PeerPrep

CS3219 Software Engineering Principles and Patterns

Final Project Report

Team Info:

Bharath Chandra Sudheer	A0218550J
Elroy Goh Jun Ying	A0217472E
Kevin Chua Kian Chun	A0164584W
Manoharan Ajay Anand	A0217703L
Ruppa Nagarajan Sivayoga Subramanian	A0217379U

1 Intro

1.1 Background

Technical interviews play a pivotal role in the recruitment process for people working in the technology sector, an industry that has been increasingly relevant and ubiquitous in today's world. Such interviews are present for jobs hiring software developers, data analysts, quality assurance, cyber analysts to system architects, and candidates are invariably required to master and be familiar with technical questions as an integral part of the evaluation process.

Many students encounter challenges and falter during their pursuit of employment opportunities and internships. These challenges encompass difficulties in articulating thought processes verbally due to a deficiency in communication skills and struggles with comprehending and solving given problems. We lack a platform for students to effectively practice technical questions for them to be better prepared for the recruitment process.

In response to this, we developed PeerPrep, an interview preparation platform with a peer matching system allowing users to jointly practice for technical interview questions. We also have a comprehensive question bank, real-time chat and coding editing, records of past questions, the ability to execute code and more.

Link to Production: https://peerprep.sivarn.com/
Repository: https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g21

1.2 Project Scope

This project aims to mitigate challenges faced by candidates, including difficulties in articulating thoughts, problem-solving, and the monotony associated with traditional interview preparation methods. By providing a collaborative and feature-rich platform, PeerPrep seeks to empower users to navigate technical interviews with greater confidence and competence. We use a microservice architecture as there is no need to couple each functionality together.

Our project is split into a frontend with 4 microservices:

- Users Microservice
- Questions Microservice
- Executor Microservice
- Innkeeper Microservice

2 Overall Description

2.1 Objectives

PeerPrep is a real-time application designed to provide a highly interactive and collaborative environment for users who aim to improve and polish their skills. It provides a platform where users can engage in problem-solving sessions, learn from mutual collaboration and adapt to improving themselves in their desired field of expertise.

2.2 Product Perspective

Peerprep aims to fulfill all the functions needed to enable a productive and enriching collaborative session for programmers.

PeerPrep allows users to send matching requests specifying their desired difficulty level of questions. They are then queued and can see their status updates. Once a suitable match for a user is found, i.e., another user who fits the specified criteria, they are paired and placed in a shared virtual room.

Within this interactive workstation, the paired users can collaborate, review and solve problems together. The application goes beyond just providing a collaborative workspace. It acts as an immersive, real-time interaction platform, offering live chat, visibility of each other's actions like cursor positions, question selections, and even the capability to share and observe live code execution results.

3 Individual Contributions

The following table summarises the technical and non-technical contributions of the members involved in the project.

Name	**Technical Contributions**	**Non-Technical Contributions**
Bharath Chandra Sudheer	Implement Innkeeper MS Implement Frontend (Matching) Implement Serverless features	Requirement Documentation Project Final Report
Elroy Goh Jun Ying	Implement Frontend (Code editor and general structure)	Requirement Documentation Project Final Report
Kevin Chua Kian Chun	Implement Frontend (Login and Tables)	Requirement Documentation Project Final Report
Manoharan Ajay Anand	Implement Executor MS Implement Users MS	Requirement Documentation Project Final Report
Ruppa Nagarajan Sivayoga Subramanian	Implement Questions MS Implement API Gateway Set up CI and deployment	Requirement Documentation Project Final Report

3.1 Subgroup for nice to haves

The following table summarises the nice to have contributions of the members involved in the project.

Subgroup	**Features**
1. Bharath Chandra Sudheer 2. Elroy Goh Jun Ying 3. Kevin Chua Kian Chun	N1: Communication (text based chat)
	N2: History (session table)
	N5: Enhance collaboration service by providing an improved code editor with code formatting, syntax highlighting for one language, syntax highlighting for multiple languages.
1. Manoharan Ajay Anand 2. Ruppa Nagarajan Sivayoga Subramanian	N3: Code execution.
	N9: Deployment of the app on the production system.
	N11: The application should have an API gateway of some kind that redirects requests to the relevant microservices.

4 Requirements

4.1 User Stories

As a...	I want to...	so that...	Priority
Student	Sign up for the platform with Gmail or Github account	I can track my progress	High
Student	Edit my profile information (e.g., name, photo, language)	personalize my profile	Medium
Student	View my history of questions done	I know what questions to practice next without repetition	High
Student	Practice mock interviews with another user	improve my interview skills and gain confidence	High

Student	Choose the difficulty level of interview questions	match my skill level and challenge myself	Medium
Student	Collaboratively edit code during a mock interview	simulate real interview scenarios and improve coding skills	High
Student	Communicate with my mock interview partner via text chat	discuss problems, ask questions, and clarify doubts	High
Student	Share my code editor with my mock interview partner	work together on coding problems in real-time	High
Student	Able to choose curated list of interview questions	explore a variety of questions to practice	High
Student	Mark question as completed during collaboration session with other user	keep track of questions that I have practiced	Low

Student	Queue for a mock interview session with a partner	find a suitable partner to practice with	High
Student	Able to execute code during mock interview session	verify that my code is correct	Low
Administrator	Manage user accounts and access control	ensure the platform's security and integrity	High
Administrator	Add new interview questions to the platform	expand the question library for students	High
Administrator	Edit questions on the platform	Make any corrections or changes to the questions	High
Administrator	Remove outdated or inappropriate questions	maintain the quality of questions available	High

4.2 Functional Requirements

The following table summarises our functional requirements for the projects. Nice to have features are emphasized with a +. For the nice to have features, we are following the numbering found in the project PDF document.

