Language string `json:"language"`  
    UserID uint `json:"user_id"`  
}
```

7.2.4 Query Sequence

7.2.4.1 Create User

Endpoint	/users
Method	POST
Content Type	application/json

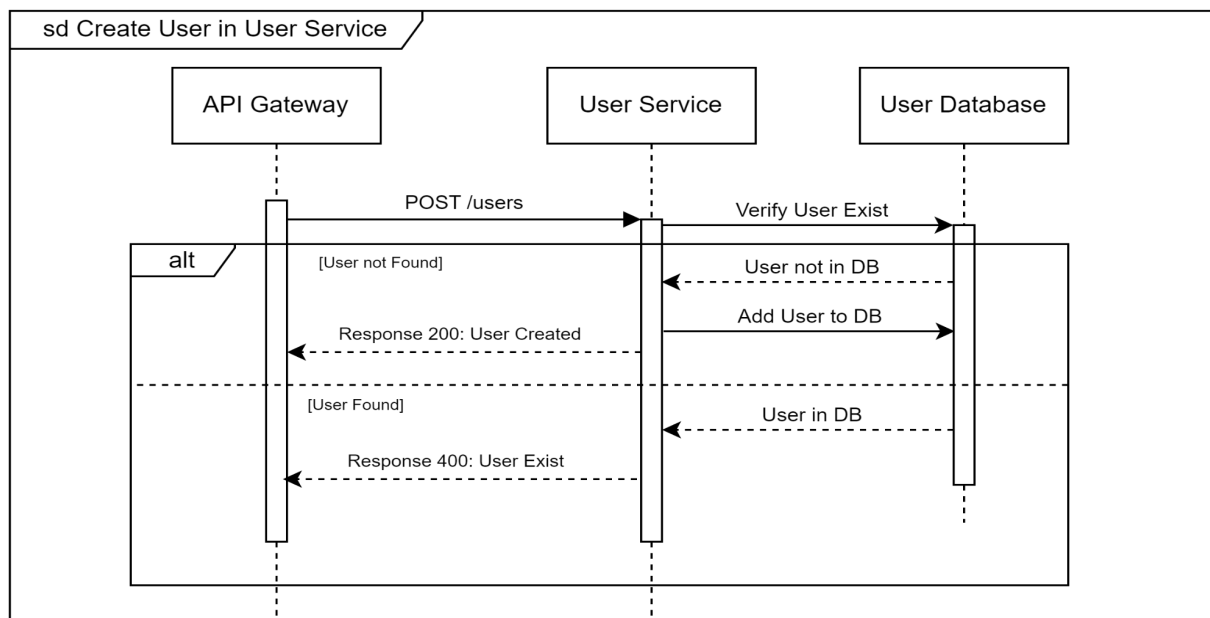


Figure 38: Create User Sequence Diagram

This illustrates the process of creating a user in user service. When the user is created in the API Gateway, it will call the /users endpoint to create the user in user service by sending the request payload.

7.2.4.2 Get User

Endpoint	/users/:authId
Method	GET

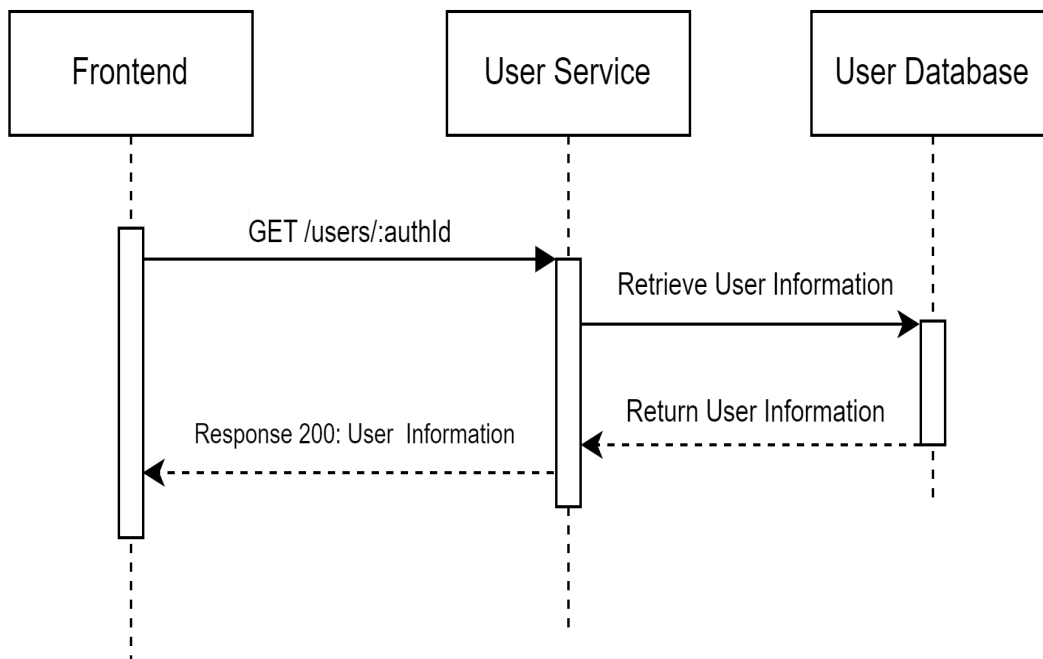


Figure 39: Get User Sequence Diagram

When provided with the authId, the user service will return the user information that corresponds to the authId.

7.2.4.3 Delete User

Endpoint	/users/:authId
Method	DELETE

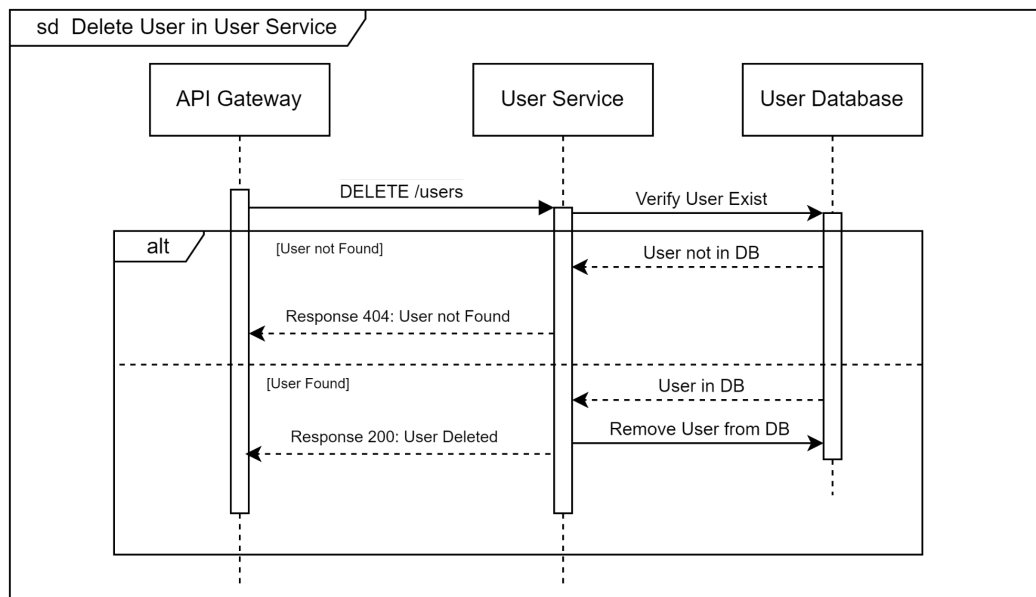


Figure 40: Delete User Sequence Diagram

Similar to creating a user, when deleting a user from API Gateway, it would also delete the user from the user service by the /users endpoint.

7.2.4.4 Update User

Endpoint	/users/:authId
Method	PUT
Content Type	application/json

This flow is similar to getting the user information. But rather than retrieving User Information, it updates the user information in the database and returns the updated user information back.

7.2.4.5 Create History

Endpoint	/history
Method	POST
Content Type	application/json

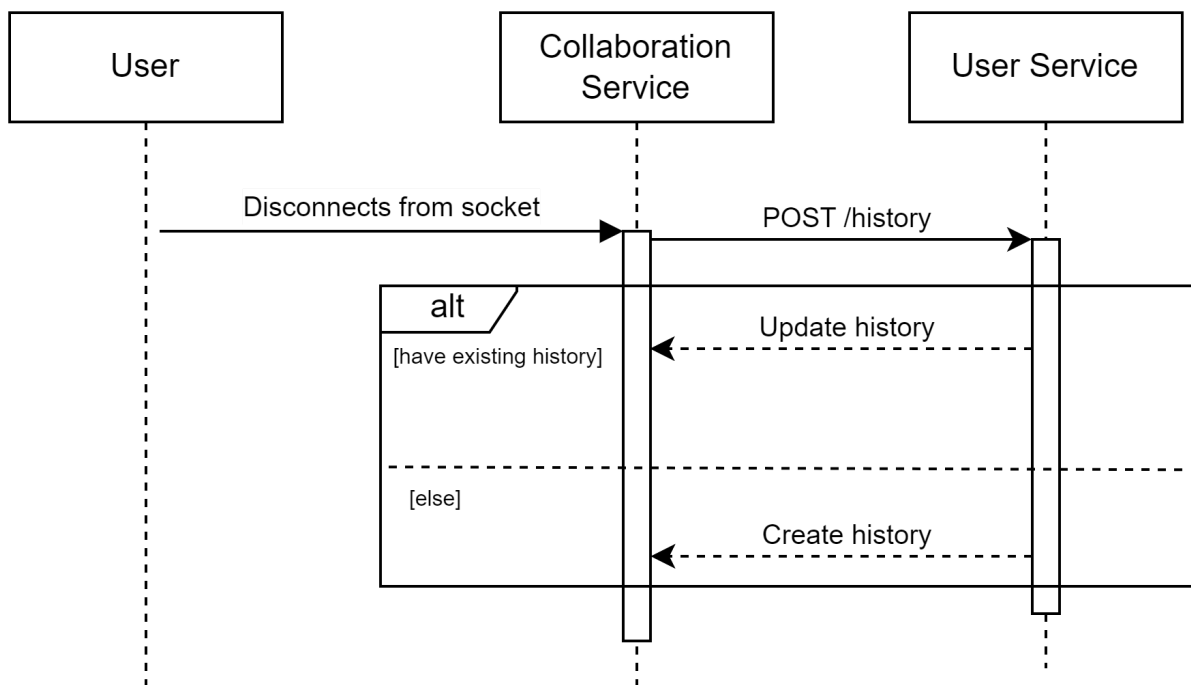


Figure 41: Create History Sequence Diagram

- This endpoint is triggered whenever a user leaves or [disconnects](#) from the socket.
 - Obtain the user_id from the user table using the username provided
 - It will check for the existence of a matching history record based on user_id , room_id and question_id in the history table
 - If a match is found, it updates the solution and language fields. This is to handle the event where a user disconnects from the socket (due to network errors or other reasons) and subsequent reconnecting, it can continue to update the same history.
 - If no existing history is found, a new history is created.

7.2.4.6 Get Histories

Endpoint	/histories/:authId
Method	GET

- Obtain the user_id from the user table using the specified authId.
- Retrieve the corresponding histories from the history table based on the user_id.

7.3 Question Service

7.3.1 Background

This service is responsible for maintaining a question repository. Users are able to view questions in the questions table and in the collaboration room on the frontend.

7.3.2 Endpoints

- GET /questions
- POST /questions
- GET /questions/:id
- DELETE /questions/:id
- GET /questions/categories
- GET /questions/complexity/:complexity

7.3.3 Components

7.3.3.1 Questions Collection Schema

