NUS
National University
of Singapore

CS3219: Project Report

Group 48

No.	Full Name (as in EduRec)	Student Number (Axxxx)
1	Aaron Tay Kai Boon	A0252386Y
2	Alyssa Png Kai Wen	A0254928R
3	Zoebelle Pang	A0262469U
4	Tang Hao Wen	A0254535A
5	Ni Shenghan	A0234874W

Table of Contents

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 (as in Edu Rec)	Signature	Date
Aaron Tay Kai Boon		10/11/2024
Alyssa Png Kai Wen		10./11/2024
Zoebelle Pang		10/11/2024
Tang Hao Wen		10/11/2024
Ni Shenghan		10/11/2024

1. Individual Contributions

No	Full Name (as in Edu Rec)	Contributions *(write point wise for different components). Extend the table as needed.*
1	Aaron Tay Kai Boon	General: Frontend: - Created the initial UI without functions for the dashboard and profile - Implement React context for Axios api instances and user information, allowing for reusability of these contexts in application. - Set up TanStack react query to simplify querying of data. Backend: - Set up initial framework for Question service with Go, Matching and collab service with express, node. Question Service: - Set up CRUD request for question service - Integrate the question API calls with the frontend component. Matching Service: - Set up Redis Cloud - Handled the implementation of the matching queue and algorithm - Integrate the matching queue implementation with the web socket Collab Service: - Handled the backend logic for the collaboration service. Ensured that conflicts are being handled when users edit code simultaneously. Nice-To-Have: - Set up the GKE cluster and deployed the microservices on the cluster. - Implemented all yaml files for deployment pods, services and ingress. - Hosted the nginx ingress on a static public IP and domain. - Implement code compilation logic for the code editor using Judge API

	Alyssa Png Kai Wen	General: Frontend: - Set-Up React - Implemented SideBar UI - Implemented Dashboard “View Question” Button - Implemented syncing of topics in Question and Dashboard page - Help with Debugging Backend: - Handling authentication checks before accessing the API User Service - Implemented the Login, Email Verification, Reset Password UI - Connect the Frontend UI to the Backend Logic Question Service - Implemented the Question UI - Implemented Filtering Logic - Implement and Connect Add Question and Update question UI - Implemented Adding of Leetcode Question Logic Matching Service - Validating the chosen topics and their corresponding complexities before user chooses to queue - Integrate assign question logic to the matching-queue Collaboration Service - Implement CodeEditor UI + Logic - Implemented Language Option UI + Logic - Handle Disconnect logic on Websocket Nice-to-Have History: - Implement History Page UI - Handled logic of how users code is being stored in the backend and displayed in the frontend
3	Zoebelle Pang	General

		- Overall UI Designs and Implementation - Created a Docker container for the React application - Previously containerized the chat Service as an extra microservice, but later removed. User Service - Implemented registration UI - Implement input validations for email formatting and password requirements Question Service - Implemented admin and non-admin configurations - Implemented delete function (Frontend + database) - Implemented function to fetch questions from API and handling authentication. - Fix filter feature Matching Service - Establish Web Socket Collaboration Service - Designed and implemented UI for collaboration page and features. - Implemented the UI for displaying backend question data Nice to have: - Implemented real-time chat interface (Socket.io) - Implement Chat UI
4	Tang Hao Wen	User Service - Implement Logout backend logic Question Service - Implement updating and deleting of questions backend logic Matching Service - Implement frontend UI while matching Collaboration Service - Reconnection to Collaboration session

		- Disconnection and exiting from collaboration session Nice to have: - Integration of Gemini into collaboration session - Implement auto inclusion from editor
5	Ni Shenghan	Backend: - Set up MongoDB on Atlas User Service: - Add email verification - Add reset password - add CORS Frontend - Help in fixing bugs - Question Service - Add create question method Matching Service - Attempted on monaco editor + yjs websocket Collaboration Service - Integrate basic features such as fetching of question Nice-to-have: - Code styling of the editor Amazon Web Service - Attempted to use EC2 instance and load balancer to deploy microservices - Attempted to use cloudfront to deploy frontend - Attempted to use EKS and elastic beanstalk - Store containers onto ECR repository

1. Introduction

1.1 Background

Technical interviews are now part of the tech job process. These interviews assess candidates' coding skills, problem-solving abilities, and computer science knowledge. One of the largest obstacles for most people taking this interview is self-doubt, rather than the lack of knowledge. The severe pressure and solitude of these exams can make even skilled people struggle.

LeetCode and HackerRank provide plenty of practice, but they may not boost candidates' confidence. These platforms allow users to solve problems, but they lack supervision and teamwork, which can boost confidence and problem-solving skills in real life. Without mentorship or collaborative input, candidates can feel overwhelmed, unsure of their approach, or stuck in habits that may not prepare them for the ever-changing technical interview field.

This gap indicates a need for a more engaging and helpful approach. Real-time comments, assistance, and insights from peers or mentors can help candidates improve their abilities and develop confidence for technical interviews. This environment simulates an interview, allowing users to solve problems together. It improves their understanding and comfort with communicating their thoughts, which are crucial for technical assessments.

1.2 Purpose

PeerPrep is a collaborative area where technical interview candidates can gain skills and confidence. PeerPrep makes interview preparation a community experience. PeerPrep simulates tech roles' dynamic, team-oriented environments by linking users with peers who share their challenge level and themes for real-time coding issue solving.

PeerPrep simulates real-world IT environments with live chat and a shared code editor. This method improves technical skills and communication by encouraging teamwork in coding and problem-solving. PeerPrep makes interview prep more collaborative and less isolated, empowering people to tackle technical interviews with confidence and readiness, connecting their practice with tech workforce abilities.

2. Project scope (Product backlog)

The Functional Requirements (FR) in this table are structured into modules (M1, M2, M3, and M4), each addressing a distinct functionality in the system. Each module contains specific functional requirements (F1.x, F2.x, etc.), representing key features or actions users should be able to perform.

2.1 Functional Requirements (+ Fulfillment)

Functional Requirements		**Priority**	**Planned sprint/ iteration**	**PR link**
M1: User Service				
F1.1	User must be able to register for a new account			
F1.1.1	User should be able to create account using their school email through a form	High	D2	link
F1.1.4	Users should receive an error message if they try to register with an existing email.	High	D2	link
F1.2	User must be able to login			
F1.2.1	Users should be able to log in using their registered email and password.	High	D2	link
F1.2.2	Users should be able to stay logged in after page refreshes	High	D2	link
F1.3	User should be able to reset account password when user forgets			
F1.4	User should be able to view and edit profile information			
F1.4.1	Users should be able to view number of questions and difficulty completed	Mid	D2	link
F1.4.2	Users should be able to view and edit personal information, email, and username	Mid	D2	link
F1.4.3	Users should be able to change the password of their account.	High	D2	link
F1.4.4	Users should be able to delete their account	Mid	D2	
F1.5	Users should be able to log out of their account			
F1.5.1	Users should stay logged out after page refreshes	High	D2	link

M2: Matching Service				
F2.1	Users should be able to find matching candidate if there are more than two users online in the same difficulty level and topic			
F2.1.1	Users are only matchable if there is another candidate that has selected the same question topic and difficulty.	High	D4	link
F2.1.2	Two matched users who have selected the same question difficulty will be randomly given a question from the repository with the corresponding difficulty.	High	D4	link
F2.2	Users should be notified if no match is found			
F2.2.1	Users should be notified and exit the matching sequence after 60 seconds of not finding a match.	High	D4	link
F2.2.2	Users should be given an option to rejoin the queue with the same topic and difficulty, rejoin the queue with a different topic and difficulty, or exit the matching process after 60 seconds of not being matched.	High	D4	link
F2.3	Application should have visual indication when users are in queue and the status of the queue.			
F2.3.1	Users should be prompted that they are in queue when waiting for a match.	High	D4	link
F2.3.2	A timer should be shown to users to indicate how long they have been queueing.	High	D4	link
M3: Question Service				
F3.1	Users should be able to access the assigned questions once the session starts			
F3.1.1	Both users matched should be able to view a question of the selected difficulty when the collaboration session starts within 10 seconds.	High	D5	link
F3.1.2	Each user should be able to view the supporting images within their assigned question if any.	Mid	D5	link
F3.2	Users should be able to view a repository of questions in a table format within a page.			
F3.2.1	Application should display questions with the corresponding difficulty, topic and description in a table to the user.	High	D2	link
F3.2.2	Users should be able to select from the pool of available topics to enter the queue.	High	D2	link

F3.3	Admin Users should be able to have access to admin features			
F3.3.1	Admin users can add unique new questions to the database with fields for title, description, categories, complexity, and link.	High	D2	link
F3.3.2	Admin users can modify existing questions, including updating the title, description, categories, complexity, and link.	High	D2	link
F3.3.3	Admin users can delete existing questions from the database	High	D2	link
M4: Collaboration Services				
F4.1	Users and their matched partner should be to code on a live editor together.			
F4.1.1	Users and the partner should be able to have an editor where one person can code while the other can view changes.	High	D5	link
F4.1.2	Users should be able to reconnect to the same code editor sessions when switching pages.	High	D5	link
F4.1.3	Users should be able to view changes to code editor upon reconnection	High	D5	link
F4.1.4	Users should be able to edit on the code editor simultaneously, with changes of both users being recorded.	High	D5	link
F4.3	Users should be able to exit the session			
F4.3.1	Users should be able to manually leave the session when needed.	Mid	D5	link

2.2 Non-Functional Requirements (+ Fulfillment)

The Non-Functional Requirements (NFR) in this table are structured into categories (N1, N2, N3, etc.), each addressing a different quality aspect of the system. Each category contains specific non-functional requirements (N1.x, N2.x, etc.), which define the desired attributes or performance targets for the application, such as security, scalability, and performance.

Non-Functional Requirements		**Priority**	**Planned sprint/ iteration**	**PR link**
N1: Security				
N1.1	User's data should be secure			
N1.1.1	User's password should be hashed before storing in database	High	D2	link
N2: Scalability				

N2.2	The system should use auto-scaling and load balancing on managed cloud services (e.g., AWS ECS/EKS, GCP GKE) to dynamically adjust resources and maintain performance.	Low	D6	link
N3: Performance				
N3.1	Web-page should be responsive with short initial load times			
N3.1.1	Web-page should load static data within 5 seconds and display it to the user.	High	D2	-
N3.1.2	Users should be able to navigate to other pages and use interactive features whilst data is still being fetched from the server.	High	D2	link
N3.2: Web-page should have low latency during collaboration				
N3.2.1	The shared code editor used for collaboration must have a latency of less than 3 seconds between when a person writes a new line of code and when their assigned partner sees the update.	High	D4	link
N3.2.2	The application should implement a real-time text-based chat service within the collaborative space, allowing participants to communicate within 500ms	Low	D6	link
N4: Usability				
N4.1	Users should be able to access and use all features on the web page with no more than 5 button clicks and minimal scrolling.	Mid	D4	-
N4.2	All features of the website must be accessible via the navigation bar, ensuring that users can easily locate and interact with them without needing to search elsewhere.	Mid	D4	-
N7: Availability				
N7.2	Application should work on different browsers.	High	D3	-

2.3 Selected Nice-to-haves

The Nice-to-Have Functional Requirements (FR) in this table are organized into categories (Nice-to-Have 1, Nice-to-Have 2, etc.), each focusing on additional, value-added features for the system. Each category includes specific functional requirements (F1, F2, etc.)

2.3.1 Functional Requirements

Functional Requirements	**Priority**	**Planned sprint/ iteration**	**PR link**

Nice-to-Have 1: Communication				
HF1:	Users should be able to send and receive messages in real-time.			
HF1.1	Users should be able to send and receive messages in real-time.	High	D5/6	link
HF1.2	Users should be able to see the name of the person sending each message.	High	D6	link
HF2:	Users should receive notifications about their connection status with other users.			
HF2.1	Users should receive a notification when another user connects or disconnects.	Mid	D6	link
Nice-to-Have 2: History				
HF3	Users should have access to a summary table of all their attempted questions			
HF3.1	Users will be able to see a table summary of all attempted questions, including details such as topics covered, complexity levels.	High	D6	link
HF3.2	Users should be able to filter and search through their submission history based on Topic and Complexity	Mid	D6	link
HF4**:**	Pop-up view to display detailed information about a specific coding attempt			
HF4.1	Users should be able to view the code snippets that they submitted in their past coding attempts	High	D6	link
HF4.2	Users should be able to view the date that the attempt was submitted	High	D6	link
Nice-to-Have 3: Code execution				
HF5	Users will still be able to see the latest code updates			
HF5.1	Users will still be able to see the latest code updates even when they navigate away or reconnect if session is active.	High	D6	link
HF6	Allow user to compile and run code.			
HF6.1	Users should be able to submit their code for an assigned coding question in the	High	D6	link

	programming language of their choice at the end of their session			
Nice-to-Have 4: Code Formatting				
HF7	User should be able to have syntax highlighting			
HF7.1	Users should be able to enable syntax highlighting for multiple programming languages, such as Python, JavaScript, HTML, CSS, etc.	High	D6	link
HF7.2	Users should be able to switch between different languages, and the syntax highlighting should update accordingly.	High	D6	link
HF7.3	Syntax errors should be highlighted within the editor to assist users in identifying and fixing code issues.	High	D6	link
HF8	The code editor should have auto-formatting for users			
HF8.1	Users should have access to recommended formatting, including indentation, line spacing, and font styling.	Mid	D6	link
Nice-to-Have 6: Incorporate generative AI to assist during the preparation.				
HF9	Code Assistance & Debugging			
HF9.1	Users should be able to input question, error messages or problematic code segments to receive suggested fixes or debug steps	Mid	D6	link
HF9.2	Users should be able to receive real-time code explanations for their code snippets, detailing functionality, logic, and potential issues.	Mid	D6	link
Nice-to-Have 9: Kubernetes - The application should have an API gateway that redirects requests to the relevant microservices.				
HF10	API Gateway & Load Balancing			
HF 10.1	The application should have an API gateway that redirects requests to the relevant microservices.	High	D6	link
HF 10.2	The API gateway should support load balancing, distributing incoming requests evenly across instances of microservices to ensure efficient resource utilization and minimize response times.	Mid	D6	link

2.3.2 Non-Functional Requirements

The Nice-to-Have Non-Functional Requirements (NFR) in this table are organized into categories (Nice-to-Have 1, Nice-to-Have 2, etc.), each focusing on additional, value-added features for the system. Each category includes specific functional requirements (F1, F2, etc.), outlining enhancements such as communication, history, and code execution capabilities that improve the user experience.

Non-Functional Requirements		**Priority**	**Planned sprint/ iteration**	**PR link**
Nice-to-Have 1: Communication				
NHF1	Real-time Messaging			
NHF 1.1	The messaging system should have a latency of less than 200 milliseconds to ensure messages are delivered in real-time without noticeable delay.	High	D6	link
NHF 1.2	System notifications and user chat messages should be visually distinct to ensure easy differentiation	Mid	D6	link
Nice-to-Have 2: History				
NF2	Efficient history retrieval			
NHF 2.1	The history retrieval system should allow users to load summaries and search results within 2 seconds for a smooth experience	High	D6	link
NHF 2.2	The summary table and detailed views should follow a consistent data format for ease of understanding, using standardized labels for topics, complexity levels	Mid	D6	link
Nice-to-Have 6: Incorporate generative AI to assist during the preparation.				
NHF3	Generative AI Assistance			
NHF 3.1	Generative AI responses should be processed within 5 seconds for detailed explanations, ensuring timely assistance	High	D6	link

NHF 3.2	Code snippets entered by users for AI assistance can be automatically included for prompt	High	D6	link
Nice-to-Have 3: Code execution				
NHF4	Reliable Code Execution			
NHF 4.1	The code execution environment should provide a 99.9% uptime, with minimal downtime to ensure consistent access for users	High	D6	link
NHF 4.2	The code execution system should support at least 5 programming languages	Mid	D6	link
NHF 4.3	Code should be executed in a sandboxed environment to prevent unauthorized access to system resources and ensure user code runs securely	High	D6	link
Nice-to-Have 4: Code Formatting				
NHF5	Consistent Code Formatting			
NHF 5.1	Syntax highlighting and auto-formatting should have a latency of less than 100 milliseconds to provide immediate feedback as users type	Mid	D6	link
NHF 5.2	Formatting should be consistent across languages, with standardized indentation, spacing, and syntax error indications	Mid	D6	link
Nice-to-Have 9: The application should have an API gateway that redirects requests to the relevant microservices.				
NHF7	API gateway that serves as a proxy to backend services			
NHF 7.1	API gateway should provide access to all backend services via a single domain	High	D6	link
NHF 7.2	API gateway should route incoming requests to the appropriate microservice based on predefined routes	High	D6	link
NHF 7.3	API gateway should have some form of load balance to distribute request across multiple instances of a microservice for scalability and reliability	Mid	D6	link

2.4 Dropped Features

Due to time constraints, we dropped certain FRs and NFRs. While these features provide additional value, they are not essential to the primary functionality of our platform.