4.2.1 User Service

	Requirements	**Priority**	**Sprint (by week)**
	M1: User Service – responsible for user profile management.		
	M1.1 Sign up for the application with Github/Google	H	5/6
	M1.2 Save preferred name, photo, and language	M	5/6
	M1.3 Gain privileged access (for admins)	H	5/6
+	N2.1 Update user activity (question completed, timestamps)	H	7/8
+	N2.2 Retrieve user activity (question completed, timestamps)	L	7/8

4.2.2 Innkeeper Service

	Requirements	**Priority**	**Sprint (by week)**
	M2: Innkeeper Service - responsible for matching users based on a certain criteria and code collaboration		

	M2.1 Accept user request to initiate matching based on difficulty of questions selected.	H	5/6
	M2.2 Alert waiting users to announce a successful match, or terminate the connection with an appropriate message at the time limit (15s).	H	5/6
	M2.3 Match unassigned users based on their provided parameters, and create a room. Assigned users will be returned to their existing room.	H	5/6
	M2.4 Accept user requests to exit a room (unassign themselves from a room).	H	5/6
	M2.5 Receives real time user edits of the text editor to update shared room state.	H	7/8
	M2.6: Streams real time updates of the room state (including text editor, active users and question selection)	H	7/8
+	N1.3: Support cursor updates from the user (visual selection / insert position) for display to the other user	L	7/8
+	N1.4: Can chat (text-based) with one another during the session	L	9/10

4.2.3 Question Service

	Requirements	**Priority**	**Sprint (by week)**
	M3: Question Service – responsible for maintaining a question repository indexed by difficulty level (and any other indexing		

	criteria – e.g., specific topics)		
	M3.1: View the list of available questions	H	5/6
	M3.2: Add question to the repository (admins)	H	5/6
	M3.3: Search or filter the list of questions	H	5/6
	M3.4: Delete a question (admins)	L	5/6
+	N4: Enable tagging of question by topic (eg. Binary Search)	L	5/6

4.2.4 Execution Service

	Requirements	**Priority**	**Sprint (by week)**
	M4: Execution Service - provides the mechanism to execute code in C++/Java/Python.		
+	N3.1: Execute the source code and view the program output	L	7/8
+	N3.2: Able to view error message for syntax errors in source code	L	9/10
+	N3.3: Able to view error message when program crashes or exceeds time limit	L	9/10

4.2.5 Frontend

	Requirements	Priority	Sprint (by week)
	M5: Basic UI for user interaction – to access the app that you develop.		
	M5.1: Able to Sign up and Login with Github/Google	H	5/6
	M5.2: Able to add/delete/view questions in the question page (admins)	H	5/6
	M5.3 Able to view profile dashboard with user activity history	H	9/10
	M5.4 Able to queue for interview room matching	H	7/8
	M5.5 Able to enter a room and choose a question by either scrolling through the list of questions or searching by title or tags	H	7/8
	M5.6 Able to mark question as completed in the room for each user	L	7/8
	M5.7 Able to close the room and exit the interview	M	7/8
+	N1 Able to view chat	L	7/8
+	N3 Able to view results of code execution	L	9/10
+	N5 Have syntax highlighting in the code editor	L	9/10

2.3 Non-Functional Requirements

The following table summarises our non-functional requirements for the projects.

	Non Functional Requirements (NFRs)	**Priority**	**Sprint**
	User Experience - Users should find the interface intuitive and must be able to navigate the application easily	H	9/10
	Privacy - user details should not be unknowingly leaked to third parties (e.g. number of questions attempted, difficulty levels, emails)	H	9/10
	Scalability - Application should be able to handle large volume of user and question data	M	9/10
	Performance - Updates to the code editor, during collaboration, should be real time with as little latency as possible	M	9/10
	Compatibility - Application should be supported on latest versions of populars browsers (eg. Chrome, Edge, Firefox)	M	9/10

