



NUS
National University
of Singapore

CS3219 Project Report

PeerPrep

Group 39

Name	Student Number
Qiu Jiasheng, Jason	A0235206N
Ho Jun Hao	A0234730M
Zoe Ang	A0264840A
Kiew Guang Ting Jerald	A0234188Y
Li Yingming	A0234174J

Declaration	4
Individual Contributions	5
Introduction	6
Project Scope	7
Functional Requirements	7
Non-Functional Requirements.....	14
Nice-to-Haves	19
Architecture	23
Design and Implementation	26
Tech Stack.....	26
Database Design.....	26
MongoDB Atlas	26
Redis.....	27
Frontend-Backend Communication	28
RESTful API.....	28
API Gateway	28
Containerization	29
Deployment.....	30
Security	31
Microservice Internal Design and Implementation	33
Frontend.....	33
Authorization and Authentication	33
API Layer	34
Code Editor	34
Libraries	34
Screenshots	35
Future Considerations	37
Communication Service	37
Chat Box	37




Video Call.....	38
Matching Service	41
Overview	41
Rabbit MQ Queues.....	41
Match Found	42
Match Not Found	43
Match Not Found (Invalid Difficulty & Topic Selection)	44
Question Service.....	44
API Endpoints	45
Question Database	45
Design Considerations.....	45
Collaboration Service.....	46
Web Socket Implementation	46
Creation of Rooms	47
Code Execution Service.....	47
API Endpoints	47
Design Considerations.....	48
User Service	49
API Endpoints	49
User Database.....	50
Design Considerations.....	50
History Service.....	51
API Endpoints	51
History Database	51
Design Considerations.....	52
Project Plan	53
Gantt Chart.....	53

Declaration

We, the undersigned, declare that:

1. The work submitted as part of this project is our own and has been done in collaboration with the members of our group and no external parties.
2. We have not used or copied any other person's work without proper acknowledgment.
3. Where we have consulted the work of others, we have cited the source in the text and included the appropriate references.
4. We understand that plagiarism is a serious academic offense and may result in penalties, including failing the project or course.
 1. We have read the [NUS plagiarism policy and the Usage of Generative AI](#).

Group Member Signatures:

Full Name	Signature	Date
Qiu Jiasheng, Jason		8 November 2024
Ho Jun Hao		8 November 2024
Zoe Ang		8 November 2024
Kiew Guang Ting Jerald		8 November 2024
Li Yingming		8 November 2024

Individual Contributions

The following table summarizes the contributions of each team member in this project.

Name	Contributions
Qiu Jiasheng, Jason	<ul style="list-style-type: none">- Room, Dashboard, Question, and Landing Page- Integrate Code Execution Service and test cases with Frontend- Integrate User Service with Frontend- UI Design
Ho Jun Hao	<ul style="list-style-type: none">- User Service (backend)- History Service (frontend + backend)- Matching Notification service- RabbitMQ- Communication service (Frontend)
Zoe Ang	<ul style="list-style-type: none">- Matching Service (backend), Redis- Nginx, Docker, GCP Deployment, Domain Routing (Cloudflare), HTTPS- Access Control- Dashboard (Frontend)- UI Design
Kiew Guang Ting Jerald	<ul style="list-style-type: none">- Question Service (Backend)- Collaboration Service (Backend)- Communication Service (Backend)- Code Execution Service (Backend)
Li Yingming	<ul style="list-style-type: none">- User Service (Frontend)- Question Service (Frontend)- Matching Service (Frontend)

Introduction

In today's tech industry, technical interviews have become a standard hurdle that aspiring software engineers must overcome to secure positions at top companies. These interviews frequently involve solving complex coding problems before a live interviewer, a skill set that many candidates find challenging to develop on their own. Effective interview preparation, therefore, often requires regular practice and exposure to a variety of coding questions that simulate the real interview experience.

The idea behind PeerPrep stems from the problem that many students struggle to prepare for technical interviews in isolation, as they lack access to real-time feedback and collaboration. Practicing with peers not only helps students deepen their understanding of coding concepts but also improves their ability to communicate and articulate their solutions, which are key skills for succeeding in technical interviews. However, finding a suitable practice partner is often challenging and time-consuming, and few platforms offer a structured, peer-based approach to interview preparation.

PeerPrep is designed to bridge this preparation gap by providing a technical interview preparation platform with a peer-matching system. By connecting students with peers who are also practicing technical questions, PeerPrep enables them to collaboratively tackle coding challenges, simulating a realistic interview environment. The platform provides students with an invaluable opportunity to practice and refine their technical and interpersonal skills in a low-stakes, supportive setting.

The live application can be found at <https://peerprep.zoeang.dev/>.

The project repository can be found at <https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39>.

Project Scope

The scope of our product is detailed through a product backlog, outlining all PeerPrep's functional and non-functional requirements. Requirements are assigned to specific milestones, ensuring a phased and prioritized approach to development. Each item in the backlog is also tracked with a fulfillment status, which is updated upon completion via a pull request.

The requirements are also tasked with different priority levels according to their importance and impact on the user experience. The following list details the different priority levels and corresponding meaning:

- 1) H: High
- 2) M: Medium
- 3) L: Low

Functional Requirements

Functional Requirements	Priority	Milestone	Fulfillment Status
M1: User Service			
M1.1 User Authentication			
M1.1.1 User Registration Description: Implement a registration system where users can sign up with email and password. <ol style="list-style-type: none"> 1. Users should be able to create an account by providing an email and password. 2. The system must check if the email is already registered. If so, the system should inform the user that the email is already used by an existing account. 3. A success message should be displayed upon successful registration, followed by a redirection to the login page. 	H	Milestone 3	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/10
M1.1.2 User Login	H	Milestone 3	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/10

<p>Description: Implement a login system that allows users to login using email and password.</p> <ol style="list-style-type: none"> 1. Users should be able to login using their registered email and password. 2. If login credentials are incorrect, an error message should be displayed. 3. On successful login, users should be redirected to the main page. 			AY2425S1/cs3219-ay2425s1-project-g39/pull/10
M1.2 Profile Management			
<p>M1.2.1 Profile Update</p> <p>Description: Allow users to update their email and password.</p> <ol style="list-style-type: none"> 1. A logged-in user should be able to update their username and password. 2. If the update is unsuccessful, an error message should be displayed. 3. On successful update, a confirmation message should be shown. 	M	Milestone 6	https://github.com/CS3219-AY2425S1-project-g39/pull/54
<p>M1.2.2 View profile</p> <p>Description: Implement a system where users can view their profile details (Username, Email, Last Login).</p> <ol style="list-style-type: none"> 1. Logged-in users should be able to see their profile details. 2. The profile should not show their password. 3. If the profile cannot be retrieved, an error message should be displayed. 	H	Milestone 6	https://github.com/CS3219-AY2425S1-project-g39/pull/54
M1.3 Password Recovery			
<p>M1.3.1 Forgot my password</p> <p>Description: Implement the mechanism to reset the password using a token sent via email.</p> <ol style="list-style-type: none"> 1. Users should be able to reset their password by clicking the “Forgot my password” button. They would then have to enter the email used. This would send 	M	Milestone 6	https://github.com/CS3219-AY2425S1-project-g39/pull/54

a link to the email for the password reset email.			
2. If the password reset is successful, a success message should be displayed, followed by a redirection to the login page.			

Functional Requirements	Priority	Milestone	Fulfillment Status
M2: Matching Service			
M2.1 User Matching Criteria			
<p>M2.1.1 Difficulty Selection</p> <p>Description: Implement a matching system where users are matched based on multiple selected difficulty levels.</p> <p>1. Users should be able to be matched with other users who have selected the same difficulty level.</p>	H	Milestone 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/33
<p>M2.1.2 Topic Selection</p> <p>Description: Implement a matching system where users are matched based on multiple selected topics.</p> <p>1. Users should be able to be matched with other users who have selected the same topic.</p>	H	Milestone 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/33
<p>M2.1.3 No Criteria Selection</p> <p>Description: Implement a matching system where users are matched to another user without selecting any criteria.</p> <p>1. Users should be able to be matched without selecting any criteria. They will be treated as having selected every option for every criterion.</p>	M	Milestone 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/33
M2.1.4 Multiple Choice Criteria Selection	M	Milestone 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/33

<p>Description: Implement a matching system where users are matched based on multiple selections from multiple criteria.</p> <ol style="list-style-type: none"> 1. Users should be able to set multiple selections for each criterion. 2. Users that are matched with multiple overlapping difficulty levels will be given a random difficulty. 3. Users that are matched with multiple overlapping topics will be given a random topic. 			3219-ay2425s1-project-g39/pull/33
M2.2 Queuing for a Match			
<p>M2.2.1 Entering the Queue</p> <p>Description: Implement a matching system where users can enter a queue to find a match.</p> <ol style="list-style-type: none"> 1. Users should be able to enter a queue to find a match after selecting one, multiple, or zero criteria. 2. The system should display a queue time to the user. 3. The system should prioritize users which entered the queue first. 	H	Milestone 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/30
<p>M2.2.2 Exiting the Queue</p> <p>Description: Implement a matching system where users can gracefully exit the queue before they find a match.</p> <ol style="list-style-type: none"> 1. Users can exit the queue gracefully by clicking a button, instead of closing the tab. 	M	Milestone 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/30
<p>M2.2.3 Handling Long Queue Times</p> <p>Description: Implement a matching system where users with long queue times are handled gracefully.</p>	M	Milestone 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/30

1. A user that has reached 5 minutes of queuing will be forcefully exited from the queue. The user will be notified that no matches can be found at the current time, followed by a suggestion to select a more flexible set of criteria.			project-g39/pull/33
---	--	--	---

Functional Requirements	Priority	Milestone	Fulfillment Status
M3: Question Service			
M3.1 Question Bank Management			
<p>M3.1.1 Questions Storage</p> <p>Description: System should store a bank of questions, categorised by difficulty and topic.</p> <p>1. The system should store enough questions such that each topic and difficulty would have varied questions for multiple attempts.</p>	H	Milestone 2	https://github.com/CS3219-pull/12
M3.2 Question Retrieval			
<p>M3.2.1 Difficulty Selection</p> <p>Description: Users should be able to select the difficulty of the question that they want to attempt.</p> <p>1. The system should be able to filter questions for the user based on difficulty.</p>	H	Milestone 2	https://github.com/CS3219-pull/18
<p>M3.2.2 Topic Selection</p> <p>Description: User should be able to select a topic that they want to focus on.</p> <p>1. The system should be able to filter questions for the user based on the topic.</p>	H	Milestone 2	https://github.com/CS3219-pull/18
M3.2.3 Randomisation	H	Milestone 5	https://github.com/CS3219-pull/18

<p>Description: System should randomly pick a question based on the difficulty and topic picked out by the user.</p> <p>1. Users should see varied questions on selecting the same topic and difficulty over multiple sessions.</p>			AY2425S1/cs3219-ay2425s1-project-g39/pull/45
M3.3 Questions Bank Updating			
<p>M3.3.1 Question Updating</p> <p>Description: Admin should be able to add more questions into the question bank.</p> <p>1. Admin should be able to add questions and their relevant information into the question bank.</p>	H	Milestone 2	https://github.com/CS3219-ay2425s1-project-g39/pull/12
<p>M3.3.2 Question Deleting</p> <p>Description: Admin should be able to delete a question from the question bank.</p> <p>1. Admin should be able to delete a question from the question bank.</p>	M	Milestone 2	https://github.com/CS3219-ay2425s1-project-g39/pull/12
<p>M3.3.3 Question Editing</p> <p>Description: Admin should be able to edit existing questions.</p> <p>1. Admin should be able to edit the content of a question in the question bank.</p>	M	Milestone 2	https://github.com/CS3219-ay2425s1-project-g39/pull/12