```
type Question struct {
    Id          primitive.ObjectID `json:"id"`
    Title       string             `json:"title"`
    Description string             `json:"description"`
    Categories  []string           `json:"categories"`
    Complexity  string             `json:"complexity"`
}
```

Main Collection for managing questions.

7.3.3.2 Categories Collection Schema

```
type Category struct {
    Id          primitive.ObjectID `json:"id"`
    Category    string             `json:"category"`
}
```

Read-Only Secondary Collection for providing categories' options when creating questions.

7.3.4 Query Sequence

7.3.4.1 Create Question

Endpoint	/questions
Method	POST
Content Type	application/json

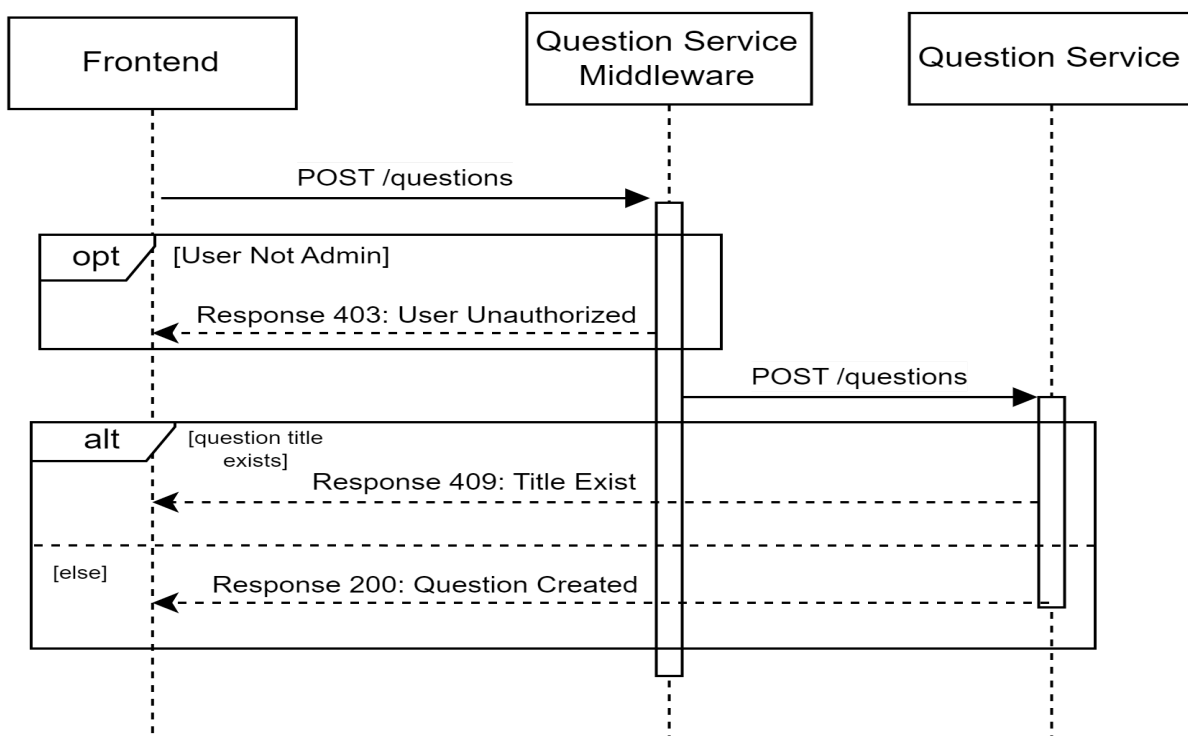


Figure 42: Create Question Sequence Diagram

- Only admin users are able to access this endpoint. Other users would not be able to create any questions.
- A document representing the question will be added into the questions collection for future retrieval and a success message would be returned.
- If the provided question title is already present in the questions collection, an error message would be returned.

7.3.4.2 Delete Question

Endpoint	/questions/:id
Method	DELETE

- Only admin users are able to access this endpoint. Other users would not be able to delete any questions and an error response would be returned.
- The specified id is converted to the corresponding MongoDB objectId.
- The document with the objectId will be removed from the questions collection and from the frontend view. A success message would be returned.
- If the objectId does not exist, an error message would be returned.

7.3.4.3 Get Question

Endpoint	/questions/:id
Method	GET

- The specified id is converted to the corresponding MongoDB objectId.
- Retrieve the document with the objectId from the questions collection.
- If the objectId does not exist, an error message would be returned.

7.3.4.4 Get Questions

Endpoint	/questions
Method	GET

- Retrieves all documents from the questions collection to populate the frontend's question table.

7.3.4.5 Get Random Question Id

Endpoint	/questions/complexity/:complexity
Method	GET

- Retrieve a random document objectId of the specified complexity from the question collection.
- This id is stored as part of the state of the collaboration room, so that both users would be able to access the same question in the room.

7.3.4.6 Get Categories

Endpoint	/questions/categories
Method	GET

- Retrieves all the categories from the categories collection to populate the Select field in the frontend's add question modal.

7.3.5 Leetcode Serverless Function

This serverless function would return a list of Leetcode questions when the Serverless HTTP URL is triggered. This serverless function utilizes the package from [dustyRAIN](#) which provides a Leetcode API for golang. This function is being deployed on GCP Cloud Function and is accessible by the endpoint given below.

Endpoint	https://asia-southeast1-peer-preps-assignment6.cloudfunctions.net/GetProblems
Query Parameters	offset: number page-size: number
Method	GET

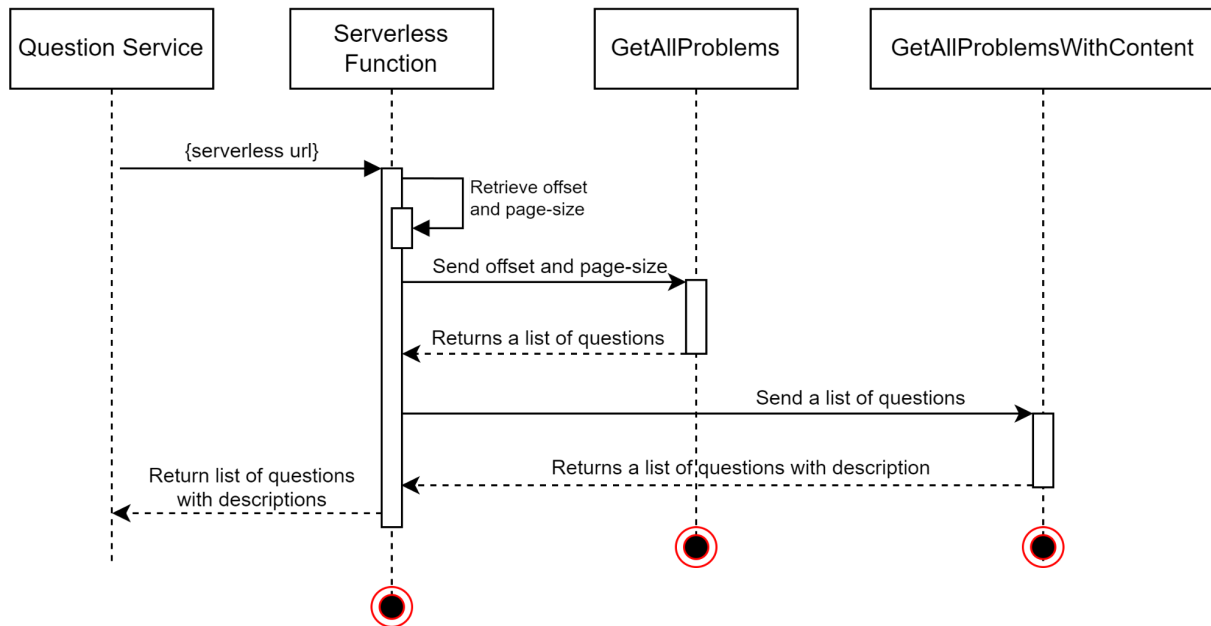


Figure 43: Get Leetcode Serverless Function