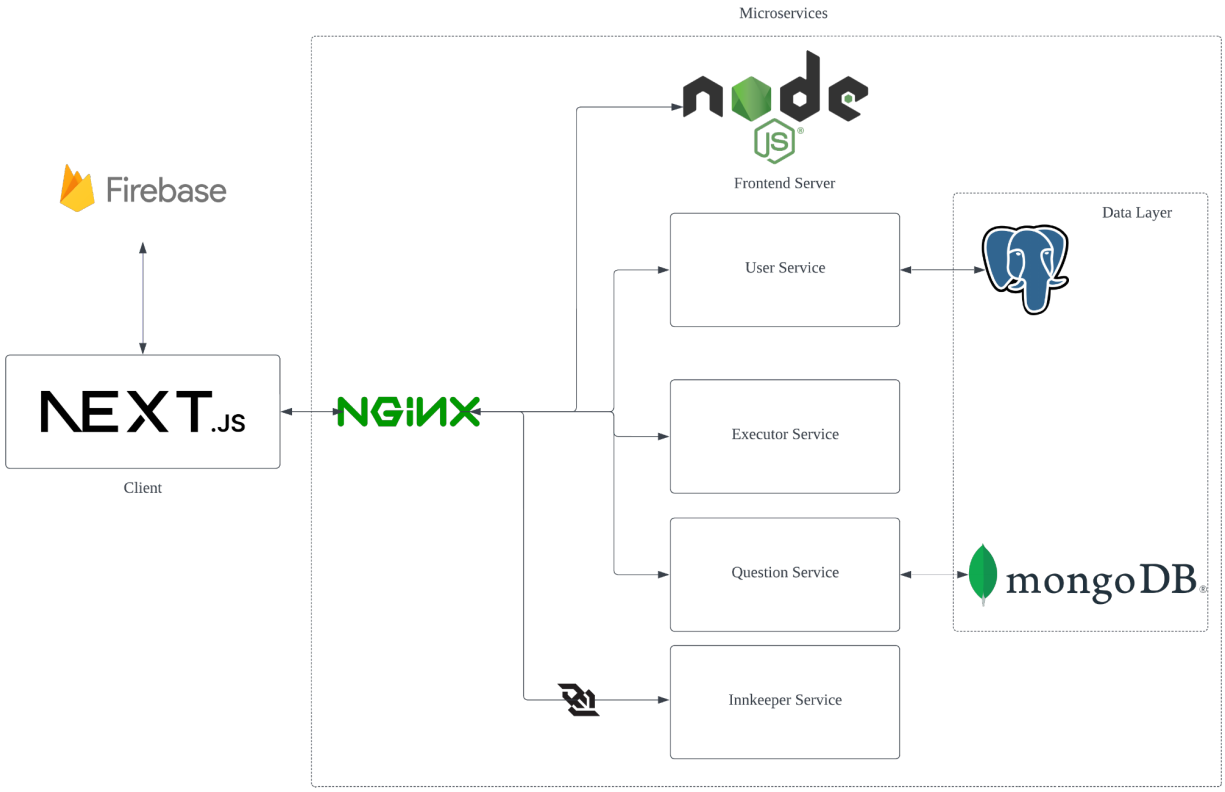
5 High Level Architecture

5.1 Microservices

PeerPrep uses a Microservice Architecture for implementing the requirements mentioned above. A Microservice architecture is desired because:

1. Functionalities are relatively modular and there is not much need to couple them together. For example, the users service can function independently of the executor service. Therefore, a Microservice Architecture allows better separation of concerns for the entire application.
2. A microservice architecture allows for large application codebases to be split into loosely coupled modules that can be run inside a containerised environment and with the right orchestration layer can be scaled independently of each other.
3. The Microservice Architecture also allows separate developers to develop and test each microservice without any dependencies on another. This speeds up development and iteration.
4. Each Microservice is free to use any tech stack necessary to solve the problem at hand giving developers great flexibility into solving the problem with the best approach in terms of language and approach.

5.2 Architecture Diagram



In the architecture above, the team has separated the various concerns and functionalities into a few main microservices:

1. Frontend Server
    a. Stores all frontend related files that will be served to the client upon request.
2. User Service
    a. Responsible for user management (i.e keeping track of user information and roles).
    b. Responsible for user activity (i.e keeping track of which user completed which question).
    c. The data is persisted on a PostgreSQL database.
3. Executor Service
    a. Responsible for code execution for code submitted by the user.
4. Question Service

    a. Responsible for question repository management
    b. The data is persisted on a MongoDB database
5. Innkeeper Service (matching / collaboration service)
    a. Responsible for handling websocket connections for users currently in a coding session.
    b. Responsible for matching users based on question difficulty.
    c. Has an in-memory queue for users waiting for a match.

In addition to these microservice, firebase is also used as an authentication as a service platform and object storage for user profile images. We decided to use firebase because:

1. Allows OAuth capabilities with Google and GitHub which our target users will likely already have an account.
2. Firebase object storage allows the use of CDNs. Although we did not explore the use of CDNs in this project, we can easily add this functionality when our user base grows to a substantial amount.

6 Technology Stack

6.1 Frontend

6.1.1 React/Next.js

React is used for our project front end due to its efficient and declarative nature. One of React's key strengths is its ability to create dynamic and interactive user interfaces by efficiently updating only the necessary components in response to changes in data or user input. The component-based architecture of React promotes modularity and reusability, making it easier to manage and scale complex applications. Additionally, React's virtual DOM (Document Object Model) optimizes rendering performance, ensuring that updates are applied with minimal impact on the overall user experience.

6.1.2 State Management (Jotai)

Jotai is chosen as the state management library for its simplicity and flexibility. As a minimalistic and lightweight state management library for React applications, Jotai allows developers to manage and share state in a straightforward manner, avoiding the complexity of traditional state management solutions like Redux. Its atom-based approach promotes a more modular and scalable code structure, making it easy to reason about and maintain application state. Jotai's focus on composability aligns well with React's component-based architecture, facilitating the creation of reusable and maintainable components. Furthermore, Jotai's API is intuitive and easy to learn, providing developers with a seamless experience for handling state in their projects.

6.2 User Service, Question Service, Innkeeper

6.2.1 Express

Express is our choice for the project backend due to its lightweight and minimalist nature that streamlines the development of robust web applications. Its simplicity allows developers to quickly set up and configure a server, facilitating rapid development cycles. Express's middleware architecture provides a modular way to handle various aspects of the application, such as authentication, logging, and error handling. Its un-opinionated nature grants developers the freedom to choose libraries and tools that best fit their project's needs. With a large and active community, extensive documentation, and a rich ecosystem of plugins, Express empowers developers to build scalable and efficient server-side applications. Its emphasis on flexibility, performance, and ease of use makes Express a go-to solution for backend development in a diverse range of projects.

6.2.2 PostgreSQL (Users Service)

PostgreSQL is often chosen as a database for managing user profiles due to its powerful features, reliability, and scalability. As a robust open-source relational database management system, PostgreSQL offers ACID compliance, ensuring the integrity and consistency of user profile data, which is crucial for applications handling user information. Its support for complex queries, indexing, and advanced data types allows for efficient storage and retrieval of user profile data and user activity data. With a strong community and a history of proven performance in large-scale applications, PostgreSQL provides a secure and flexible foundation for managing user profiles.