Functional Requirements	Priority	Milestone	Fulfillment Status
M4: Collaboration Service			
M4.1 Real-time Code Editor			
<p>M4.1.1 Implement Real-time Code Editor</p> <p>Description: Set up a collaborative code editor that allows both users to type code in real time.</p>	H	Milestone 5	https://github.com/CS3219-ay2425s1-project-g39/pull/12

<ol style="list-style-type: none"> 1. Two matched users should be able to simultaneously edit code in the shared editor. 2. Changes made by one user should be reflected in the other user's editor. 3. If connection is lost, the editor should notify both users. 			project-g39/pull/45
M4.2 Session Management			
M4.2.1 Session Initiation Description: After users are matched, initialize a session that allows for collaboration on a specific question. <ol style="list-style-type: none"> 1. Once a match is found, a collaboration session must be created with a unique session ID. 2. The session must be linked to a specific question retrieved from the question service. 3. Both users should be notified and directed to collaborative workspace once the session starts. 	H	Milestone 5	https://github.com/CS3219-AY2425S1-cs3219-ay2425s1-project-g39/pull/45
M4.2.2 Session Termination Description: Allow users to gracefully terminate a collaborative session. <ol style="list-style-type: none"> 1. Either user should be able to end the session by clicking a "Terminate session" button. 2. Both users should receive a notification once the session has ended. 3. The session must be logged for record-keeping purposes. 	H	Milestone 5	https://github.com/CS3219-AY2425S1-cs3219-ay2425s1-project-g39/pull/45
M4.2.3 Disconnection Description: Implement a mechanism that allows disconnected users to reconnect within 5 minutes before the session ends. <ol style="list-style-type: none"> 1. If one user disconnects, the user should be able to reconnect to the room while 	M	Milestone 5	https://github.com/CS3219-AY2425S1-cs3219-ay2425s1-project-g39/pull/45

the other user is still connected to the room			
2. If both users disconnect, the room should be active for 5 minutes, in which either or both users should be able to reconnect to the room in that time			
3. If both users do not reconnect within 5 minutes, the session should be terminated automatically.			

Non-Functional Requirements

Non-Functional Requirements	Priority	Milestone	Fulfillment Status
NF1: Performance			
NF1.1 Response Time			
<p>NF1.1.1 Enhance Real-time Collaboration Performance</p> <p>Description: Ensure the Collaboration Service can handle real-time interactions efficiently. The system should remain responsive and fluid during collaborative sessions.</p> <ol style="list-style-type: none"> 1. The real-time collaboration experience (i.e., chat messages, video call, code editor) should have a latency of less than 200ms. 2. The system should be able to handle up to 50 concurrent sessions without an increase in latency. 	H	Milestone 5	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/45
<p>NF1.1.2 Optimize Website Load Times</p> <p>Description: Improve the website load times to ensure that users can access the application quickly. This includes optimizing both initial load and subsequent page navigations.</p> <ol style="list-style-type: none"> 1. The website should load the homepage within 3 seconds. 2. Subsequent page navigation should occur within 2 seconds. 	H	Milestone 6	Fulfilled

NF1.1.3 Improve Data Fetch Times Description: Optimize the data fetching mechanisms in microservices to ensure that data is retrieved and displayed promptly during user interactions. <ol style="list-style-type: none"> 1. Data fetching for user profiles, questions, and matches should occur within 1 second. 2. The system should handle concurrent data requests within 1 second. 	H	Milestone 6	Fulfilled
NF1.2 Match Time			
NF1.2.1 Fast Match Time Description: Once the queue has at least 2 users that have entered with compatible selected criteria, a match should be made quickly. <ol style="list-style-type: none"> 1. 2 compatible users should be matched within 5 seconds of the second user entering the matchmaking queue. 	H	Milestone 4	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/30

Non-Functional Requirements	Priority	Milestone	Fulfillment Status
NF2: Availability			
NF2.1 Up-time			
NF2.1.1 Health Check for Microservices Description: Develop and implement health check mechanisms to monitor the status of all microservices. <ol style="list-style-type: none"> 1. Services must respond to health check requests, with response times under 200ms and accurate statuses reported in at least 99% of cases. 	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/13
NF2.2 Cloud Deployment			
NF2.2.1 Infrastructure Setup	H	Milestone 6	Fulfilled

<p>Description: The system shall be deployed on a cloud platform, GCP.</p> <ol style="list-style-type: none"> 1. The system shall leverage cloud services for key components including databases, and compute instances. 2. The system shall have at least 95% service level agreement in Singapore. 			
--	--	--	--

Non-Functional Requirements	Priority	Milestone	Fulfillment Status
NF3: Usability			
NF3.1 Ease of Use			
<p>NF3.1.1 Simplify Access to Core Features</p> <p>Description: Ensure that all core features of the application can be accessed within 3 clicks from the homepage to enhance user experience and navigation.</p> <ol style="list-style-type: none"> 1. Users should be able to reach User Service, Matching services, question services, and collaboration services from the homepage in 3 clicks or fewer. 2. Each core feature should be accessible through clearly labeled and intuitive navigation elements. 	M	Milestone 6	Fulfilled

Non-Functional Requirements	Priority	Milestone	Fulfillment Status
NF4: Portability			
NF4.1 Deployment Flexibility			
<p>NF4.1.1 Containerize Microservices</p> <p>Description: Implement Docker containerization for all microservices to ensure consistent deployment across different environments.</p>	H	User Service: Milestone 3	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-

<ol style="list-style-type: none"> 1. All microservices should have Docker files with clear build and run instructions 2. The application should run successfully in a local environment using Docker Compose 		Question Service: Milestone 3	project-g39/pull/13
		Matching Service: Milestone 4	
		Collaboration Service: Milestone 5	
		Nice-to-Haves: Milestone 6	

Non-Functional Requirements	Priority	Milestone	Fulfillment Status
NF5: Security			
NF5.1 Authentication			
<p>NF5.1.1 Implement JWT for User Authentication</p> <p>Description: Integrate JWT to secure user authentication, ensuring only verified users can access PeerPrep services.</p> <ol style="list-style-type: none"> 1. Users must log in using JWT, with no unauthorized access in security tests over 100 sessions. 	H	Milestone 3	https://github.com/CS3219-AY2425S1-project-g39/pull/47
NF5.2 Data Encryption			
<p>NF5.2.1 Encrypt User Data in Transit</p> <p>Description: Enable HTTPS for all communications</p> <ol style="list-style-type: none"> 1. All HTTP requests are redirected to HTTPS, and communications are encrypted with TLS. 	H	Milestone 6	https://github.com/CS3219-AY2425S1-project-g39/pull/47
NF5.2.2 Secure Password Storage	H	Milestone 3	https://github.com/CS3219-AY2425S1-project-g39/pull/47

Description: Store all user passwords using hashing. 1. Passwords must be hashed with salt, with no raw password storage.			AY2425S1/cs3219-ay2425s1-project-g39/pull/10
NF5.3: Cloud Deployment in Google Cloud Platform (GCP)			
NF5.3.1 Security Description: The system shall implement leverage on the cloud deployment to improve our security. 1. The system shall use GCP's encryptions protocols such as AES-256 to protect all data stored or transmitted. 2. The system shall use GCP's automatic management of SSL/TLS certificates for the website, ensuring secure HTTPS connections without manual certificate handling.	H	Milestone 6	Fulfilled

Non-Functional Requirements	Priority	Milestone	Fulfillment Status
NF6: Modifiability			
NF6.1 API Gateway Setup			
NF6.1.1 Single Entry Point Description: The API gateway should act as a single point of entry for all client requests. 1. The API gateway should handle all HTTPS requests and route them to the appropriate microservice.	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/13
NF6.2 Microservice Redirection			
NF6.2.1 Routing Logic	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/13

Description: System should have clear routing logic within the gateway for the appropriate microservice. 1. User input and requests should go to the appropriate microservice.			3219-ay2425s1-project-g39/pull/13
---	--	--	--

Nice-to-Haves

Nice-to-Haves	Priority	Milestone	Fulfillment Status
N1: Communication Service			
N1.1 Chat			
N1.1.1 Chat Box Description: The collaboration room should include a chat box for the users to communicate with each other 1. The chat box should relay messages between the users in a timely fashion	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/48
N1.2 Video			
N1.2.1 Video Call Description: The collaboration room should include a video chat 1. Both users should be able to see each other during the whole time of the collaboration session	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/48
N1.2.2 Sound Description: Both users should be able to talk to each other over the video call 1. Both users should be able to hear each other speak	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/48

Nice-to-Haves	Priority	Milestone	Fulfillment Status
---------------	----------	-----------	--------------------

N2: History Service			
N2.1 Past Questions Attempted			
N2.1.1 Log of Past Questions Description: The system should keep a log of questions that the user has attempted <ol style="list-style-type: none"> 1. The log should have the details of the question (title, difficulty, topics) stored 2. The log should automatically update after the collaboration session has fully terminated 	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/42
N2.2 Description of Past Attempt			
N2.2.1 View of Past Attempt Description: The user should be able to click into a log to see their previous attempt <ol style="list-style-type: none"> 1. The user should be able to see the code attempted for the question at the time of termination of the room 	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/42

Nice-to-Haves	Priority	Milestone	Fulfillment Status
N3: Code Execution Service			
N3.1 Languages Provided			
N3.1.1 Common Languages Description: The system should provide a few common coding languages for the users to code in <ol style="list-style-type: none"> 1. The system should accept code of 3 different coding languages (Python, Java, C++) 2. The user should be able to select their preferred coding language in the collaboration room 	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/49
N3.2 Execution of Code			
N3.2.1 Code Execution	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/49

Description: The user should be able to run their code and get back a result 1. The user should be able to click a button to run their code			AY2425S1/cs3219-ay2425s1-project-g39/pull/49
N3.2.2 Test Cases Description: The user should be able to see test cases and the system should be able to run the code against test cases 1. The system should run the code against given test cases and return a result to the user (Correct, Wrong, Error) 2. The system should allow for custom test cases that the user is able to input	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/49

Nice-to-Haves	Priority	Milestone	Fulfillment Status
N4: Syntax Highlighting			
N4.1 Syntax Highlight			
N4.1.1 Color Differentiation Description: The system should provide different colors for different syntax 1. The system should highlight the code based on the language selected by the user	H	Milestone 6	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/49

Nice-to-Haves	Priority	Milestone	Fulfillment Status
N5: Deployment			
N5.1 Deployment to Cloud Services			
N5.1.1 Google Cloud Platform (GCP) Description: The system should be deployed on GCP	H	Milestone 6	Fulfilled

1. The system should be deployed on GCP to ensure availability			
--	--	--	--

Nice-to-Haves	Priority	Milestone	Fulfillment Status
N6: API Gateway			
N6.1 API Gateway			
<p>N6.1.1 Nginx</p> <p>Description: The system should have an API gateway (Nginx) for its microservices</p> <p>1. The system should use Nginx as an API gateway to redirect requests to the various microservices</p>	H	Milestone 2	https://github.com/CS3219-AY2425S1/cs3219-ay2425s1-project-g39/pull/13

Architecture

Our platform uses a **microservices** architecture to support its range of functionalities in a way that's scalable, modular, and robust. Microservices allow us to design self-contained services, each focused on a specific feature, enabling greater flexibility and independence in development and deployment.