This diagram illustrates the flow of the serverless functions. It takes in 2 query parameters, the offset and pagesize. As the problems in the Leetcode API are ordered, we require the offset to know the index of the question we want to receive next and the pagesize specify the number of questions we want to receive starting from the offset. With these 2 parameters, we would be able to retrieve the list of problems from the GetAllProblems function. However, the GetAllProblems does not return the description of the questions and thus, the list of questions would have to go through the GetAllProblemsWithContent function to retrieve all the description of the questions.

7.3.6 Database Seeding

On startup, the question service will check the number of documents in the questions collection. If there are less than 100 documents, the question service will query the [leetcode serverless url](#) to obtain 100 LeetCode questions to populate the Database such that the total number is at least 100.

For any of the questions, if their categories are not already present in the categories collection, they would be added in.

7.4 Matching Service

7.4.1 Background

This service is responsible for the matching of users based on the difficulty level of questions. Users will first indicate to queue on the frontend, which will then interact with the Matching Service for all queuing, canceling, and matching related functionalities.

7.4.2 Service Architecture

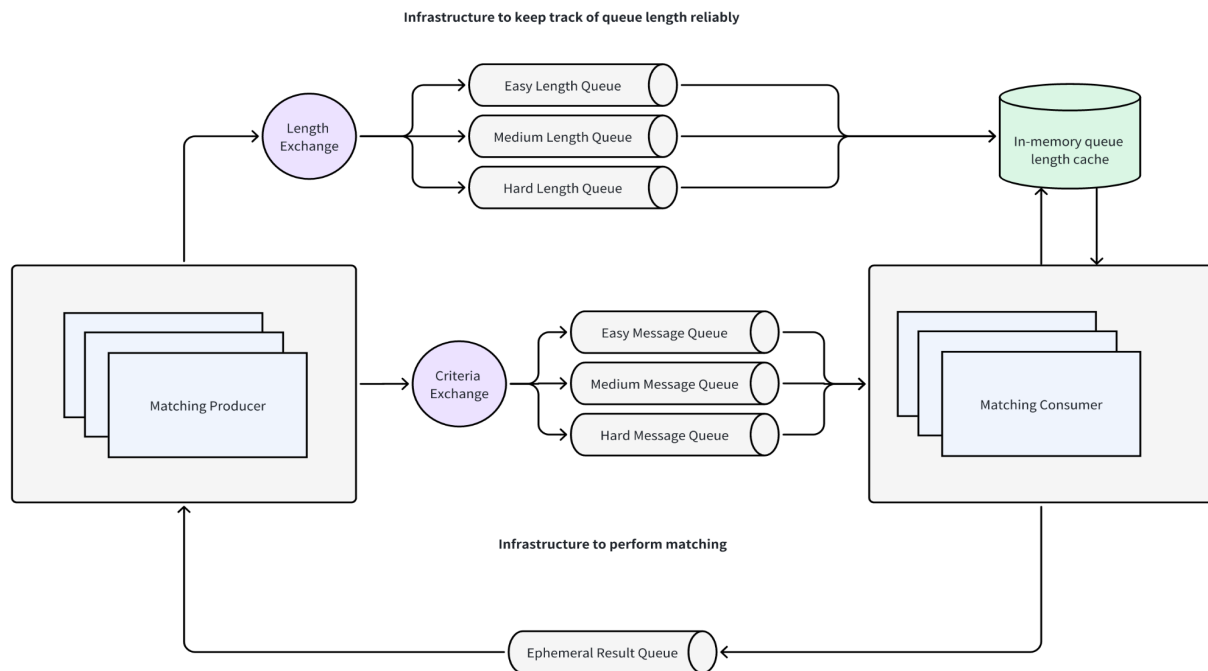


Figure 44: Service Architecture Diagram

The above provides a high level architecture of the matching service in our GKE cluster. Each producer and consumer has multiple replicas to serve high traffic matching in peak periods. The architecture heavily considers the pitfalls of a distributed system where each instance of a producer or consumer has their own in-memory address space, which forces communication using RabbitMQ point-to-point and fanout exchanges to synchronize the multiple instances.

Some of the multi-instance issues we accounted for are:

- Two consumers detect a match, and each consume 1 message each, unable to form a match since we require a single consumer to perform the matching with 2 messages
- User initiating a cancel request but this knowledge is not known to all producers
- Length of message queue must be made known to all consumers, not just one

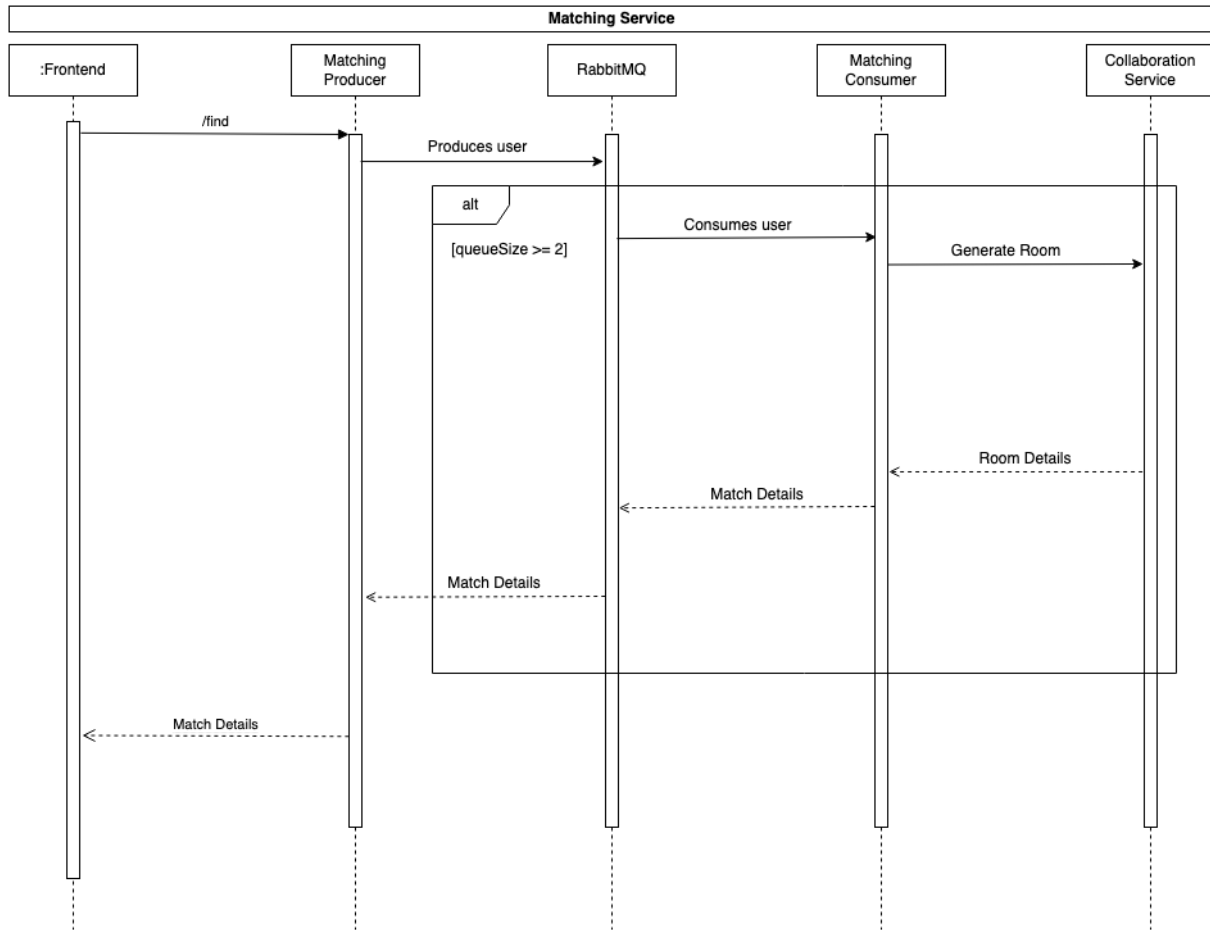


Figure 45: Matching Service Sequence Diagram

In a happy-path use case, the user selects a matching preference and initiates a match. This sends an API request to the matching producer and spins off a goroutine to handle this request. On the producer goroutine, a timer of 30 seconds starts, using Golang inbuilt context with timer. The producer then produces a message into the specialized message queue that has multiple matching consumers continually polling the length of the message queue.

Once the length of the message queue is more than or equal to 2, meaning at least 2 users are in the queue, the consumer consumes them and matches them with the help of the [collaboration service](#). The match details are then sent back in the API response for the frontend to redirect the users to the collaboration room.

7.4.3 API Endpoints

- POST `/match/find`
- POST `/match/cancel`

7.4.4 Components

7.4.4.1 Queueing Mechanism

Endpoint	/match/find
Method	POST
Content Type	application/json