6.2.3 MongoDB (Question Service)

MongoDB is a preferred choice for storing a read-heavy repository of questions due to its NoSQL, document-oriented design, which aligns well with the nature of question data that may vary in structure. In scenarios where rapid and flexible access to data is crucial,

MongoDB's schema-less model allows for the storage of questions with diverse attributes without requiring a predefined schema. The JSON-like BSON format facilitates efficient querying and retrieval of data, making it suitable for read-heavy workloads. MongoDB's horizontal scalability and automatic sharding capabilities are advantageous for handling increased read demands, ensuring that as the question repository grows, performance remains optimal. Additionally, MongoDB's support for secondary indexes and caching mechanisms enhances its ability to handle frequent and diverse read operations on a repository of questions, making it a practical choice for applications prioritizing quick and scalable access to question data.

6.2.4 CRDT (Innkeeper)

Implementing the Innkeeper service in a NodeJS environment means that we have access to the wide array of NodeJS packages by the developer community. The y-doc Node.js package is chosen as a CRDT (Conflict-Free Replicated Data Type) library for its efficiency, ease of use, and robust collaborative editing capabilities. CRDTs are particularly valuable in distributed systems where multiple users interact with shared data concurrently. The y-doc library simplifies the implementation of CRDTs in Node.js applications, providing a seamless way to handle concurrent edits and updates across distributed nodes without the risk of conflicts. Its collaborative data synchronization features are well-suited for real-time collaboration scenarios, making it an ideal choice for applications that require shared, live editing experiences. The y-doc library's focus on simplicity and its integration with Node.js environments make it a reliable tool for developers seeking to implement CRDTs in their applications, ensuring consistent and conflict-free data synchronization in distributed systems.

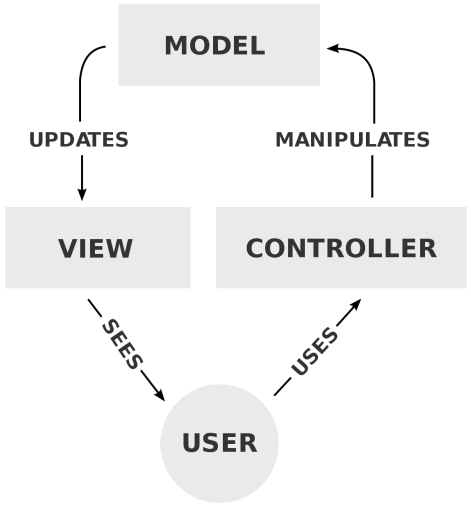
6.3 Executor Service

C++ is chosen for the execution of user-submitted code due to its combination of performance, versatility, and low-level system access. Microservices handling code execution demand efficiency, and C++'s compiled nature and proximity to the hardware provide a performance advantage critical for rapid code execution. Its ability to interact with system resources directly allows for fine-grained control over processes and resource utilization, essential in a secure and isolated environment where user code needs to be executed. With its balance of speed, control, and system-level capabilities, C++ proves to be a pragmatic choice for implementing a microservice designed to execute user-submitted code reliably and efficiently.

7 Design Patterns

7.1 Model-View-Controller (MVC)

In our User, Question and Innkeeper services, MVC design pattern is used extensively.

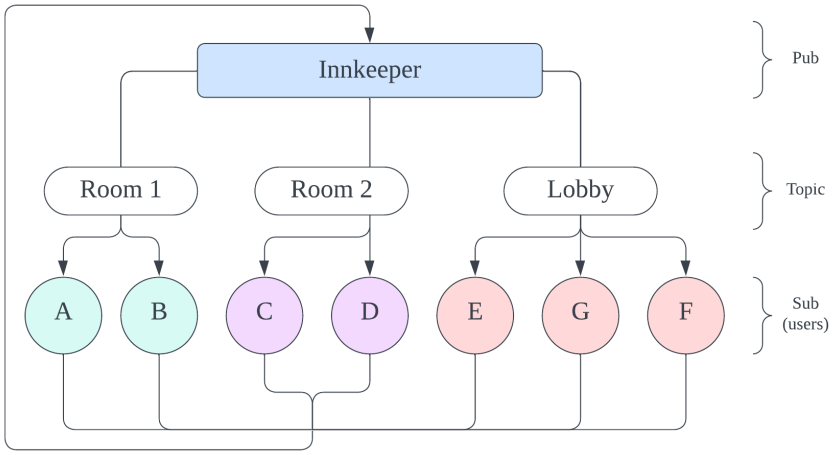


In the context of backend development, the Model-View-Controller (MVC) design pattern organizes the software architecture into three key components. The Model encapsulates the application's data and business logic, representing the backend's core functionality. The View focuses on presenting data to users, typically in the form of responses sent to the client. Finally, the Controller acts as an intermediary that handles incoming requests from the client, processes them, updates the Model accordingly, and triggers the appropriate View to render the response. This separation allows for a clear distinction between data manipulation, user request and response processing, and model validations, contributing to maintainability, scalability, and code organization in the backend of an application. MVC is widely adopted in backend frameworks, aiding developers in building robust and modular server-side systems.

7.2 Publisher-Subscriber Pattern

In this architecture all actions can be treated as the production, detection, consumption and reaction of events. Clients will make requests to the Innkeeper, which consumes these events and then reacts to it appropriately, updating state and in turn creating additional events. This ties in very well with the CRDT framework used (discussed later in this document), and allows for Innkeeper (matchmaking / collaboration microservice) to be the central authority and field all clients requests appropriately.