Why did we avoid monolithic architecture?

A monolithic architecture combines all functionalities within a single application, which would have constrained our ability to scale and update individual features independently. Additionally, while we ultimately settled on both JavaScript and TypeScript for development, this decision wasn't fixed initially. A monolithic approach would have required us to use a single language across the entire application, limiting our flexibility to choose the best language for each component.

We also chose to deviate from the module's suggested milestones, occasionally adjusting our pace by doing it in certain weeks. With a monolithic structure, the tightly coupled codebase could have made it challenging to develop and deploy features independently.

Scalability was another initial priority. While we didn't implement orchestrators or scaling solutions in the end, a monolithic setup would have restricted us to vertical scaling, limiting our ability to address bottlenecks by scaling specific features independently.

How do microservices address these challenges?

The microservices architecture overcomes all these challenges. By separating each service, we gained the flexibility to use different programming languages as needed. Since each service is largely independent, development was more straightforward to parallelize, allowing us to assign work in a way that supported concurrent progress. Additionally, this setup opens the door to adding orchestrators, load balancers, and scaling mechanisms in the future, giving us the flexibility to scale individual services as needed.

Of course, microservices come with their own trade-offs. Compared to monolithic architectures, microservices introduce added complexity and network latency from inter-service API calls. We also needed tools like Docker to containerize each service. However, we felt the benefits of microservices—independent scalability, flexible language choice, and streamlined team workflows—more than justified these additional considerations.

PeerPrep comprises seven main microservices, each dedicated to a core feature:

1) **Question Service:**

Manages and organizes coding questions available on the platform, ensuring users are provided with diverse and relevant problems.

2) **User Service:**

Handles user information, authentication, and security, safeguarding sensitive data and managing user access across the platform.

3) **Matching Service:**

Matches users based on problem topics and difficulty levels, enhancing the likelihood of successful collaborative sessions.

4) **Collaboration Service:**

Facilitates real-time collaboration on coding problems, enabling users to work together in a shared coding environment.

5) **Communication Service:**

Supports chat and video call functionalities, enabling users to communicate directly while working together. This service enhances collaborative experience by bridging the gap between coding and conversation.

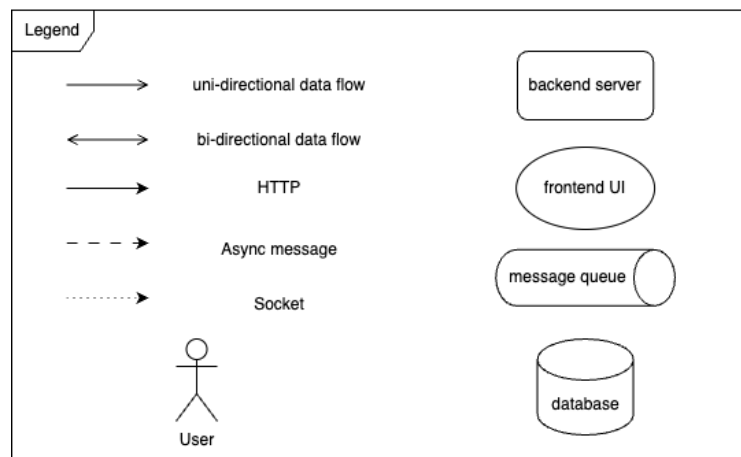
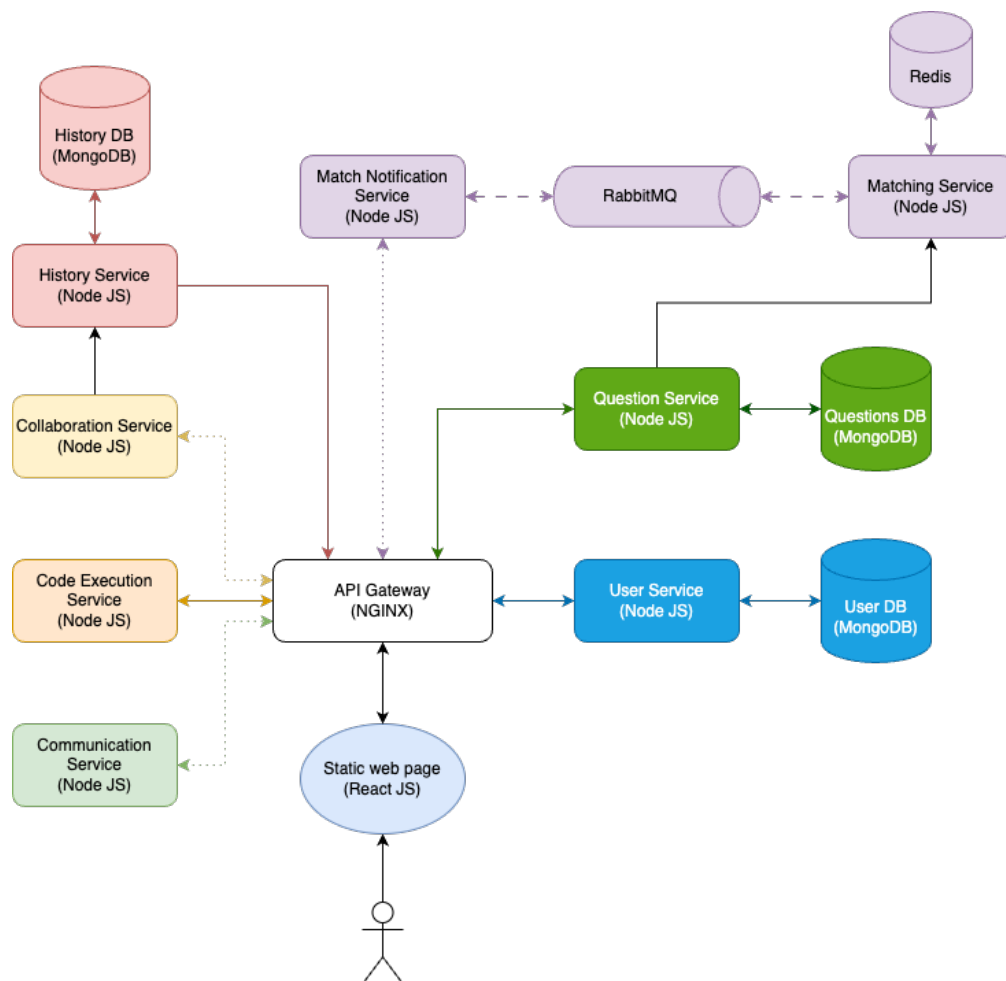
6) **History Service:**

Manages records of past coding attempts, allowing users to review and learn from previous sessions.

7) **Code Execution Service:**

Provides an isolated and secure environment for users to run and test their code, ensuring code runs safely without affecting other parts of the system.

The following architecture diagram illustrates PeerPrep's microservices and how they interact with each other:



Design and Implementation

This section describes the design and implementation of the overall application.

Tech Stack

In general, this is the tech stack we used for the frontend and backend services:

Frontend

- 1) **TypeScript**: Statically typed superset of JavaScript that adds type-checking and other features to improve code reliability and maintainability
- 2) **React**: JavaScript library for building dynamic, component-based user interfaces

Backend

- 1) **JavaScript**: Flexible, high-level scripting language used widely for both front-end and back-end web development
- 2) **Node.js**: Runtime environment that allows JavaScript to be executed on the server side
- 3) **Express**: Minimalist web application framework for Node.js, providing a robust set of tools for building and handling HTTP requests and responses

More technologies and libraries are used within each microservice based on their requirements. They are detailed further in the later sections.

Database Design

The two microservices with a database are the User Service and Question Service. The User Service persists user information and the Question Service stores the questions.

Rather than sharing one single database, we chose to have two separate databases, following the **Database-Per-Service Pattern**. In doing so, we ensure that each microservice is responsible for its own data and logic, supporting a separation of concerns between microservices. Furthermore, the databases are isolated from database changes in other services and can be scaled independently.

MongoDB Atlas

In terms of implementation, we chose MongoDB, a **NoSQL** database, over a traditional SQL database to better accommodate the dynamic and scalable nature of our platform.

Why NoSQL?

NoSQL databases like MongoDB are schema-less, allowing us to store data without rigid structures. This flexibility was essential as it enabled us to add, remove, or modify fields during development without disruptive schema migrations.

Additionally, NoSQL databases are designed for horizontal scaling, making them ideal for applications with varying workloads. As our user base grows, MongoDB can handle increased data and traffic by distributing it across multiple servers, helping the platform stay responsive under load.

Why not SQL?

SQL databases enforce a fixed schema, requiring us to predetermine the structure of all tables. This rigidity would necessitate schema migrations with every structural change. SQL databases are also harder to scale horizontally, making them less suited to handle a growing user base in a flexible, distributed way.

Choosing NoSQL meant sacrificing SQL's advanced querying capabilities, but we determined that our queries would primarily be simple select, filter, and insert operations. We also accepted the trade-off of relaxed data consistency in exchange for flexibility and scalability, as most of our data—aside from user data—did not require strong consistency.

Atlas vs. MongoDB image

We chose MongoDB Atlas over a Docker-hosted MongoDB image because Atlas offers a fully managed infrastructure with built-in monitoring, high availability, and reliability. With automated scaling, backups, and data replication across regions, Atlas ensures that our database remains performant and resilient without manual intervention.

Using MongoDB Atlas allows us to focus on development rather than database management, as it handles optimization and infrastructure needs. This managed setup provides a more efficient, secure, and reliable database solution compared to a self-hosted Docker container, which would require significant maintenance and oversight.

Redis

Redis was used particularly for the implementation of Matching Service.

Why Redis?

Redis was first chosen due to the nature of the service. The data in the Matching Service need not be persistent, as if the server or service was restarted, sockets between the clients and the servers would need to be re-established anyways. As such, it would make sense to drop all current queues. This means that not conducting snapshots and clear data on server restart would be in line with our use case.

Additionally, Redis supports tagging, allowing us to allocate multiple tags for multiple Difficulties and Topics which the users select. This allows for fast retrieval and ease of use.

Why not a traditional database?

Firstly, this is a school project, so we wanted to try learning something new, like Redis.

Secondly, Redis mainly runs on system memory. While this is a limited resource, it does mean that writes and retrievals from it will be faster compared to traditional database. We do understand however that this benefit may be marginal due to the low data rate which we are dealing with.

Frontend-Backend Communication

RESTful API

In our application, frontend-backend communication is facilitated through Representational State Transfer Application Programming Interfaces (RESTful APIs), which enables smooth, stateless interactions between the client and server. RESTful APIs follow a standardized set of HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations, making it simple for the frontend to retrieve or manipulate data in the backend. Each endpoint in the API represents a specific resource, such as user data or application content, which the frontend can access by sending HTTP requests to well-defined URLs.

This RESTful design makes our API scalable and flexible, as it can be easily extended with new endpoints or integrated with other services without major architectural changes. It also ensures consistency in data handling, improving both usability and security.

API Gateway

To manage and secure our RESTful API, we utilize an API Gateway built on NGINX, which acts as a single-entry point for all client requests to backend services. It is responsible for handling routing and access control, directing each request to the appropriate microservices within our backend infrastructure. By consolidating these responsibilities at

the gateway, we simplify backend logic and ensure that each service remains isolated and secure.