Functional Requirements		**Priority**	**Planned sprint/iteration**
M1: User Service			
DF1.1	User must be able to register for a new account		
DF1.1.2	Users should receive a confirmation email with an activation link after registration.	Low	After D7
DF1.1.3	Users should be able to activate their account by clicking the link in the confirmation email.	Low	After D7
DF1.2	User must be able to login		
DF1.3	User should be able to reset account password when user forgets		
DF1.3.1	Users should be able to request a password reset link via their email.	Low	After D7
DF1.3.2	Users should receive a confirmation email after successfully resetting their password.	Low	After D7
F1.4	User should be able to view and edit profile information		
F1.5	Users should be able to log out of their account		
DF1.5.2	Users should automatically be logged out after 30 mins of inactivity	Low	After D7
M2:	Matching Service		
DF2.1	Users should be able to find matching candidate if there are more than two users online in the same difficulty level and topic		
DF2.1.3	If there are more than two matchable candidates in the queue, priority should be given to those that entered the queue first.	Low	After D7
DF2.2	Users should be notified if no match is found		
DF2.3	Application should have visual indication when users are in queue and the status of the queue.		
DF2.3.3	Users should be allowed to view the numbers of users in queue for each topic.	Low	After D7
DF2.4	Users should be able to use the app while waiting for a match.	Low	After D7
M3: Question Service			
DF3.1	Users should be able to access the assigned questions once the session starts		

F3.1.4	Users should be able to switch to a different question if needed.	Low	D5
DF3.2	Users should be able to view a repository of questions in a table format within a page.		
DF3.3	Admin Users should be able to have access to admin features		
	M4: Collaboration Services		
DF4.1	Users and their matched partner should be to code on a live editor together.		
DF4.3	Users should be able to exit the session		
DF4.3.1	Users should be automatically exited from the collaborative session once the session timer runs out.	Low	D5

Non-Functional Requirements		**Priority**	**Planned sprint/ iteration**
N1: Security			
DN1.1	User's data should be secure		
DN1.1.2	User's account details should be protected by an additional layer of security beyond just username and password.	Low	D6
DN1.1.3	User's communication should be encrypted in transit to protect user data and privacy.	Low	D6
N2: Scalability			
DN2.1	The application should support up to 10,000 concurrent users without degradation in performance	Low	D6
DN.2.3	The deployment should be managed using container orchestration tools like Kubernetes to manage load balancing and traffic distribution	Low	D6
N3: Performance:			
DN3.1	Web-page should be responsive with short initial load times		
DN3.1.3	The system must minimize load times for users regardless of their geographical location by optimizing the delivery of static assets like stylesheets, scripts, and media files.	Low	D6
N3.2	Web-page should have low latency during collaboration		
DN4: Usability DN7: Availability			
DN7.1	Users should be able to access the website 24/7, without significant downtime or disruptions	Low	D3

3. Technology Stack

Component	Framework, Technologies, Tools
Frontend	React Chakura UI - UI component Library Tanstack - Data Table component Library CodeMirror - Code editor component Library
Backend: Microservice	Go: - Question Service Express.js: - User Service - Matching Service - Collaboration Service
Authentication	JSON Web Tokens (JWT)
Collaboration	Websocket
API Gateway	NGINX Ingress Kubernetes
Containerisation	Docker
Cloud Service	Google Cloud Platform (GCP) - Cloud Run to host front end Google Kubernetes Engine (GKE)
Project Management	Jira Board Github
Database	MongoDB hosted on Atlas Cloud
Queue	BullQ, Redis

4. Project Plan - Software Development Life-Cycle (SDLC)

4.1 Scrum Agile Process

Scrum is an agile project management framework that helps to improve workflow and productivity in a team, especially for complex projects.Each Sprint is guided by a specific Sprint Goal that the team strives to achieve by the end of the iteration. The tasks required to complete this goal are distributed among team members, who work

together to meet the Sprint Goal. By breaking down the project into smaller iterations, the Scrum process allows teams to adapt to changing requirements and promotes continuous improvement.

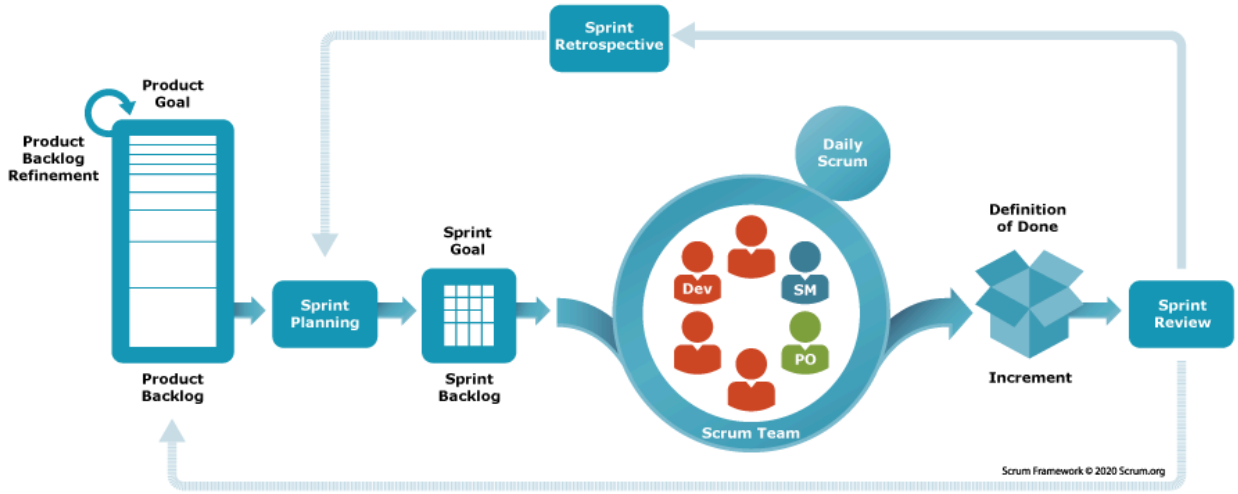


Image taken by https://www.scrum.org/resources/what-scrum-module

A Sprint normally concludes with a Sprint Retrospective and Review, where the team demonstrates what they have worked on during the Sprint. The team will also reflect on what went well and what could be improved on for the next Sprint.

At the start of every sprint, a key ceremony is done called Scrum is Sprint Planning. This involves the team meeting together at the beginning of each Sprint to discuss the Sprint's goal and pick tasks from the Product Backlog to add to the Sprint. It allows the whole team to visualize the workload needed for the Sprint and ensure equal deliverables for each member of the team.

4.2 Project Management Details

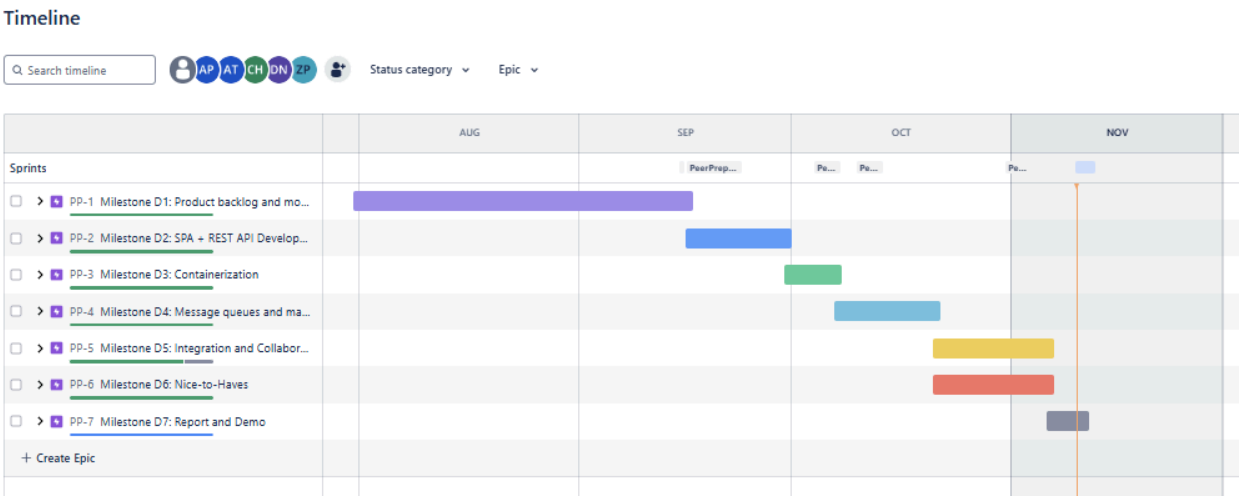
Our team worked in weekly Sprints, using the JIRA Board to organize tasks and manage sprint activities. We held regular meetings to plan each Sprint and align tasks with our Sprint Goal, typically connected to a milestone or epic. Adopting the Scrum framework helped enhance collaboration and communication within our team. Given everyone's additional commitments, Scrum enabled us to evaluate each other's workloads and fostered a clear focus on completing each Sprint's tasks, ensuring progress toward both functional and non-functional requirements. This approach made

the project feel less overwhelming and improved transparency in tracking each team member's productivity and challenges.

Our project workflow began with a Sprint Planning Meeting, where we assigned JIRA tickets to align with the Sprint Goal. Each ticket corresponded to a GitHub branch, with details outlining task requirements and labeled "[Backend]" or "[Frontend]." Tickets were tracked as "To Do," "In Progress," or "Done," giving everyone visibility into what others were working on. After pushing their branch, team members would create a PR for review and testing. Once reviewed, branches were either merged into the main branch or sent back for further revisions, refactoring, or bug fixing.

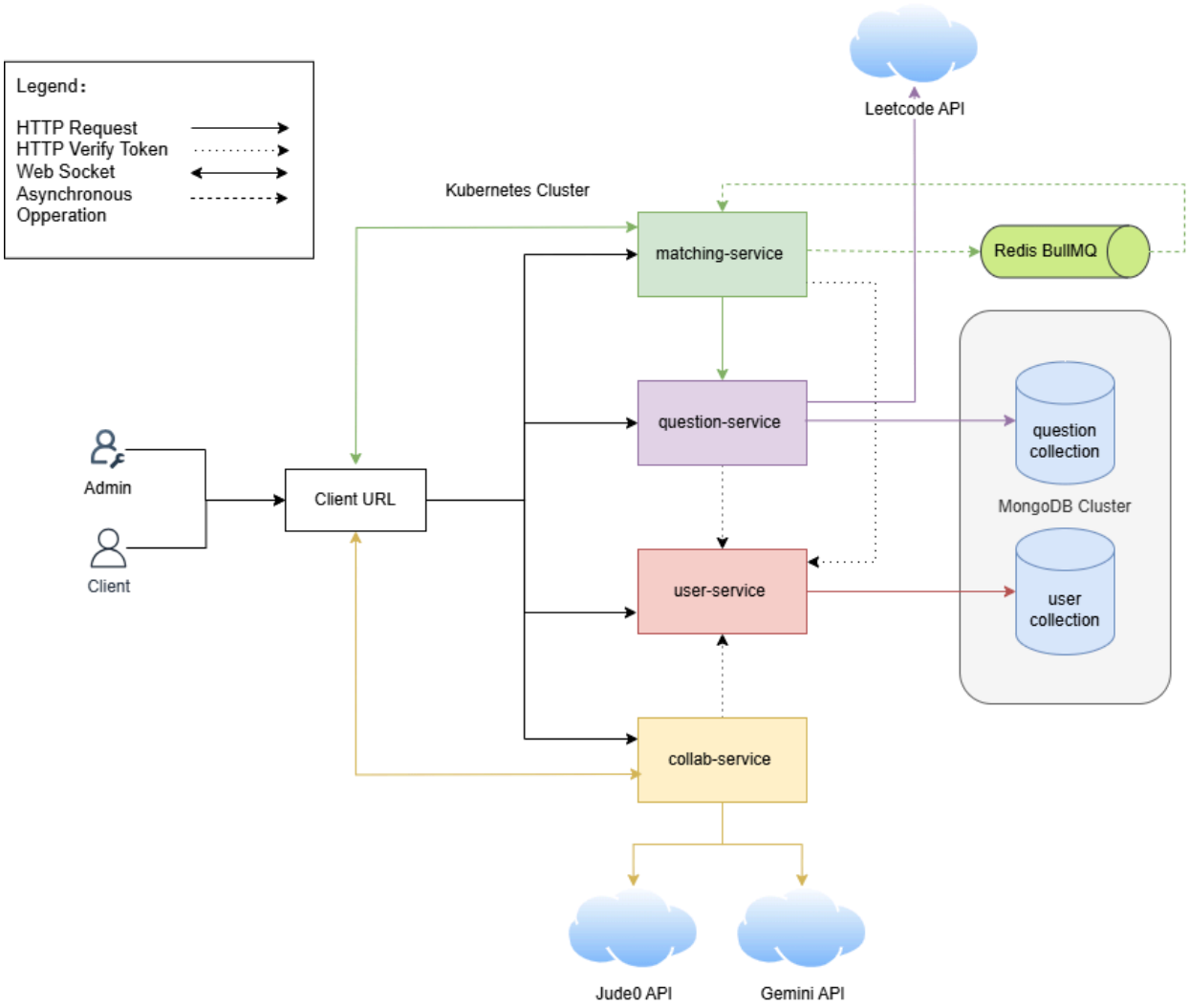
4.3 Gantt Chart

This is the overall Gantt chart for our project. We organized our JIRA Epics based on the Project Milestones. The green or blue bars indicate the percentage of each Epic that is done, in progress, or yet to be started.



5. Architecture Design

5.1 Architecture Overview



5.2 Architecture Considerations

5.2.1 Microservice VS Monolith

Our team adopted a microservice architecture instead of a monolith architecture due to the following reasons.

1. Services can be developed and deployed independently.
2. Services can be scaled independently.
3. Services are loosely coupled from one another

While our team recognises that developing a monolith architecture will allow for simplicity in development and deployment, we believe that the limits of scalability and having future slowdown in development due to a tightly coupled code base outweighed its benefits.

A key consideration in the development of our application was the ability to work independently in parallel. Microservice allowed our team each member's preferred technological stack as the development of each service is independent of one another. This allowed our team to have faster development cycles and each member can work on different services with minimal conflicts.

Our team developed Peerprep with the requirement of being able to handle a larger number of users in the future and have prioritized scalability in the development of our application. By adopting a microservice architecture, our team was able to make use of orchestration tools to scale our services horizontally depending on which service requires more resources at the time.

One consideration of our team was that we wanted Peerprep to be a project that we can continuously improve on. A loosely coupled codebase meant that we could easily add-on features to existing services due to the modularity of each service. This also allows the extensibility of more services.

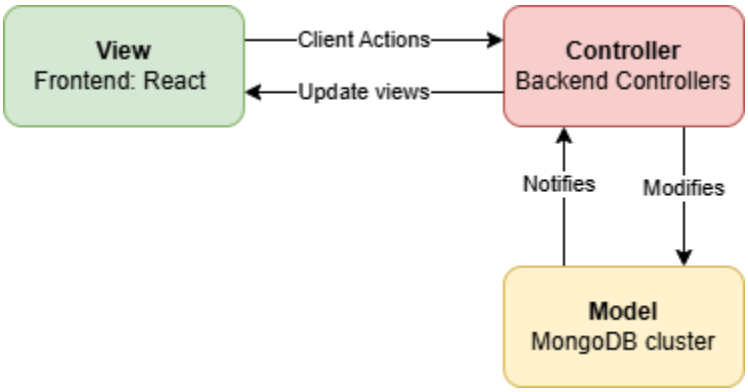
5.2.2 Shared Database per Service

We adopted a collection-per-service pattern, where each service only modifies its own set of collections.

While we recognise a microservice architecture typically adopts a database per service pattern due to its loose coupling and more granular control of scaling the database as required by a service. However, Peerprep is still small in scale and is still in the development phase. As such, the ease of managing a single database and the lower cost in having only a single database was more beneficial for us.

Moreover, due to development being limited to a small team size, it is easy for us to enforce data access restrictions at a service level. By ensuring that each service only modifies its only collection, we also managed to preserve logical level data independence as changes to a service collection will not affect another service. This makes it easier to switch to a database per service pattern in the future.

5.2.3 Model View Controller (MVC)

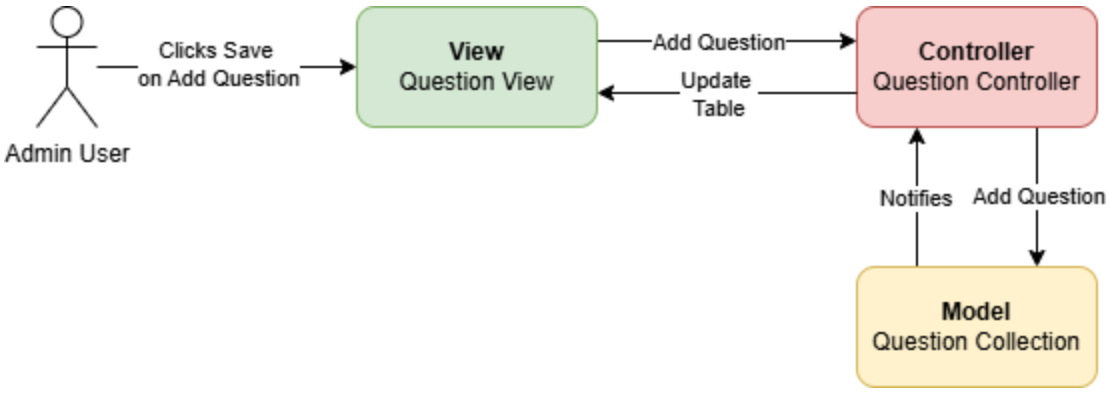


In our project, the application is structured according to the MVC (Model-View-Controller) pattern, where:

1. Model: The MongoDB Cluster serves as the data storage, holding collections such as questions, users, and other resources.
2. View: The React components provide the user interface, presenting interactive views for users to manage and interact with the application.
3. Controller: The backend, implemented as controllers, handles incoming requests, manages application logic, and communicates between the Model and the View.

5.2.3.1 Example: Question Service MVC

Consider the situation where an admin user adds a new question to the question list. When the admin clicks the "Save" button in the Question View, a **QuestionRequest** is sent as a POST request to the backend question controller. The controller processes this request by generating a unique ID for the question and validating if the question inputs are valid. Upon a valid request, the controller then updates the MongoDB question collection. The controller then responds with an HTTP status, either confirming the successful addition of the question (http.**statusOK**) or throwing an error. Based on this response, the UI updates with a notification, displaying either a success toast or an error message.



5.2.3.2 Benefits of MVC

Our team has chosen the MVC architecture for several key reasons:

1. **Clear separation of concerns.** Each component serves its own functionalities and makes code easier to manage and develop. This clear separation of logic among components allows for changes to be made to components to less likely affect other components. This makes it easier for the application to adapt to new requirements over time.
2. **Parallel development**. This benefited our iterative development cycle as each team member is able to work on individual components separately for the same service in a milestone.This separation speeds up the development process, enabling us to complete the application faster than with other architectural patterns.

