Peer Prep

A CS3219 Project

Group 52

[Application]

[Repository]

Developers

Truong Minh Duong (A0244143N)
Anshumaan Tyagi (A0244508E)
Rehman Sajid (A0214081U)
Lee Wen Jun (A0235164J)
Tan Jian Wei (A0223779R)

Introduction

Background

With recent mass layoffs by technology giants such as Amazon, Google and Meta, the technology job market appears bleak. It is becoming harder for students to secure these highly sought-after jobs and internships with the influx of skilled and experienced talent back into the job market. Many of these openings require students to have to complete multiple rounds of technical interviews ranging from Online Assessments to technical whiteboard-style interviews. Platforms like LeetCode and Hackerrank help students practice these technical skills, by providing a way for students to solve coding questions from a large range of categories and difficulties.

However, students might be unsure if their approach in solving a question is optimal and if there could be a better approach in solving the questions. They could also face difficulty in explaining their solutions to others, and justify their reasonings behind their approach.

Purpose

Peer Prep aims to serve as an alternative to traditional coding practice platforms by providing a platform where they can participate in solving coding questions through pair programming.

1. The platform would help students explore alternative methods of solving questions by seeing how someone else would approach the question.
2. The platform would allow students a way to practice justifying their methods of solving questions to others - something that is often required in whiteboarding-style technical interviews.
3. The platform would allow students to work together to tackle more difficult questions that they might be unable to solve individually.

Scope

Peer Prep is tailored for university students who are seeking to practice their technical skill sets with regards to practicing coding questions as well as justifying their approach.

Contributions

Truong Minh Duong

Frontend	1. UI Implementation: a. Refresh dialogs and notification 2. Linked Collaboration refresh question to the Collaboration Service Microservice
Backend	1. Collaboration Service a. Refresh question status handling b. Fetch new question from Question Service c. Refresh cancelation 2. User Service a. Architecture Design b. Developing Schema and endpoints

Report	1. Project requirements
	2. Collaboration service - backend
	    a. Nice to Haves
	    b. Must to haves
	    c. Sub groups
	3. Future plan

Anshumaan Tyagi

Backend	3. Architecture Design 4. Matching Service a. Integration of Firestore b. API Endpoints c. Deployment on cloud run 5. Execution Service a. Configured Load Balancer to allow https requests to VM instances 6. User Service a. Configured Load Balancer to allow https requests to VM instances 7. Question Service a. Configured Load Balancer to allow https requests to VM instances

Report	4. Architecture Diagram
	5. Microservices
	    a. Matching Service
	6. Application Design
	    a. Architecture Decisions
	        i. Firestore Vs Rabbit Mq
	        ii. Compute Engine vs Cloud Run
	    b. Design Patterns
	        i. Load Balancer

Rehman

Backend	Question Service 1. Deploying and configuring MongoDB 2. Manually populating Questions 3. Developing Schema and endpoints 4. Containerizing the service
Frontend	Matching Service 1. Develop the Matching Page 2. Utilizing Redux to manage state 3. Implementing API calls to handle matching logic on Frontend (Matching Pages, Dashboard Page) Questions 1. Sorting of questions based on complexity 2. Sorting of questions based on name
Report	1. Monolithic vs Micro-Service Architecture 2. Reflections 3. Question Service a. Overview b. Endpoint Details 4. Model View Controller (MVC) a. Overview b. Diagrams c. Details

Lee Wen Jun

Frontend	1. UI design a. Web Theme b. Component designs c. Login, Logout, Dashboard, Loading, and Collaboration page 2. UI Implementation a. Loading page b. Collaboration page - Question section - Code editor section - Chat + Terminal section 3. Code Editor a. Support for multiple language b. Syntax highlight for multiple language 4. Code Execution a. Linked code editor with execution service b. Presented execution output on frontend
Report	1. Front End a. Monaco Editor b. Firebase React Hooks c. RTK Query d. Design Considerations 2. Collaboration Service - Front end a. Question Section b. Code Editor Section c. Chat + Terminal Section 3. Code Execution Service a. Overview b. Process

Tan Jian Wei

Frontend	1. Architecture Design 2. UI Implementation: a. Login Page b. Dashboard Page c. History Page d. Question Bank and Details 3. Linked Collaboration Page to the Collaboration Service Microservice 4. Linked Firebase Authentication functionality
Backend	1. Collaboration Service a. Architecture Design b. Integration of Socket.IO with Redis c. Code Storage, Room Creation, and Messaging d. Configured Compute Engine and deployed Collaboration Service 2. Execution Service a. Configured Compute Engine and deployed Judge0 3. User Service a. Added History Functionality where users could save and retrieve their previous attempts. 4. Firebase Authentication a. Set up Firebase Authentication to help manage the User's sessions b. Added route guards to authenticate and authorize access to User Service, Question Service, and Collaboration Service.
Report	1. Introduction 2. Frameworks a. Vite.js + React 3. Notable Packages a. Monaco Editor

	b. Firebase React Hooks c. RTK Query 4. Microservices a. User Service 5. Application Design a. Architecture Decisions i. Socket.IO vs Cloud Pub/Sub

Requirements

User Roles & Permissions

The following table shows the roles that are being catered to by Peer Prep:

Role	Permissions and Capabilities
User	Users refer to accounts created using the user creation page. They are granted access to Peer Prep's features such as: 1. Viewing all Questions 2. Matching with other users 3. Collaborating with other users 4. Basic management of their own accounts
Admin	An Admin has all the Permissions and Capabilities of a User in addition to: 1. Creating a new Question 2. Editing an existing Questions 3. Deleting an existing Question

Features

Feature	Description
User Management	1. Users are able to create an account 2. Users are able to log in 3. Users are able to change passwords 4. Users are able to delete their accounts 5. There are 2 user roles: User, Admin
Questions	Users 1. Users are able to view all Questions 2. Users are able to view the details of a Question 3. Users are able to view their latest saved attempts for a Question Admins: 1. Admins are able to create new Questions 2. Admins are able to edit existing Questions 3. Admins are able to delete existing Questions
Matching	1. Users should be able to match with another user to collaborate on a random question 2. Users should be able to match with another user to collaborate
Collaboration	1. Users are able to write code and view the code written by the other user in real time 2. Users are able to choose which language they wish to code in 3. Users are able to communicate with each other via text messages

Project Requirements

1. Must have (Mx):

- ❖ M1: User Service – responsible for user profile management.
    - ➢ We implemented user service, more details below
- ❖ M2: Matching Service – responsible for matching users based on some criteria.
    - ➢ We implemented matching service, more details below
- ❖ M3: Question service – responsible for maintaining a question repository indexed by criterias.
    - ➢ We implemented question service, more details below
- ❖ M4: Collaboration service – provides the mechanism for real-time collaboration between the authenticated and matched users in the collaborative space.
    - ➢ We implemented collaboration service, more details below
- ❖ M5: Basic UI for user interaction – to access the app that you develop.
    - ➢ We implemented the frontend for our application, consisting of login pages, dashboard pages, user management page, matching and collaboration pages.
- ❖ M6: Deploying the application.
    - ➢ We deployed on Google Cloud Platform. Application is running and can be accessed via this link..

2. Nice to have (Nx):

- ❖ N1: Communication: Implement a mechanism to facilitate communication among the participants in the collaborative space.
    - ➢ We implemented a messaging service in real time. The chat window is on the same page as the shared editor.
- ❖ N2: History: Maintain a record of the questions attempted by the user.
    - ➢ We implemented the functionality in the user service. The user service stores a user's previous attempts to a question, keeping only the latest saved attempt to a question at any given time.
- ❖ N3: Code execution: Implement a mechanism to execute attempted solution/code in a sandboxed environment, and retrieve+present the results in the collaborative workspace.

    - We integrate Judge0 Code Execution Service to execute code in the editor.
- N4: Enhance question service to enable managing questions, for example, tagging (by topic, popularity, etc.,), retrieving questions on the fly during a session initiation.
    - Questions in the question database are managed by topics and difficulty; We also implemented a refresh question feature during a session in collaboration service.
- N5: Enhance collaboration service by providing an improved code editor with code formatting, syntax highlighting for one language, syntax highlighting for multiple languages.
    - We integrate Monaco Editor to handle syntax format and highlight for multiple languages.
- N9: Deployment of the app on the production system (AWS/GCP cloud platform)
    - We deployed on Google Cloud Platform. Application is running and can be accessed via this link..

Subgroup	Nice to have features
Lee Wen Jun Tan Jian Wei	N1, N2, N3
Truong Minh Duong Anshumaan Tyagi Rehman Sajid	N4, N5, N9

Functional Requirements

The following sections detail the various Functional Requirements of Peer Prep, with each of the tables corresponding to a feature domain and each of the Functional Requirements prioritized based as follows:

Priorities	
High	Required to deliver on the purpose of the project
Medium	Required for a good user experience
Low	Implemented if there are excess resources

User Management

S/N	Description	Task	Priority
F1	User Management		
F1.1	User Creation		
F1.1.1	The application must allow Users to create accounts using an email and password.	M1	High
F1.1.2	The application must ensure that user accounts are created with a proper email format and a secure password of at least 8 characters.	M1	High
F1.2	User Session		
F1.2.1	The application must allow users to log in using their existing Peer Prep account.	M1	High

F1.2.2	The application must allow users to log out of Peer Prep from any page.	M1	High
F1.3	User Account Management		
F1.3.1	The application must allow users to change their password after logging in.	M1	High
F1.3.2	The application should have 2 roles (User and Admin) and be able to distinguish between the 2 roles.	M1	High
F1.3.3	The application must allow users to delete their account after logging in.	M1	Medium

Matching Service

S/N	Description	Task	Priority
F2	Matching Service		
F2.1	Matching with other users		
F2.1.1	The application must allow users to match with other users to collaborate on a random question.	M2	High
F2.1.2	The application must allow users to match with other users to collaborate on a question based on their selected difficulty and/or category.	M2	High
F2.1.3	The application must allow users to cancel their matching process, if they so wish to.	M2	High
F2.1.4	The application should time out the matching process if the user has been searching for a match for more than 1 minute.	M2	Low

Question Service

S/N	Description	Task	Priority
F3	Question Service		
F3.1	Question Bank		
F3.1.1	The application must show all questions to users.	M3	High
F3.1.2	The application must allow users to view question details of all questions.	M3	High
F3.1.3	The application should allow users to sort questions by their title and complexity.	N4	Medium
F3.2	Question Management		
F3.2.1	The application must allow administrators to create, edit and delete questions.	M3	High
F3.3	Question History		
F3.3.1	The application must retain a user's latest saved attempts at a question.	N2	High
F3.3.2	The application must allow users to view their previous attempts for a question.	N2	High
F3.3.3	The application must store all latest saved attempts in all the languages used for a user's attempt at the same question.	N2	Medium
F3.3.4	The application should allow a user to restore their previously saved code to a question they are currently working on.	M4, N2	Medium

Collaboration Service

S/N	Description	Task	Priority
F4	Collaboration Service		
F4.1	Pair Programming		
F4.1.1	The application should allow users to write code and view the other user’s code in real time.	M4	High
F4.1.2	The application should show the details of the question that the users are currently working on.	M4	High
F4.1.3	The application should allow users to choose their preferred programming language they wish to attempt the question in.	M4	High
F4.1.4	The application should provide programming-language specific suggestions and syntax highlighting, and code formatting.	N5	Medium
F4.1.4	The application should allow users to re-fetch a new question with the same user, allowing them to attempt a new question with the same user.	N4	Medium
F4.2	Code Execution		
F4.2.1	The application must be able to execute code written by the user.	N3	High
F4.2.2	The application must display to the user the result from the code execution.	N3	High
F4.3	Communication		
F4.3.1	The application should allow users to communicate in real time using text messages.	N1	High
F4.4	Performance		

F4.4.1	The application should store all chat messages exchanged between users.	N1	Medium
F4.4.2	The application should store all code written by users.	M4	Medium

Non-Functional Requirements

The following sections detail the various Non-Functional Requirements of Peer Prep, with each of the tables corresponding to the respective categories.