Furthermore, we use it to control Cross-Origin Resource Sharing (CORS) behavior, ensuring that all requests being parsed to the backend come from our domain, ensuring that all data is encrypted. It also has the capacity for load balancing and other more advanced features should it be required in the future

Containerization

Docker was chosen as the containerization tool for PeerPrep's various microservices for the following reasons:

1) **Portability:**

Docker containers package each microservice with all its dependencies, configurations, and environment settings, ensuring that it can run consistently across any platform that supports Docker. This containerized approach enables developers to create images that work seamlessly on their local machines, in testing environments, or on production servers without modification. This is especially usefully considering the variance in platform between developers and the deployment server.

2) **Isolation of Services:**

As each microservice operates in its own container, completely isolated from the others. This isolation ensures that the services don't interfere with each other's dependencies or environment configurations, reducing compatibility issues and making debugging simpler. This also aligns with the microservices philosophy of having independently deployable units that can be maintained and scaled without impacting other services.

3) **Scalability and Resource Efficiency:**

Docker containers are lightweight compared to traditional virtual machines, allowing us to run multiple services on the same host with minimal overhead. For PeerPrep, this means we can efficiently scale services by spinning up additional container instances as needed, improving resource usage and allowing our infrastructure to handle increased demand dynamically.

Deployment

PeerPrep is deployed via **Google Cloud Platform (GCP)**.

Why the Cloud?

A lot of the advantages are not limited to GCP specifically but also to the broader group of all major cloud providers like AWS and Azure. It is affordable, considering school credits or the free ones that come with GCP. Even so, GCP quoted about \$7 per month.

We have the choice of location. With GCP, we could select different regions, ranging from Singapore to USA. While we of course chose Singapore, since that's where our target demographic lies, it would be just as easy to relocate it to USA should we want another instance for international users.

Additionally, there are a lot of other things which are done well by default in cloud deployments. A major plus would be their security, with the default levels of server hardening being good. There is a slew of other advantages, like network capacity, and discrete static IPv4 addresses, which numerous very nice to have, but we won't particularly touch on.

Cloud vs. Self Hosting

When considering deployment, we additionally considered self-hosting the site, since one of our members already had a hardened, internet exposed server, with docker and other useful tools already installed, making it trivial to deploy on the server, routing the new domain using nginx. We however decided against to.

The major advantage here which we were after though, is availability. While there are work arounds when it comes to self-hosting, like hosting redundant servers in multiple locations and routing them through Cloudflare in case of a power or internet service outage, it would still involve some disruption to the service. GCP on the other hand is almost confirmed to be always available with major cloud providers settling their own redundancy, thus guaranteeing a monthly availability for single VMs of 99.9%.

Using a cloud-based solution additionally comes with the peace of mind that even if your VM gets compromised, it's only your VM on GCP. It won't or shouldn't lead to your private home network being compromised. For self-hosting however, it is common for attackers to persist better on other, less fortified machines, especially IoT devices, once the network is penetrated.

With GCP it is also easier to update the service. With a single server implementation, updating the server usually requires taking it down, updating it, before making it available

again. With GCP however, we could just spin up a new VM, even with our own image if we wanted. We then would put the updated code on the new VM, test it, ensure that it runs according to our specification, before redirecting the domain to the new VM using Cloudflare. Then the old VM could just be deleted or reused the next time we'd like to update the service. While there is still disruption to current users, the services could be checked or monitored to ensure that the update occurs when there are little to no users; And the service would immediately be made available again.

Security

When building PeerPrep, we implemented several security mechanisms:

Obfuscated Origin Server

By using Cloudflare's proxy, our origin server's IP address is hidden, ensuring that only Cloudflare's IP addresses are publicly exposed. This obfuscation prevents direct attacks on our server, adding a protective layer between attackers and the origin infrastructure.

DDoS Protection

Cloudflare provides robust protection against Distributed Denial-of-Service (DDoS) Attacks for our application, leveraging its industry-leading network to identify and block potentially harmful traffic. This service effectively mitigates large-scale attacks, maintaining uptime and performance during attempted DDoS events.

SSL Encryption

The application at peerprep.zoeang.dev is secured with an SSL certificate issued by Cloudflare, ensuring that data exchanged between clients and our servers remains encrypted. NGINX is configured to redirect all HTTP requests to HTTPS, enforcing secure, encrypted communication.

Edge SSL certificates from Cloudflare secure data transmission between clients and the Cloudflare proxy, with a short 7-day expiration that limits the time available for key cracking. This setup enhances security by making MITM attacks highly improbable.

Origin SSL certificates secure data between Cloudflare and the origin server, ensuring that if the origin's IP were discovered, data remains encrypted during transit between the proxy and origin.

CORS Protection

Cross-Origin Resource Sharing (CORS) settings are configured to prevent Cross-Site Request Forgery (CSRF) attacks from malicious websites that may be open on a client's browser, safeguarding against unauthorized actions on behalf of users.

Whols Privacy

Domain privacy settings are enabled, concealing the domain owner's identity in public Whols records, which adds a layer of anonymity and security for the application's ownership information.

Password Protection

Passwords are encrypted between the client, proxy, and server using SSL. They are also salted and hashed before being stored in the database, so as to not make them readable should the data or the database be compromised.

Access Control

Access control is implemented for the admin page, where Create, Update and Delete operations are limited to the administrator. Additionally, access control is implemented for other operations, like matching and collaboration, only allowing users with a valid JWT token to access these services.

Microservice Internal Design and Implementation

This section details the design and implementation of each microservice. Of which, two of them involve more complex internal designs that we shall explore with greater depth – Communication Service and Matching Service.

Frontend

To ensure maintainability and clean boundaries between the frontend's different functionalities, we organized the frontend into several parts:

- 1) **APIs**: Consolidated API request declarations
- 2) **Components**: Shared, reusable frontend components used across the application
- 3) **Hooks**: Shared hooks for state management used across the application
- 4) **Pages**
- 5) **Types**: Shared TypeScript types used across the application

The above structure is referenced from [alan2207/bulletproof-react](https://github.com/alan2207/bulletproof-react), a simple and scalable architecture for building production ready React application.

Authorization and Authentication

The frontend handles authentication and authorization through an **AuthProvider**, which uses React Context to manage user session data using JWTs and user IDs stored in the local storage.

When a user logs in, the backend issues a JWT, which is stored in local storage alongside the user ID. The AuthProvider component provides a context for the rest of the app, making it easy to access authentication status, user details, and authorization information across frontend components. The use of local storage ensures that the session persists even when the page is refreshed, which essentially allows users to stay logged in even after closing the browser.

The frontend also uses protected routes to control access, checking for the presence of a JWT token in local storage to determine if a user is authenticated. If the token exists, the user is allowed to proceed; otherwise, they're redirected to the login page. We decided to avoid doing backend verification on each route access, reducing network requests and improving performance, while the backend still verifies the token's validity for any API calls. This setup ensures quick route access management while maintaining secure authentication for sensitive actions.

API Layer

The API layer is organized into backend microservices that the frontend communicates with via a **shared Axios instance**. This instance intercepts all outgoing requests, automatically attaching the JWT token in the headers to authenticate each request. It also intercepts responses, providing a centralized approach to handle errors consistently and extract messages in a standardized format, which improves the reliability of error handling and simplifies debugging across different microservices.

Code Editor



For the shared code editor, the frontend integrates **CodeMirror**, a code editor component available for React. We chose CodeMirror because it provides a rich selection of features, including syntax highlighting and cool color schemes. We enabled syntax highlighting for Python, JavaScript, and C++. In doing so, we enhance the readability of the code editor and the user experience by emulating a modern code editor.

Beyond these basic features, CodeMirror also offers search/replace, autocompletion, undo history, which are features that we can utilize in future iterations for a better user experience.

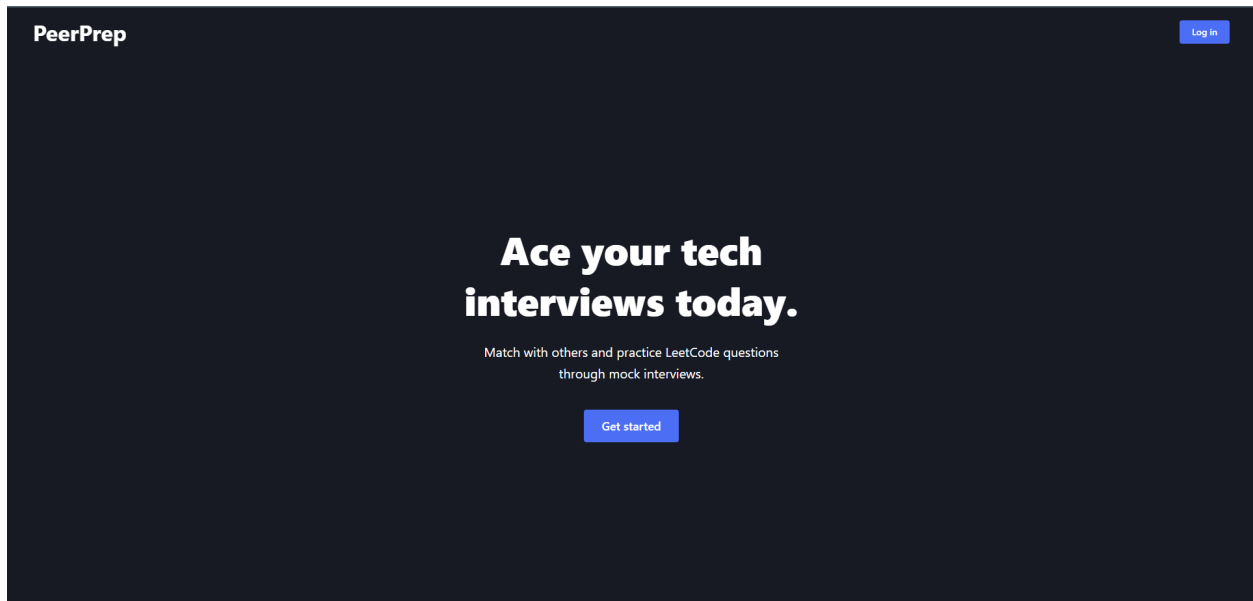
Libraries

PeerPrep also relies on the following libraries:

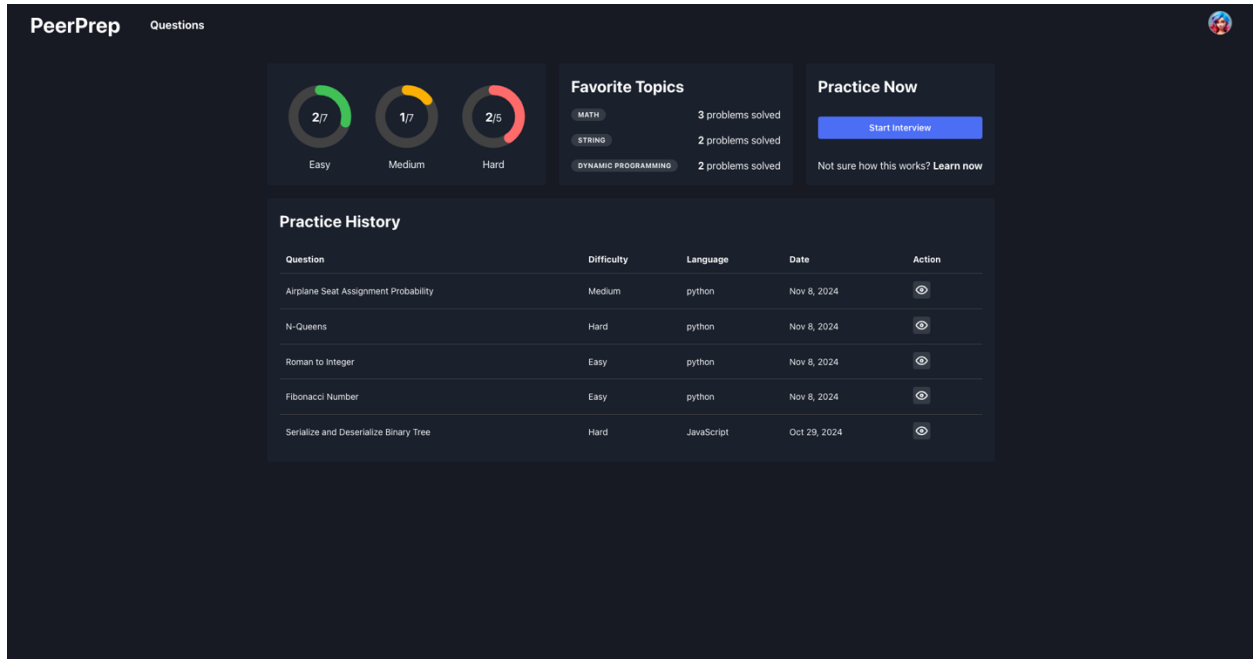
- 1) **Mantine**: Modern React component library that provides a comprehensive set of customizable, accessible UI components
- 2) **Vite**: Fast, modern build tool and development server that optimizes the bundling and hot reloading of frontend projects
- 3) **React Router**: Library for managing and implementing dynamic routing in React applications