```
type MatchRequest struct {  
    Username      string `json:"username"`  
    MatchCriteria string `json:"match_criteria"`  
}
```

```
type MatchResponse struct {  
    MatchUser      string `json:"match_user"`  
    MatchStatus    int    `json:"match_status"`  
    RoomId         string `json:"room_id"`  
    ErrorMessage   string `json:"error_message"`  
}
```

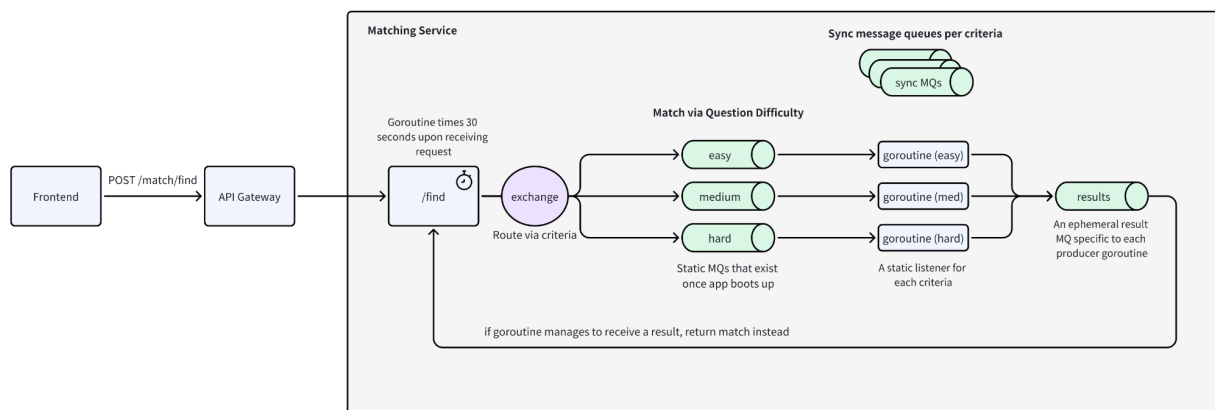


Figure 46: Queueing Mechanism

For the queuing functionality, each matching consumer will **poll** the message queues every 2 seconds to check the size of it via the local in-memory cache. Once the MQ is size 2 or more, a valid match can be initiated. Our choice of 2 seconds polling would be

to minimize computation given the traffic of our application. If we set it any shorter, the nodes would be performing wasteful computation and polling.

In a distributed architecture where we have more than one matching consumer instance, e.g., 2 matching consumers snooping for matches on the easy criteria MQ, we use an additional sync MQ to communicate between all consumers snooping on the same criteria. This sync MQ ensures that only one consumer is consuming from the request MQ at any one time, preventing race conditions where two consumer instances detect a match and they each consume only one user.

This means that it does not matter which producer / consumer instance our load balancer routes the request to. The entire matching service will perform correctly in both a single instance or multi-instance cluster.

7.4.4.2 Cancel Mechanism

Endpoint	/match/cancel
Method	POST
Content Type	application/json