Performance

S/N	Description	Task
NF1	Performance	
NF1.1	Response Time	
NF1.1.1	The application should allow users to log in and log out within 2 seconds of pressing the respective buttons.	M1
NF1.1.2	The application should immediately reflect any changes to the question bank made by the administrator.	M3
NF1.1.3	The application should allow users to log in immediately after user account creation.	M1
NF1.1.4	Users should be able to view code changes made by the other user when collaborating within 1 second of the change being made on the other user's client.	M4
NF1.2	Throughput	
NF1.2.1	The application should be able to handle at least 1,000	

	concurrent requests to the backend API.	
NF1.3	Scalability	
NF1.3.1	The application should be able to scale up to accommodate high user traffic by increasing the number of pods active, within 10 minutes.	

Security

S/N	Description	Task
NF2	Security	
NF2.1	Authentication	
NF2.1.1	The application should authenticate users before granting them any access to the application.	M1
NF2.1.2	The application should check if a user has a valid session token from the client before executing any requests, and refetch a new session token if expired.	M1
NF2.1.3	The application should deny any access to the server without a valid session token.	M1
NF2.2	Authorization	
NF2.2.1	The application should restrict the ability for users to Create, Edit and Delete Questions if they are not an admin.	M1
NF2.3	Data Encryption	
NF2.3.1	The application should utilize HTTPS so that all communication between the client and the server is encrypted using TLS.	

Usability

S/N	Description	Task
NF3	Usability	
NF3.1	User Interface	
NF3.1.1	The application should allow the user to navigate to the main page from any page.	M5
NF3.1.2	The application should visually distinguish between normal text and buttons that a user can click on.	M5
NF3.1.3	The application should be responsive in its clicks, with each user action having a corresponding feedback within 1 second.	M5
NF3.1.4	The application’s feedback to the users should be clear and indicate clearly whether their action has been successful or resulted in an error.	M5
NF3.1.5	The application should ensure all components are adequately sized based on their purpose.	M5

Reliability

S/N	Description	Task
NF4	Reliability	
NF4.1	Availability	
NF4.1.1	The application should have an uptime of 99% at all times.	
NF4.1.2	The application should be able to be restarted or rolled-back to a previously working iteration within an	

	hour.	
NF4.2	Fault Tolerance	
NF4.2.1	The application should continue to function with minimal disruption in the event of a server failure.	
NF4.2.2	The application should automatically restart in the event of a server crash.	

Maintainability

S/N	Description	Task
NF5	Maintainability	
NF5.1	Code Maintainability	
NF5.1.1	Code should follow coding standards and be well-structured.	
NF5.1.2	Each section of code should be familiar to at least 2 developers at any given time.	
NF5.2	Modularity	
NF5.2.1	The application should be implemented with a microservice architecture.	
NF5.3	Deploying New Releases	
NF5.3.1	The application's individual components should be able to be deployed within 5 minutes of a new release of that respective component.	
NF5.3.2	The deployment of a new release of a microservice should be backwards compatible until all legacy dependencies are removed.	

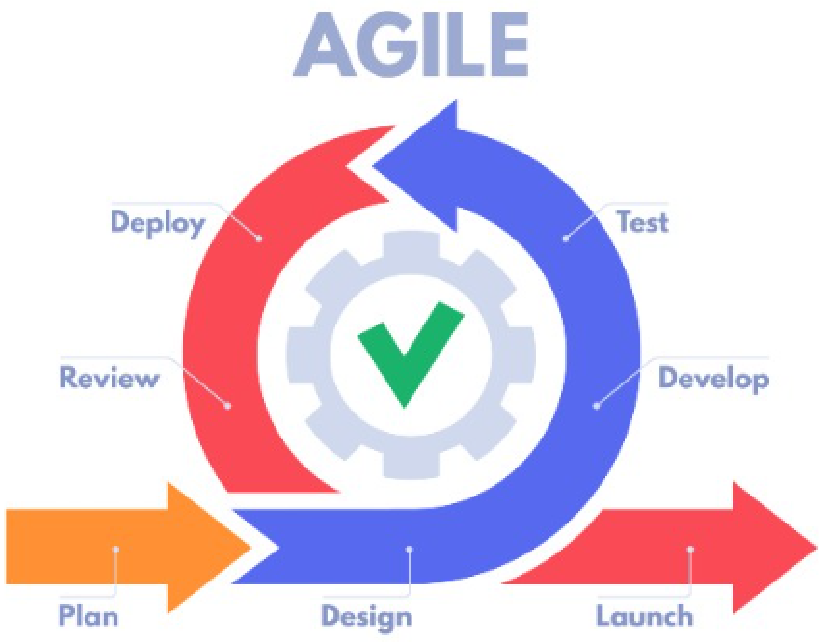
Software Development Process

GitHub

We used GitHub for our version control software. It allowed us to work individually on our own branches without interrupting the master and dev branch that we were using for deployment and development respectively. Branches would be merged into the dev branch through a pull request where we would look through each others code before merging them into the dev. To avoid large merge conflicts in the main development branch, we would make it a rule of thumb to merge from the dev branch to the feature branch first before merging into the dev branch again. This allowed us to resolve any conflicts we had with the dev and have cleaner pull requests.

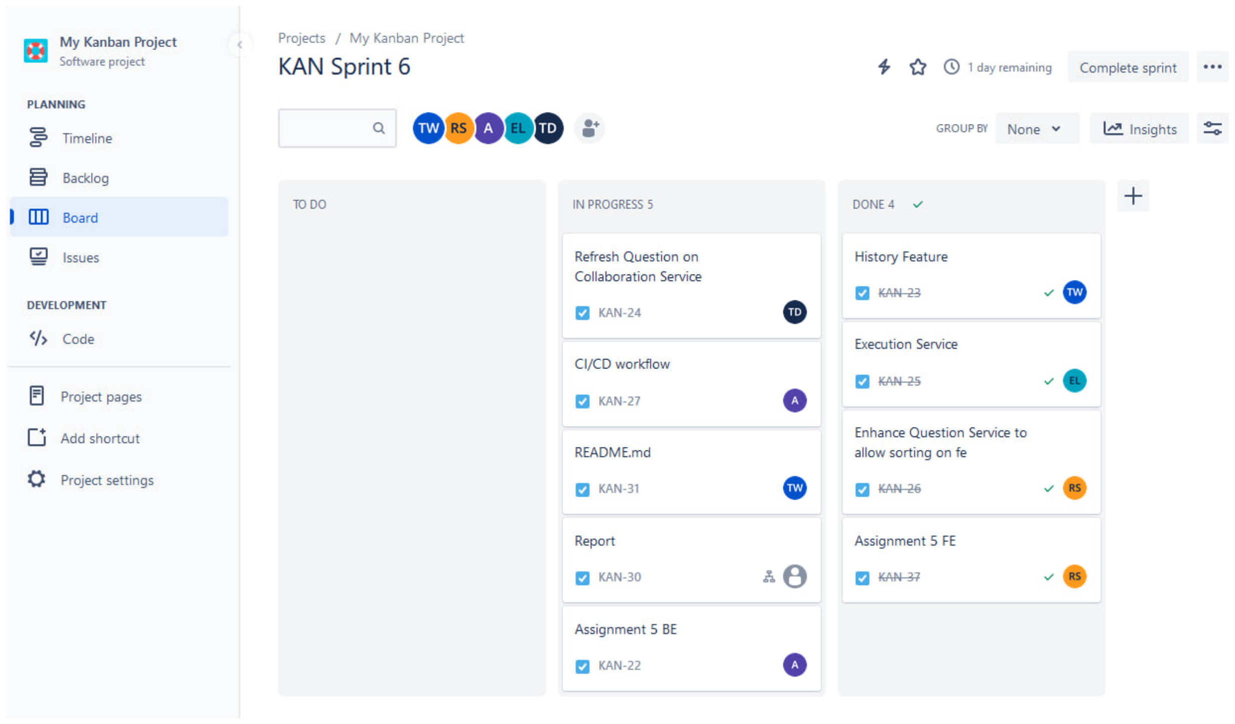
Additionally, GitHub also allowed us to tag our releases and manage our submissions easily.

Agile Workflow with JIRA



Agile Workflow

We employed an Agile Workflow enabled with JIRA. During our weekly meetings on Thursday at 10am, we would update what each of us will be implementing this week and what blockers we have faced and whether we need more assistance or time to deliver on a certain feature. We would also go over what our progress was like and what tasks we have yet to complete to hit the requirements for the project.



KAN Sprint 6

TO DO	IN PROGRESS 5	DONE 4
	Refresh Question on Collaboration Service — KAN-24 — TD	History Feature — ~~KAN-23~~ ✓ — TW
	CI/CD workflow — KAN-27 — A	Execution Service — ~~KAN-25~~ ✓ — EL
	README.md — KAN-31 — TW	Enhance Question Service to allow sorting on fe — ~~KAN-26~~ ✓ — RS
	Report — KAN-30	Assignment 5 FE — ~~KAN-37~~ ✓ — RS
	Assignment 5 BE — KAN-22 — A	

Our JIRA Board

During each meeting we would reference the JIRA Board to track which tasks have yet to be completed. We would also use the JIRA Board to track the progress of each member's tasks during the sprint.