Screenshots

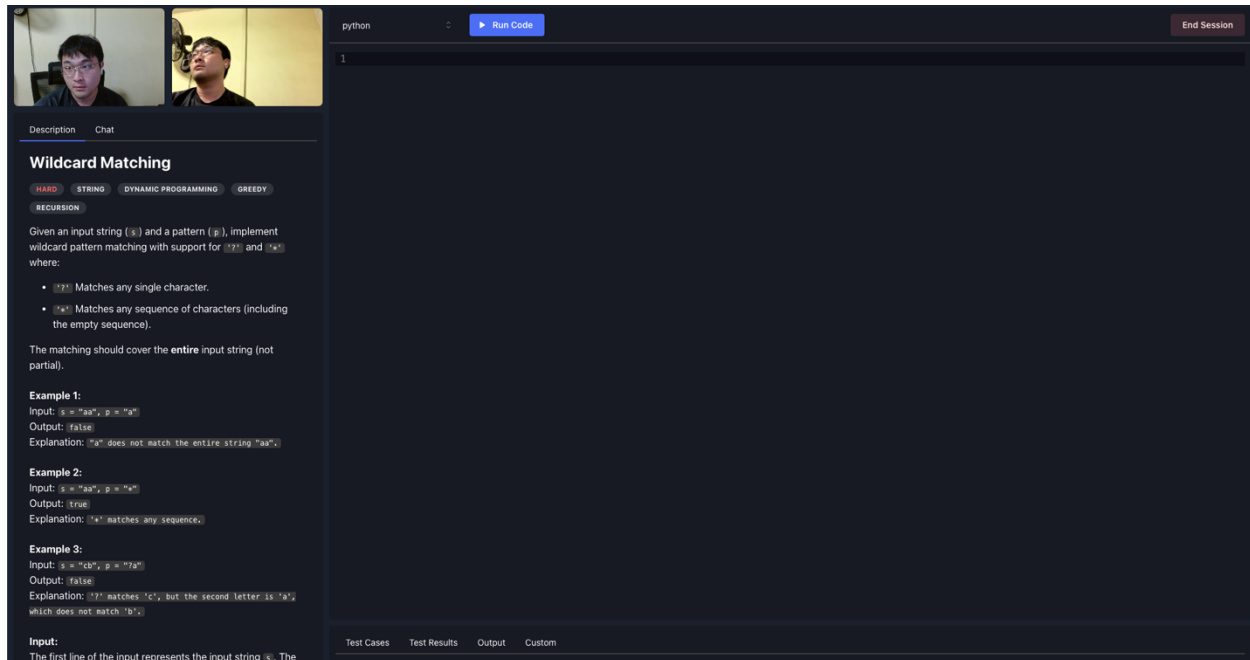
The following images are screenshots of the frontend.



Landing Page



Dashboard Page: Users can start new interviews and view their past attempts.



Wildcard Matching

HARD **STRING** **DYNAMIC PROGRAMMING** **GREEDY** **RECURSION**

Given an input string (*s*) and a pattern (*p*), implement wildcard pattern matching with support for `'?'` and `'*'` where:

- `'?'` Matches any single character.
- `'*'` Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

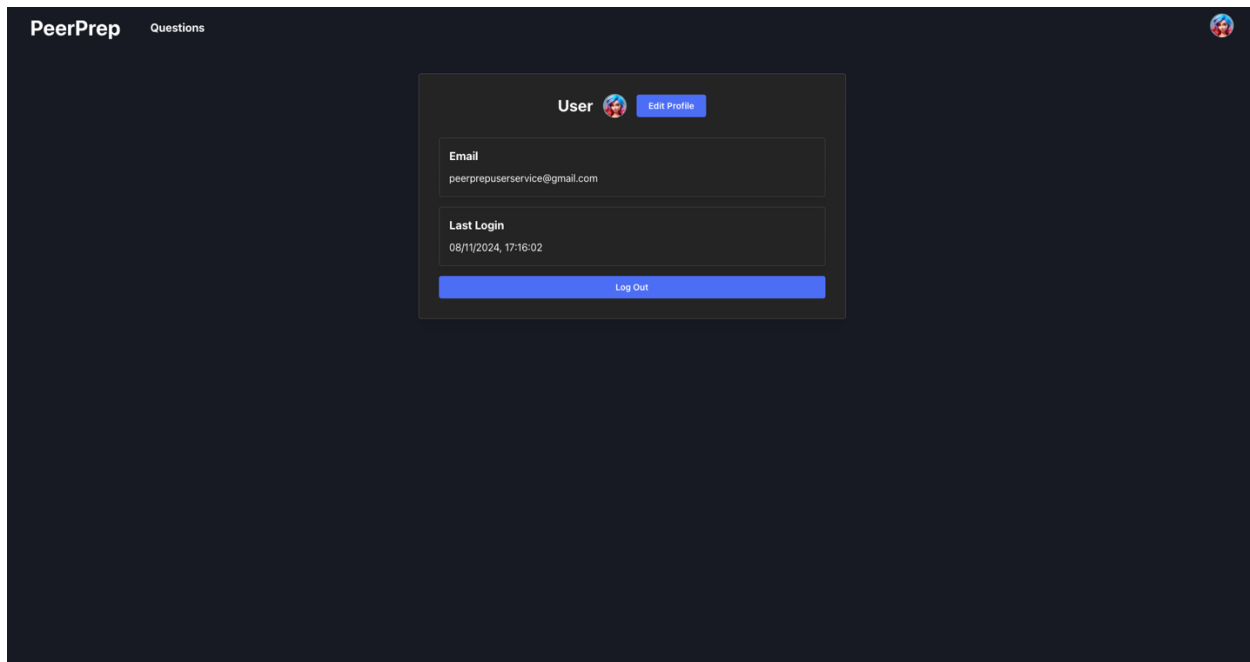
Example 1:
Input: `s = "aa", p = "a"`
Output: `false`
Explanation: `"a"` does not match the entire string `"aa"`.

Example 2:
Input: `s = "aa", p = "a*"`
Output: `true`
Explanation: `"a"` matches any sequence.


Example 3:
Input: `s = "ab", p = "?*a"`
Output: `false`
Explanation: `"?"` matches `"c"`, but the second letter is `"a"` which does not match `"b"`.

Input:
The first line of the input represents the input string *s*. The

Collaboration Page: Matched users can conduct practice interviews.



PeerPrep Questions

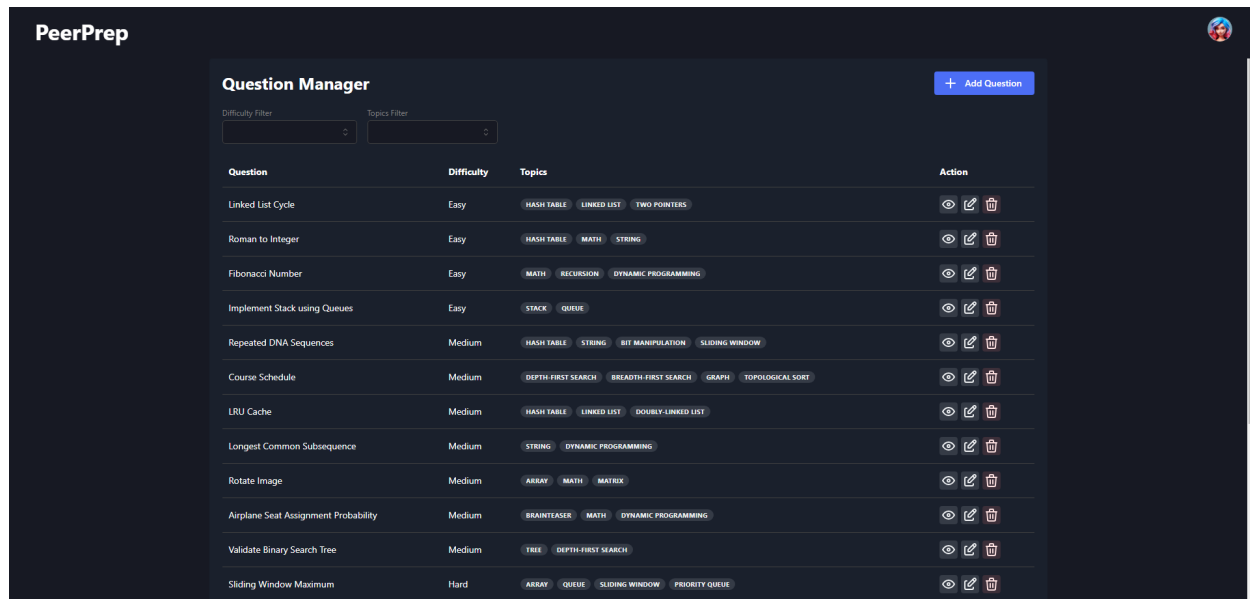
User  [Edit Profile](#)

Email
peerprepuserservice@gmail.com

Last Login
08/11/2024, 17:16:02

[Log Out](#)

Profile Page: Users edit their profile and log out.



Admin Page: Admin users can create, edit, and delete questions.

Future Considerations

One observation during development is that, as the application grows and adds more features, components are becoming increasingly interdependent. Future iterations of the frontend could adopt a more modular folder structure, moving away from the current flat setup, by segregating feature-specific code from globally shared code. This would involve organizing the codebase into distinct folders for each feature, where each directory encapsulates all related components, utilities, hooks, and styles. Globally shared code could then reside in a separate shared or common folder, improving maintainability and making it easier to manage dependencies as the application scales.