Since in all scenarios clients only need to be concerned with creating and responding to events, the Innkeeper takes charge of being the broker and then notifies all interested subscribers for their specific topic(s) of interest. In our particular situation we have these topics: Users who are not matched will be subscribed to lobby events (new matches, send to room etc), and users who are matched will only be subscribed to their room events (chat messages, code edits, execution results and so on).



7.3 Data Access Object (DAO) Pattern

The DAO Pattern is a software design pattern that provides an abstraction layer for accessing a data source, typically a database. It separates the business logic of an application from the details of how data is stored and retrieved. The DAO Pattern involves

creating a set of interfaces and classes responsible for performing CRUD (Create, Read, Update, Delete) operations on a specific data entity. By encapsulating the database interactions within DAO classes, developers can achieve a modular and maintainable codebase. DAOs abstract the underlying database details, allowing for easier changes to the data access layer without impacting the rest of the application. This pattern is particularly useful in scenarios where different data sources or database technologies might be employed, as the DAO provides a consistent interface regardless of the specific implementation details.

In our question service, we have used the DAO pattern to abstract the data interactions with a DAO object. This means that code that is related to the database operations, in this case the operations between MongoDB, is contained in this object. This means that the rest of the application can treat database operations like a black box through this DAO object. Furthermore, if we were to change to a different database solution in the future, only this DAO object needs to be changed.

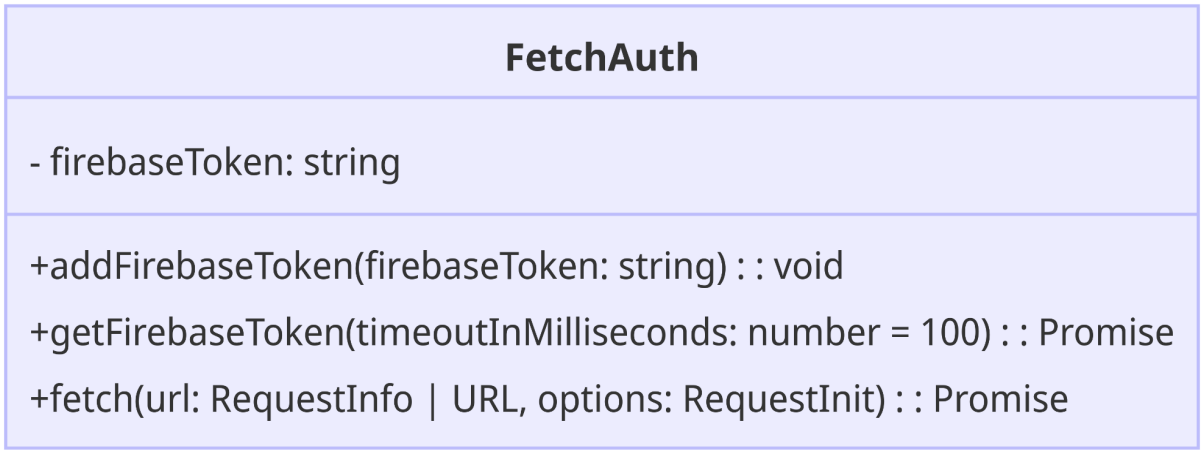
```
const getAllQuestions = async () =>
  QuestionModel.find().then((questions) => questions.map((q) =>
q.toObject()));

const getQuestionById = async (id: string) =>
  QuestionModel.findById(id).then((question) =>
question?.toObject());

const getQuestionsByGroupOfIds = async (ids: string[]) =>
  QuestionModel.find({ _id: { $in: ids } }).then((questions) =>
    questions.map((q) => q.toObject())
  );

const createQuestion = async (
  title: string,
  description: string,
  tags: [String],
```

```
  difficulty: string
) => {
  const question = new QuestionModel({ title, description, tags,
difficulty });
  return question.save().then((question) => question.toObject());
};

const updateQuestion = async (
  id: string,
  title: string,
  description: string,
  tags: [String],
  difficulty: string
) => {
  const updatedQuestion = await QuestionModel.findByIdAndUpdate(
    id,
    { title, description, tags, difficulty },
    { new: true }
  );
  return updatedQuestion?.toObject();
};

const deleteQuestion = async (id: string) => {
  const deletedQuestion = await QuestionModel.findByIdAndDelete(id);
  return deletedQuestion?.toObject();
};

export const QuestionDao = {
  getAllQuestions,
  getQuestionsByGroupOfIds,
  getQuestionById,
  createQuestion,
  updateQuestion,
  deleteQuestion,
};
```

Code snippet of the DAO object

7.4 Singleton Pattern

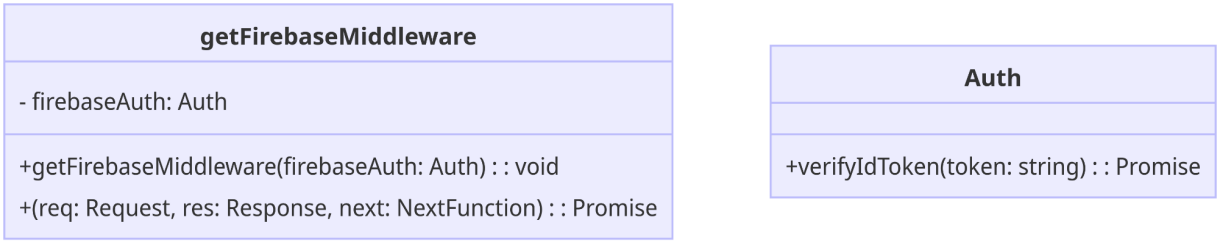
In the frontend architecture, the decision to implement the singleton pattern for authentication was driven by the necessity for a unified and shared approach to managing crucial data, specifically the Firebase authentication token. The singleton pattern ensures that a single instance of the FetchAuth is created and shared across the entire application.