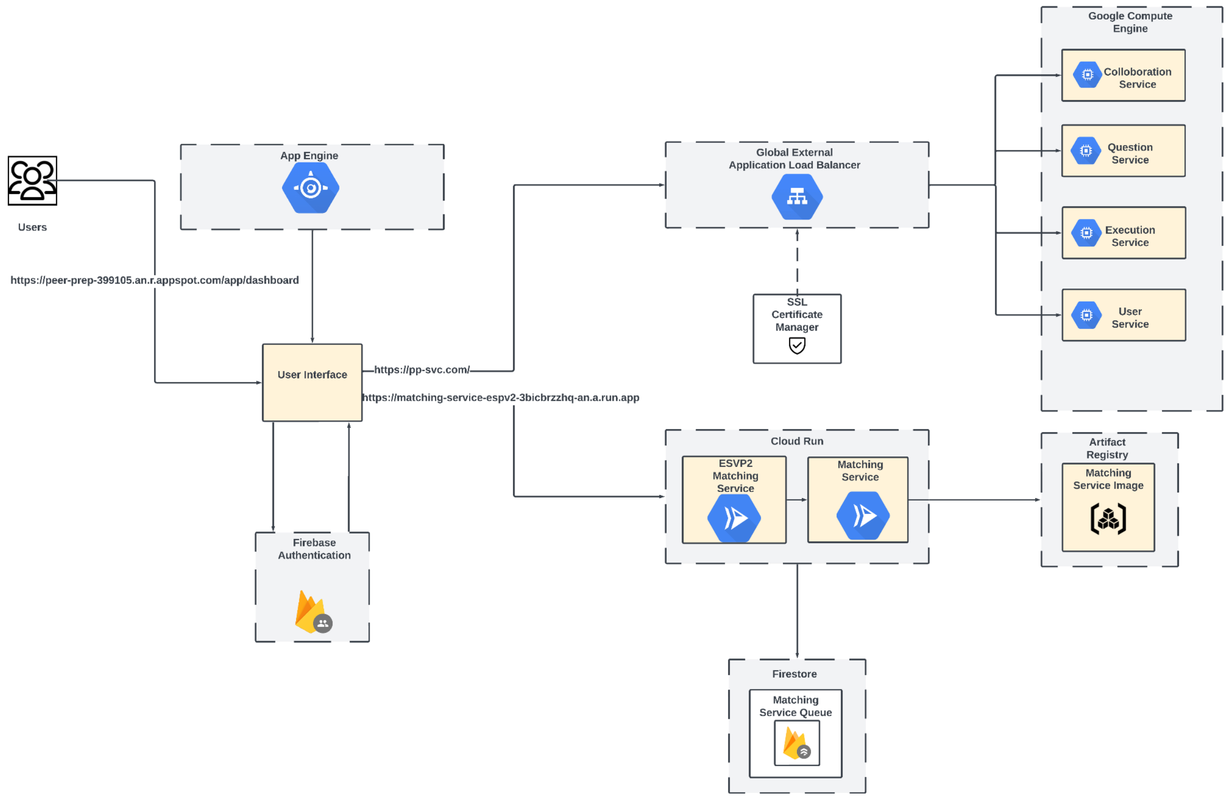
After each meeting, we would spend the rest of the time until the next meeting to develop and test the features implemented.

Application Design

Technology Stack

Frontend	React Material UI SASS Redux
Backend	Express Firebase Authentication Docker
Database	MongoDB Postgres Firestore
Servers	Google Cloud Platform

Architecture Diagram



Architecture Decisions

Microservices vs Monolithic

A microservices architecture design refers to the division of responsibilities in the application to ensure independence across functionalities. The client would interact with these microservices either through a gateway or individually. In our use case, we implemented them such that each microservice has a load balancer associated with it and can scale up horizontally by spawning more pods.

On the other hand, a monolithic architecture design is the traditional design pattern where a single server is responsible for all functionalities, consolidating the features of the application into a single API server. As opposed to the microservices architecture, the monolithic architecture is scaled vertically by increasing CPU and RAM on the server itself.

	Microservices Architecture	Monolithic Architecture
Benefits	<u>**Scalability**</u> Improved scalability, as each microservice can be scaled independently based on demand. <u>**Flexibility**</u> Allows for flexibility in choosing different technologies and tools for different microservices, promoting innovation. Isolation: Fault isolation; issues in one microservice do not affect others. <u>**Autonomous Development:**</u> Enables autonomous development and deployment of individual microservices, enhancing agility. <u>**Team Autonomy**</u> Teams can work independently on different microservices, promoting autonomy.	<u>**Simplicity**</u> Easier to develop, test, and deploy as the entire application is contained within a single codebase. Simplifies development and maintenance. <u>**Performance**</u> Can be more efficient in terms of performance, as there's no overhead of inter-service communication. Development Speed: Faster development time as developers work on a single codebase and have a clear understanding of the entire system. <u>**Ease of Debugging**</u> Easier to debug and trace issues as the entire application is a single unit. Database Transactions: Easier to manage database transactions, as they can be done within a single database.

Potential Issues	**Complexity** Increased complexity in terms of inter-service communication, monitoring, and management. **Latency** Potential for increased latency due to inter-service communication. Operational Overhead: Greater operational overhead, including managing deployments, monitoring, and debugging in a distributed environment. **Consistency Challenges** Ensuring data consistency and transaction management across microservices can be challenging. **Testing Complexity** Testing becomes more complex due to the need for integration testing.	**Scalability** Limited scalability, as the entire application must be scaled horizontally, even if only one component requires more resources. **Flexibility** Less flexibility in terms of technology stack and development tools, as all components must align. Longer Deployment Cycles: Longer deployment cycles, as the entire application needs to be deployed, lead to potential downtime. **Maintainability** Can become difficult to maintain and update as the application grows.

Decision

Decided to use Microservice Architecture.

Justification

We chose this architecture because it allowed for easier distribution of work as we can work on separate services in parallel with minimal dependency and merge conflicts.

Firestore Database vs RabbitMQ (Anshumaan)

Firestore is a NoSQL document database within Google Cloud Platform's Firebase. It organizes data into documents stored in collections, offering real-time updates, scalability, and serverless architecture. Notable features include offline support, security rules for access control, and easy integration with Firebase Authentication. For our use case, we would use it to store unmatched users and matched users, basically the database would house our queues.

RabbitMQ is a message broker software that facilitates communication between distributed systems. It acts as a middleman, enabling different parts of a system to communicate by passing messages. RabbitMQ supports various messaging patterns, including point-to-point and publish/subscribe, providing a reliable and scalable solution for decoupling components in a distributed architecture. It's widely used in applications where different services or components need to exchange information asynchronously. For our use case we would need different queues organized by topic and difficulty, and these queues would be linked to different consumers that match unmatched users.

Potential Issues

Issues with RabbitMQ:

Difficult to Deploy: Deploying RabbitMQ can be challenging for some users, especially those new to message broker systems. Configuration and setup may require careful consideration.

Requires In-Memory Storage: RabbitMQ primarily uses in-memory storage for message queues. While this can enhance performance, it may also limit the system's ability to handle very large message volumes or maintain messages during system outages.

Adds Complexity: Introducing RabbitMQ into a system adds complexity, particularly in terms of managing message queues, ensuring message delivery, and handling potential failures.

Not Scalable: As we will need to use VMs to deploy RabbitMQ it would make scalability even harder to Configure.

Harder to Filter and Search Queues: Filtering and searching queues in RabbitMQ requires additional effort or may not be as straightforward compared to other systems.

Issues with Firestore:

Slower than RabbitMQ as it Uses Cloud Storage: Firestore, being a cloud-based database, might introduce some latency compared to the in-memory storage used by RabbitMQ, potentially leading to slower performance in certain scenarios.

More Expensive for Large-Scale Applications: Firestore's pricing structure may become a concern for large-scale applications, as costs can increase with higher usage and storage requirements.

Decision

We decided to use Firestore for our matching queues.

Justification

Choosing Firestore for our application is a strategic decision based on several advantages. Firestore offers ease of implementation and deployment, providing a user-friendly development experience. Its seamless integration with the Google Cloud environment ensures a cohesive and well-supported development ecosystem. Utilizing Firestore authentication enhances security, ensuring controlled access to the database. The compatibility with cloud-run containers simplifies accessibility, promoting a streamlined deployment process. Firestore's capabilities for straightforward filtering and querying contribute to efficient data management. Additionally, Firestore's support for atomicity in data retrieval lays the foundation for scalability, allowing our application to grow seamlessly as needed. Considering our current deployment scale, Firestore proves to be a cost-efficient solution, aligning well with our application's present requirements and future scalability considerations.

Socket.IO vs Cloud Pub/Sub

Cloud Pub/Sub is an asynchronous stateless messaging service that allows for the subscription and publishing of messages. It allows our users to communicate asynchronously with rapid response times. We initially considered this to be used in our collaboration service where each user would collaborate by sending each other asynchronous updates to each other when they make any changes on their client.

Socket.IO is a stateful real-time communication library built on the WebSocket protocol, facilitating bi-directional and event-driven interaction between clients and servers. We considered this in our implementation as it would allow us to maintain a room state on our servers and would allow users to interact with the room state via their respective clients.

Potential Issues

Cloud Pub/Sub

Since it is an asynchronous messaging system, it is stateless and it would make it difficult to retain the state of the room after users decide to stop collaborating. It would also make it difficult to restore a user's current session if they were to disconnect from the room without additional guards such as manually setting up a way to store and retrieve the previously saved room state before their disconnection.

There could also be issues with latency as messages first have to be published before they are delivered to subscribers. In a collaborative code environment, speed is important, so keeping latency to a minimum is essential in ensuring a smooth user experience.

There can also be issues with managing the connection between users. Given that some questions might be very challenging and users might take a while to finish, prolonged connections might become difficult for Cloud Pub/Sub to manage.

Socket.IO

As the Redis database that Socket.IO uses to store the room state is locally hosted within the Compute Engine server instance, there is a scalability issue. We are unable to spawn more instances if there is an overload of users so we would need to scale the instance vertically instead. Another solution would be to run a Redis database externally on Google Cloud Platform itself but due to cost issues, we decided against that for now.

Additionally, since Socket.IO only manages the events, we would need to manage the connection manually. This refers to needing to account for disconnections and reconnections as well as concurrency issues when it comes down to modifying the Redis database.

Decision

Ultimately, we chose to go with Socket.IO

Justification

For our use case and project in its current form, scalability was not something that was of utmost concern for us. Additionally, the benefits of having the room state stored far outweighed the scalability brought by Cloud Pub/Sub. Users would be able to disconnect from the application and easily pull the latest room state without any complex systems, reducing any overhead. Additionally, Socket.IO provided high performance and low latency, allowing our users to view code changes in real-time.

Compute Engine vs Cloud run

Feature	Cloud run	Compute Engine
Type of Service	Containers as a Service(Caas)	Infrastructure as a Service
Flexibility and Control	Cloud run abstracts away infrastructure details, offering less control but simplifying deployment.	Compute Engine provides complete control over Vms, allowing for customization of the operating system, software, networking
Scaling	Automatic scaling based on incoming request traffic, scales down to zero when idle.	Manual scaling or managed auto-scaler configurations, providing more control over scaling.
Deployment	Simplified deployment for containerized applications using containers.	Highly customizable VM instances, allowing flexibility in deployment configurations.

	Generates SSL certified URLs.	Does not Generate SSL certified urls.
Use Cases	Ideal for variable workloads, microservices architectures, and event-driven applications.	Well-suited for applications with specific hardware requirements, traditional monolithic applications, and when detailed infrastructure control is necessary.
Management	Fully managed environment, minimal manual intervention, and automated scaling.	Requires manual management of VMs, allowing for hands-on control over configurations.
Billing	Pay per request and resource consumption scales down to zero when idle for cost optimization.	Pay for reserved VMs or on-demand usage, providing more predictable costs but requiring careful planning for cost efficiency.