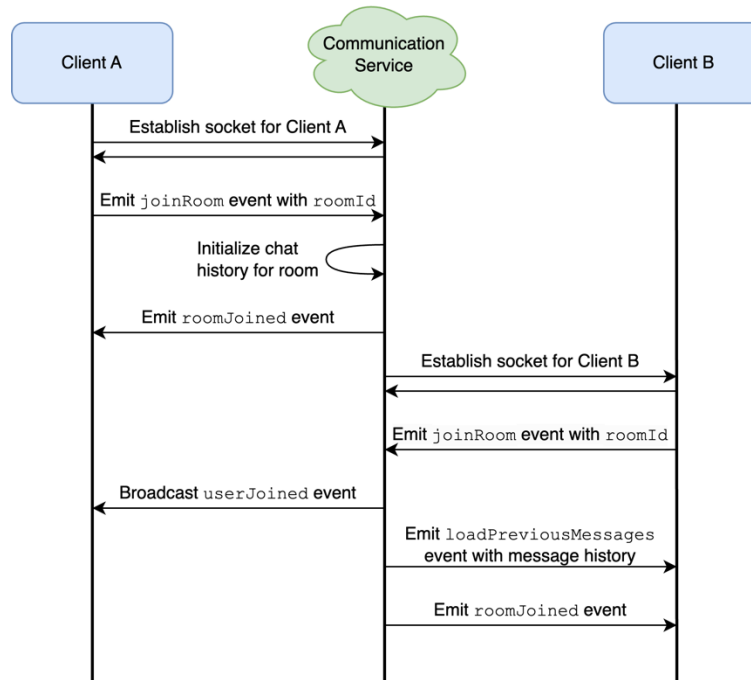
Communication Service

Our Communication Service features a video call and chat box. This requires an exchange of camera, audio, and text messages between our two users.

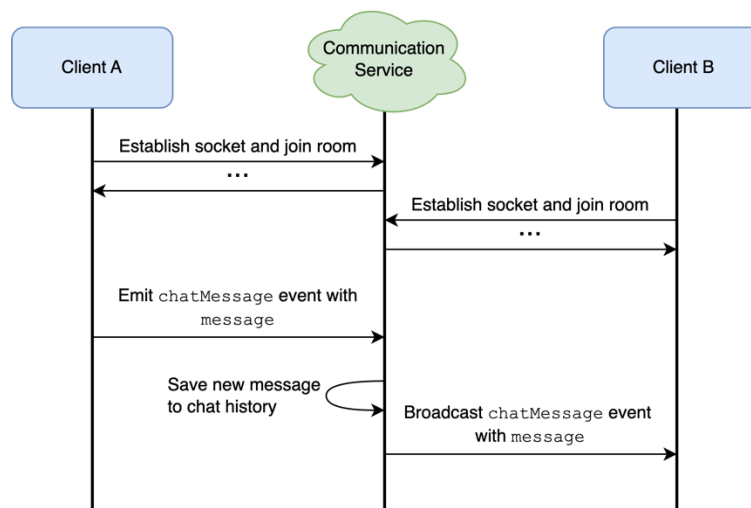
Chat Box

The exchange of text messages is handled through **web sockets**, where our server helps to store a history of the conversation as backup in case a user unexpectedly disconnects.

The following diagram illustrates the process of two clients joining a room.



After the rooms are created, a user can emit a message to the server, which is then broadcast to other users in the same room. The server essentially acts as a ‘middleman’, facilitating the exchange of messages between clients through sockets. The following diagram details the process of sending messages.



Video Call

The camera and audio exchange for our video call proved to be much more challenging. We decided on using **WebRTC (Web Real-Time Communications) peer-to-peer (P2P)**

connections for video calls, instead of routing through our central server, as it offers several advantages:

1) **Reduced Latency:**

In a P2P connection, data is sent directly between peers, minimizing the distance and number of hops the data must travel. This reduces latency, leading to lower lag and a smoother, real-time experience, which is especially important for interactive applications like video calling.

2) **Bandwidth Efficiency:**

By bypassing a server and connecting directly, WebRTC P2P connections reduce the need for additional bandwidth on a central server. Each participant only needs to handle their own data stream, avoiding the overhead and potential bottlenecks caused by server-based routing.

3) **Scalability:**

Since there is no need for a central server to route media data, scaling to more users for 1-to-1 calls is simpler. Each new P2P call doesn't impose additional load on your infrastructure, making it cost-effective and more easily scalable for individual connections.

4) **Enhanced Privacy & Security:**

WebRTC encrypts media streams by default using Datagram Transport Layer Security (DTLS) with Secure Real-Time Transport Protocol (SRTP), ensuring that audio and video data remain private between the participants. With a direct P2P connection, there's less chance of data interception since it bypasses central servers, making the connection inherently more secure.

Using the same web socket that handles text messages, our communication service facilitates the P2P connection by exchanging the WebRTC 'offer', 'answer' and 'candidate' components.

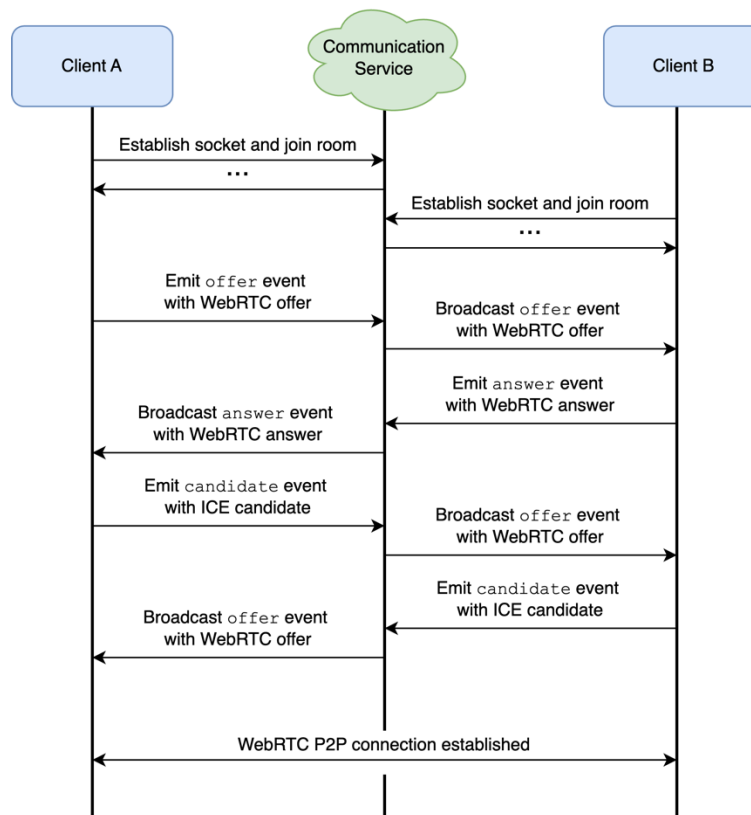
1) **Offer:** Session Description Protocol (SDP) message created by the user who initiates the WebRTC connection.

- a. Contains information about the media format, codecs, network information, and desired connection parameters, allowing the other peer to understand what kind of connection is requested.

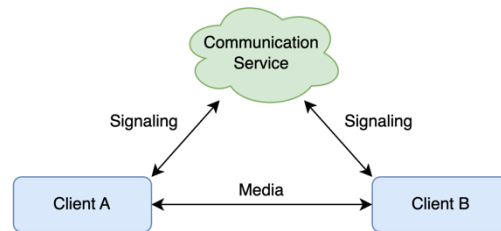
2) **Answer:** SDP message created by the user who receives the offer.

- a. Contains similar information to the offer but tailored to acknowledge and accept the offer's parameters.
- 3) **Interactive Connectivity Establishment (ICE) Candidate:** Message connecting information on how to reach a specific peer across different network paths.

These three components allow WebRTC peers to negotiate and establish a direct connection, creating an efficient, real-time path for media or data transmission. The following diagram highlights the process for establishing a WebRTC P2P connection between two clients.



The WebRTC components are exchanged through our server to establish a direct connection between clients. Video and audio are then exchanged **directly** between clients through the WebRTC P2P connection. The following diagram represents a high-level architecture of the video call feature. Note that the video and audio never go through our server.



Matching Service

Our matching service involves matching users who are currently online to each other, according to specifications provided by the users. This requires us to establish and maintain a connection with our users until a match is found.

Overview

The matching service used 2 express servers, a Rabbit MQ instance for communication between servers, and a Redis instance, to store clients currently waiting.

The first express server, the Match Notification Service, establishes a socket with the client while matching, to ensure that the client is online while searching for its partner. If a client disconnects while matching, the client is also automatically removed from the queue, ensuring that clients don't get matched with clients who are offline.

The second express server, the Matching Service, handles the matching itself. It sends and receives information with the Match Notification Service using RabbitMQ. It gets information about users and their Difficulty and Topic Selection, matches the user if there is a match and if not puts the new user in Redis to be matched. The timestamp when the user arrives is also stored, to ensure that the users are matched in a First-In-First-Out fashion.

RabbitMQ is used for asynchronous communication between Match Notification Service and Matching Service.

Redis was also used for storing users while users are waiting to be matched. Users are tagged by their Difficulty and Topic selections to make it easier to find a match when a new user is found.

Rabbit MQ Queues

Search Queue

- Transmits information when matching is initiated
- Information Transmitted: User ID, Difficulty Selection, Topic Selection

- Match Notification Service → Matching Service

Match Found Queue

- Transmits information when a match is found
- Information Transmitted: User ID 1, User ID 2, Session ID, Question ID
- Matching Service → Match Notification Service

Disconnect Queue

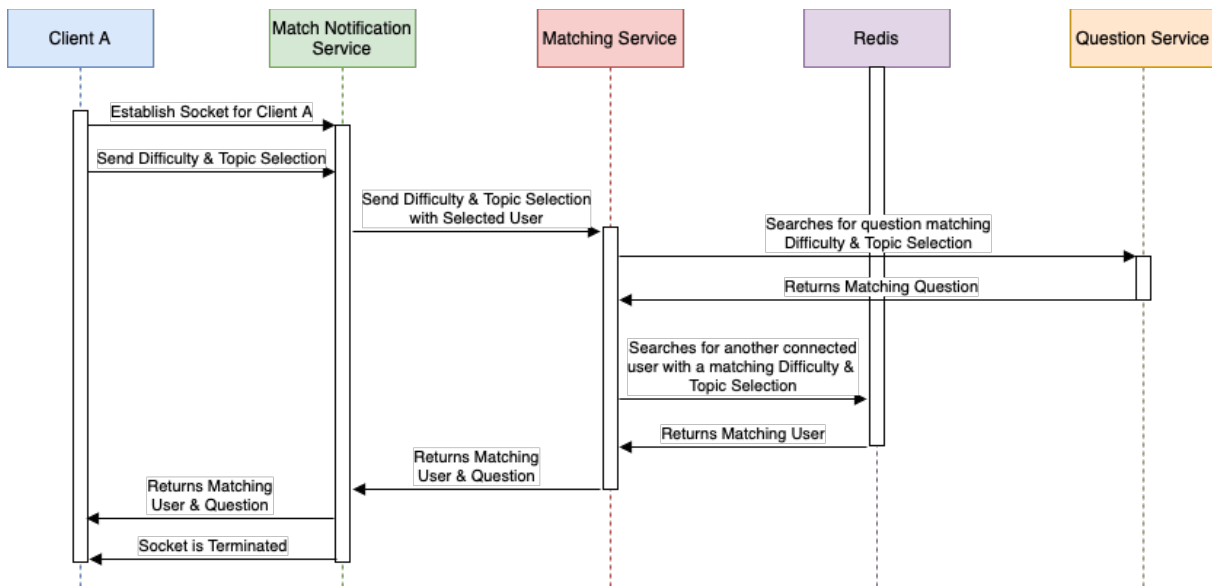
- Transmits information when a user terminates their socket prematurely
- Information Transmitted: User ID
- Match Notification Service → Matching Service

Error Queue

- Transmits information when the matching service experiences a user error
- Information Transmitted: User ID, Error Tag
- Matching Service → Match Notification Service