```
type CancelRequest struct {  
    Username      string `json:"username"`  
    MatchCriteria string `json:"match_criteria"`  
}
```

```
type CancelResponse struct {  
    CancelStatus bool `json:"cancel_status"`  
}
```

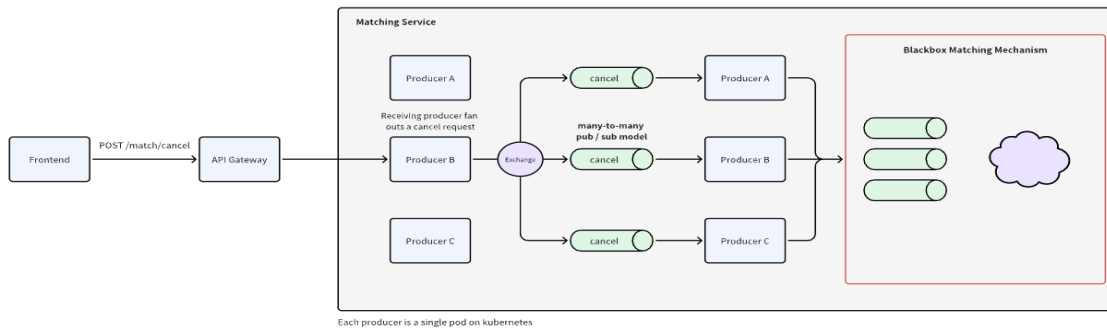



Figure 47: Cancel Mechanism

Assuming a multi-instance architecture, when a user manually cancels the matchmaking, the request is routed to a single random producer by our load balancer. This request is then fanned out to other producer instances via RabbitMQ's fanout channel. This way, all producers are updated on whichever user has initiated a cancel request and can keep track of the canceled users in their own in-memory cache.

7.4.4.3 Length Tracking Mechanism

```
type MessageQueueLengthRequest struct {
    Increment      int    `json:"increment"`
    MatchCriteria string `json:"match_criteria"`
}
```

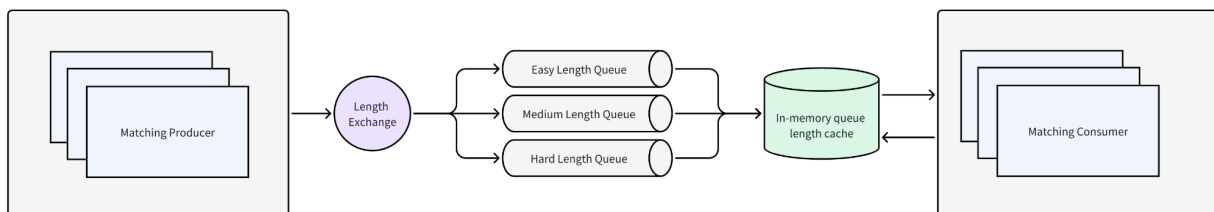


Figure 48: Length Tracking Mechanism

Initially, we relied on the RabbitMQs API management endpoints which polls for the length of the message queues via an exposed port 15672. However, these statistics are only updated in a 5 second interval, and accounting for network latency, this delay can be more than 5 seconds. Since our matching algorithm is time sensitive, we had to ditch this approach and build our own message queue length tracker. This involved using the matching producer to publish results to a consumer subscribed channel which allows all

matching consumers to update their own in-memory cache containing the length of the message queue they are monitoring.

7.5 Collaboration Service

7.5.1 Background

This service provides the mechanism for real-time collaboration (e.g., concurrent code editing) between the authenticated and matched users in a collaborative space.

7.5.2 Service Architecture

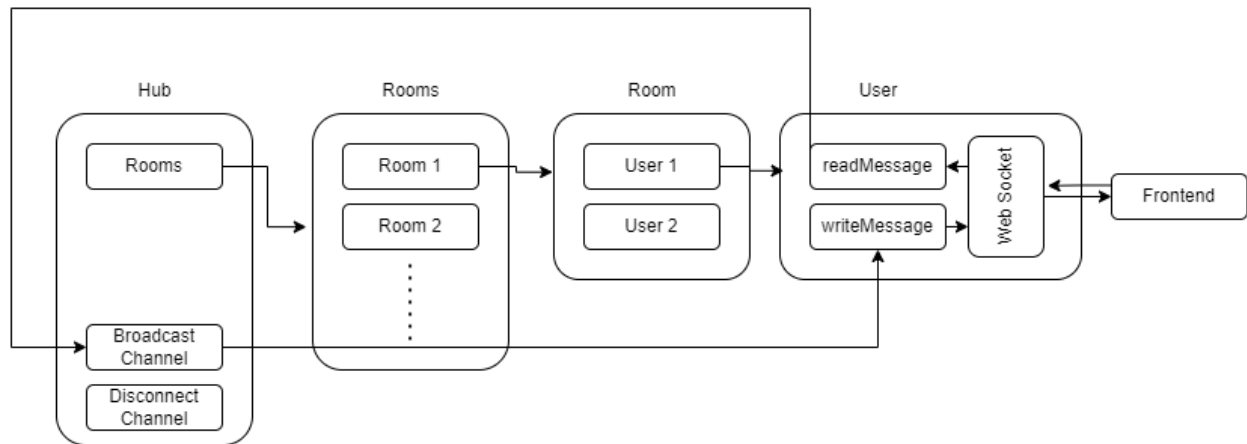


Figure 49: Collaboration Service Architecture

7.5.3 API Endpoints:

- POST /room
- GET /ws/:roomId
- GET /ws/:roomId/:username

7.5.4 Components

7.5.4.1 Hub Schema

```
type Hub struct {
    Rooms map[string]*Room
    BroadcastChannel chan *Message
    DisconnectChannel chan *User
}
```

As our server needs to actively listen to new connections, we created a hub using goroutine instead of running an infinite loop as our server will hang. A hub keeps track of a list of ongoing collaboration rooms for the matched users and is responsible for

sending the websocket messages to the opposite user in each room through a broadcast channel.

7.5.4.2 Room Schema

```
type Room struct {  
    Id string  
    Users map[string]*User  
    QuestionId string  
}
```

A room is responsible for maintaining the list of users that are matched with each other, so the hub knows which user to send the message to.

7.5.4.3 User Schema