Decision

We decided to use both cloud run and compute engines.

Justification

We used cloud run for completely cloud-deployed services such as matching service. The matching service hosts a REST API to add users to a queue (a cloud-deployed database) and to find matches from a collection (A cloud-deployed database) so it does not require extreme control over hardware configurations. Deploying matching service on cloud run allows for simpler deployments and integration as cloud run instances are SSL certified, thus not requiring a load balancer to access the instance. Not to mention , these instances are automatically scaled thus again rendering the use of external resources (such as a load balancer) to manage the instance futile.

We used a compute engine for hardware requiring services such as User Services(in-memory database) , collaboration service (websocket needs stateful containers), and questions service (database requirements). Deploying these services as a VM allowed for easier deployment as otherwise they would need extra external resources for eg. cloud databases to be deployed as cloud-run instances. These databases would also add to our cost thus making it cost inefficient. As these VM instances are managed by a load balancer, our application could be made scalable in future iterations.

Design Patterns

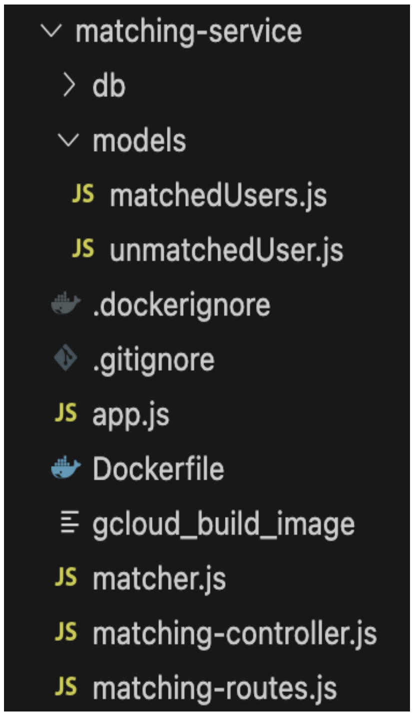
Model-View-Controller

We used MVC pattern for our services such as questions service and matching service to decouple sub-components with different responsibilities.

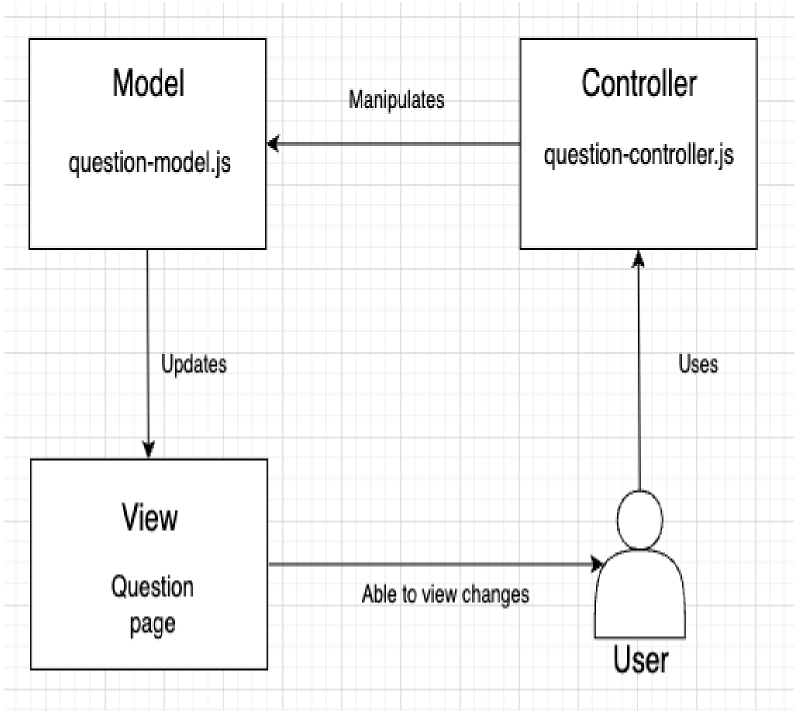
Model: Represents the application's data and business logic. Our model layer for each service was in the form of schemas that helped to interact with our database.

View: Is responsible for presenting the data to the user and capturing user input. It displays information from the Model and sends user commands (e.g., button clicks, form submissions) to the Controller for processing. Our View layer is within the frontend.

Controller: The Controller acts as an intermediary between the Model and the View. It receives user input from the View, processes it (typically involving interactions with the Model), and updates the Model or the View accordingly.



```
matching-service
  db
  models
    matchedUsers.js
    unmatchedUser.js
  .dockerignore
  .gitignore
  app.js
  Dockerfile
  gcloud_build_image
  matcher.js
  matching-controller.js
  matching-routes.js
```

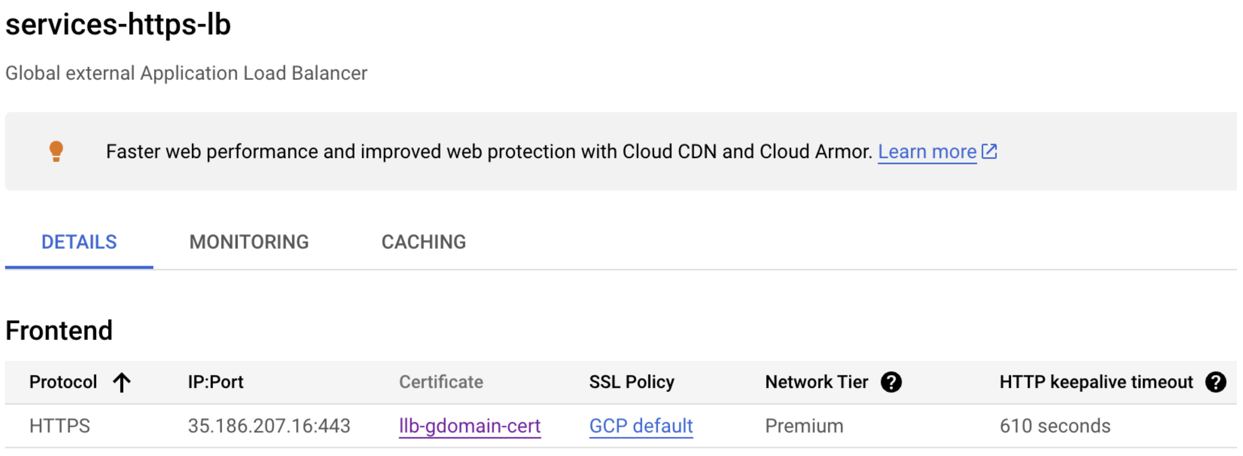


Benefits	Details
Decoupling	MVC promotes decoupling between the components, allowing each part of the application to evolve independently. Changes in one component (e.g: Model) do not directly affect the others (e.g:View or Controller).
Maintainability	The separation of concerns in MVC makes the codebase more maintainable. Our team can work on one component without affecting the others, making it easier to debug, test, and enhance the application.

Load Balancer

We implemented a Global External Load Balancer within the Google Cloud Platform to manage the routing of requests. Given that our Virtual Machine instances on the Google Compute Engine lack SSL certificates, we required a gateway for our HTTPS frontend client to access these instances. The Load Balancer essentially functions as a router, directing requests based on the URL path.

To configure the load balancer, we purchased a domain through Google Cloud and utilized GCP's managed SSL certificates to secure certification for the domain.



services-https-lb

Global external Application Load Balancer

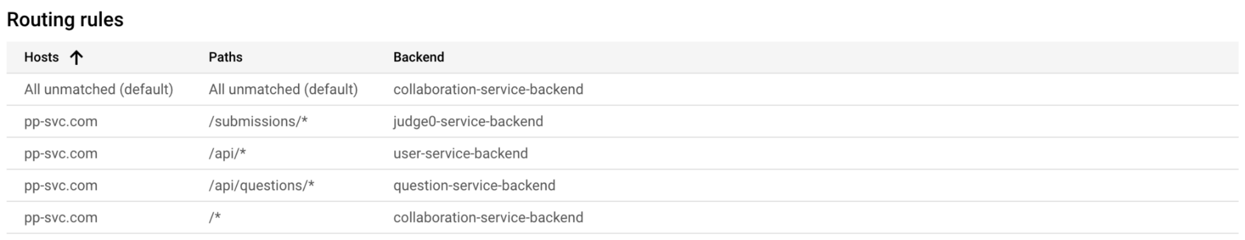
Faster web performance and improved web protection with Cloud CDN and Cloud Armor. Learn more

Frontend

Protocol	IP:Port	Certificate	SSL Policy	Network Tier	HTTP keepalive timeout
HTTPS	35.186.207.16:443	llb-gdomain-cert	GCP default	Premium	610 seconds

Load Balancer Frontend

Routes



Routing rules

Hosts ↑	Paths	Backend
All unmatched (default)	All unmatched (default)	collaboration-service-backend
pp-svc.com	/submissions/*	judge0-service-backend
pp-svc.com	/api/*	user-service-backend
pp-svc.com	/api/questions/*	question-service-backend
pp-svc.com	/*	collaboration-service-backend

Load Balancer Routes

Routing URL	Backend Services
pp-svc.com/*	Collaboration Service
pp-svc.com/submissions/*	Execution Service
pp-svc.com/api/*	User Service
pp-svc.com/api/questions/*	Questions Service

Frontend

Frameworks

Vite.js + React

The choice of React came naturally as we had experience working with React and it would be easier to work with a technology stack that we were already familiar with rather than starting from scratch.

Choosing to use Vite.js in conjunction with React was because of its speed and modern build setup, providing significant performance benefits during development. The instantaneous server start and rapid hot reloads make the development workflow notably faster in comparison to traditional bundlers such as Webpack.

When integrating Vite.js into React, it takes advantage of React’s declarative and component-based architecture, allowing for a significantly cleaner and maintainable codebase. It also works well with React being that it is modular and aligns with React’s component-based nature, resulting in a streamlined development process.

When integrated with React, Vite.js takes advantage of React's declarative and component-based architecture, allowing for a clean and maintainable codebase. The seamless integration of Vite.js with React facilitates a quick and painless setup, enabling developers to focus more on building features and less on configuration.

Notable Packages

Monaco Editor

Monaco Editor is a powerful, open-source code editor developed by Microsoft. It is the same editor that powers Visual Studio Code and is designed to be embedded in web applications. Monaco Editor provides a rich set of features, including syntax highlighting, autocompletion, code navigation, and more.

Firebase React Hooks

Firebase React Hooks is a set of React hooks provided by the Firebase JavaScript SDK. Firebase is a platform developed by Google for building web and mobile applications, offering various services like authentication, real-time database, storage, and more. The Firebase React Hooks simplify the integration of Firebase services into React applications.

RTK Query

RTK Query is part of the Redux Toolkit, a state management library for React applications. RTK Query simplifies data fetching, caching, and state management by providing a set of tools and abstractions. It is designed to work well with Redux, but it can also be used independently.