5.2.4 REST API

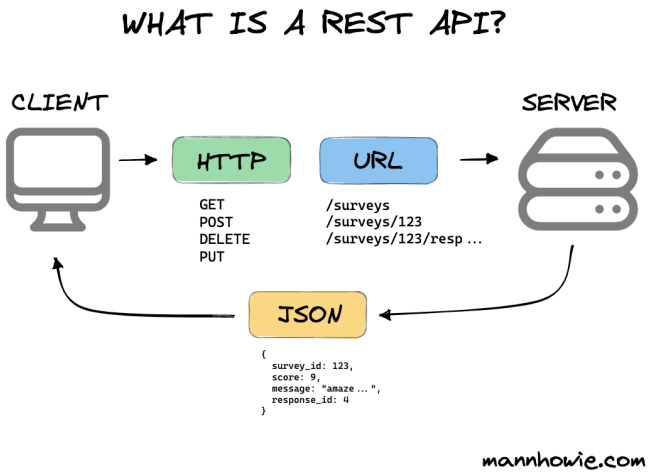


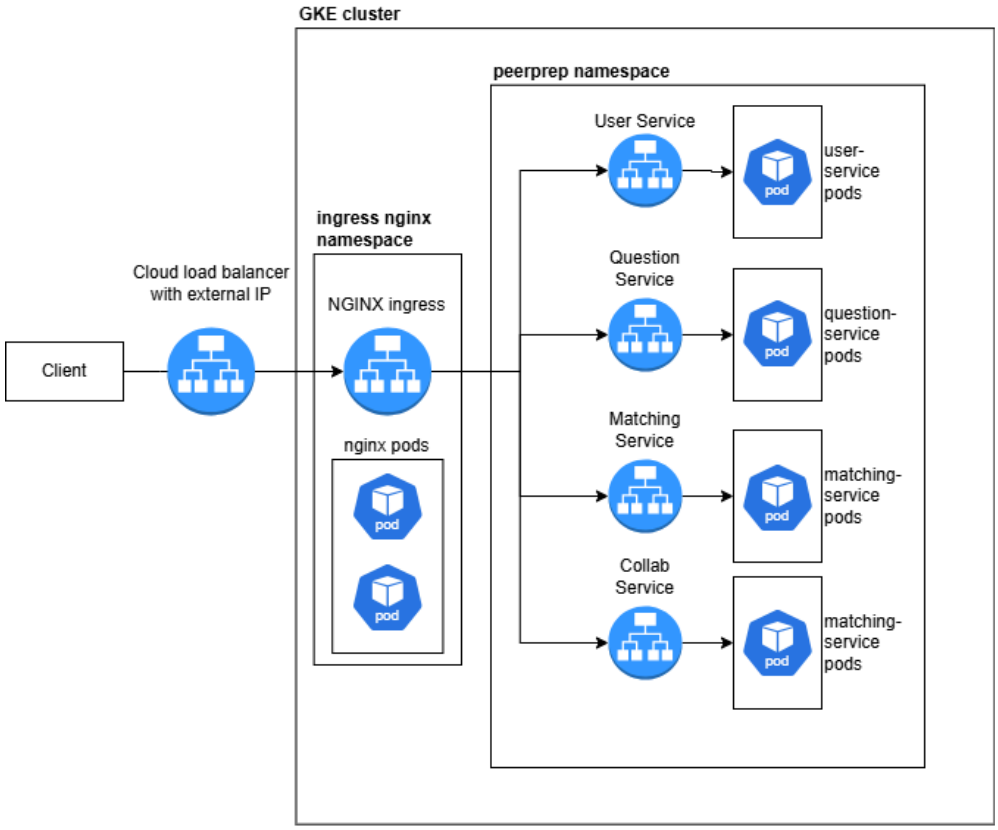
Image taken from https://mannhowie.com/rest-api

We chose to implement a REST API for our project to facilitate data access between the server (in our case, the database) and the client (our application). REST APIs allow our services to operate more independently, promoting scalability, usability, accessibility, and flexibility which are all highly beneficial for our project.

One of the main advantages of REST APIs is their loosely coupled structure. This enables easier updates and maintenance as each service can evolve independently, making it suitable for our microservice architecture. Additionally, the RESTful approach offers standardized data access, making our application accessible to various clients and adaptable to future integrations.

However, REST APIs come with some drawbacks. The stateless nature of REST can lead to decreased network performance, as each request must include all necessary data, resulting in potentially repetitive data transmission over a series of requests. Although this may affect performance, our application requirements align well with a stateless architecture, given the need for scalable and distributed services.

5.2.5 NGINX ingress (API gateway)



Our team decided to adopt an API gateway architecture as part of our microservice architecture due to the following reasons.

1. Simplifying the architecture. Using an ingress serves as a single point of entry to our application, while keeping the rest of our services private within the cluster. As such, we only require the public IP address of the external load balancer instead of having a public IP address for each of our individual services.
2. Ease of management of services. NGINX Ingress simplifies routing by allowing configuration of routes directly with Kubernetes Ingress resources. This allows us to route our request to the different services via the same domain with slightly different paths. This also allows for security measures to be implemented easily as all requests to the backend goes through a single endpoint.
3. Scaling. NGINX Ingress can do load balancing across multiple service instances, allowing large volumes of traffic to be handled in a balanced way.

5.3 Deployment Choice

5.3.1 Google Kubernetes Engine (GKE)

Our team decided to deploy our application on the GKE autopilot cluster due to the following reasons.

1. GKE clusters are based upon Kubernetes open-source cluster management system.
2. GKE offers automated management, scaling that is optimized for Kubernetes best practices and manages security of the underlying infrastructure.

A key consideration for using GKE was having access to kubernetes as an orchestration tool. This allowed our team to leverage on the advantages that kubernetes provides, such as horizontal pod autoscaling and the use of kubernetes load balancers. This is essential for our requirements of our application to run smoothly and handle more users in the future. One of the key non-functional requirements for the application is ensuring continuous availability, 24/7. Leveraging on Kubernetes addresses this, as Kubernetes automatically replaces failed pods with a health replica, ensuring the desired number of pod replicas is always maintained. This minimizes downtime and ensures application availability to users at all times.

GKE helped simplify cluster management and improve operational efficiency. GKE autopilot clustered manages much of the operational overhead, allowing our team to focus on developing and deploying your applications. GKE autopilot cluster provides

automatic scaling of nodes based on the workload demand. This is also more cost efficient for our team as charges are based on the resource requests of the pods.

6. Messaging Patterns

6.1 Pub-Sub Pattern

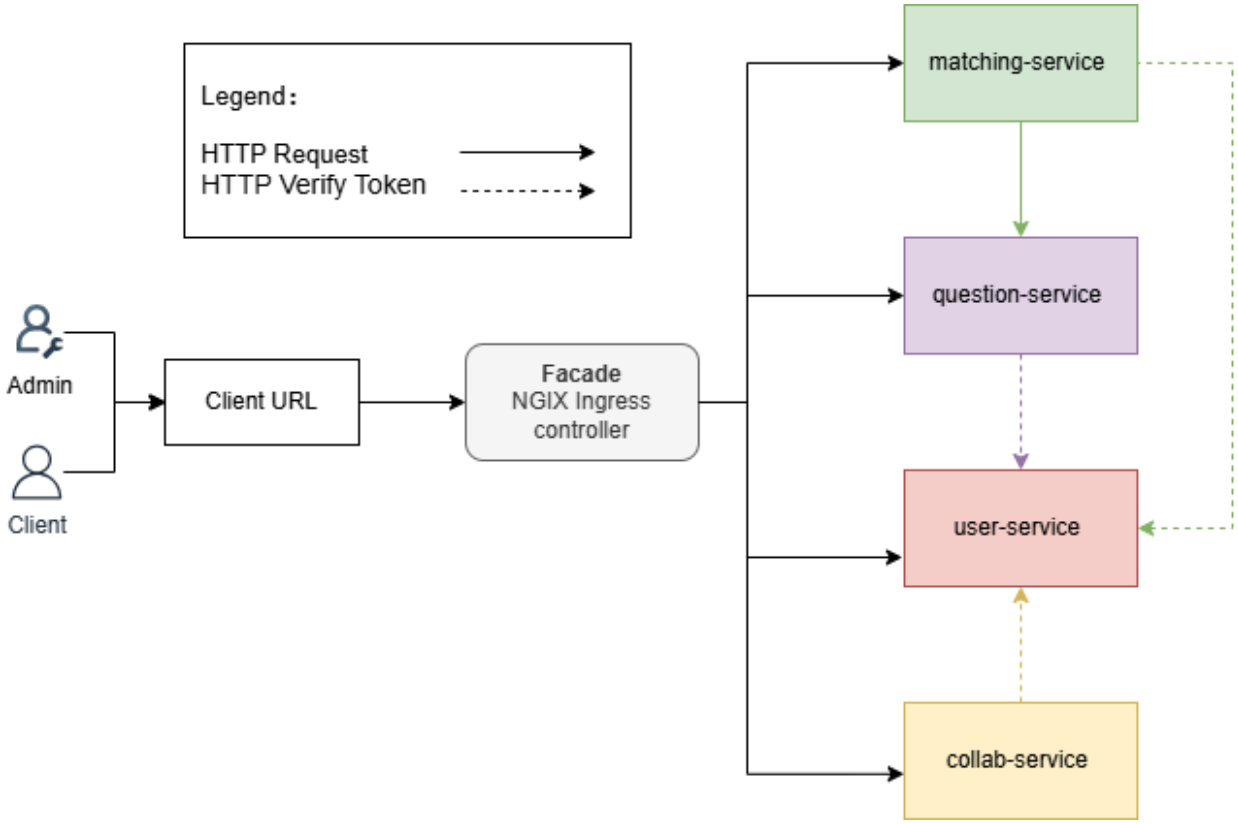
Our team employed the pub-sub pattern within our matching service via Redis with the help of the Bull MQ library. Refer to image in Section 7.3.1. We felt that the pub-sub pattern was most suitable for our application due to the following reasons.

1. **Decoupling of the components**. This allowed for asynchronous communication, since publishers can publish events to the queue without waiting for a response and waiting for subscribers to be listening. This is suitable for our application use case, as it delivers real-time experience to end users. Users will be able to join the queue and be notified upon joining of the queue. Users will be blocked to wait for a response on the result of their match before being allowed to continue work on the client.
2. **Ability to scale easily**. The number of subscribers listening to the same queue can be increased easily. This allows for load balancing and distribution of work, as different subscribers can handle different parts of the workload. This allows for our matching service to scale and support a larger user base while also ensuring the responsiveness of application by keeping matching times short.

7. Design Patterns

7.1 Facade Pattern

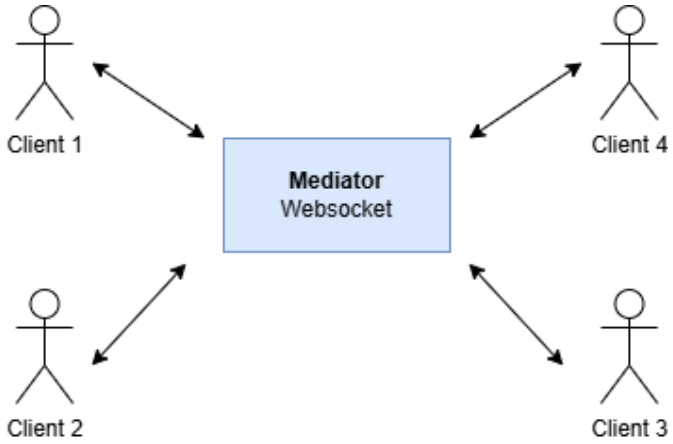
Our team implemented an API Gateway, which acts similarly to the Façade pattern but operates at the architectural level. Like a Façade, the API Gateway provides a unified interface, simplifying how our frontend interacts with various backend services. Rather than having the frontend communicate with individual services directly, the API Gateway abstracts these interactions, centralizing the logic and coordination between services.



This added layer of abstraction brings several advantages:

1. The API Gateway reduces complexity for the frontend, enabling it to make requests through a single endpoint rather than managing multiple service interactions.
2. Since the API Gateway manages inter-service communication, any changes in backend services are isolated from the frontend, making it easier to update or scale specific services without affecting the user interface.

7.2 Mediator Pattern



In our real-time collaboration service, the WebSocket server functions as a mediator for the clients. The Mediator Pattern is defined as an object that centralizes the interaction between a set of objects, allowing their communication to be managed without them directly referencing each other.

For example, in our chat service, when a client sends a message, the WebSocket server acts as the mediator by determining which matched client should receive that message, often using a room ID. The server then forwards the message to the relevant client. In this setup, the clients do not communicate directly with each other; instead, they rely on the server (the mediator) to handle and distribute messages.

This pattern mirrors the Mediator Pattern, where the WebSocket server serves as a central communication hub. Clients interact with the server rather than with each other, allowing us to maintain better control over the flow of messages and interactions. This also allows us to scale the system more easily, since adding new clients or modifying message-handling logic can be done in the server without impacting the clients themselves. In this way, WebSocket can be seen as applying the **Mediator Pattern** at the architectural or system level.

8. Implementation Information

8.1 User Service

8.1.2 JWT Token

8.1.2.1 Authentication

When a user login, it will send a request to retrieve the access tokens which is valid for 1 day, from the storage and verify it with JWT_SECRET from the env file in the user

service. Upon successful validation, the user will be redirected to the question page. In the event if it failed due to a wrong email or password, it will throw *401 error*. If the user does not exist in the database, it will throw a *500 internal server error*.

8.1.3.1 Authorisation

The token is then utilized by the frontend to interact with the REST APIs and websockets on the backend. This is achieved by making a request to the /auth/verify-token endpoint in the user service. If the token is valid, the server responds with a 200 status code; if invalid, it returns a 401 status and redirects the user to the login page. This process ensures that unauthorized users cannot access, modify, or delete data in the database.

8.2 Question Service

8.2.1 Role-based access control (RBAC)

Users for Peerprep are categorized into Admin and non-Admin. Admin users can update any user's privilege. The server verifies the user associated with the JWT token is an admin user from *isAdmin* in the body response and allows the update of requested user's data. If the authorisation check fails, a *403 error* is thrown. Admin users are able to perform CRUD operations on the questions through the frontend.

Non-admin users can only update their own data. The server will validate if the user id matches the id in the JWT token. Users then can create, update, delete their own profile data after successfully matched.

8.2.3 Question History (N2)

8.2.3.1 Frontend Details

On the Collaboration page, when a user clicks the "End Session" button, the session will end for both users, and the question will be saved to each user's history.

On the History page, users can browse their question history by retrieving their previous question attempts via the user service. The attempts and their respective dates are displayed in a table format. Users can choose to retry questions in new sessions, aiming to improve their past solutions. They also have the option to navigate to LeetCode to explore additional solutions or review the question independently.

8.2.3.2 Backend Details

The **currentCode** and assigned **question** state variable is stored both in each user's EditorView. Any changes to the code made by a user are saved as a string through CodeMirror's onChange event handler. This code is only synchronized and saved for both users when edits are made to the code editor. If the code is left at its default state,

the question is not considered attempted and it will not be saved for the users when they leave the session.

If one user ends the session, a "`endSession`" event is sent to the websocket server that broadcasts a "`leaveSession`" to the socket.io room. This triggers the frontend to send a HTTP request to the user service. This request will store the assigned question to the collaboration session along with the user's code in the userCollection under the user's questions.

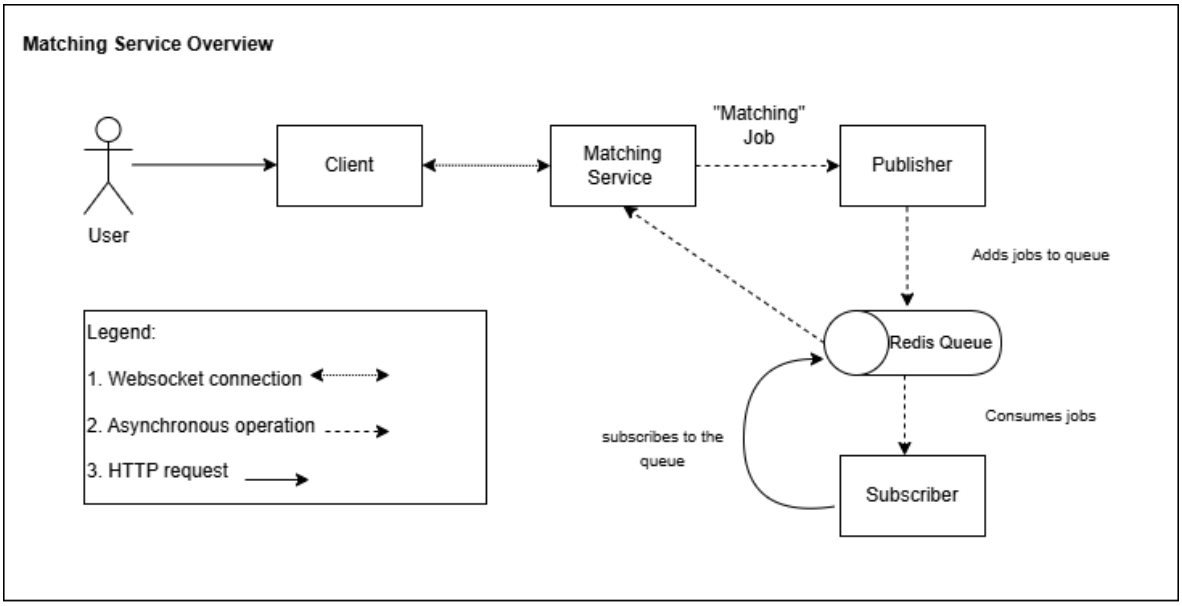
8.2.3.1 Design Decisions

Storing question history via the front end ensures that the collaboration service remains decoupled from the user service. Therefore, the user service alone manages the storage and handling of users' question data and history. By keeping these responsibilities separate, the collaboration service can focus solely on facilitating live interactions, while the user service independently manages historical data and persistence. This separation enhances modularity, making the system more maintainable and scalable, as each service operates independently within its scope without unnecessary dependencies.

By saving the user's attempt in the user collection, managed by the user service, is justified because it ensures that all user-specific data remains centralized and secure within a single service. Since each attempt is tied to an individual user, it makes sense to store it alongside other personal data in the user collection. This approach consolidates all user-related information, including question history, into one place, improving data integrity and making it easier to retrieve, update, and manage user records.