```
type User struct {  
    Connection *websocket.Conn  
    MessageChannel chan *Message  
    RoomId string  
    Username string  
    Solution string  
    Language string  
}
```

Schema meant for collaboration only

A user object in the room is responsible for maintaining the websocket connection and channel to receive messages that are sent from the other user. It is also responsible for maintaining a channel to read any messages that are sent from the hub, so that it can listen to the changes in code and update the code at the frontend.

7.5.5 Query Sequence

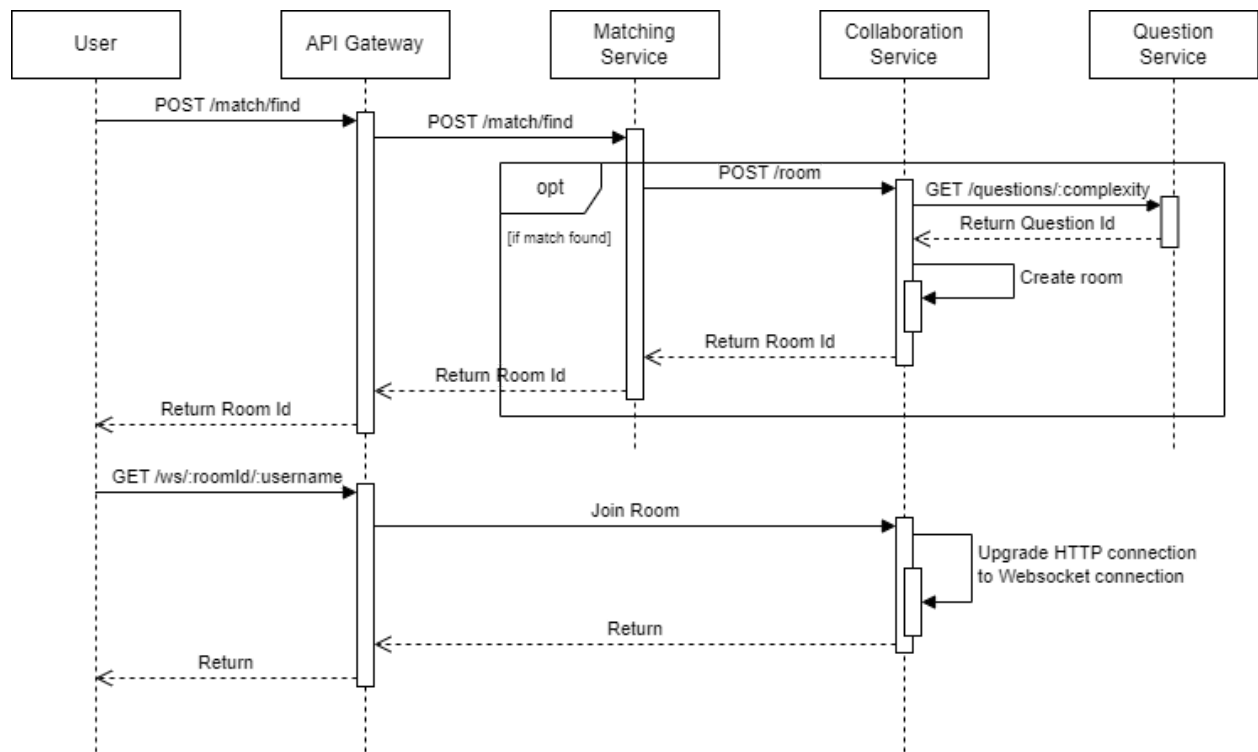


Figure 50: Match Query Sequence Diagram

7.5.5.1 Create Room

Endpoint	/room
Method	POST
Content Type	application/json

- This endpoint is triggered when the matching service successfully matches 2 users.
- The Hub creates a room for the matched users, assigned with a randomly generated roomId.
- It uses the complexity preferred by both matched users to [get a random question id](#) and stores it in the room.

7.5.5.2 Join Room

Endpoint	/ws/:roomId/:username
Method	GET

- A user is created with the roomId and username provided. His/Her interaction to the server is upgraded from a HTTP request into a websocket connection.
- The room with the specified roomId will keep track of the created user in the Users map.
- Reading and writing of messages through this connection is established.
- When there are 2 users in the room, broadcast to both users that the matched partner has entered the room.

7.5.5.3 Get Question Id

Endpoint	/ws/:roomId
Method	GET

- Fetches the question id stored in the specified room.
- Meant to synchronize the two matched users with the same question.

7.5.6 Message Mechanism

7.5.6.1 Message Schema

```
type Message struct {  
    Content string  
    RoomId string  
    Username string  
    Type string  
}
```

7.5.8.2 Message Types

- **Code** : Sent to the other user every time a user edits the code in the editor for real time updates.
- **Language**: Sent to the other user upon changing the language in the code editor.
- **Chat**: Sent to the other user upon clicking send in the chatroom modal.
- **Enter**: Sent to each user once when there are 2 users in the collaboration room to notify that the other user has joined.
- **Exit**: Sent to notify the other user the moment he/she leaves the room by any means, e.g refresh, close the browser.

7.5.8.3 Reading Messages

Messages are read from the user's websocket connection. They are initially in JSON format, which would be unmarshalled into the Message struct format.

The messages are then passed into the hub's broadcast channel. For every message received by the hub, it will check the roomId and username, and pass the message to the other user's Message Channel of that room.

Should there be any error in reading from the websocket, the user connection is closed and the user would be removed from the hub via its Disconnect Channel.

7.5.8.4 Writing Messages

For every message that comes in from the user's MessageChannel, it will be written into a JSON object and sent through the user websocket connection.

If the message type is either code or language, the user's current solution and language will be updated respectively.

Should there be any error in reading from the MessageChannel, the user connection is closed.

7.5.8.5 Disconnection

History Response Body