Design Considerations

Design considerations are the thoughtful and strategic decisions made during the planning and development phases of a project to ensure that the end product meets specific goals and standards. These considerations encompass a wide range of factors that collectively contribute to the overall success and effectiveness of a design. Key elements include user experience, scalability, security, and integration with other components.

User-Centric Design:
Prioritizing the needs and preferences of the end-user to create an interface that is intuitive, accessible, and visually appealing.

Scalability:
Designing a system that can seamlessly grow and adapt to accommodate increased demands, new features, and evolving requirements.

Cross-Browser Compatibility:
Ensuring consistent functionality and appearance across various web browsers, minimizing compatibility issues.

Security:
Implementing measures to safeguard user data, protect against vulnerabilities, and ensure secure communication with backend services.

Integration with Backend Services:
Establishing effective communication and data exchange between the frontend and backend components of a system.

Feedback and Notifications:
Keeping users informed about system status, errors, and updates through clear and concise feedback mechanisms.

Microservices

User Service

The User Service can be broken into the User Service microservice and the Firebase Authentication that we employ.

Firebase Authentication

Peer Prep utilizes Firebase's Authentication system to provide session management. It provides us with a mechanism to use a user's email and password to log into the application. In our use case, a simple email + password system was sufficient to authenticate users. Additionally, with Firebase Authentication automatically handling the password encryption and authentication flow, it improved the overall security of our application.

Additionally, Firebase Authentication also provided us with session tokens that we could use to authenticate calls to our backend servers and access to the application on the front end.

Firebase Authentication also has integrated features such as password reset and account deletion which further enhanced security and reduced our workload by eliminating the need for manual intervention.

Users

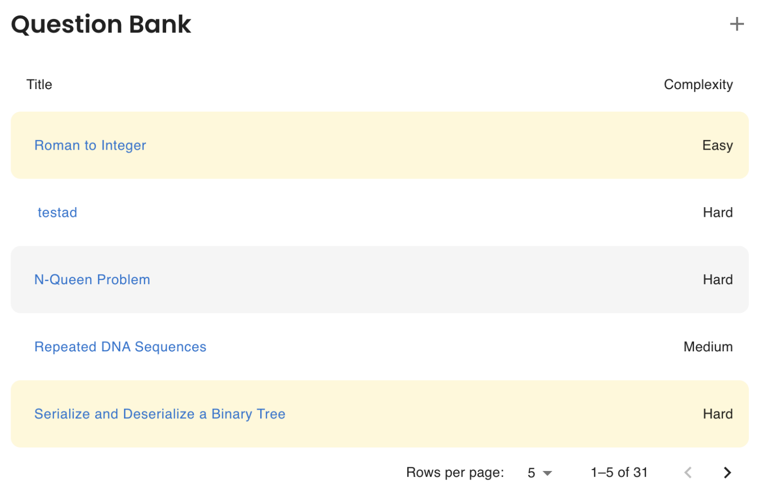
The User Service is responsible for tracking Users and stores their Firebase ID and the Email they use to log in. Whenever a User Account is created in Firebase, we also keep a store a record in our database of this account.

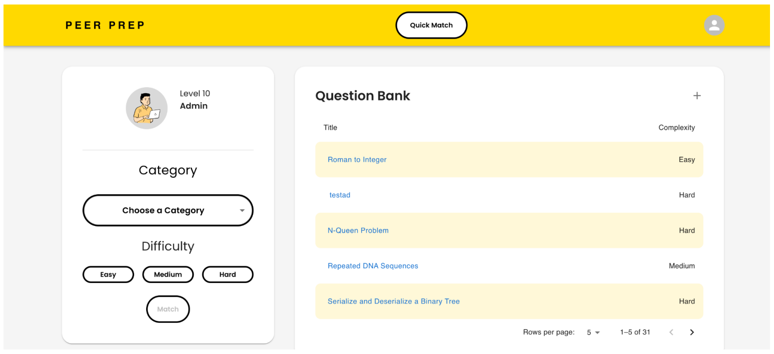
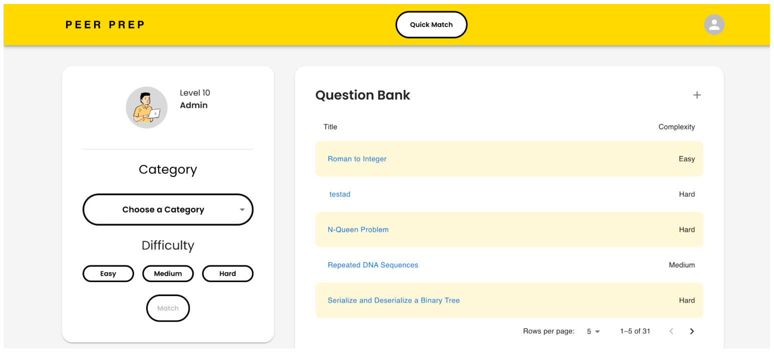
History

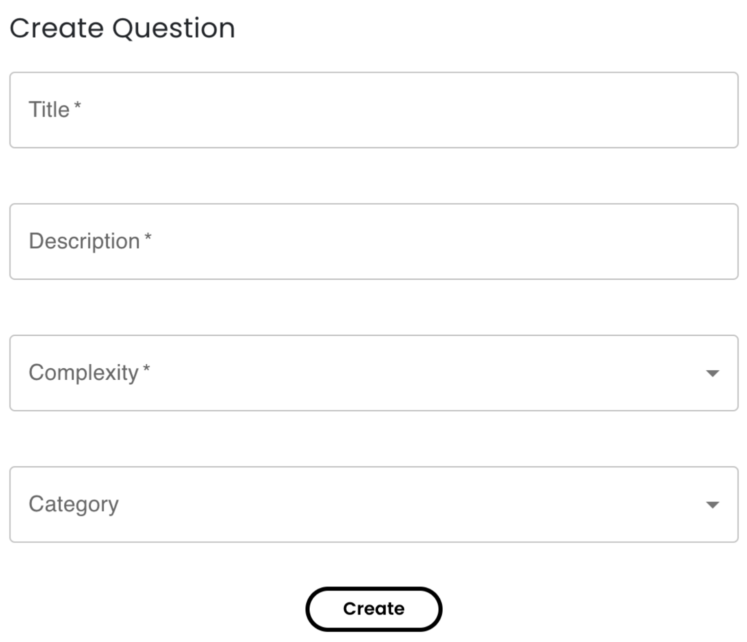
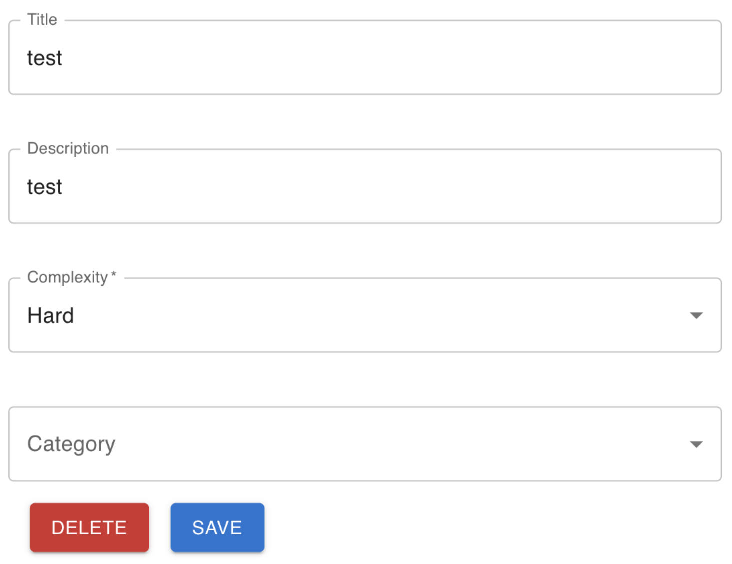
The User Service also stores a user's previous attempts to a question, keeping only the latest saved attempt to a question at any given time. The attempts are uniquely identified by the language of choice, the question, and the user's Firebase ID. This allows a user to save individual attempts for different languages for the same question.

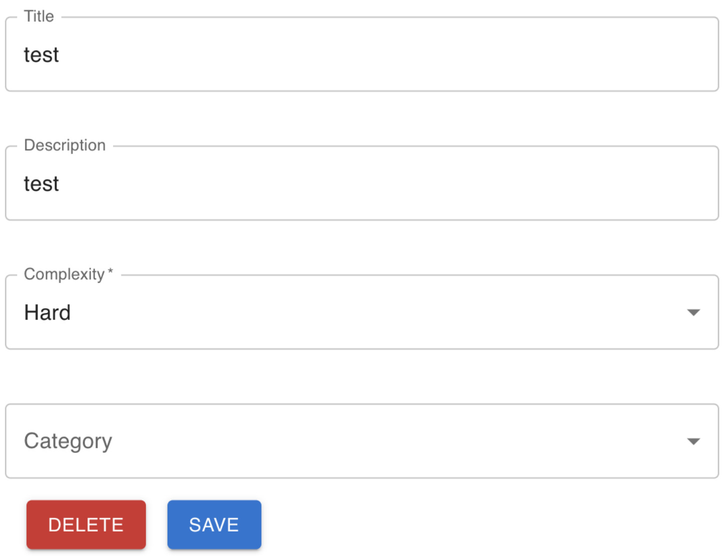
Question Service

The question service is responsible for managing the questions stored in the MongoDB database. The endpoints in the table below allows the frontend to interact with the database.

Endpoint	Functionality
GET /questions	Enables frontend dashboard to fetch, render and sort all the questions.  **Question Bank** + Title · Complexity Roman to Integer · Easy testad · Hard N-Queen Problem · Hard Repeated DNA Sequences · Medium Serialize and Deserialize a Binary Tree · Hard Rows per page: 5 · 1–5 of 31
GET /questions/random	Used when the user chooses to Quick Match rather than matching by category and difficulty.

	 PEER PREP Quick Match Level 10 Admin Category Choose a Category Difficulty Easy Medium Hard Match Question Bank Title — Complexity Roman to Integer — Easy testad — Hard N-Queen Problem — Hard Repeated DNA Sequences — Medium Serialize and Deserialize a Binary Tree — Hard Rows per page: 5 1–5 of 31
GET /questions/random-filtered	Used when the user chooses to match by category and complexity rather than Quick Match. Provides frontend with a randomly selected question based on filters of category and difficulty selected.  PEER PREP Quick Match Level 10 Admin Category Choose a Category Difficulty Easy Medium Hard Match Question Bank Title — Complexity Roman to Integer — Easy testad — Hard N-Queen Problem — Hard Repeated DNA Sequences — Medium Serialize and Deserialize a Binary Tree — Hard Rows per page: 5 1–5 of 31
POST /questions/new	Allows for creating of new questions through this form. The question details are sent over the request body.