Match Found

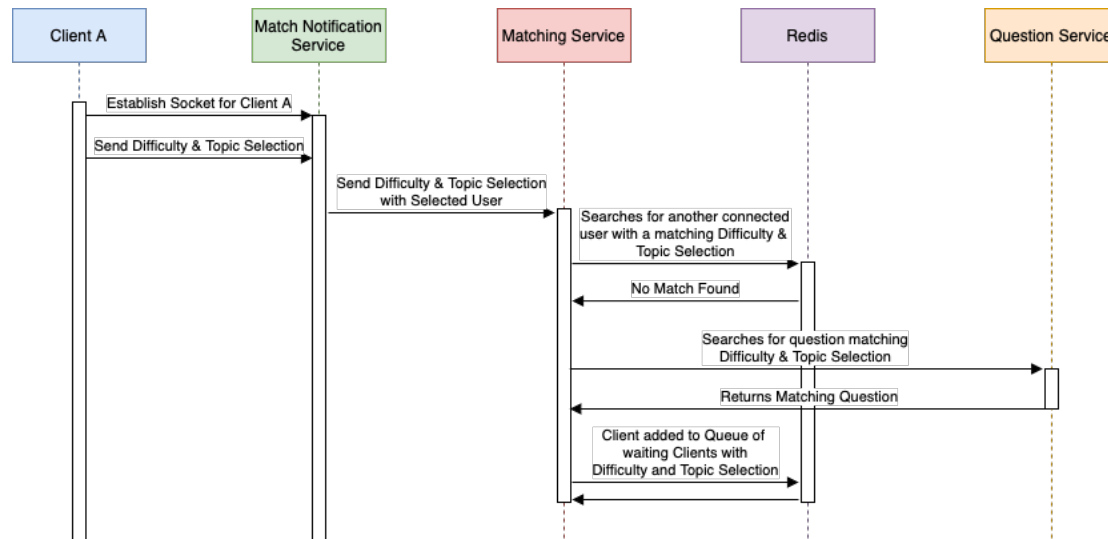
1. Establish socket with Client A using Match Notification Service
 - 1.1. Ensures that the Client remains online for the duration of matching
2. Difficulty and Topics selection is sent to Matching Service
3. Redis is queried for matching user
 - 3.1. Tries to find a connected user who's Difficulty and Type matches that of the current user's
 - 3.2. Client B is found
4. Question is selected using the overlapping selections of Client A and Client B's Difficulty and Type
5. Matching Service returns the matching question and user to the Match Notification Service
6. Match Notification Service returns the matching question and Partner ID to both Client A and Client B



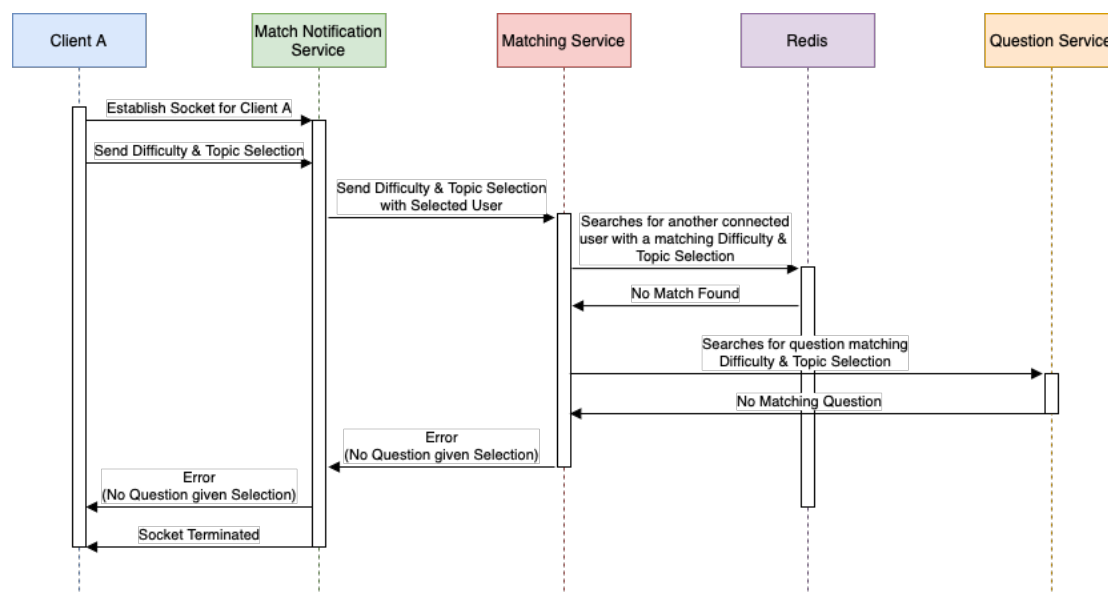
Match Not Found

1. Establish socket with Client A using Match Notification Service
 - 1.1. Ensures that the Client remains online for the duration of matching
2. Difficulty and Topics selection is sent to Matching Service
3. Redis is queried for matching user
 - 3.1. Tries to find a connected user who's Difficulty and Type matches that of the current user's
 - 3.2. No matching user is found
4. Question is search for given selected Difficulties and Topics
 - 4.1. If no valid question is found given client's selection, socket is terminated
 - 4.2. Ensures client is not waiting for non-existent question
5. Client A and its Difficulty and Topic selection is added to Redis Queue to wait for question

Match Not Found (Valid Difficulty & Topic Selection)



Match Not Found (Invalid Difficulty & Topic Selection)



Question Service

Our question service manages and organizes coding questions available on the platform, ensuring users are provided with diverse and relevant problems. We implemented RESTful APIs which were most suitable for the CRUD functions we needed for questions. Our main design decisions were concerning filter criteria and formatting.

API Endpoints

The Question Service is designed as a RESTful API that communicates over HTTP. The following key API endpoints cover essential user functions:

- 1) Create a Question
- 2) Read
 - a. All questions
 - b. One question
 - c. Questions of a certain difficulty
 - d. Questions of a certain topic
 - e. Questions that fit a custom set of criteria
 - f. A random question that fits a custom set of criteria
 - g. All topics
- 3) Update a Question
- 4) Delete a Question

Question Database

We use MongoDB to manage question data. The question data schema includes the following fields:

- 1) id (number)
- 2) title (string)
- 3) description (string)
- 4) topics (string [])
- 5) difficulty (string)
- 6) testCases (testCase [])

The test case schema is built within the question schema to support our code execution functions, with the fields:

- 1) input (string)
- 2) answer (string)

Design Considerations

Schema & Database Structure

We considered whether the test cases should be built within the question schema or whether they should be a separate database managed by the code execution service. We decided to have the test cases inside of the question schema to avoid tightly coupling the

question and code execution services, particularly when questions (and their test cases) are created, updated, or deleted.

Filter Criteria

It was important to implement getting questions by a custom set of criteria. This was to support our matching service which, upon a successful match of two users, would also need to assign them a suitable question.

Formatting

We started out using the raw text provided by the course for our question descriptions. However, we realized that storing the question description in HTML allowed us to better display code and mathematics excerpts relevant to each question. It greatly improved the formatting of our question descriptions for users.

Collaboration Service

Our collaboration service facilitates real-time collaboration on coding problems, enabling users to work together in a shared coding environment.

Web Socket Implementation

The primary responsibility of the collaboration service is to ensure that the two users' code editors are in sync. For this purpose, we decided on the implementation of a web socket, as opposed to using RESTful APIs, as it aligns better with several key NFRs.

- 1) **Responsiveness:** Web sockets provide instant, bidirectional communication, allowing the system to reflect changes made by one user immediately on another user's editor. This supports low-latency responsiveness, crucial for a smooth collaborative editing experience.
- 2) **Scalability & Efficiency:** Since web sockets keep a single persistent connection open per user session, they minimize the network overhead associated with frequent requests. This approach is more resource-efficient and scalable, as it reduces the strain on the server even when handling high-frequency updates from multiple users.
- 3) **Consistency & Real-Time Synchronization:** Web sockets help maintain real-time data consistency between users by instantly broadcasting each change, ensuring that both editors stay in sync without requiring complex polling or data fetching. This meets the NFR of always maintaining data accuracy and synchronization in a collaborative environment.

Creation of Rooms

We considered using RabbitMQ for Matching Service to initiate the creation of rooms in the Collaboration Service (and Communication Service) but ultimately decided to create the room upon a request sent from the Frontend.

Benefits of RabbitMQ

The benefits of using RabbitMQ for this purpose would give us greater assurance that the rooms created are legitimate and assigned correctly. There would be less need for input validation between the Frontend and Collaboration Service. It would also synchronize Collaboration Service rooms with Communication Service rooms, to ensure that both are running and for the right users.

Loose Coupling

However, we felt that the benefits of Rabbit MQ were outweighed by the cons. Our implementation prioritizes loose coupling, to avoid unnecessary dependencies between Matching Service and Collaboration Service, as well as possibly Communication Service. This allowed for easier development and testing as well.

Latency Issues

We decided to avoid a potential latency issue, where the Frontend client is attempting to join a Collaboration Service room that has not yet been created. Our implementation avoids this completely by creating the room instantly upon receiving the Frontend request.

Possible Future Extension

We also considered a possible feature that would extend from our current services. This feature would allow a user to invite a chosen friend to collaborate on a question together. This manual creation of a room by a user bypasses the matching service. To support this feature and other possible future extensions, we decided to prioritise flexibility in our choice of implementation.

Code Execution Service

Our Code Execution Service provides an isolated and secure environment for users to run and test their code, ensuring code runs safely without affecting other parts of the system.

API Endpoints

The Code Execution Service is designed as a RESTful API that communicates over HTTP. The following key API endpoints cover essential functions:

- 1) Get Supported Languages
 - a. Returns an array of supported languages (string [])
- 2) Execute Code
 - a. Input takes in 3 fields
 - i. code (string)
 - ii. language (string)
 - iii. input (string)
 - b. Outputs
 - i. output (string)
 - ii. isError (boolean)

Design Considerations

Isolation of Code Execution

Each code execution request is isolated by creating a unique directory, preventing interference between user sessions and reducing the risk of file conflicts or unauthorized file access across requests.

Resource Limiting

Memory and execution time are strictly limited to prevent excessive resource consumption or denial of service attacks. 'ulimit' is used to cap memory usage, while timeouts are enforced on both compilation and execution to guard against infinite loops or excessive memory use.

Language-Specific Handling

The design accommodates multiple languages (Python, JavaScript, and C++) with language-specific commands for compilation (for C++) and execution. This makes the system adaptable and extensible to other languages with similar configuration blocks.

Error Handling & User Feedback

Clear error handling differentiates between compilation errors, timeouts, and memory errors. Error messages are returned to the user, providing feedback on issues in code execution, which is useful for debugging.

Temporary File & Directory Cleanup

After code execution, temporary files and directories are removed, preventing file buildup that could otherwise consume disk space over time. This cleanup promotes efficient resource management and maintains system hygiene.

Security Considerations

Limiting system calls and containing execution in a controlled environment reduces potential security risks associated with running untrusted code, such as code injection or unauthorized access to system resources. Additionally, the unique directory for each execution session limits data leakage between users.

User Service

We decided to build our own User Service, rather than using the provided one. While both services use **JSON Web Tokens (JWT)** for authentication, our custom service includes additional features, such as a "forgot password" function, providing a more complete and flexible user experience.