```
{  
  room_id: string  
  question_id: string  
  title: string  
  complexity: string  
  solution: string  
  language: string  
  username: string  
}
```

Upon closing of the user connection, the question's complexity and title will be retrieved using the [get question API](#). These with room_id, question_id, solution , username, and language will form the history request body and call the [create history API](#) to create/update the history.

The room will be deleted after both users disconnect from the room.

8. Deployment

8.1 Summary

Peerpreps is deployed through Google Kubernetes Engine (GKE) that is offered on Google Cloud Platform (GCP). Our services run on an autopilot cluster. The below architecture supports high availability ([NF3](#)) and scalability ([NF5](#)).

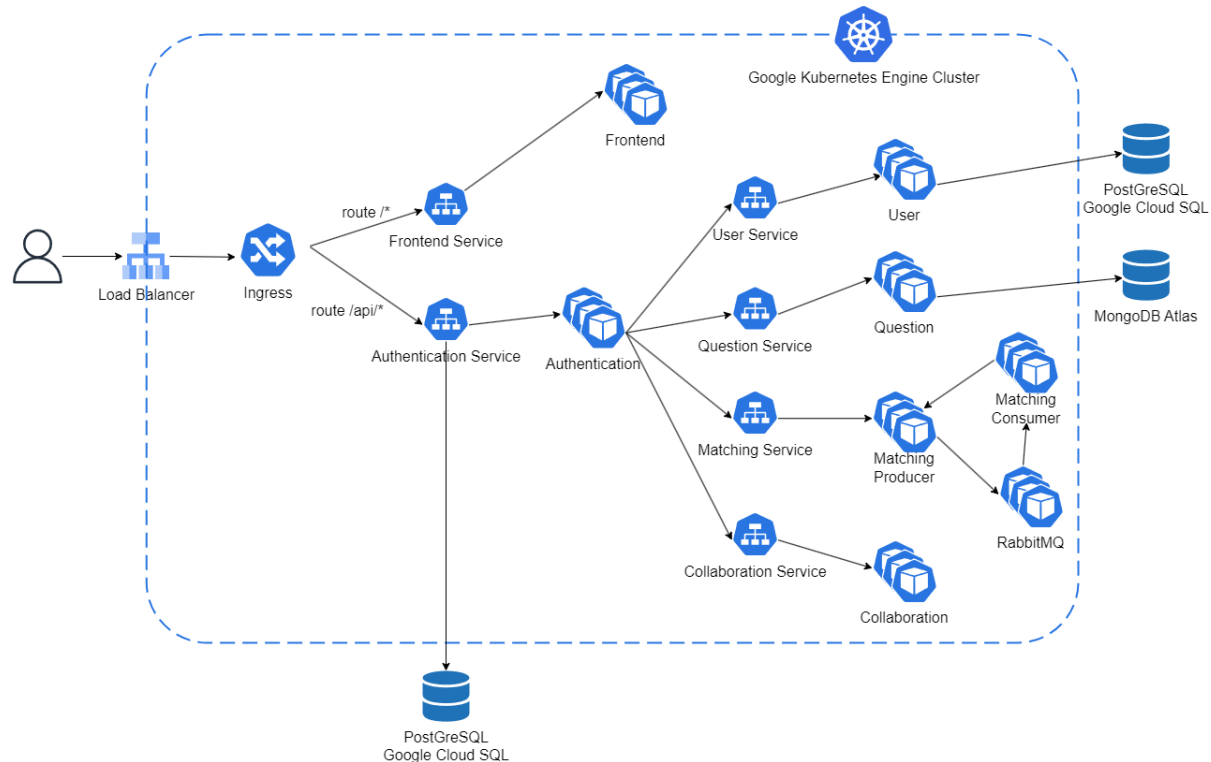


Figure 51: Deployment Diagram

8.1.1 Replica Sets

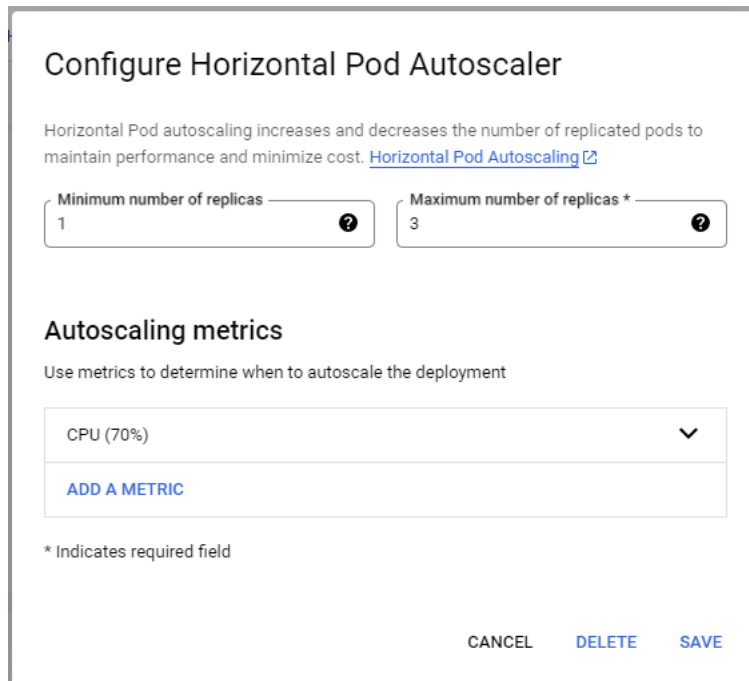
Our requirement for high availability can be met because Kubernetes maintains a state of our deployments. If one pod is down, and the required number of replica sets is not met (in our case: 1), Kubernetes will deploy a pod again to ensure that the requirements are met. If we want to ensure a higher availability, we can configure to have more replica sets in our initial deployment, rather than just 1, so that if one pod fails, another pod is still available. Our choice of 1 pod per service is due to our application being low in user traffic, meaning we do not need many instances. Depending on how this user traffic scales, we can also easily scale up the number of minimum pods to meet the user demand.

In addition, GKE clusters support rolling deployments. This allows us to incrementally update all our pods without the users experiencing any downtime. Likewise, this circles back to providing high availability.

8.1.2 Horizontal pod auto-scaler & Service discovery

Our infrastructure also satisfies high scalability, as we have configured the deployment for each service to scale to 3 replicas when they hit a 70% CPU usage. Moreover, when scaling, each pod doesn't have to worry about the IP addresses of the individual pod and which pod to route to, as everything is handled by the Service for each deployment. Pods will only have to send the request to the Service, and it will route to the pods and distribute traffic based on its own algorithms.

Below is an example of the horizontal pod auto-scaler:



Configure Horizontal Pod Autoscaler

Horizontal Pod autoscaling increases and decreases the number of replicated pods to maintain performance and minimize cost. [Horizontal Pod Autoscaling](#)

Minimum number of replicas ? Maximum number of replicas * ?

Autoscaling metrics

Use metrics to determine when to autoscale the deployment

▼

[ADD A METRIC](#)

* Indicates required field

CANCEL DELETE SAVE

Figure 52: Horizontal Pod Autoscaler

8.1.3 Ingress

Finally, we have an Ingress (Nginx Ingress) as a reverse proxy which helps to route the different endpoints to either the frontend or API gateway. To deploy the Ingress, we used Helm to deploy the respective resources that are already pre-configured by other developers, so we do not have to manually configure everything on our own. Afterwards, we can just use GCP to generate a global IP address for us, and attach it to the Ingress.

Below are all the different Kubernetes resources that we have deployed on GKE:

```
thomas@DESKTOP-1VP4EUM:~/ay2324s1-course-assessment-g13$ kubectl get pods
NAME                                READY    STATUS    RESTARTS    AGE
api-gateway-77878d4545-gqjfh        1/1      Running   3 (7h2m ago)  7h10m
collaboration-service-749dcf49c7-rlccd  1/1      Running   0            161m
frontend-7bb6d746f7-tjkq2          1/1      Running   0            125m
matching-consumer-5bcb574794-sdff8   1/1      Running   0            8m8s
matching-producer-795894c97b-2d4hc   1/1      Running   0            60m
nginx-ingress-ingress-nginx-controller-b888fbb49-vdvh8  1/1      Running   0            161m
question-service-64d8684c75-kjxmb    1/1      Running   0            161m
rabbitmq-0                           1/1      Running   0            127m
user-service-bd46d9c57-7dvgx         1/1      Running   2 (7h4m ago)  7h10m
thomas@DESKTOP-1VP4EUM:~/ay2324s1-course-assessment-g13$ kubectl get svc
NAME                                TYPE                CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
api-gateway-service                ClusterIP           34.118.234.51   <none>           80/TCP           6d6h
collaboration-service              ClusterIP           34.118.236.79   <none>           80/TCP           6d2h
frontend                          ClusterIP           34.118.232.88   <none>           80/TCP           6d1h
kubernetes                        ClusterIP           34.118.224.1    <none>           443/TCP          6d8h
matching-consumer                  ClusterIP           34.118.229.166  <none>           80/TCP           24h
matching-producer                  ClusterIP           34.118.232.67   <none>           80/TCP           6d5h
nginx-ingress-ingress-nginx-controller  LoadBalancer      34.118.229.43   34.143.144.123  80:30887/TCP,443:31644/TCP  6d6h
nginx-ingress-ingress-nginx-controller-admission  ClusterIP           34.118.232.155  <none>           443/TCP          6d6h
question-service                   ClusterIP           34.118.227.66   <none>           80/TCP           6d3h
rabbitmq                           ClusterIP           34.118.229.170  <none>           5672/TCP,15672/TCP  6d5h
user-service                       ClusterIP           34.118.225.206  <none>           80/TCP           6d6h
thomas@DESKTOP-1VP4EUM:~/ay2324s1-course-assessment-g13$ kubectl get ingress
NAME            CLASS    HOSTS                                ADDRESS      PORTS    AGE
ingress-resource  <none>   34.143.144.123.nip.io               34.143.144.123  80      6d6h
thomas@DESKTOP-1VP4EUM:~/ay2324s1-course-assessment-g13$ kubectl get configmap
NAME            DATA    AGE
configmap       15       6d7h
configmap.yaml  0        6d7h
kube-root-ca.crt  1        6d8h
nginx-ingress-ingress-nginx-controller  1        6d6h
thomas@DESKTOP-1VP4EUM:~/ay2324s1-course-assessment-g13$ kubectl get secrets
NAME                                TYPE    DATA    AGE
nginx-ingress-ingress-nginx-admission  Opaque    3        6d6h
secrets                                Opaque   11       6d8h
sh.helm.release.v1.nginx-ingress.v1    helm.sh/release.v1  1        6d6h
```

Figure 53: Kubernetes Resources

8.2 Significant Decisions

8.2.1 Cloud Service Choice

In order to achieve scalability and service discovery, there are many options that are already abstracted by GCP to simplify deployment, such as Google App Engine (GAE). As such, we have to decide between GAE and GKE. As GAE is a more abstracted service, it is significantly more expensive than GKE. Moreover, GAE has more restrictions and fewer configurations, which would be troublesome for us to implement our own infrastructure. For example, we are deploying RabbitMQ as a stateful set, which is not supported by GAE. Thus, we have decided to go with GKE.

8.2.2 Cluster Mode

When creating a kubernetes cluster in GKE, we could choose between the autopilot and standard mode. We eventually chose the autopilot cluster because the autopilot mode offers many built-in features such as node autoscaling and preconfigured security and cluster configurations. This reduces a lot of cluster set up and maintenance efforts, allowing us to focus our efforts on deploying the application correctly.

8.2.3 Database Choice

When choosing how to host our databases, we were given two choices. The first being hosting databases by ourselves, and the second being using third party solutions. If we want to host databases by ourselves, it is harder to scale and manage the data. If we have misconfigured some settings, our data could be easily lost. Thus, we have chosen to use third party solutions, such as MongoDB Atlas and GCP SQL instances, which automatically helps us scale vertically if a certain threshold holds, despite them being more expensive than hosting ourselves.

9. Project Management

9.1 Progress Tracking

In order to keep track of our development process, we have utilized the project feature on Github, which allows us to create and assign tickets, and keep track of the tickets' statuses. This allows our team to have a good understanding of everyone's progress and work together more efficiently. Instead of using Scrum, our team updates each other asynchronously via weekly standups, and this helps us to increase our efficiency as we no longer have to host meetings that consume our time.

We set team agendas week by week, and distribute the taskings amongst each other, depending on the portion that we are in charge of. This allows for clear task distribution and goals. At the end of each week, we will collectively hold a joint debugging session to test our master branch to ensure that whatever we have set out to do is accomplished.

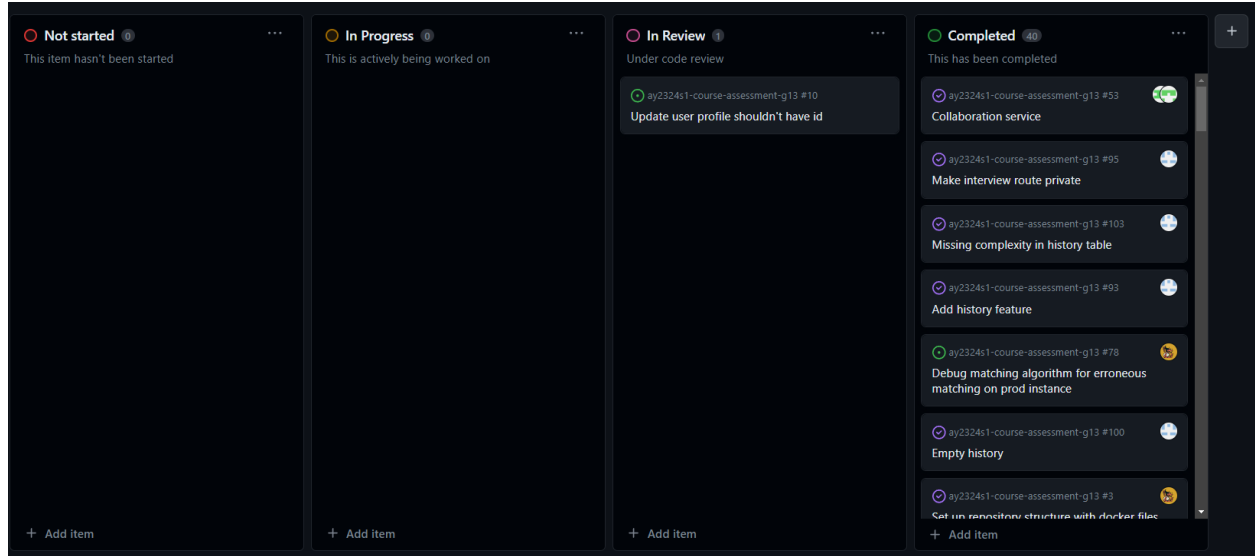


Figure 54: Github Project Management

9.2 Technical Communication

In order to facilitate communication amongst one another in terms of how we should integrate backend and frontend components, we use a shared Notion document where we detail out every API contract for each resource that we expose. This document contains all the resource usage details and caveats, which allows us to work with one another really efficiently. The image below shows an example of an API contract that details how to interact with the auth upgrade endpoint.

Request

```
POST /auth/upgrade HTTP/1.1
Content-Type: application/json

{
  "oauth_id": 1234,
  "oauth_provider": "GitHub",
}
```

This illustrates the flow of upgrading a user. Only Super Admin is able to upgrade the roles of selected users. The selected user information would be passed over as request payload.

+ :: Downgrade Auth User Role Flow

- Using GET `/auth/user/downgrade` endpoint

The process to downgrade a user is similar to upgrading a user. It can only be done by Super Admin and the selected user who is being downgraded would have its information passed over as request payload.

Figure 55: Notion API contracts

10. Suggestions for improvements

1. There is currently no mechanism to store any secrets. Vault can be used in this project for every member to keep track of secrets instead of passing them around in a telegram group. Some of the tools we can use include HCP Vault and AWS Secrets Manager.
2. CD workflows can be added to streamline deployment process instead of manually deploying to the GKE.
3. CI workflows can be added to test whether the code works as expected. Different Github rules such as master branch protection, update branch before merging can be enforced to ensure good software engineering practices.
4. It is possible to configure DNS and buy a domain & TLS certificate to make the application more secure and user-friendly.
5. We could have used Preview Deployments by Vercel to view our changes in a live environment without having to merge our branch and risk rollbacks.
6. Rather than creating our own authentication service, we could have leveraged on an Identity Provider like Auth0.
7. Infrastructure as Code (IaC) tools such as Terraform can be used to keep track of revisions of deployment infrastructure. This will allow the deployment team to reuse revisions and minimize human errors while deploying.

8. We could expand our matching options beyond the complexity level to include question categories and/or their preferred languages. This enhancement allows them to filter questions based on specific categories, empowering them to focus on weaker areas that require extra practice or on topics frequently tested by companies. Including their preferred languages also ensures that both users can practice and refine their skills in a language they are comfortable with, fostering a more synergistic and productive learning environment.
9. We could also provide a solution code so the user can review and understand how to solve the problem. This valuable learning resource not only enhances users' understanding of coding challenges but also fosters continuous learning and substantial growth in their programming skills.
10. Instead of using toasts as a mechanism to notify users that they have unread messages in the chat room since it triggers on every new message and clutters the screen, we could use a sound based notification, similar to current social media applications.

11. Reflections and learning points

Throughout this project, we had the opportunity to explore a variety of technologies, adding an extra layer of interest and enrichment to our overall experience. We delved into new concepts, such as how messages can travel between different parts of a system, how to use Kubernetes and what microservices and Docker are. While it was initially challenging to learn all these new technologies, our team tackled these hurdles together.

Despite our team members possessing strong foundations in software engineering, we encountered challenges during the development phase. The integration of new code often led to the introduction of bugs when merging pull requests into the master branch. This demands extra time and effort to fix the bugs. It then became clear that writing unit tests is crucial for preventing such problems. However, due to time constraints, most of our time was spent fixing bugs rather than implementing tests. Looking back, we recognize that starting unit tests from the beginning could have saved us time and resources in the long run.

This experience underscores the importance of proactive testing and how it can contribute to a more efficient development process. Despite the challenges faced, our journey through this project has been one of continuous learning and growth, reinforcing the idea that a solid foundation, coupled with the right practices, is key to successful software development.