	 Create Question Title * Description * Complexity * Category Create
DELETE /questions/:id	Allows for deletion of questions from the question bank stored in our database based on question id.  Title: test Description: test Complexity *: Hard
PUT /questions/:id	Updates question based on request body. Endpoint is called after question details are edited and SAVE

	button is clicked.  Title test Description test Complexity * Hard Category DELETE SAVE

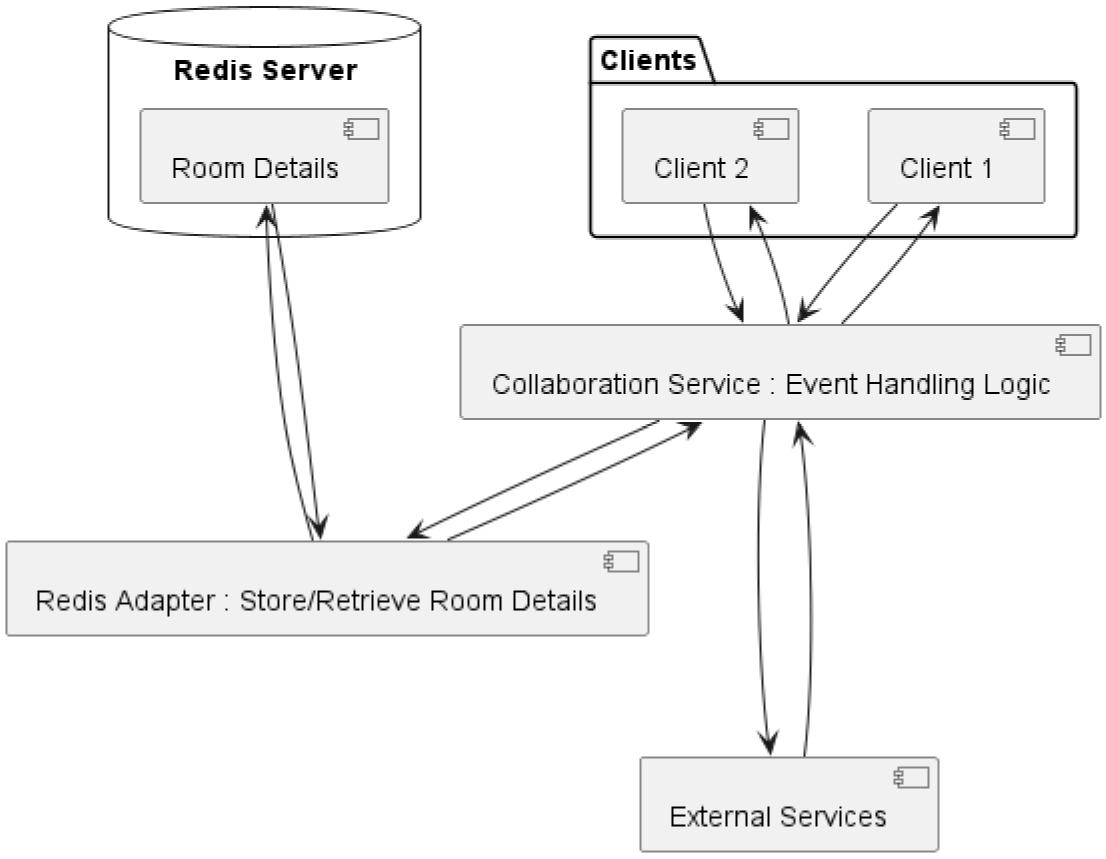
Collaboration Service

The frontend collaboration page integrates three essential sections: Question Section, Code Editor Section, and Chat + Terminal Section.

Question Section	The Question Section serves as the foundation for collaboration by providing users with a randomly generated technical interview question. This section includes: 1. Question Title: Clearly states the question's title. 2. Difficulty Level: Indicates the difficulty level of the question, allowing users to gauge the complexity of the problem.

	3. Description: Offers a detailed description of the problem, guiding users on the specific requirements and constraints.
Code Editor Section	The Code Editor Section is where collaborative coding takes place. It features code editors, allowing simultaneous editing and real-time collaboration. This section includes: 1. User’s code editor: Displays the code editor for the user, enabling them to write, modify, and execute code. 2. Partner's code editor: Displays the code editor for the user's partner with read-only code.
Chat + Terminal section	The Chat + Terminal Section is a dynamic space that combines communication and feedback during collaboration. This section includes: 1. Chat Feature: Enables users to communicate with their partner, discuss strategies, share insights, and seek help. 2. Terminal Output: Displays the user’s terminal output, allowing them to view the results of their code execution.

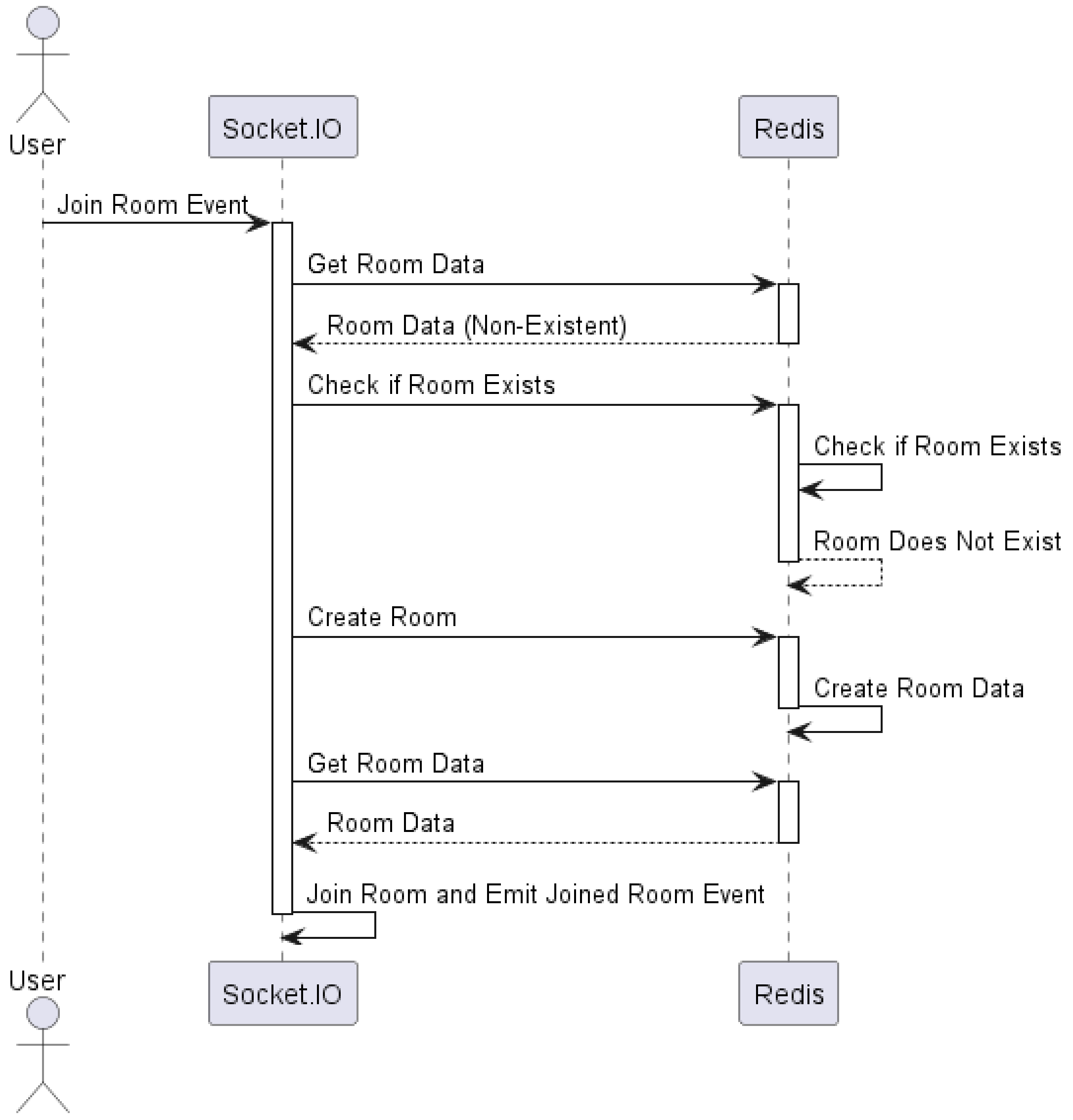
The backend collaboration service includes SocketIO server and Redis database. We choose Redis because of speed and low-latency data access.


Redis Server
Room Details
Clients
Client 2
Client 1
Collaboration Service : Event Handling Logic
Redis Adapter : Store/Retrieve Room Details
External Services

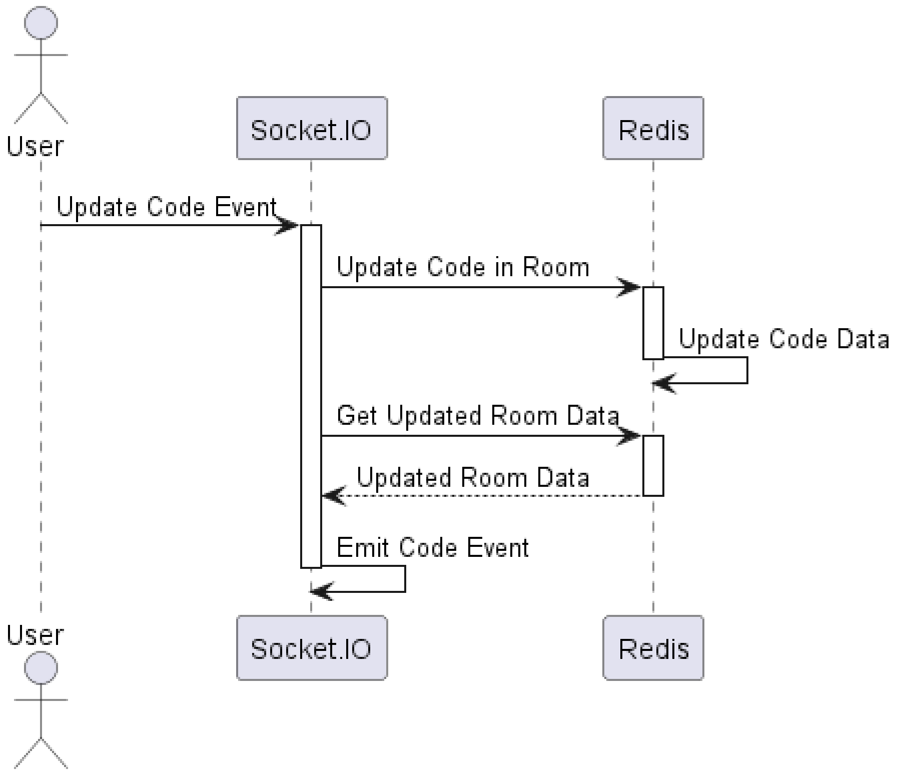
There are three groups of functionality:

1. Room management: including creating, joining room and retrieving room details.

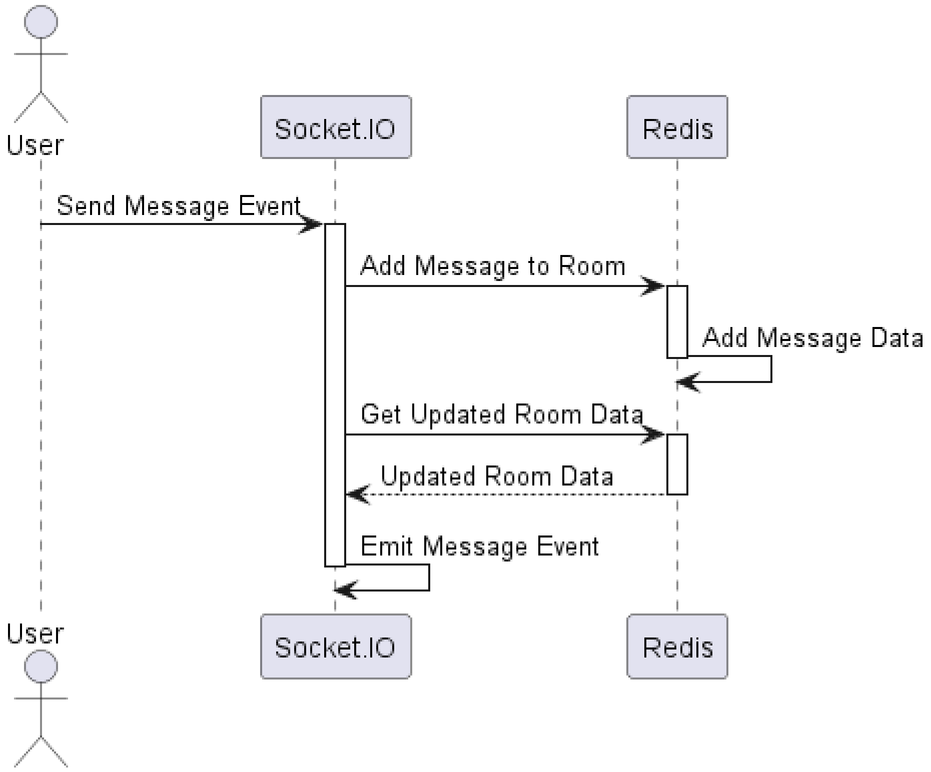
   Below is a sequence diagram when user joining a new room:



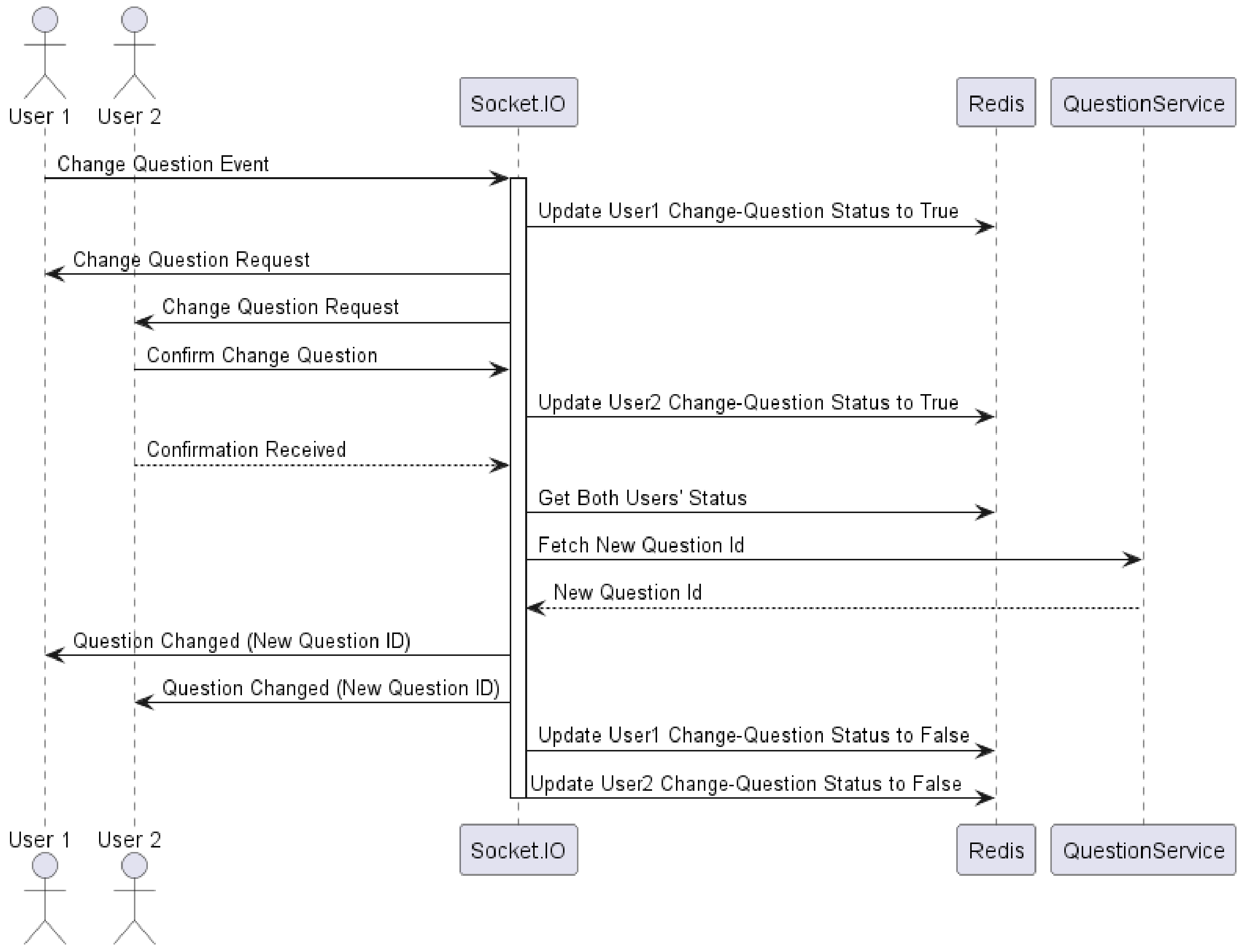
2. Code and message management: including updating code and message in the room details.
   Below are two sequence diagram of updating code and message:
   Updating code:



Updating message:



3. Refresh question management: including handling requests for changing questions, confirming the change and canceling the change.
   Below is a sequence diagram when both user agree to change to new question:


User 1
User 2
Socket.IO
Redis
QuestionService
Change Question Event
Update User1 Change-Question Status to True
Change Question Request
Change Question Request
Confirm Change Question
Update User2 Change-Question Status to True
Confirmation Received
Get Both Users' Status
Fetch New Question Id
New Question Id
Question Changed (New Question ID)
Question Changed (New Question ID)
Update User1 Change-Question Status to False
Update User2 Change-Question Status to False

   Currently, new questions will be fetched randomly (and different from the old question). We choose this design because random selection encourages users to explore a wider range of content.

Code Execution Service

The Code Execution Service is a crucial component linked to the code editor in the front end, providing users with the ability to execute their code and receive real-time feedback on its performance. This service leverages on an open-source online code execution system known as judge0. Below is the execution workflow.

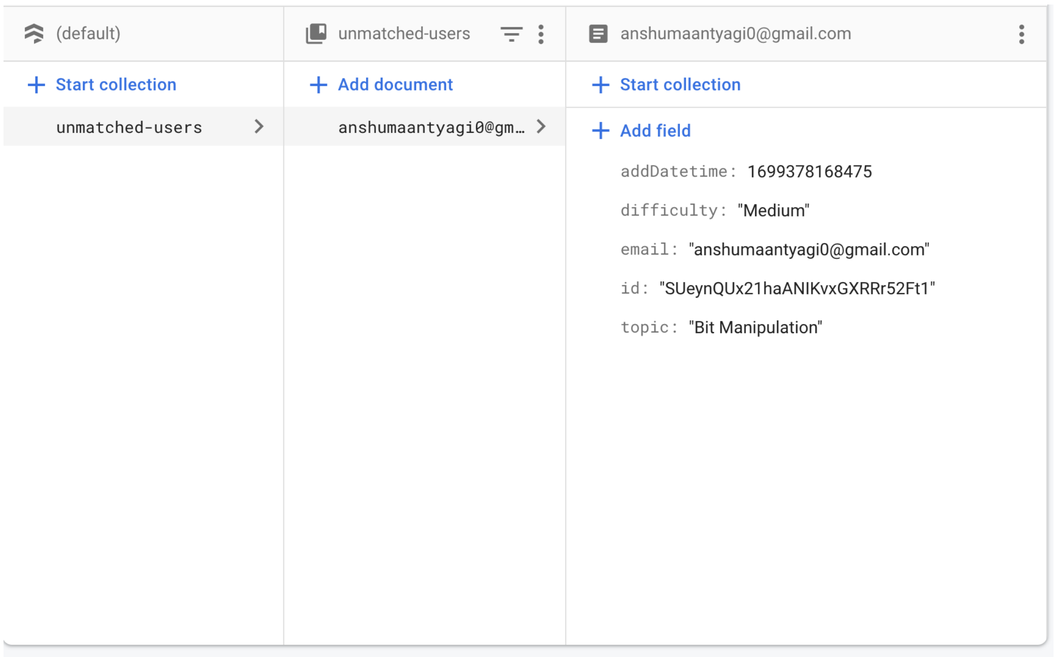
Process	Description
Code Execution Trigger	Upon completing their code in the code editor, users can initiate the execution process by clicking on a "Submit" button.
API Integration with Judge0	The trigger event sends the user's code to an API endpoint provided by judge0
Submission Token Retrieval	Upon successful submission, a unique submission token is generated and returned to be used as a reference to retrieve the execution status and output.
In Queue Status	Initially, the status of the code execution is set to "In Queue" as the submitted code is being processed by the judge0 system.
Status Check Loop	The execution service continuously polls the judge0 API using the submission token to check the status of the code execution.
Output Retrieval	Once the status transitions to "Accepted," indicating that the code execution is complete, the execution service retrieves the output generated by the executed code.
Display in Terminal Output	The obtained output is then displayed in the Terminal Output section in frontend, allowing users to review and analyze the results of their code execution.

Matching Service

The purpose of Matching Services is to match users based on their chosen topic of preference and difficulty.