8.3 Matching Service

8.3.1 Matching Service Overview

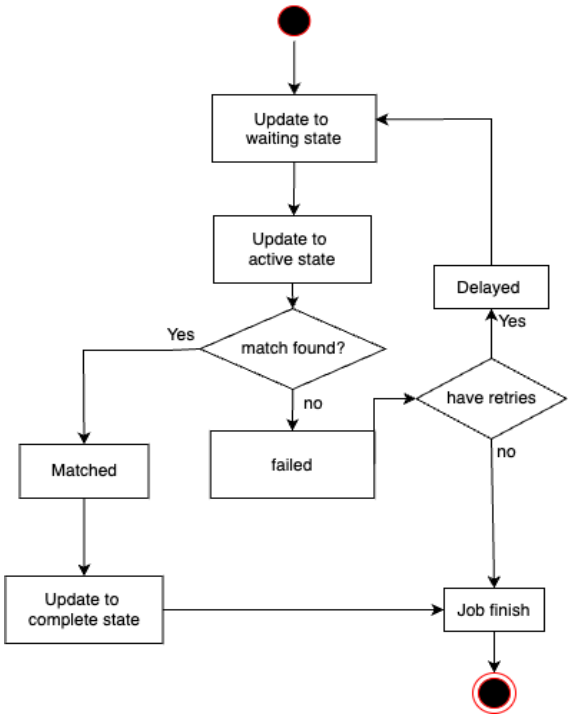


Our matching services use a transient asynchronous communication to match users. This ensures loose coupling of the matching service with the client. In the event that the matching service is down, the client can still function.

1. When the user joins the queue via the client, a websocket connection is established with the matching-service and the user is assigned a socket Id.
2. The user is then notified about if he has successfully joined the queue on the client.

3. The queue publisher adds an event or “job” into the queue. Any available subscribers will then listen for jobs in the queue and consume it, handling user matches.
4. Once two users are matched, they are notified and placed into a unique chat room (created dynamically based on their `socketIds`) to facilitate real-time communication.

8.3.2 Matching Implementation



The matching service matches users based on the following criteria.

1. Users will only be matched if they have selected the same question difficulty and topic.
2. Users will be matched on a first come first serve basis.

The matchmaking process consists of several steps:

- **Job Delays**: Users are initially placed in a "waiting" state in the queue. If no match is found within a certain time frame, the job may be delayed and retried, based on a backoff strategy.
- **Matching Process**: The matchmaking process operates in several stages:
    1. **Delayed Jobs**: The algorithm first checks for users in a delayed state who are potentially suitable for a match. If a matching user is found (i.e., with the same `topic` and `difficulty`), the job is promoted and moved to the active queue.
    2. **Active Matching**: If no match is found among delayed jobs, the system proceeds by checking users in the "waiting" state. It compares their `topic` and `difficulty` and attempts to match users. Once a match is found,

both users are removed from the waiting queue and added to the active queue.

3. **Question Assignment**: Once a match is found, the system fetches a question ID from a separate question service. This question ID is used to assign a relevant question to the matched users. This is done through the `fetchQuestionId` function, which interacts with the external service to fetch the correct question based on the `topic` and `difficulty`.

- **Failure Handling**: If no match is found after multiple attempts, the job will be marked as "failed," and a notification will be sent to the users indicating that no suitable match could be found. This is handled in the `matchingQueue.on("failed")` event listener.

8.3.3 Redis as message broker

In the matching service, Bull Queue (BullQ) and Redis are used to efficiently manage user matching requests through a job queue system. BullQ creates a Redis queue where each user's matching request is added as a job, which is then processed by a worker to find suitable matches. The system leverages Redis to handle job retries, delays, and backoff configurations, promoting jobs when a match is found or retrying after delays.

We decided to adopt Redis as our message broker due to the following reasons.

1. **Very high throughput** due to it being an in-memory data store. Peerprep is meant to be a real-time collaboration application and ensuring matches are made fast, reducing latency between users joining the queue and finding the match.
2. Support for the **pub-sub messaging** pattern. Refer to Section 5.1.

While we recognise the trade-off of lacking data persistence when using a Redis queue compared to Kafka or RabbitMQ. However, we felt that the trade-off for a fast memory access is worth it over data persistence as storing information about users in the matching queue is not essential for our application, as users can always rejoin the queue again in the event data is lost when the system is down.

8.3.4 Websocket communication

The matchmaking system relies heavily on real-time communication using **Socket.IO**. We decided to use a websocket connection to a matching service due to the following reasons.

1. Users can be notified instantly upon a match. This is benefical compared to polling as multiple http requests must be made at intervals. This incurs overhead in terms of having to make multiple http requests and having to store matched user data status in the backend.
2. Ensures users are connected before matching. In the event that one user disconnects after a match is found, the socket is able to detect it and notify the other user upon failure.

8.4 Collaboration Service

8.4.1 Socket.io and Rooms

Socket.IO is a library that enables real-time, bidirectional communication between clients and servers, often used for applications that require live data updates, like chat or collaborative tools. In this case, rooms are virtual spaces in which specific clients (sockets) can connect and receive targeted messages.

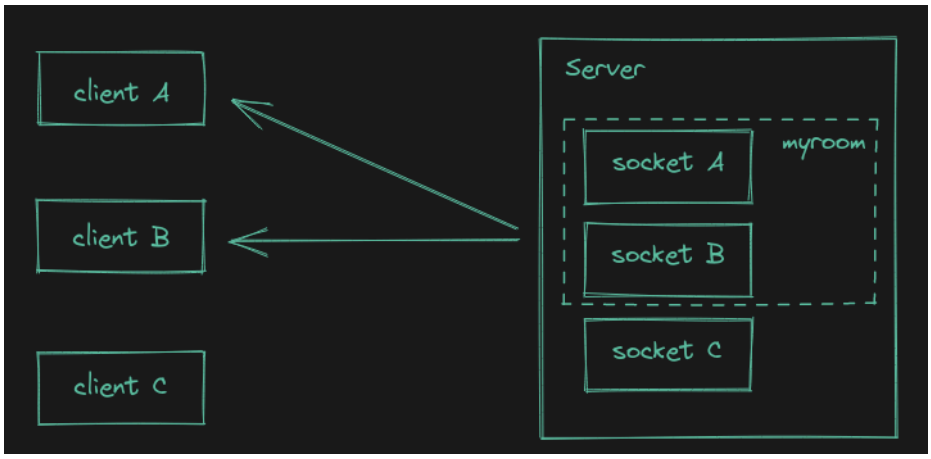


Image taken from https://socket.io/docs/v3/rooms/

In Socket.IO, each socket connection is assigned a unique, randomly generated identifier (Socket#id), ensuring that each user has a distinct `socketId`. To facilitate collaboration, when the matching service pairs users, they are assigned a unique roomId specific to their collaboration session. Each user's `socketId` then joins the WebSocket room corresponding to this `roomId`. All real-time collaboration data, such as code changes and text-based communication, is transmitted within this room using `io.to(roomId).emit()` events.

By assigning matched users to a shared `roomId`, Socket.IO allows messages to be broadcasted exclusively to users in that room, ensuring that only session participants receive the shared code updates and messages. This architecture enables secure,

efficient real-time collaboration, isolated from other active sessions, and scales well for multiple concurrent collaborations.

8.4.2 Scalability of socket.io

We chose to leverage on socket.io instead of other libraries such as yjs.websocket due to the horizontal scalability of the socket.io. Socket.io provides redis adaptors that allow for horizontal scalability of our collaboration service, while ensuring that sockets across scaled servers can communicate with each other.

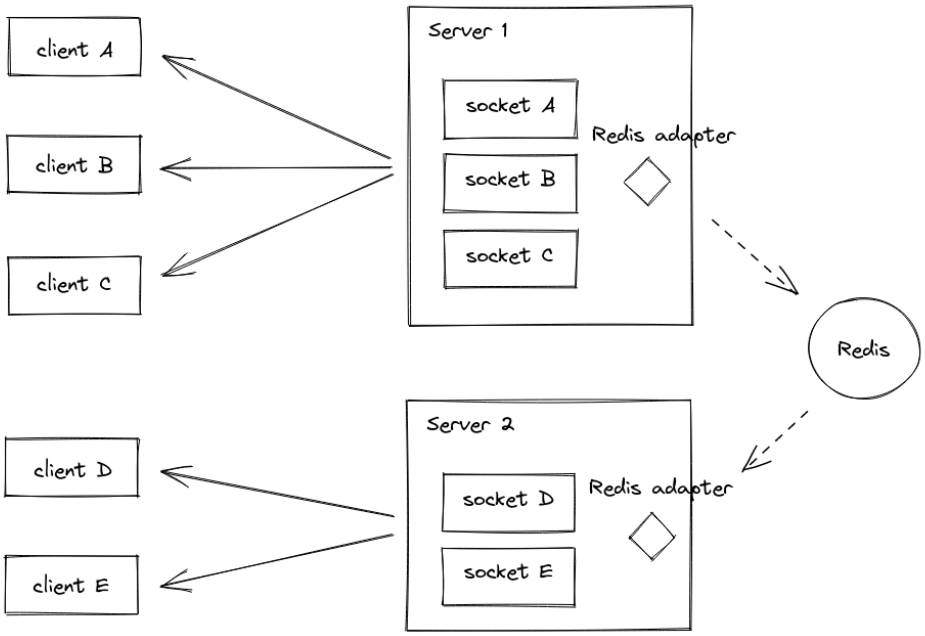


Image taken from https://socket.io/images/broadcasting-redis.png

8.4.3 Communication (User live chat)

The chat service leverages **Socket.IO** for web connection to enable real-time, bidirectional communication between users within a collaborative session. When two users are matched for collaboration, they are assigned a unique roomId that creates a private virtual room within Socket.IO. Each user's unique `socketId` joins this room, allowing all chat messages and collaboration data to be contained within that specific session.

Within this room, Socket.IO enables the chat service to broadcast messages exclusively to the users in the session. Using `io.to(roomId).emit()`, chat messages are sent instantly to both users, ensuring that only the participants of the specific collaboration session receive the communication. This structure isolates the session from other active sessions, enhancing both privacy and efficiency. The chat service, supported by Socket.IO's room-based architecture, provides a seamless, real-time messaging experience, essential for collaborative tools where immediate interaction is crucial.

Initially, we created a separate WebSocket microservice for live chat. However, we later decided to integrate the chat service within the Collaboration Service for these reasons:

- **Reduced WebSocket Connections**: Combining live chat with the collaboration service under a single WebSocket reduces resource consumption and simplifies user session tracking and resource management.
- **Simplified Horizontal Scaling**: Combining chat and collaboration in one microservice allows for synchronized scaling, making it easier to maintain session consistency across nodes and preventing latency or issues that could arise if the services were separated.

Message/Event	Description	Direction
`sendMessage`	Sends a chat message from one user to others in the room	Client to Server
`receiveMessage`	Delivers chat messages to users in the room	Server to Client
`joinRoom`	Adds a user to a specified chat room	Client to Server
`userJoined`	Notifies when a user joins a room	Server to Client
`userDisconnected`	Notifies when a user leaves the chat room	Server to Client
`endSession`	Indicates the end of a chat session	Client to Server / Server to Client

8.4.4 Code Editor

The code editor has to allow both users to work together seamlessly in a shared environment. We utilize CodeMirror and Socket.io to achieve that. This system supports

seamless collaboration by providing real-time updates, version control, and persistent state, ensuring users experience continuity even across disconnections.

We used **CodeMirror's collaboration extension** to help manage real-time updates from both users during collaborative coding sessions. This extension provided utilities for collaborative editing based on operational transformation.

Despite the additional complexity of implementation using codemirror, we chose it over the Monaco editor as it did not provide conflict resolution utilities. We felt that conflict resolution during live coding sessions was important and it helped ensure a smooth collaboration session where changes from both parties can be recorded without affecting each other.

CodeMirror also provided additional plugins that allowed us to also have syntax highlighting for different languages, making it a desirable code editor to have in our application.

Message/Event	Description	Direction
`getDocument`	Requests the initial document when a user connects	Client to Server
`getDocumentResponse`	Returns the current document state to the requesting user	Server to Client
`pushUpdates`	Sends code updates to the server, including version information	Client to Server
`pullUpdates`	Requests missing updates or versions from the server	Client to Server
`pullUpdateResponse`	Returns the requested updates	Server to Client
`receiveUpdates`	Broadcasts code updates to other connected users	Server to Client

8.4.5 Code Execution (Judge0 API)

The **Code Execution** feature in our collaborative code editor provides users with the capability to write, edit, and execute code within the same environment. This

functionality enhances the collaborative experience by allowing users to test their code in real time.

We chose Judge0 API for our application because of its seamless code execution across multiple languages, essential for collaborative coding. Its real-time feedback helps users instantly verify code outputs or errors, enhancing productivity. By offloading execution to Judge0, we maintain low-latency performance and scalable infrastructure, which result in more responsive and efficient collaboration.

Message/Event	Description	Direction
`compileRequest`	Sends code and language information for execution	Client to Server
`compileResult`	Returns the result of code execution, including output or errors	Server to Client
`executionError`	Notifies user of any execution errors encountered during processing	Server to Client

8.4.6 AI Chat (Gemini API)

The **Gemini Chat** component is an AI-driven conversational feature integrated within our collaborative environment. This feature leverages Google's **Generative AI** capabilities to provide real-time assistance and responses, making it an invaluable tool for users needing support, guidance, or feedback. By combining a responsive chat interface with advanced language models, Gemini Chat enables seamless interactions between users and an AI assistant.

Gemini Chat provides a text input field where users can type their queries. This input is managed using the `input` state, which captures and stores the user's query. The chat maintains a history of the conversation through the `messages` state. Each entry in this array includes the user's message and the corresponding AI-generated response.

We integrated the Gemini API into our collaborative workspace to streamline user access to support, minimizing the need to switch between multiple tools or interfaces. This integration reduces context switching, allowing users to maintain focus and productivity within the workspace. Additionally, incorporating Gemini Chat within the collaboration section enables seamless inclusion of user-generated code in text

prompts, based on user preferences, thus reducing repetitive copy-pasting and enhancing overall workflow efficiency.

9. Set-up Instructions

9.1 Structure Overview

Our repository consists of two main folder: frontend and backend

The deployments are split into local and cloud:

1. Local deployment using NPM
2. Deployment using containerisation method: Docker
3. Deployment using kubernetes on Minikube
4. Cloud deployment using Google Cloud

9.2 Cloning of repositories and setting up environment variables

You will need to create accounts for MongoDB, Redis, RapidAPI, Gemini API to deploy locally. Example links can be found in env.example. The steps to deploy both locally and on cloud are shown below.

Clone the repositories from here. Refer to the .example.env to set-up your environmental variables

Steps to deploy locally using docker compose:

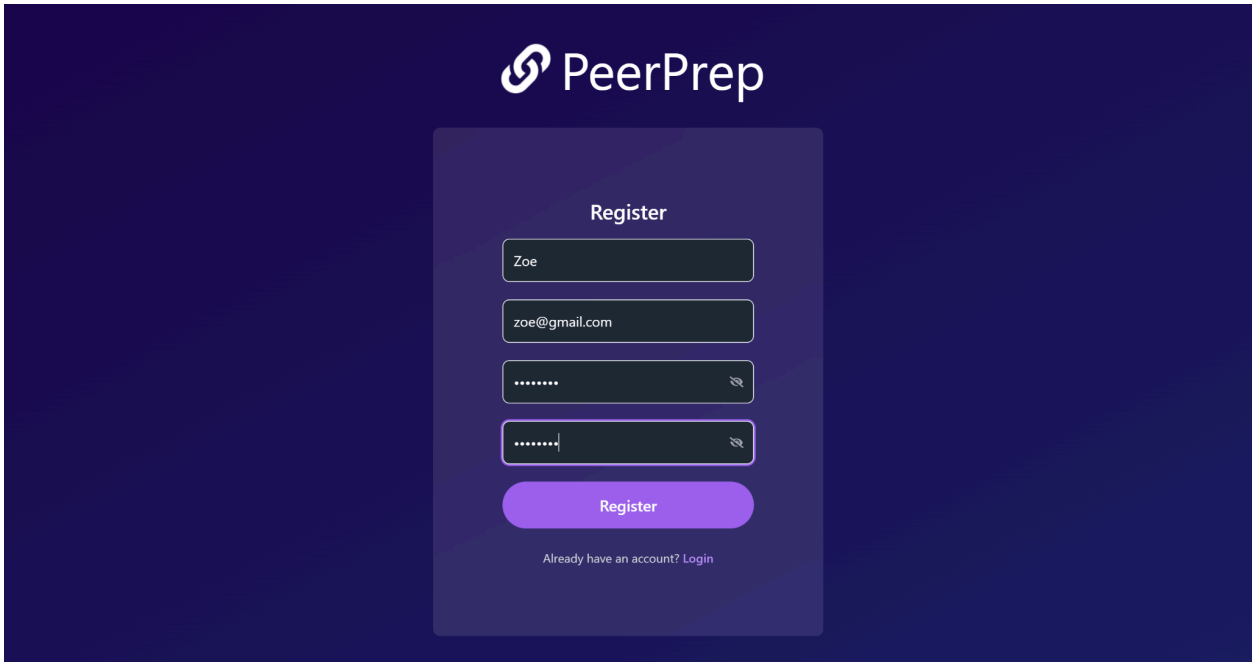
1. Ensure that Docker is installed in your local device. Open docker.
2. Go to ./backend. Run docker compose up.
3. After finishing, each microservices are running as images in Docker, go to http://localhost/5173 to view the application.

Deployment on Google Cloud

We built and pushed images of our microservices to the Google Artifact Registry. We then created an autopilot cluster on Google Kubernetes Engine. To deploy the microservices on the cluster, use the google cloud CLI to configure kubectl. After configuration, run `kubectl apply -f` for all the kubernetes files within the kubernetes folder. This will create a deployment of the microservices from the images in our Google Artifact Registry. We also needed to install the nginx ingress controller into our cluster.