This approach is particularly beneficial as every request made to the backend should have a consistent token which is updated when the user logs in or has an ongoing session. We opted to design our own singleton class instead of using the default Firebase functionality as we wanted flexibility when interfacing with our backend.

By employing the singleton pattern for authentication, we guarantee a centralized and standardized mechanism for handling the Firebase auth token, promoting efficiency, and avoiding redundant instances or conflicting states across the frontend architecture. This design choice contributes to a more cohesive and streamlined authentication process throughout the application.

7.4 Chain of Responsibility Design Pattern



The Chain of Responsibility pattern is a behavioral design pattern where a request is passed through a chain of handlers. Each handler decides either to process the request or to pass it to the next handler in the chain. In the provided code, the concept of the Chain of Responsibility is applied in the middleware function by handling different steps of the request sequentially.

We use the Chain of Responsibility pattern in both the Users and Question service. I will describe how it works for our Question service:

1. Token Verification**:**
    - The middleware starts by checking if a Firebase token is provided in the request headers.
    - If no token is provided, it immediately sends an error response using `handleCustomError` and stops further processing.
2. Firebase Token Verification:
    - If a Firebase token is provided, it proceeds to verify the Firebase token using `firebaseAuth.verifyIdToken(firebaseToken)`.
    - If the verification fails, it catches the error and sends a server error response using `handleServerError.`
3. Role Authorization:
    - If the HTTP method is not a GET request, it performs additional authorization steps.

- It makes a request to the Users service to fetch the user's profile using the provided Firebase token.
- If there is an issue with the request or the user does not have a valid role, it sends an error response and stops further processing.
- If the user is not an admin, it sends an error response indicating that only an authorized admin can perform the action.

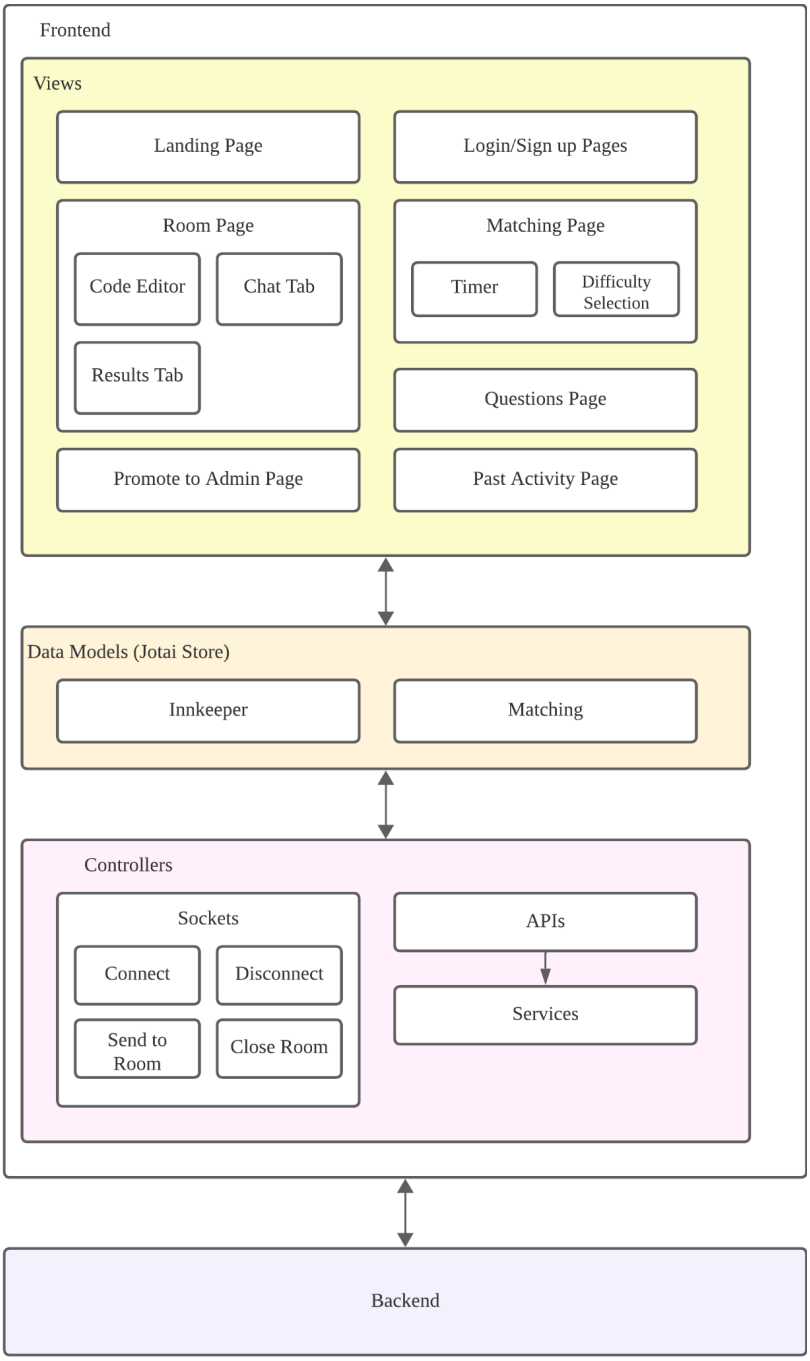
4. Next Function Call**:**
    - If all the previous steps are successful, it calls the `next()` function, allowing the request to move to the next middleware or route handler in the chain.
5. Error Handling**:**
    - If any error occurs at any step, it catches the error and sends a server error response using `handleServerError.`

Thus, this middleware acts as a chain of handlers where each step is responsible for a specific aspect of request processing. If a step cannot handle the request, it passes the responsibility to the next step in the chain. This approach makes the code modular and allows for easy extension or modification of the request processing pipeline.