API Endpoints

The User Service is designed as a RESTful API that communicates over HTTP. The following key API endpoints cover essential user functions:

- 1) Login: Authenticates the user and issue JWT for future requests.
 - a. Input:
 - i. Username
 - ii. Password
 - b. Output
 - i. JWT token
- 2) Register user: Registers a new user
 - a. Input:
 - i. Email
 - ii. Username
 - iii. Password
- 3) Forgot password: Sends a password reset link to the user's email address
 - a. Input
 - i. Email
- 4) Reset password: Resets the user's password using the token from the reset email
 - a. Input:
 - i. password
- 5) Change profile details: Update the authenticated user's profile
 - a. Input:
 - i. Username
 - ii. Password
 - b. Output

- i. Id
 - ii. Username
 - iii. Email
 - iv. Last login
- 6) Get user details: Retrieve the authenticated user's profile details
 - a. Input:
 - i. None (JWT token in header)
 - a. Output
 - i. Id
 - ii. Email
 - iii. Username
 - iv. Last login
- 7) Validate token: Validate the provided JWT to confirm if it's still valid
 - a. Input
 - i. None (JWT token in header)

User Database

We use MongoDB to manage user data. The user data schema includes the following fields:

- 1) email (string)
- 2) username (string)
- 3) hashedPassword (string)
- 4) lastLogin (date)
- 5) isAdmin (boolean)
- 6) resetPasswordToken (string)
- 7) resetPasswordTokenExpiry (date)

Design Considerations

Unit Testing

We implemented unit tests using Jest and Supertest to ensure the reliability and functionality of the API.

Password Security

All user passwords are hashed using the bcrypt library before being stored in the database, ensuring that sensitive information is protected.

Forgot Password Feature

The "forgot password" feature sends an email with a reset link and token to the user's registered email address (if it exists). By clicking the link, users are redirected to a secure page where they can reset their password.

JWT Token Security

JWT tokens are generated with an expiration time to enhance security. While JWTs provide the convenience of persistent authentication, allowing users to remain logged in without repeatedly entering credentials, the expiration time limits the window of risk. If a token is leaked or compromised, its validity is short-lived, reducing the chances of unauthorized access. This approach also forces periodic re-authentication, further strengthening security.

History Service

The History Service serves as a central repository for storing and retrieving users' past coding attempts. This feature is crucial for PeerPrep, as it enables users to review previous collaboration sessions, reflect on their progress, and identify areas for improvement. With this service, users can access a record of past coding sessions, providing a valuable reference for future learning and skill development.

API Endpoints

The core functionality of the History service is handled by a CRUD server built with Node.js. This server allows users to retrieve their past attempts (GET) and automatically records new session data (POST) at the end of each collaboration session. We do not support PUT and DELETE operations since the History service serves as a permanent log of all previous attempts.

History Database

The history data is stored in a dedicated NoSQL database (MongoDB). Although the schema is flexible, we require six essential fields:

- 1) `userIdOne` (string)
- 2) `userIdTwo` (string)
- 3) `textWritten` (string)
- 4) `programmingLanguage` (string)
- 5) `questionId` (number)
- 6) `datetime` (date)

When designing our schema, we considered whether to include only the `questionId` or to also store the `questionDetail`, `questionName`, and `questionDifficulty`. Including these

additional fields would reduce latency by eliminating the need for a second API call to the Question service to fetch these details and join the data. Ultimately, we decided not to include these fields to avoid potential data inconsistencies between the History and Question services. This inconsistency could occur if the History service stores outdated question data, while updates are made in the Question service.

Design Considerations

Event Sourcing

With event sourcing, instead of storing only the latest state, we would store all state-changing events. Actions like “session started,” “code run,” or “code change” would be logged as events.

Event sourcing would provide a comprehensive history of user actions, allow for easier rollbacks to previous states, and support detailed audit trails. However, this approach introduces complexity, as the current state must be reconstructed from a series of events, which can impact response times (an NFR). It would also result in significantly more data storage than a simple CRUD system, making it more challenging and costly to manage over time.

Command Query Responsibility Segregation (CQRS)

With CQRS, read and write operations are handled by separate services. This setup could be advantageous for scaling if the History service experienced a heavy imbalance between read and write operations.

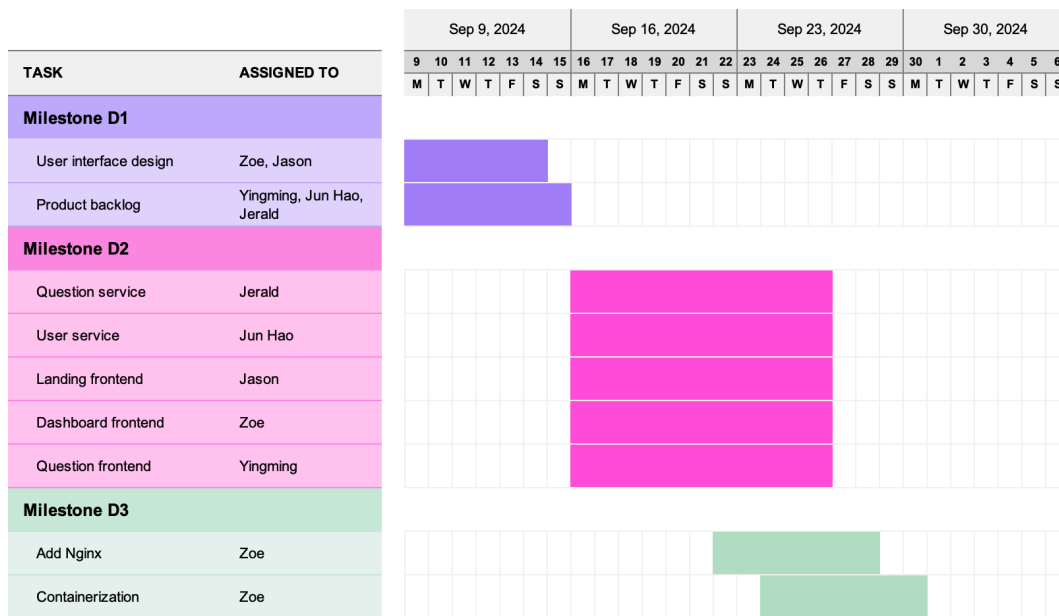
However, CQRS is typically more suitable for complex applications with significantly different read and write models. For a straightforward service like the History service, it would add unnecessary complexity to the architecture.

Ultimately, the CRUD and NoSQL model provides a simple, scalable, and effective way to manage history data, aligning well with our requirements.

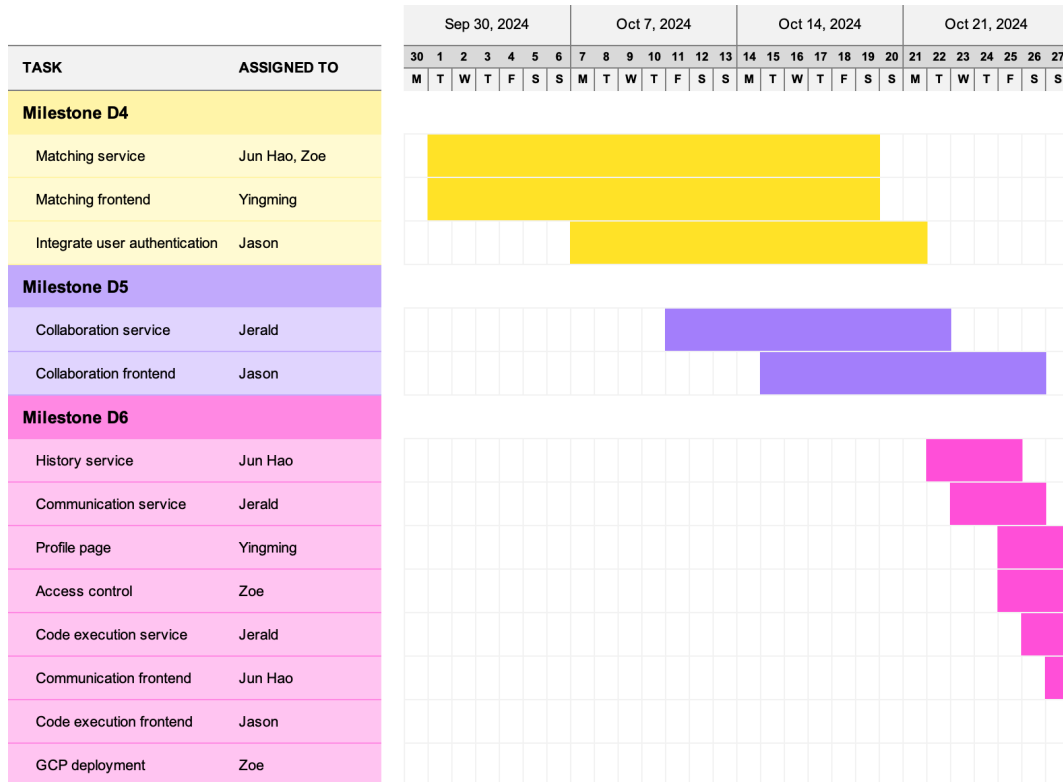
Project Plan

We used an **Agile** methodology with weekly sprints, aligning our progress with the milestone deadlines provided in the course. This iterative approach allowed for continuous development, integration, and testing, enabling us to adapt quickly based on regular feedback and ensure steady advancement toward key technical milestones, such as developing, integrating, and deploying microservices within the project timeline.

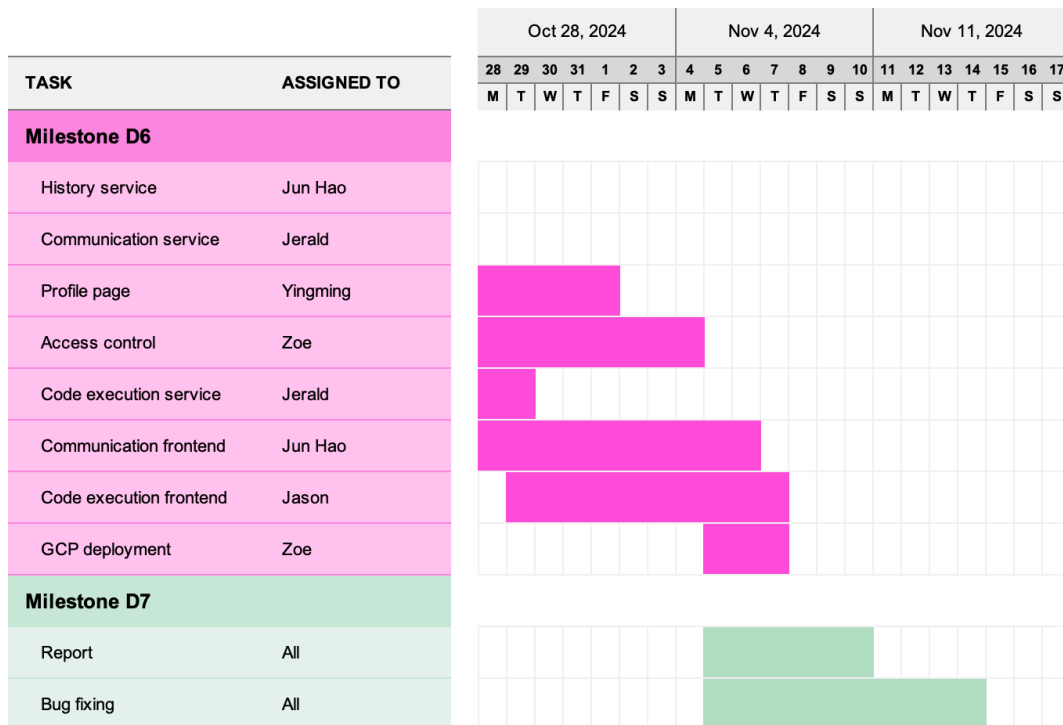
Gantt Chart



Weeks 1-4



Weeks 5-8



Weeks 9-11