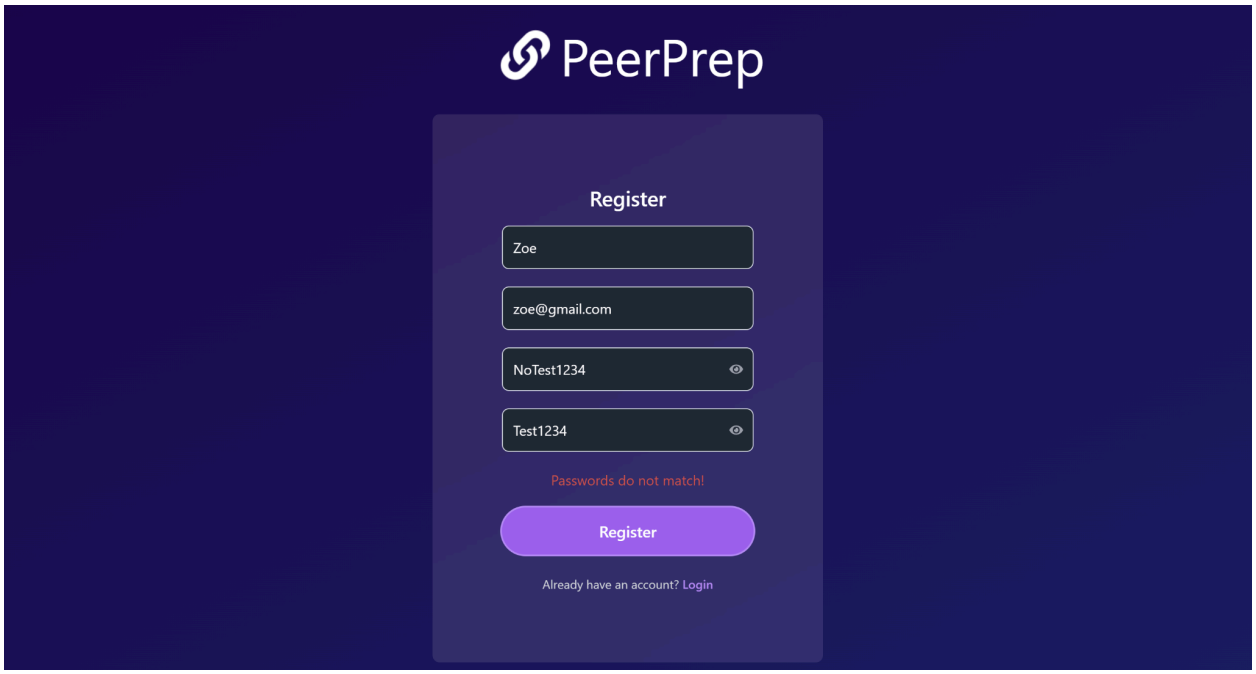
10. Application screenshots

Register Page



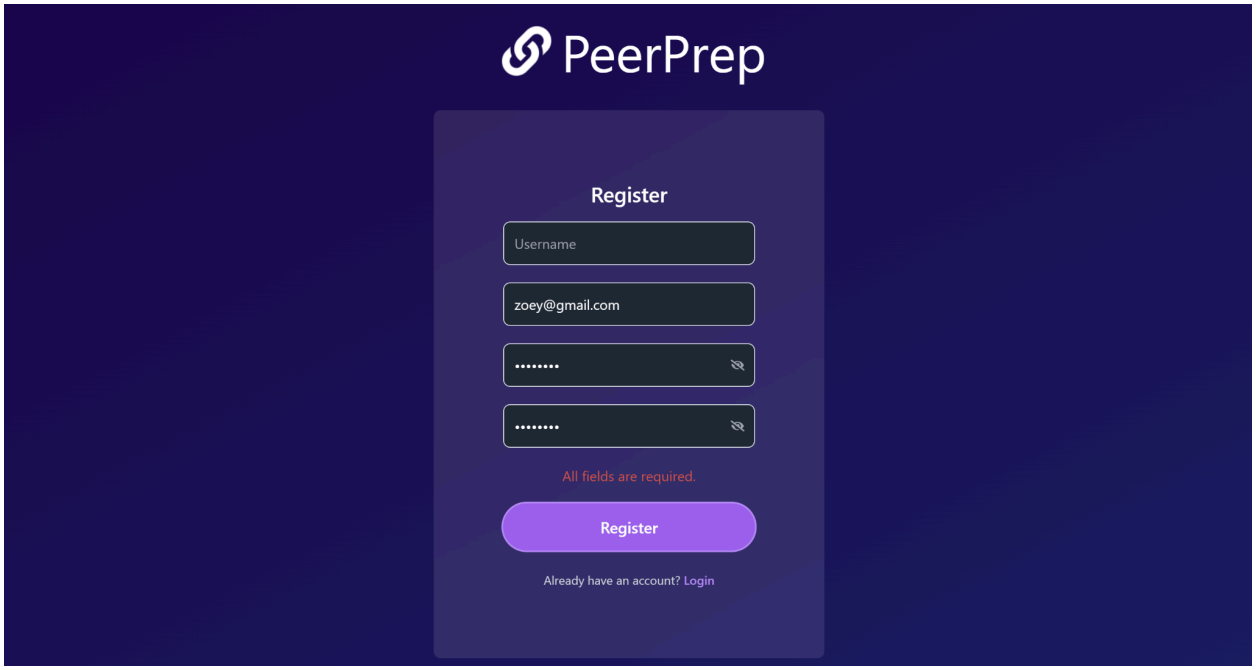
Password match validation

A warning message in red text, "Passwords do not match!" is displayed



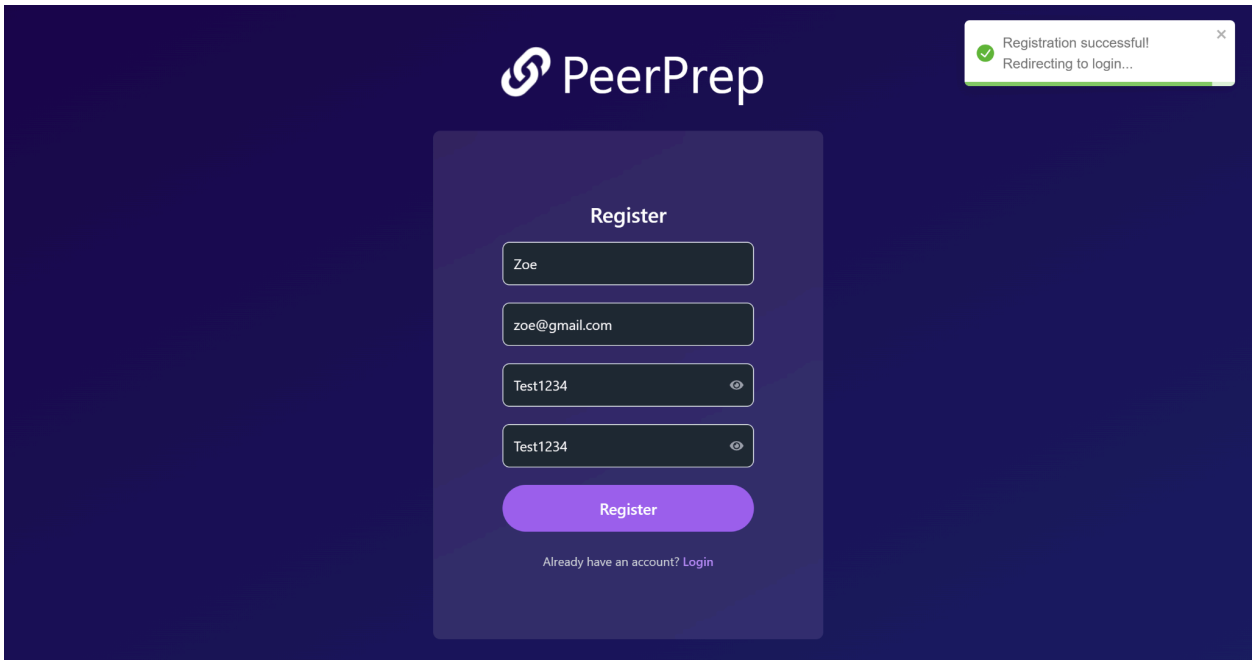
Any empty field

A warning message in red text, "All fields are required." is displayed


PeerPrep
Register
Username
zoey@gmail.com
All fields are required.
Register
Already have an account? Login

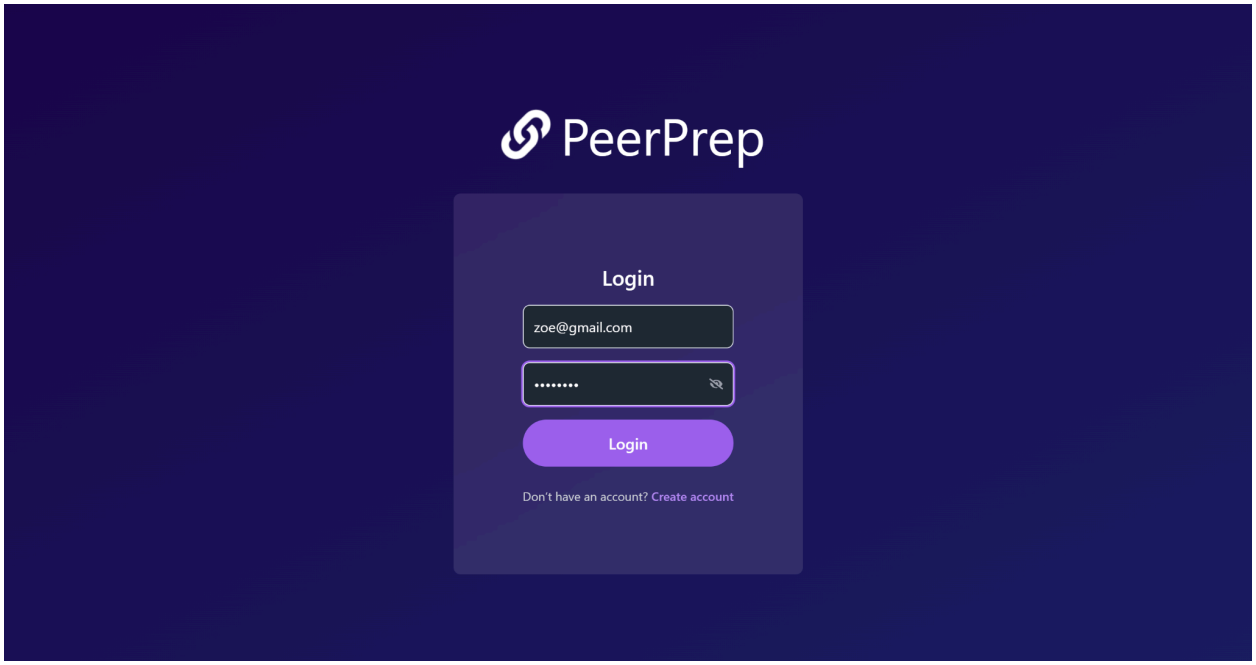
Successful Registration

A success notification is displayed in the upper-right corner of the screen with a green checkmark and the message "Registration successful! Redirecting to login...".



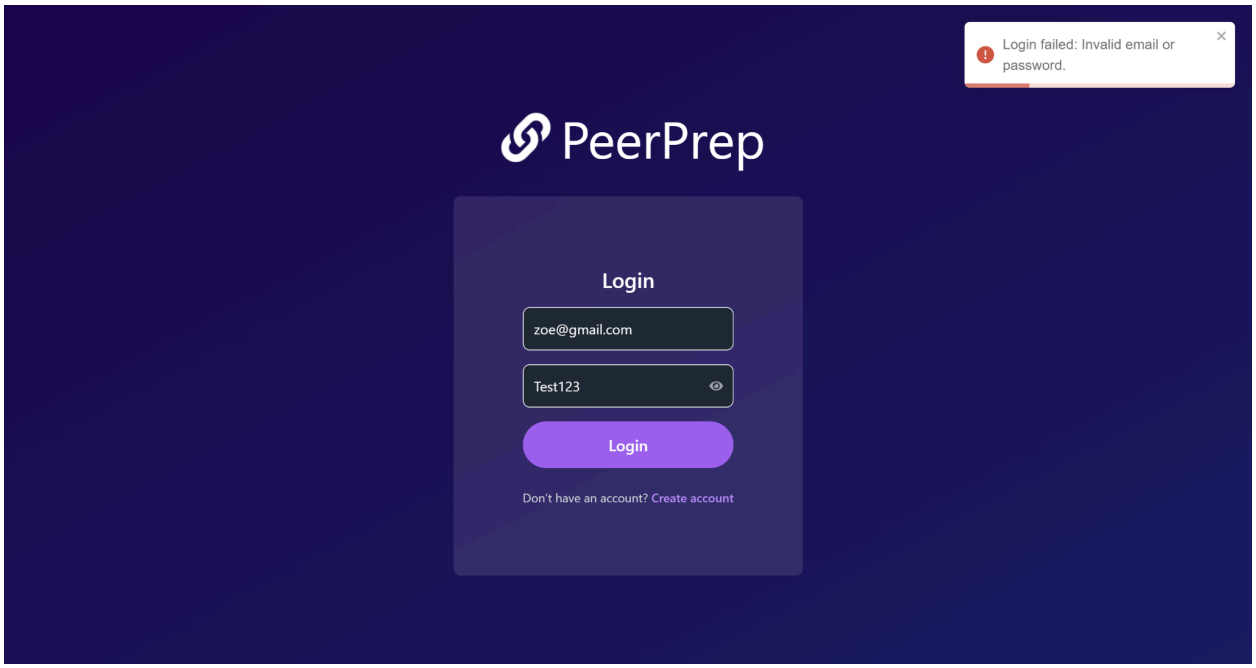
Login Page

Password is hidden.



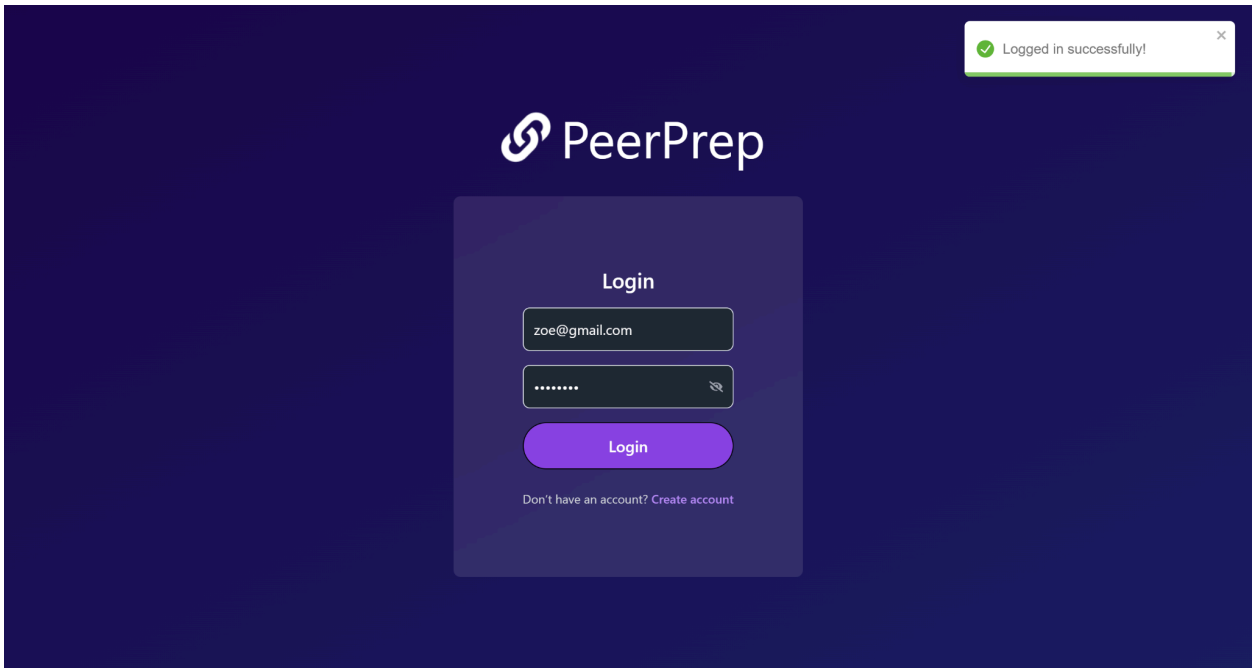
Wrong password

An error notification is displayed in the upper-right corner of the screen with a red cross and the message "Login failed: Invalid email or password".