8 Microservices

8.1 Frontend

8.1.1 Model-View-Controller (MVC) Architecture



In our front-end development, we've applied the MVC pattern to ensure a clear division of responsibilities.

Frontend (View): User interfaces including various pages like Landing, Login/Signup, Room with a Code Editor and Chat, Matching with Timer and Difficulty Selection, Questions, Admin Promotion, and Past Activity.

Data Models (Model): Manages application data within our Jotai Store, with Innkeeper and Matching models for state and matching logic. Upon retrieval, backend data is stored in a Jotai store, our chosen state management solution, which acts as the model within our MVC framework. The controller components monitor the Jotai store for any changes to the data, passing updated information to the view components as necessary. Consequently, any modifications to the Jotai store prompt the view components to refresh, displaying the most recent data to the user.

Controllers: Handle interactions between the View and Model, with Sockets for real-time communication (Connect, Disconnect, Send to Room, Close Room) and APIs interfacing with Services for business logic.

The flow of control and data moves from the Frontend (View) to the Controllers, which manipulate the Data Models (Model), and back to the Frontend, adhering to the MVC principles of separation of concerns.

8.2 User Service

8.2.1 Database Schema

User service database schema

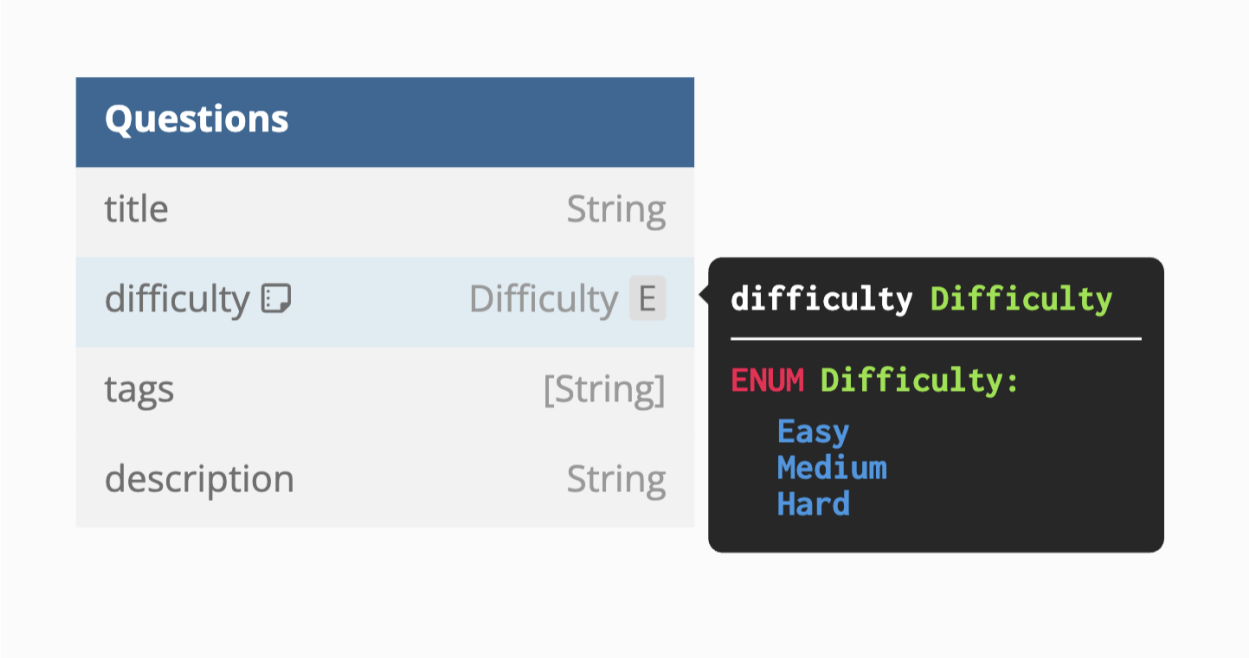
There are 2 tables in the PostgreSQL database as shown above. The profiles table is used for profile management and the activity table is used for activity management. The uid field refers to the uid of the firebase user.

In the activity table, we have a composite primary key consisting of both the uid and questionId fields, so when the user completes a new question, a new record is created with the current date. When a user redos a question, the submitted field of the old record is updated to reflect the current date.

As you can see with the schema, we have chosen not to use foreign keys. We want to develop the application in a quick manner and having foreign keys in the schema would have restricted the flexibility.

8.3 Question Service

8.3.1 Database Schema



Question service database schema

8.4.1 User Authorization

Question microservice needs to determine whether a user has administrative privileges before allowing create, update, or delete operations. To achieve this, the Question service initiates an HTTP request to the User service, querying the user's role.

This communication pattern enables a clear separation of concerns: the Questions microservice focuses on managing questions-related logic, while the Users microservice specializes in user-related functionality. This separation of concerns enhances maintainability, as changes in one microservice do not necessarily affect others, promoting a more flexible and resilient microservices architecture.