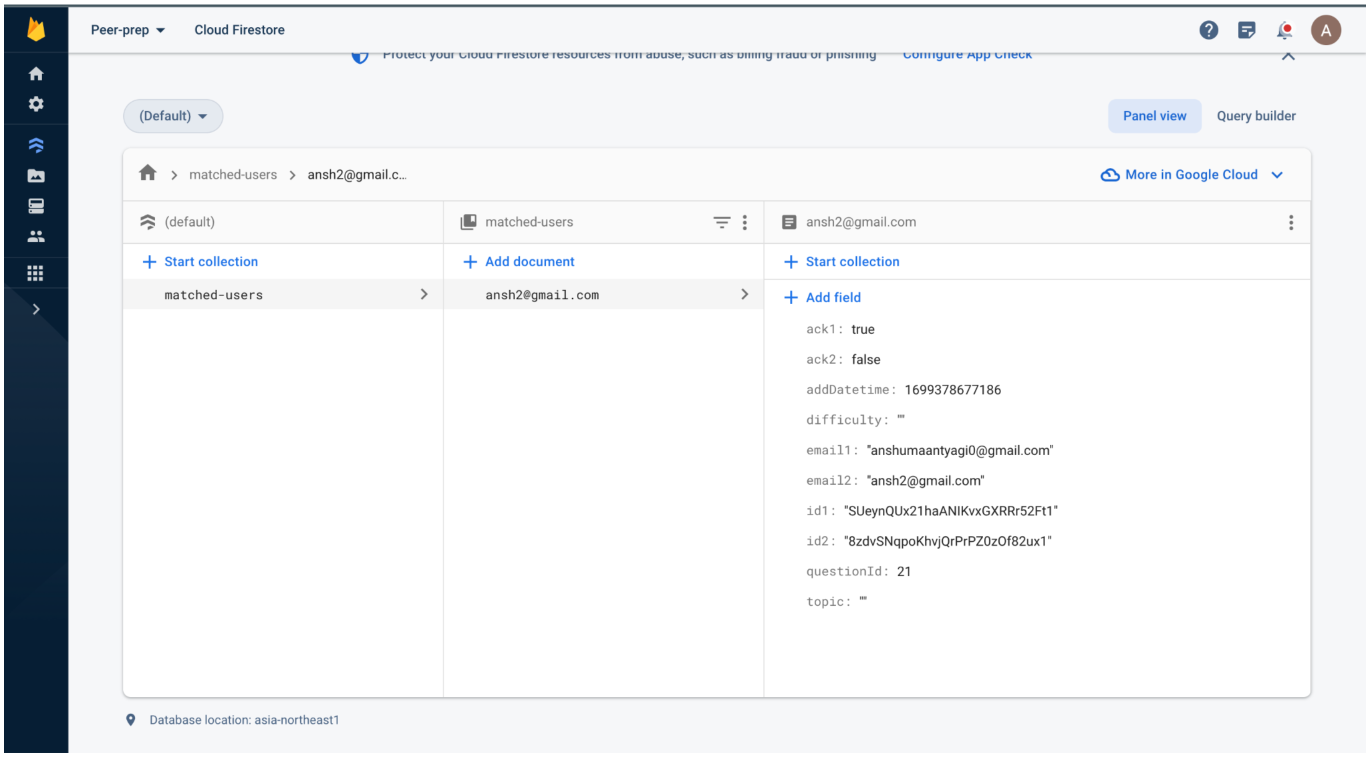
Firestore

The queue for Unmatched users and matched users is stored in firestore. Each time a new user wants to match , the user is added to the unmatched queue. After two users with the same topic and difficulty are added to the queue , they are matched and the match information is stored in the matched collection (stores matches) and both matched user's information is deleted from the unmatched queue to accommodate future matches for the users.
From here, these matches can be queried by clients by referencing the user's email in the request body. Once requests to query both users are made to this collection, the matched information is deleted to accommodate storing future matches for both users.



```
addDatetime: 1699378168475
difficulty: "Medium"
email: "anshumaantyagi0@gmail.com"
id: "SUeynQUx21haANIKvxGXRRr52Ft1"
topic: "Bit Manipulation"
```

Unmatch Queue before matching users



```
ack1: true
ack2: false
addDatetime: 1699378677186
difficulty: ""
email1: "anshumaantyagi0@gmail.com"
email2: "ansh2@gmail.com"
id1: "SUeynQUx21haANIKvxGXRRr52Ft1"
id2: "8zdvSNqpoKhvjQrPrPZ0zOf82ux1"
questionId: 21
topic: ""
```

A Match stored in Matched Collection

API Endpoints

Endpoint	Functionality
POST /find-match	Store’s User Information inside the unmatched queue if no suitable match. If a suitable match is found, the oldest valid match is selected and both user's information is stored in the matched collection. Returns successful on successful addition to the queue.

POST /check-match	Queries the matched collection to check if any valid matches for a specific user (supplied in the request body) exist. Returns successful the matched user if a user is found otherwise null.
GET /remove-user	Queries both unmatched queue and matched collection to check if any record of the user exists and if it does, the record is deleted. Its purpose is to handle time-out or match cancellation requests from the frontend's side. Returns successful on successful removal of the user's data.

Other Important Points

Valid Unmatched Users

A user in the queue for less than or equal to a minute. (Frontend time out time).

Valid Matches

Matches that have existed in the collection for less than two minutes. (2 * Front end time out time)

Atomicity

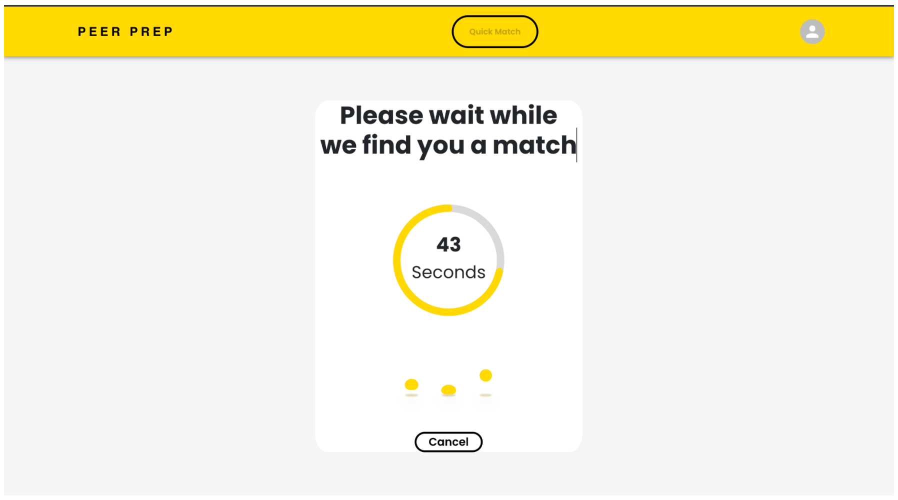
To disallow concurrent matches , whenever a matching service instance accesses the firestore database , the database is locked, meaning other requests from other clients have to wait until that transaction is completed. This allows for a more robust and scalable matching service.

Docker

The docker image of the service is deployed on the artifact registry. The cloud run instance uses the docker image stored in the artifact registry to run a container to handle client requests.

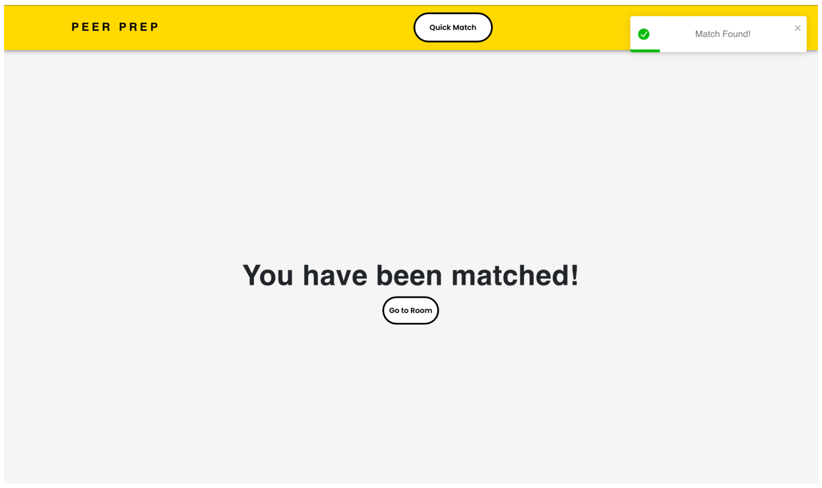
Frontend

A timer is displayed notifying the search for matches.



Timer

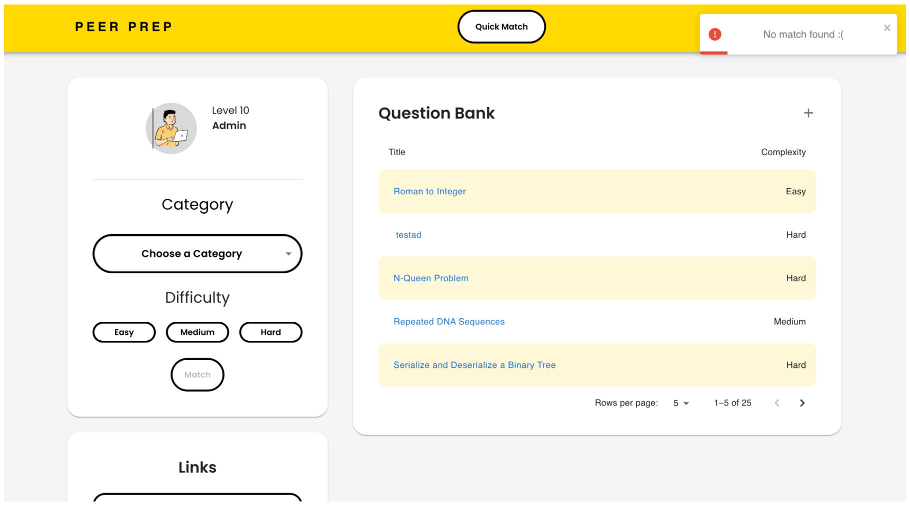
After a successful match is found user is instructed to join their collaboration room.



You have been matched!

Matched Page

On an unsuccessful match, the user is redirected to the home screen and a notification is sent.



Title	Complexity
Roman to Integer	Easy
testad	Hard
N-Queen Problem	Hard
Repeated DNA Sequences	Medium
Serialize and Deserialize a Binary Tree	Hard

No Match Found Notification

Future Plans

Enhancements

1. In future versions, we will consider supporting the option to choose criterias or entirely random for refreshing questions.
2. In future versions, we will consider implementing a level system to gamify the experience.
3. In future versions, we will consider implementing friends and community features to enhance the social and collaborative aspects of our platform.
4. In future versions, we will consider implementing a discussion forum in question service for users to discuss their solutions.

Improvements

Currently user data is stored locally. In the future we will consider switching to a shared database for scaling and synchronization.

Reflections

1. We learned that it is important to integrate our services consistently from the start. This would have minimized the issues we faced near the end when trying to integrate and deploy everything we have been developing separately.

2. We also learnt the importance of having weekly sprint meetings to update one another on our progress. This has helped us stay on track to achieve our targets as a team. Having to share our progress with the team at every stand-up meeting, naturally forces us to put in consistent effort so that we can show what we have managed to complete during the sprint.

3. Looking back we now realize how important it was for us to plan our requirements and solution architecture well. This helped us break down a big

task, developing the Peer Prep platform, into more granular requirements that we could further split by developer and week.

4. We learnt the importance of planning the API design early. Our microservices are all interacting with each other through the API endpoints exposed. Hence, deciding a list of requirements of what these endpoints should take in and output allows the developers to work on their own service independently.