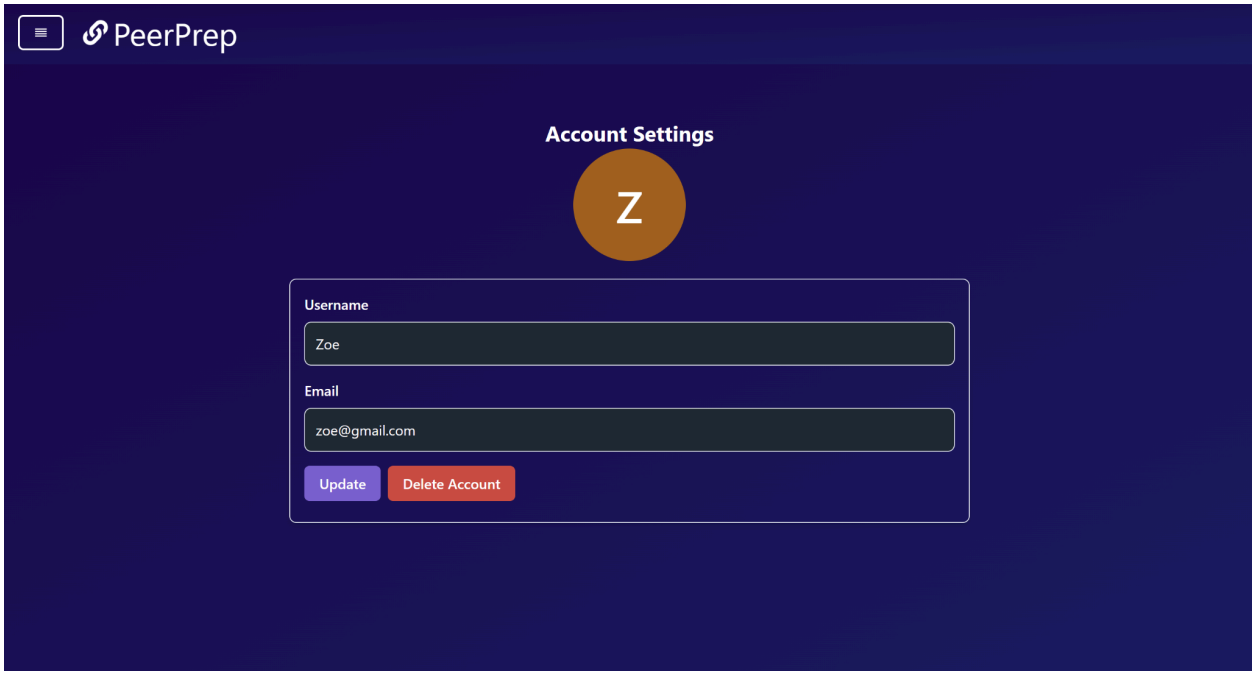


Successful sign in

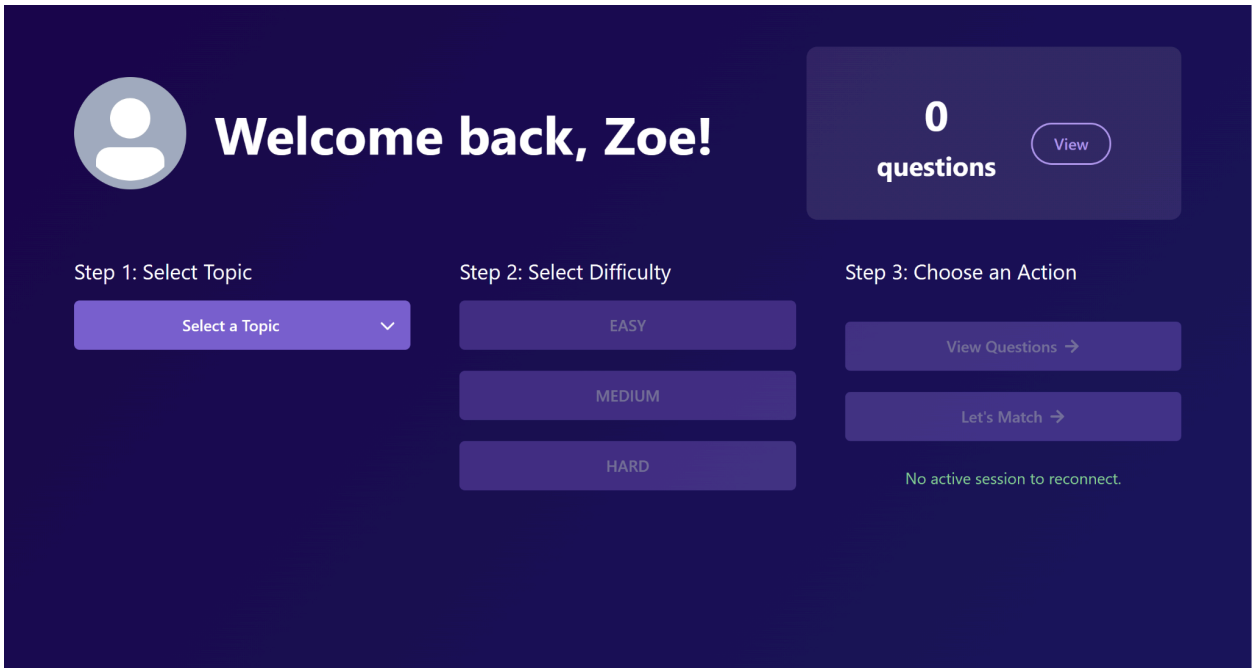
A success notification is displayed in the upper-right corner of the screen with a green checkmark and the message "Logged in successfully!".



Account Settings Page

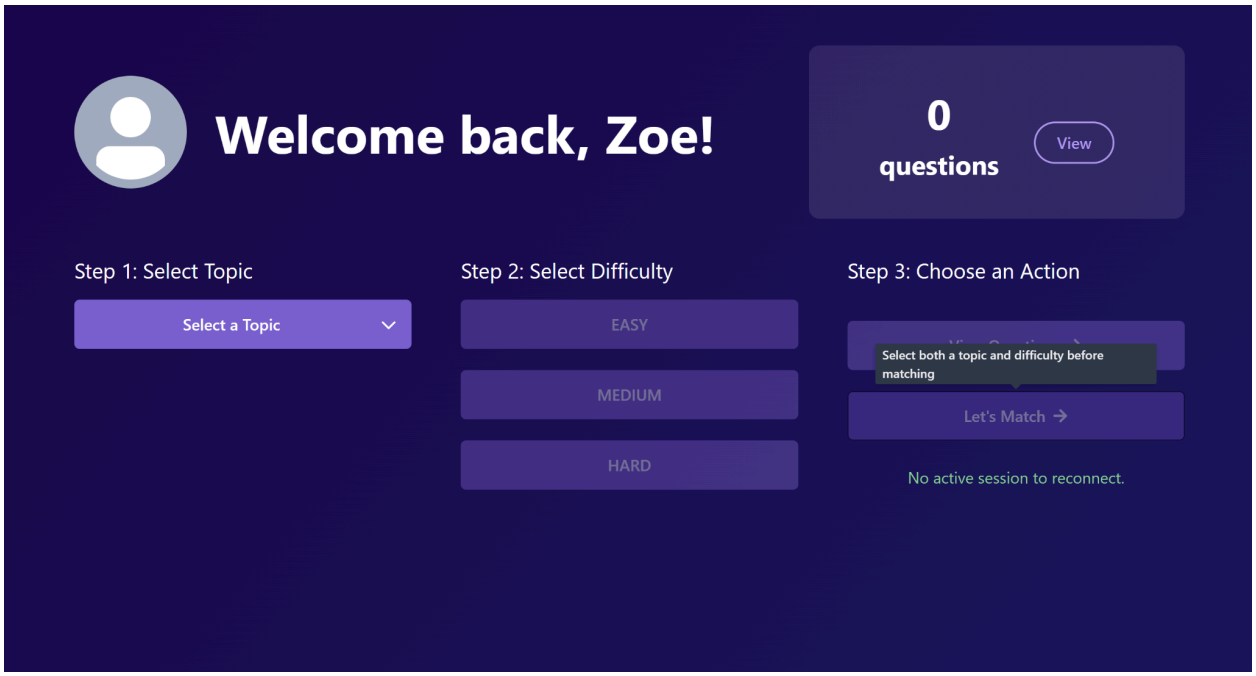


Home Page with no active session

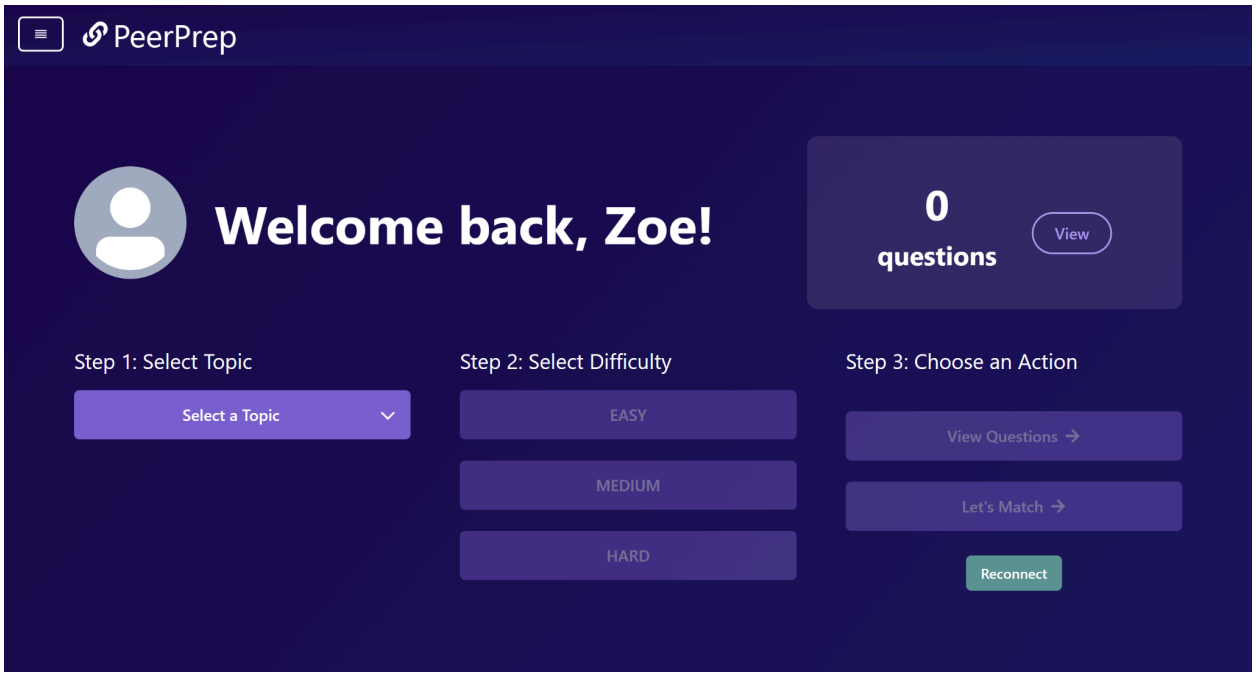


Tooltip enhancement

The tooltip appears when the user hovers the cursor over the "Let's Match" button and reads, "Select both a topic and difficulty before matching."

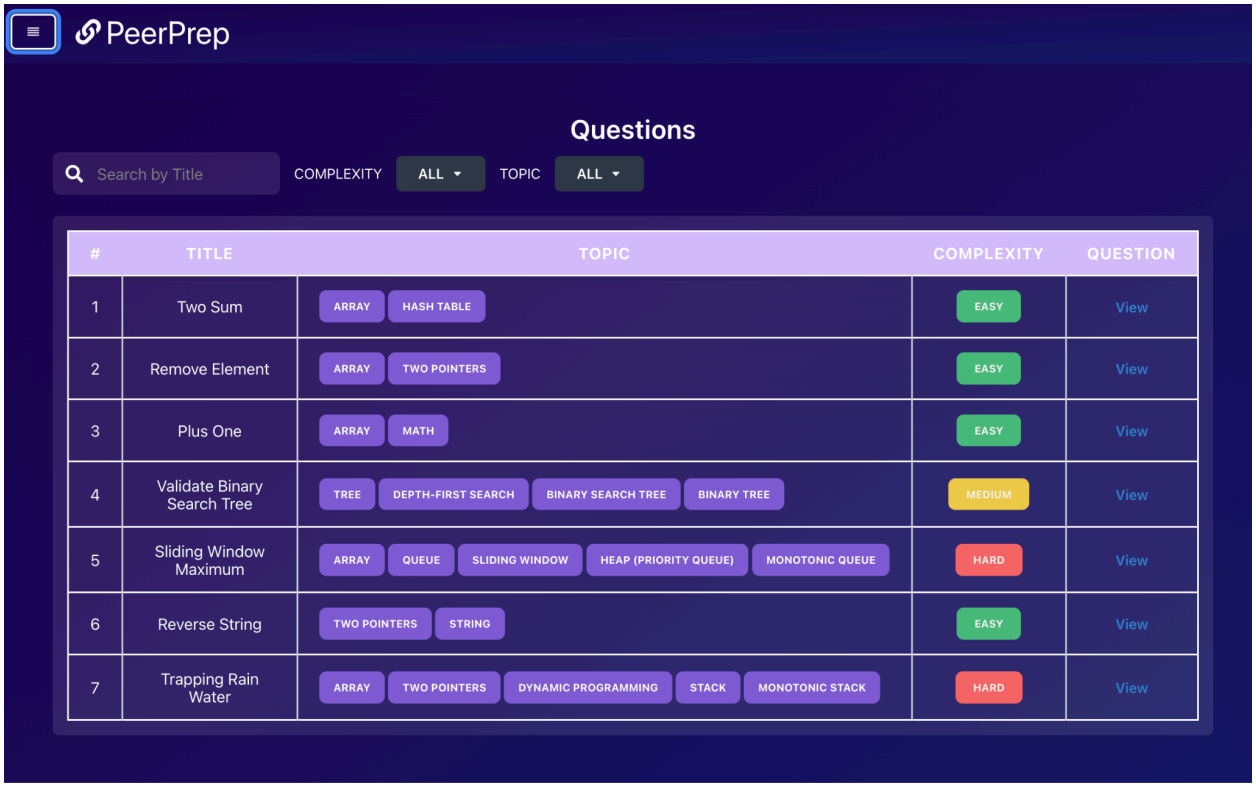


Home Page with active session ongoing

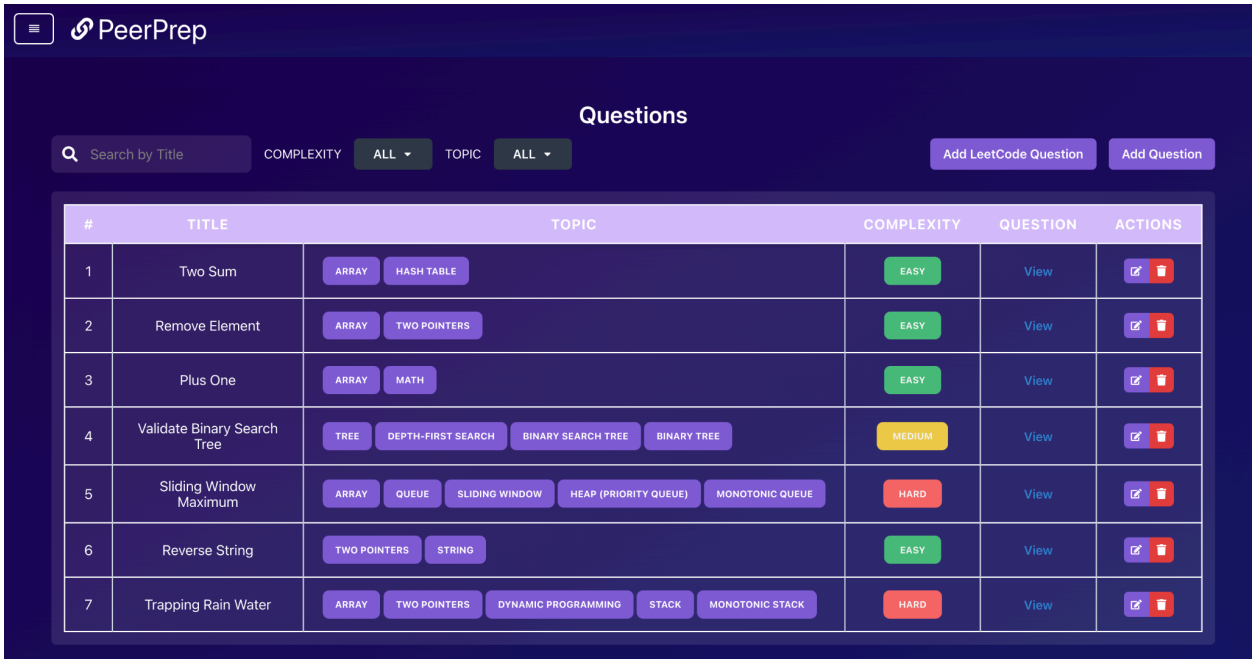


Questions Page

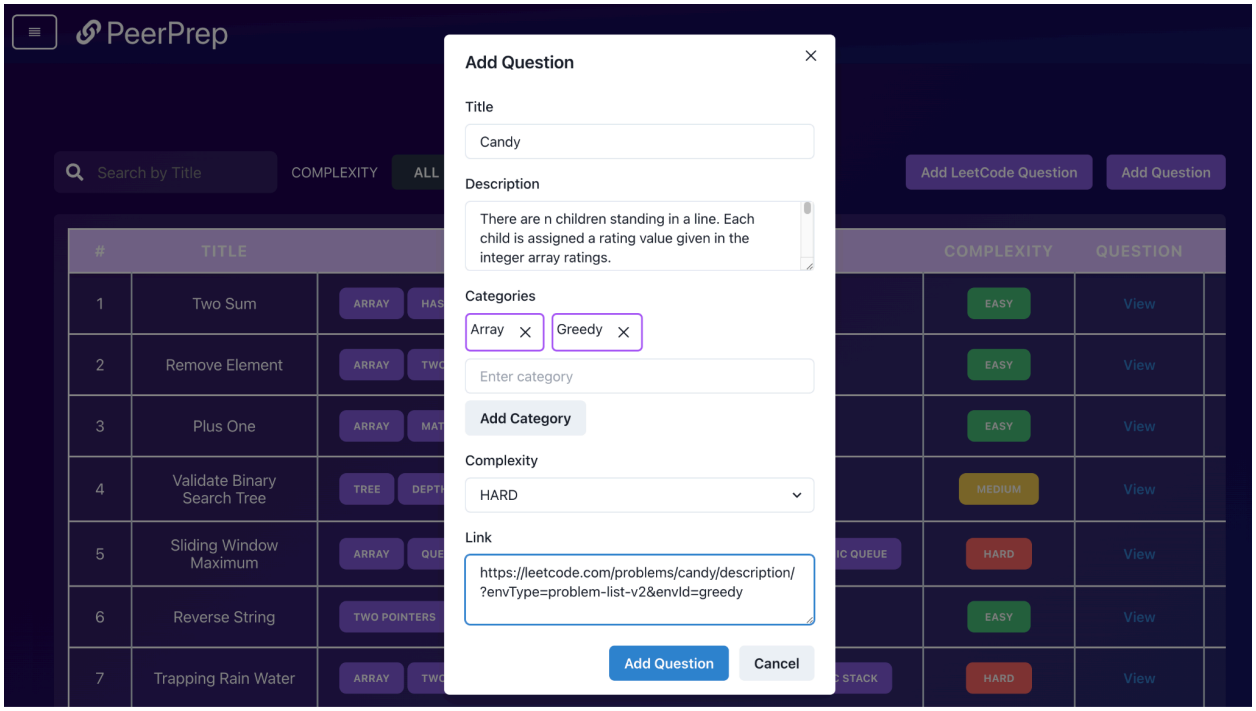
Non-admin:



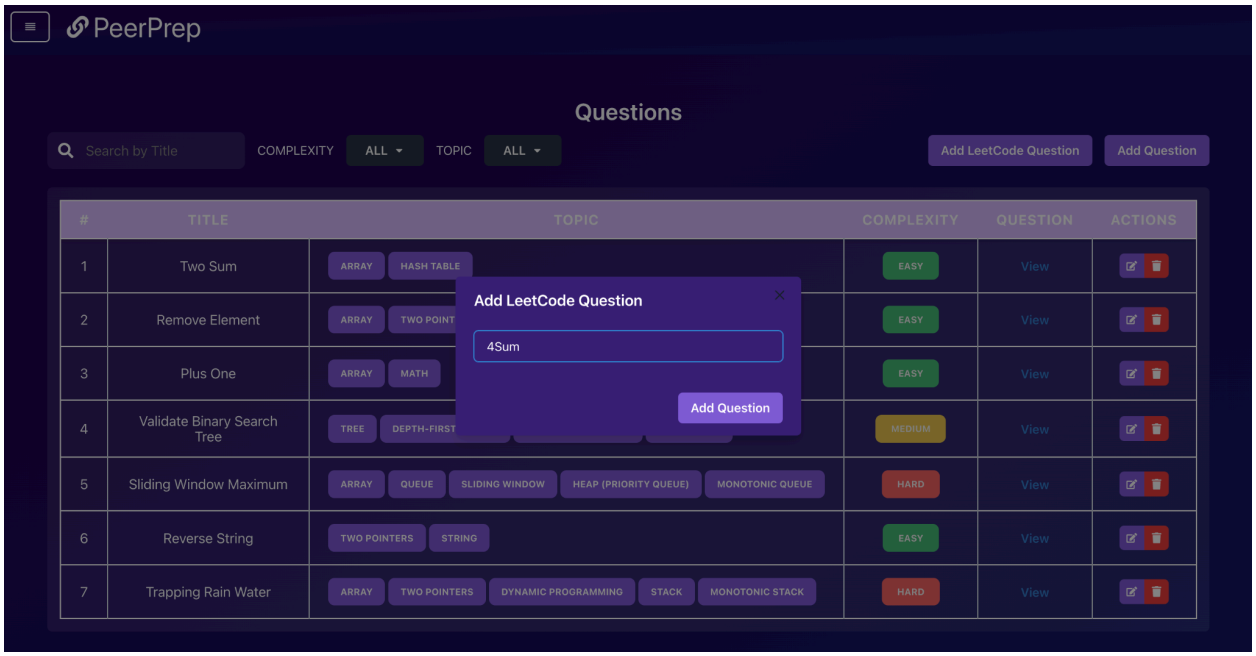
Admin:



Addition of custom question:

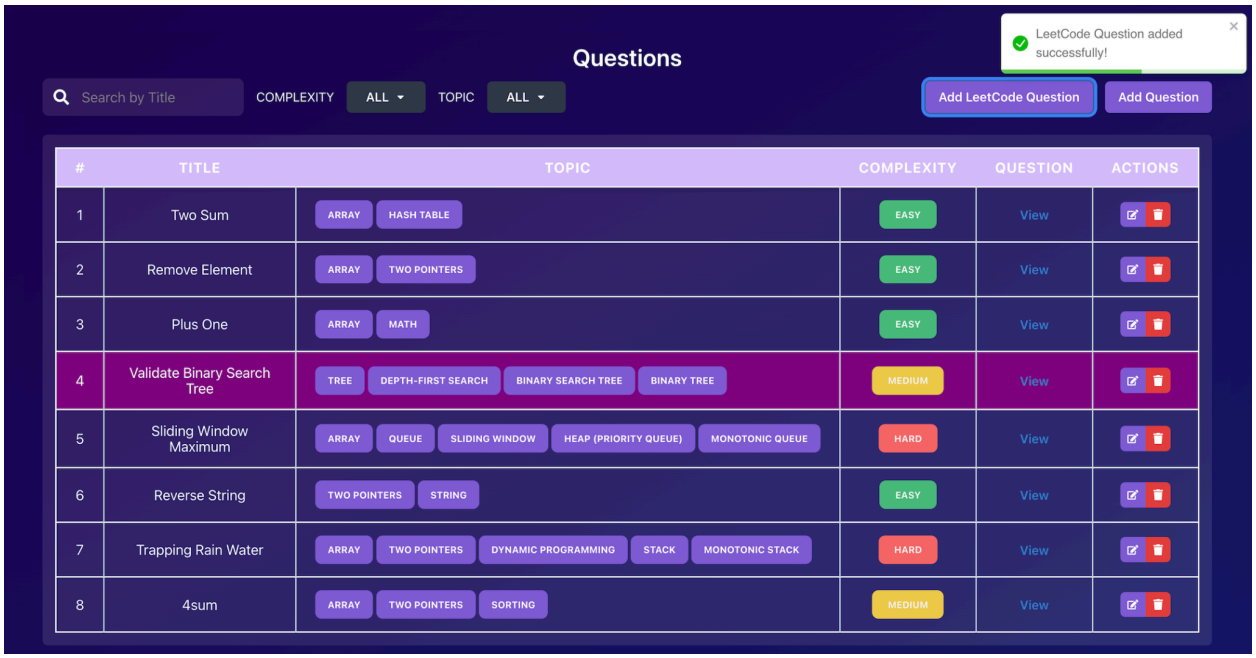


Addition of Leetcode question:



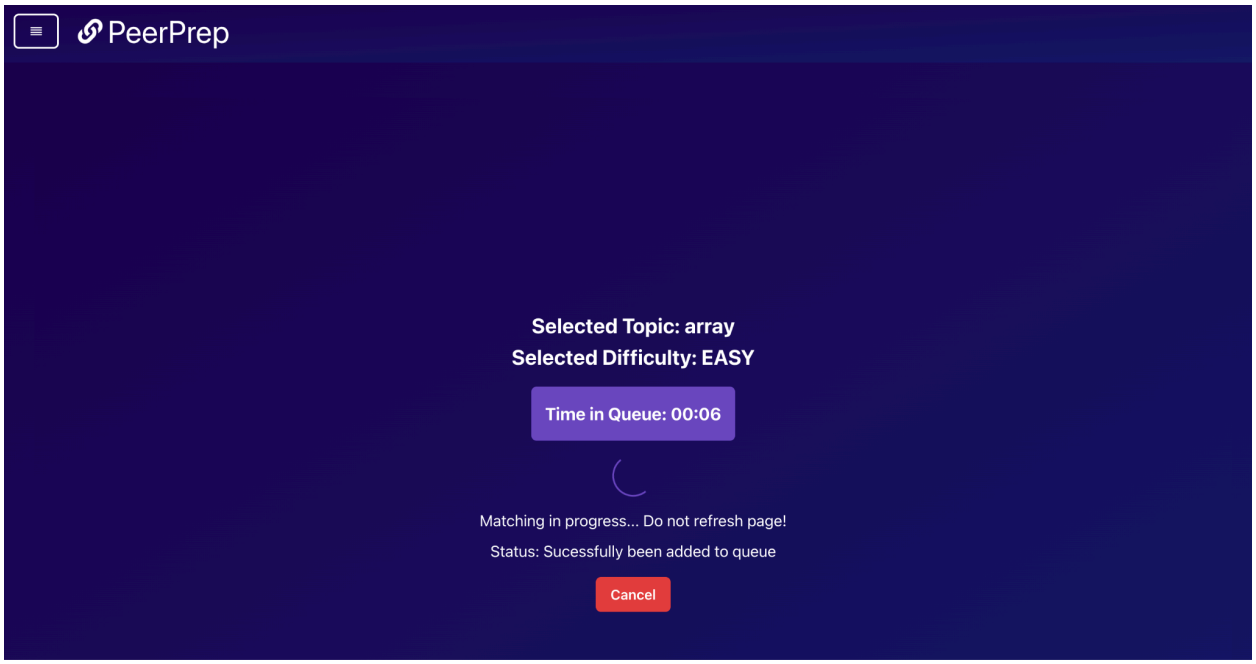
LeetCode question successfully added

A success notification is displayed in the upper-right corner of the screen with a green checkmark and the message "LeetCode Question added successfully!".

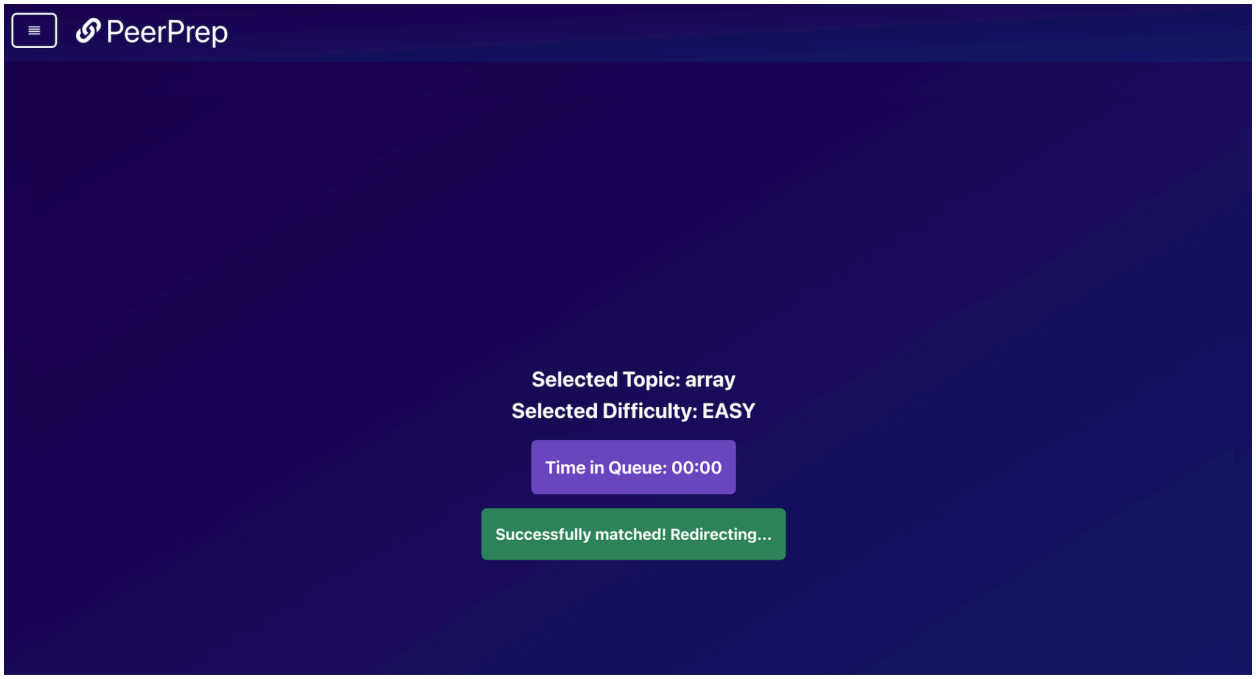


Matching Page

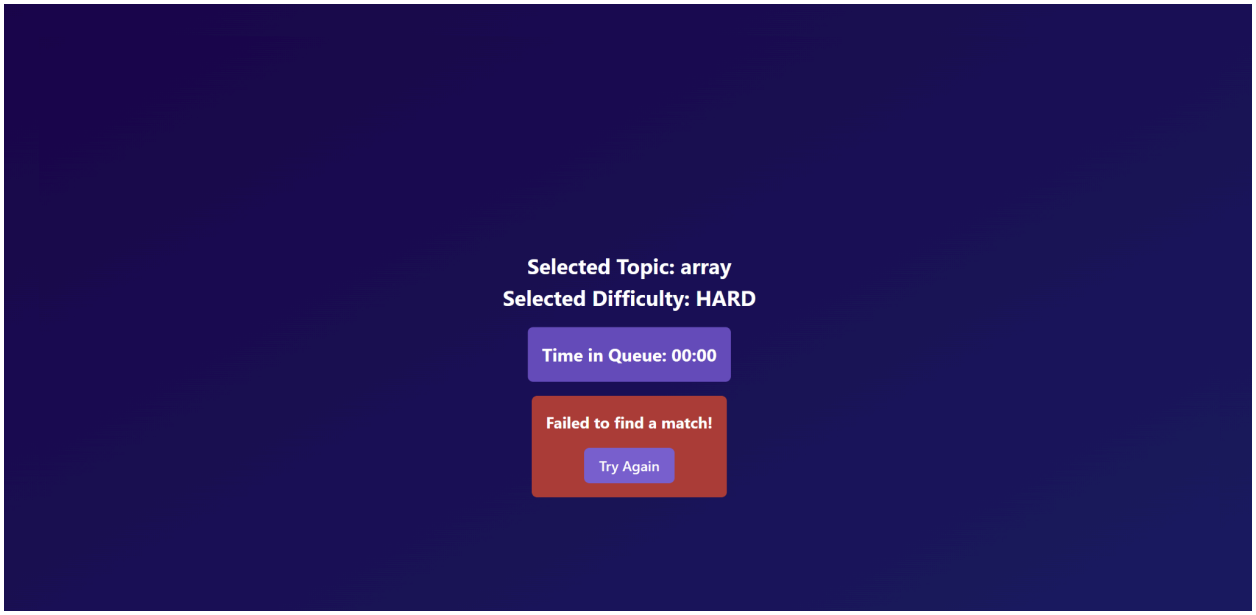
Matching in progress



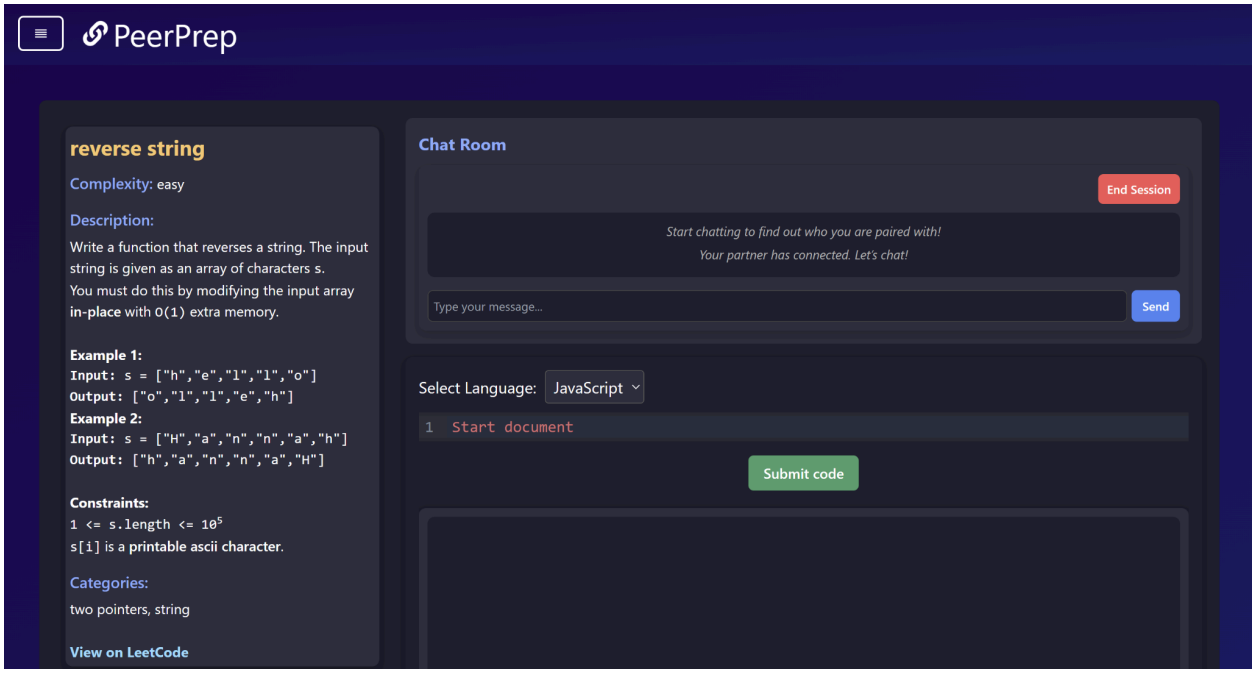
Match found



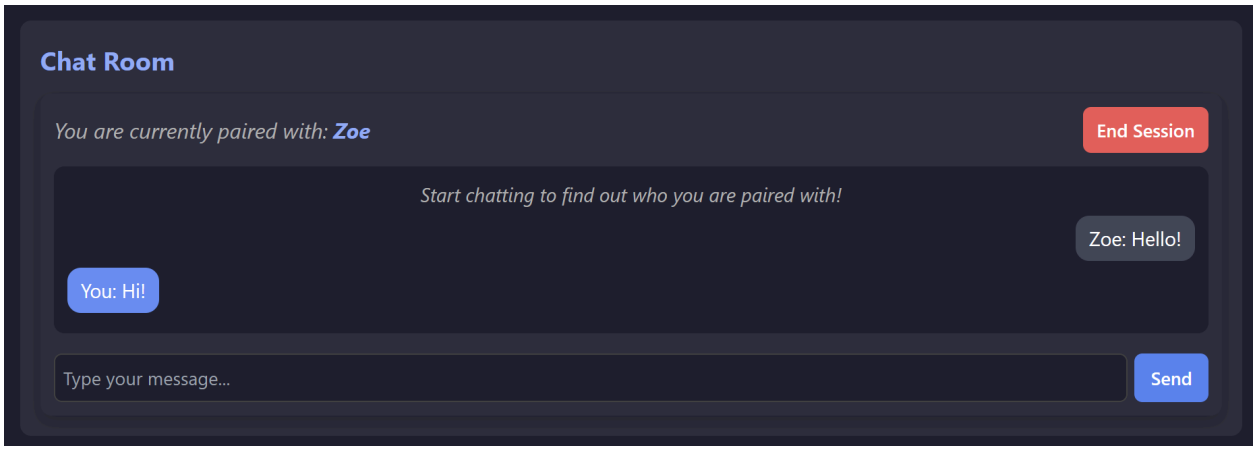
Match not found



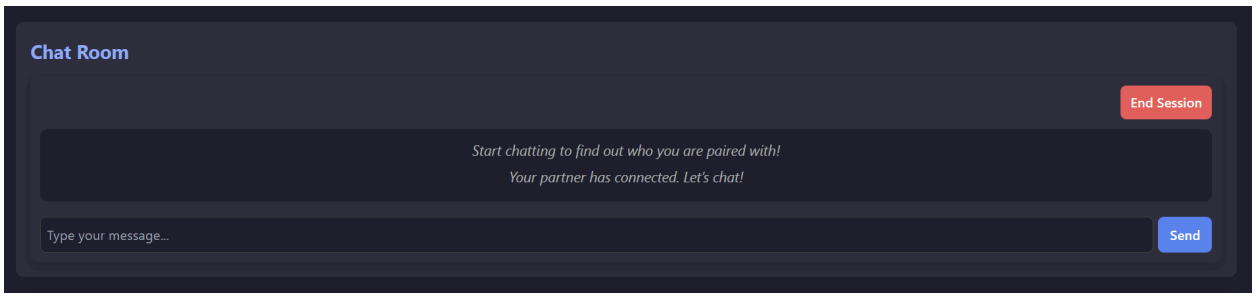
Collaboration Page



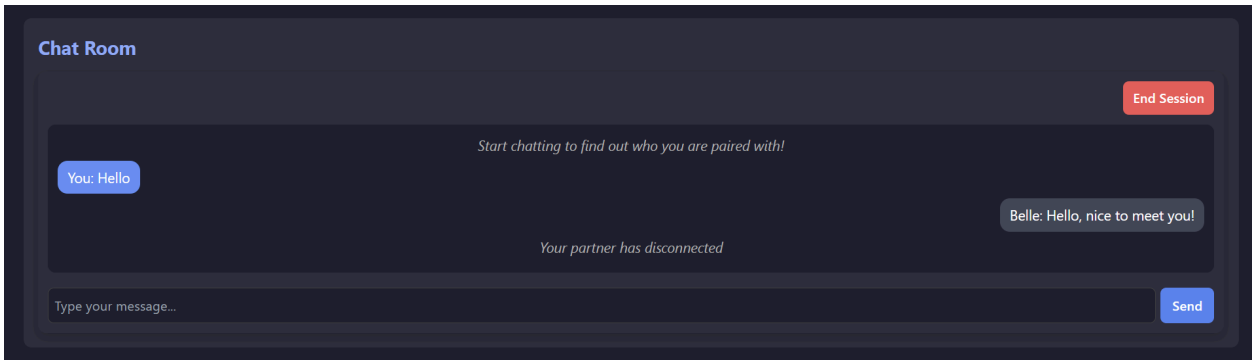
Live Chat



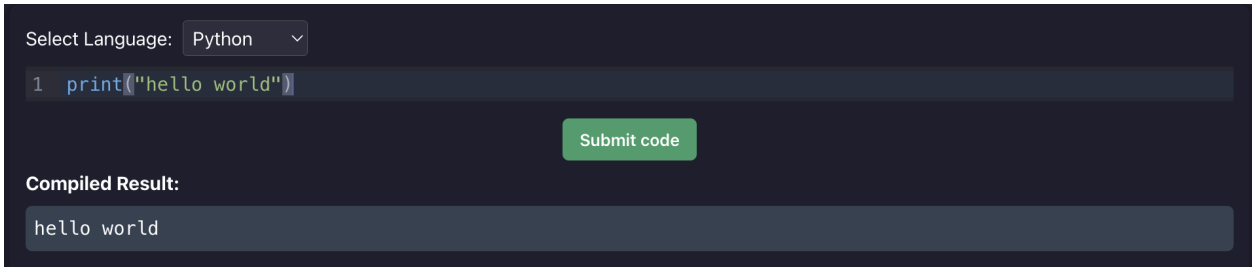
When a user is connected to the room first, a notification appears saying, 'Your partner has connected,' when another user joins, indicating that the session is ready for chatting



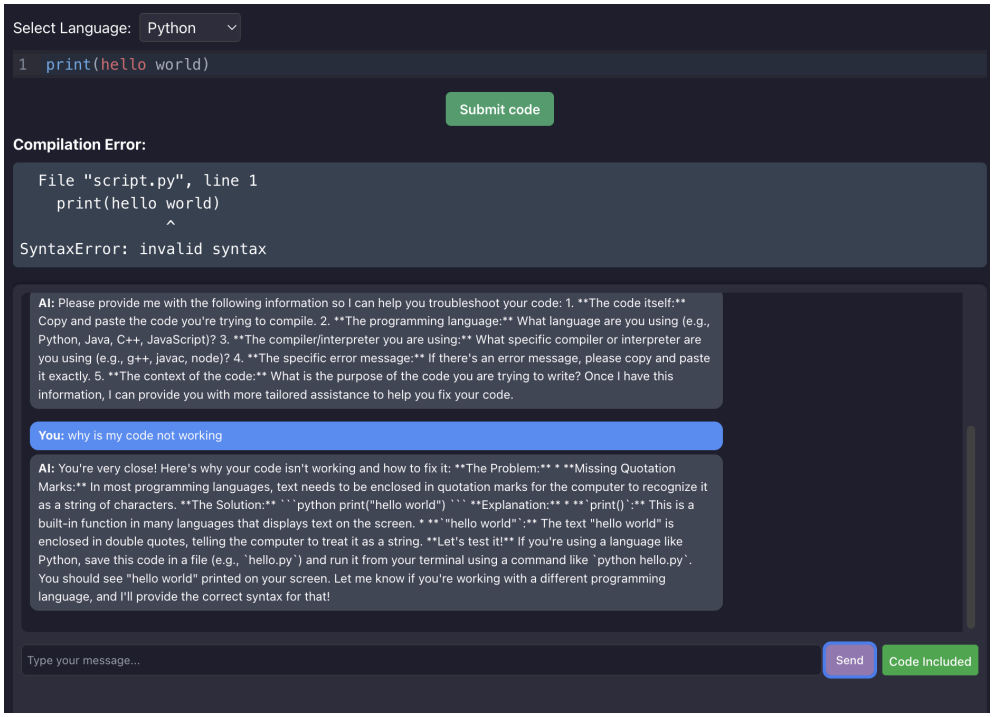
When their partner disconnects, a notification appears saying, 'Your partner has disconnected,' indicating the other user has left the session.



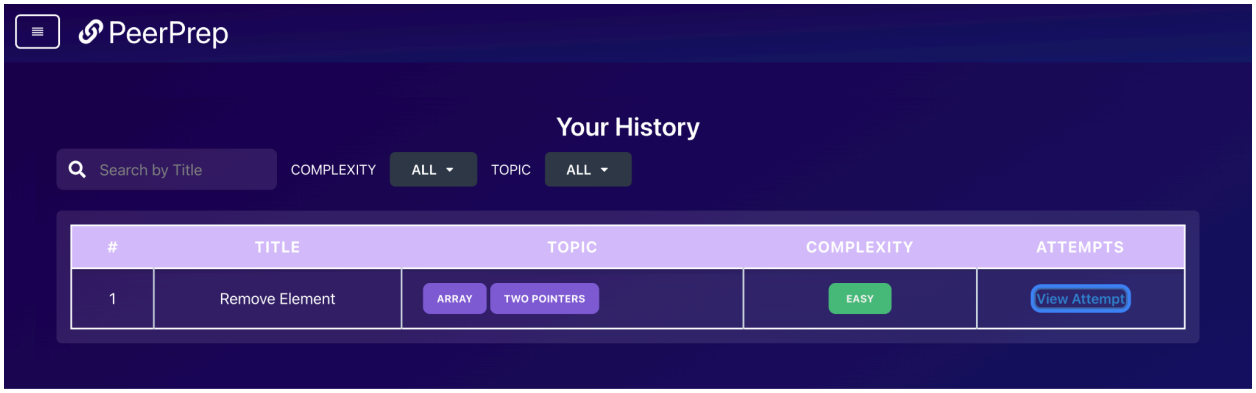
Successful compilation:

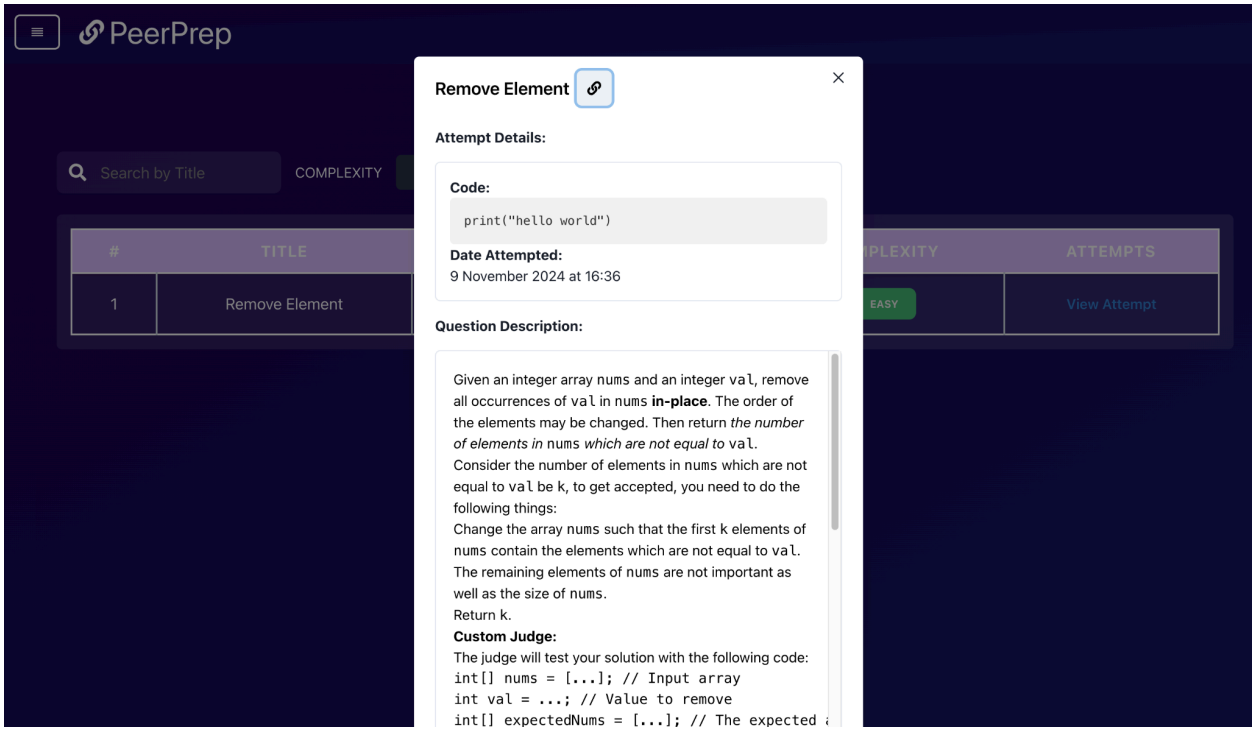


Failed compilation:

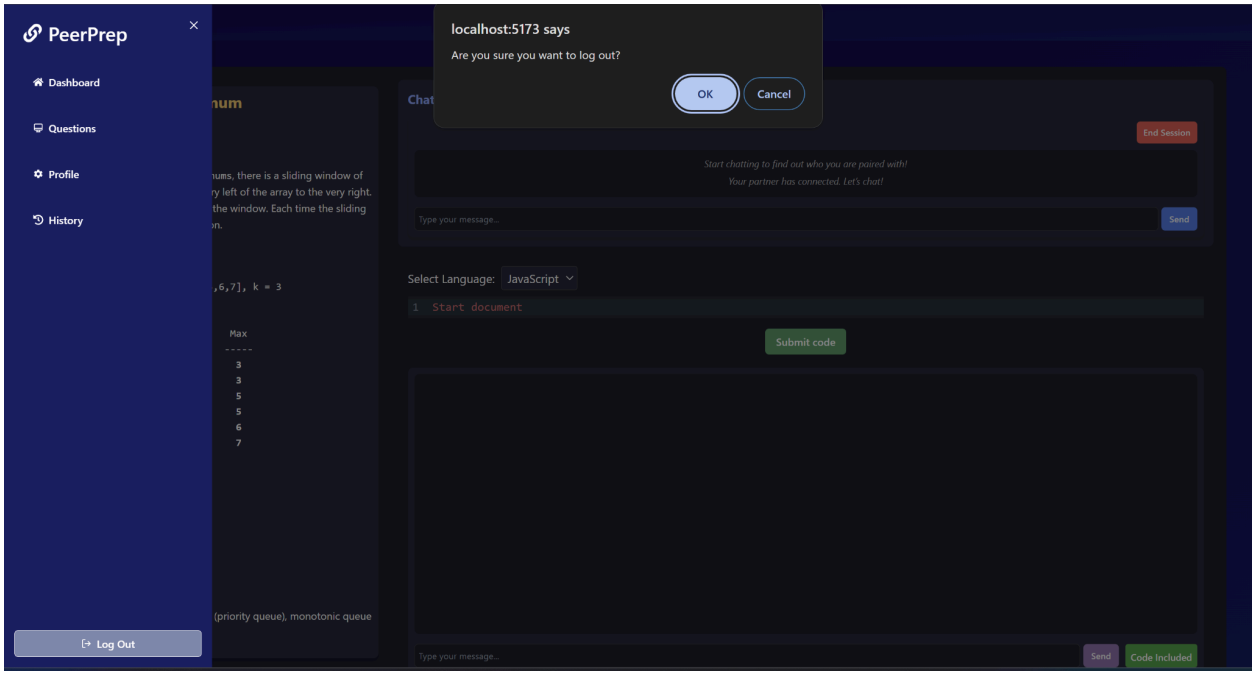


History Page

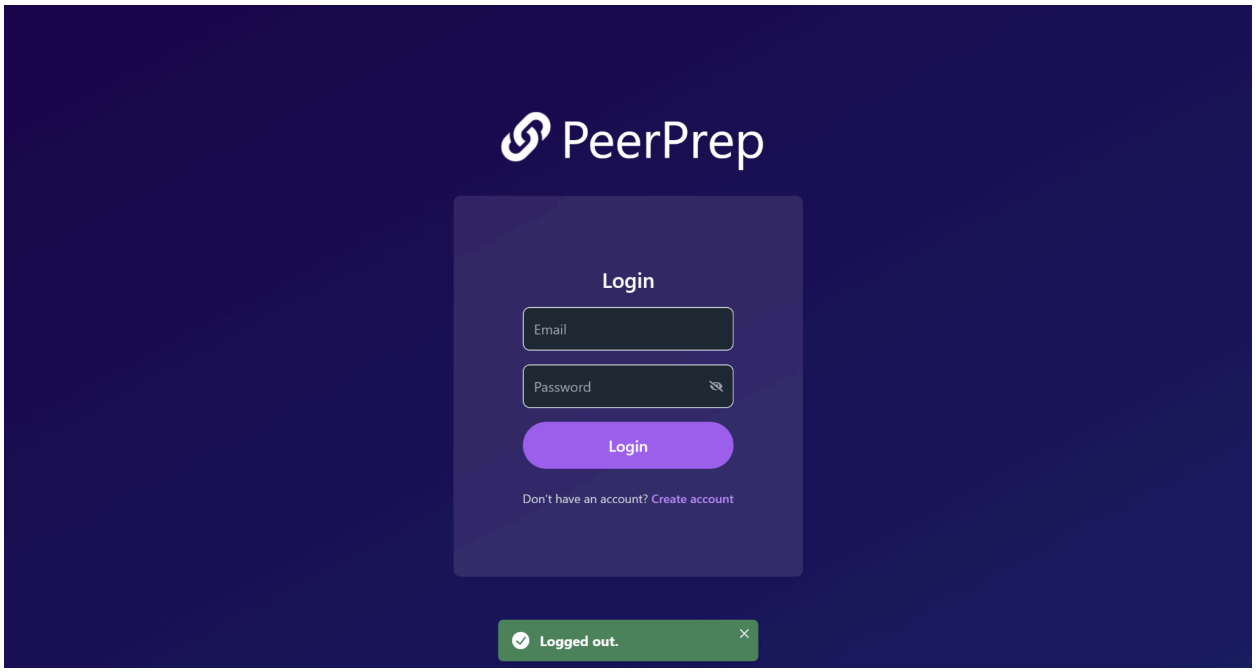




Logged out



After logging out, Logged out notification was shown.



11. Future Enhancement

In addition to implementing the drop feature in version 2.4, our team recognizes that there is significant potential for further enhancements in this project. While we have made good progress, we believe that there are several areas where improvements can be made to elevate the overall experience and functionality.

1. One enhancement would be to get our entire application on Cloud. This can be achieved by hosting the frontend of the application on **Google Cloud Run** to communicate with our deployed GKE Cluster via the API gateway. However, due to time constraints, we are unable to create an SSL connection for our hosted backend service, preventing our hosted client service from interacting with our .**Using Let's Encrypt** certificates can be used to ensure secure connections between the frontend and backend. With this, users will be able to access the application regardless of their Wi-Fi network or location, improving the overall accessibility and reliability of the service.

2. Currently, if the WebSocket server goes down, any unsaved code in the room is lost. To enhance the collaboration service, we plan to implement a feature that will automatically save and restore the code either on a database or using persistent message brokers in the event of server disruptions. This will prevent any data loss and ensure that users can continue their work seamlessly, even if there is an interruption in the server connection.

3. When a user disconnects or leaves the session, the chat history is not saved. We plan to address this by adding functionality that will save chat messages upon reconnection. This will allow users to scroll through their chat history after reconnecting, ensuring that important conversations are accessible even if they leave or rejoin the session.

4. To streamline development and deployment, we are considering integrating Continuous Deployment (CD) into our workflow. Using GitHub Actions, we can automate the deployment process by connecting Google Kubernetes Engine (GKE) and Cloud Run. This integration will facilitate smoother updates and maintenance, improving both the reliability and efficiency of our deployment pipeline.

12. Reflection

1. We have learned the importance of effective project management and have developed a strong workflow. This workflow has improved our communication, allowing us to delegate tasks more efficiently and maintain transparency regarding our other commitments and challenges.

2. We learnt the importance of having task allocation based on difficulty. We overlooked the difficulties in some of the tasks in the milestones, leading to struggles in meeting the deadlines. Our team adopted a development cycle to split tasks for each milestone among each other. However, this meant that challenging milestones had less time to complete. One example would be our nice-to-have of deploying our application on cloud and setting up an API gateway. The learning curve of Devops was steep and there were many unexpected challenges faced. Due to starting late for such challenging tasks, we struggled to troubleshoot and had to keep finding alternatives to make things work quickly when we got stuck with an issue that took long to resolve with no progress. This made our team realize that better time management for more challenging tasks was crucial during development.

3. We have also recognized the importance of testing after integration. Since there are frequent code changes and multiple branches open to handle our respective tasks, merging branches can introduce bugs. Therefore, testing each branch thoroughly before merging is essential to maintain code quality and functionality.

4. We learn that it is essential to have good documentation and a clear code structure. As we moved along, we noticed having clear documentation and sticking to coding standards allowed us to work together more effectively and cut down confusion during the integration.

In conclusion, this project has deepened our understanding of software engineering concepts and practices. Despite its challenges, we have gained valuable knowledge, not only in theory but also through practical application within the project.