8.4 Executor Service

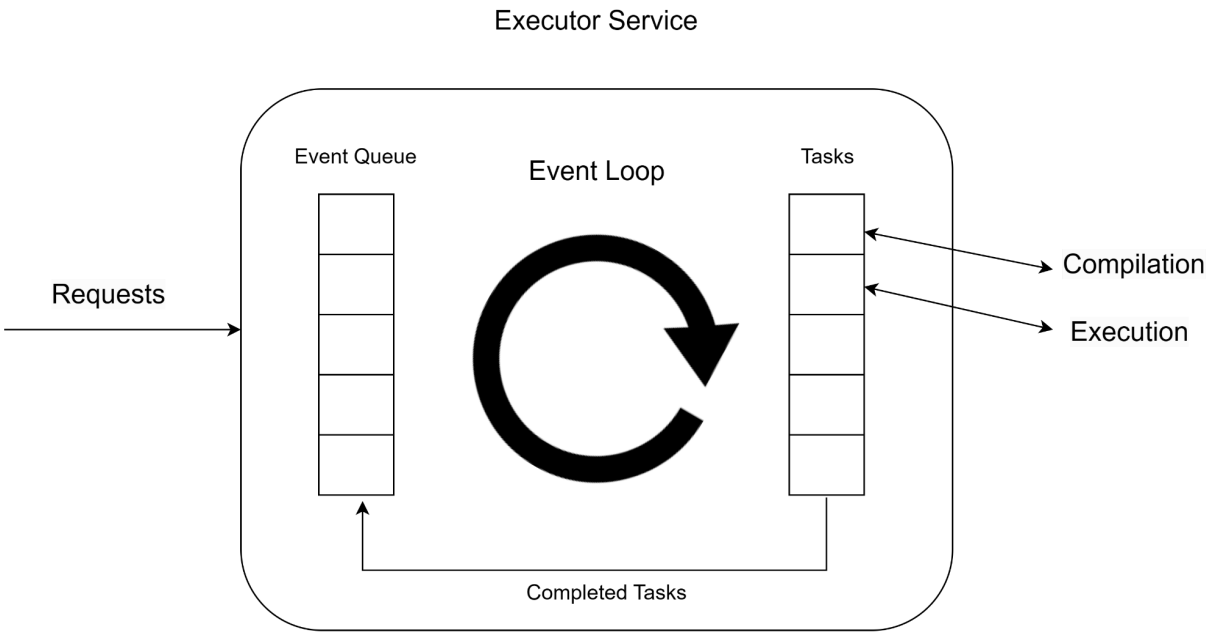
8.4.1 Background

Users of our application can execute source code in C++, Java or Python. All source code submitted by users are stored in uniquely named directories to ensure that multiple users are able to run their code concurrently without conflicts.

The executor service creates and runs 2 processes, one for compilation and the other for execution, sequentially. There is no compilation process created for Python source code due to the nature of the language. For each of these processes, the executor service waits for the status of the process in a nonblocking manner. For the execution process, there is a time limit of 10 seconds before the process is killed. Depending on the status of the process, either an error message or the output of the program is returned to the user.

The executor service was coded in C++ to make use of Linux APIs that allow granular control of process management.

8.4.2 Event-Loop Architecture



The executor service makes use of the event-loop architecture. This architecture revolves around a single-threaded event loop to manage and process events within an application.

At the core of this architecture is an event loop, which continuously checks for and processes events. The event loop manages the flow of events, handling both incoming events (such as user interactions, I/O operations, or messages from other systems) and triggering appropriate callbacks or handlers for these events.

Another feature of this architecture is non-blocking I/O operations. Instead of blocking the execution thread while waiting for I/O (like file system operations or network requests) to complete, the event loop delegates these operations to underlying system APIs and continues processing other events. When the I/O operation completes, a callback is triggered to handle the result.

Having a single thread of execution allows us to avoid the additional challenges of multi-threaded programs such as handling race conditions and synchronization. However,

the use of the event-loop architecture still offers good scalability as the application is able to handle large numbers of concurrent requests and connections.

8.5 Innkeeper

8.4.1 Background

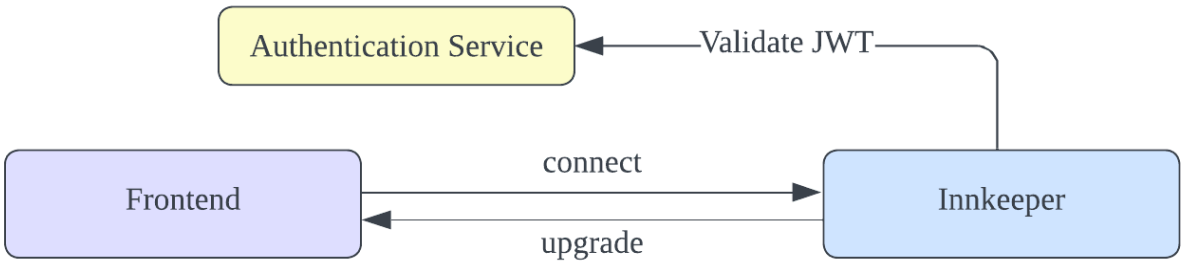
Innkeeper is a microservice developed to offer matching and collaboration services. It functions through a SocketIO service that a frontend can call to start a connection. After the user sends along a matching request based on the question difficulty, they are put in a queue where their statuses can be monitored. If another user's criteria match, the two user profiles are transferred together into a room.

Once in that room, Innkeeper continues its real-time connection services, accommodating features like real-time text chat messages, question selection, integrated code execution results, and enabling real-time edits on the collaborating space. Added features also allows for both users to see one another's cursor, and share execution results.
ß
The architecture of Innkeeper is based on event production and consumption. Clients initiate particular events, which the Innkeeper then consumes, processes and reacts to appropriately. These events range from a match request, a state update, a chat message, or a code change. To accommodate these various demands, Innkeeper utilizes a publish/subscribe pattern, users can subscribe to certain 'events' (lobby or room events) based on their current status, and be notified about any new events of that kind.

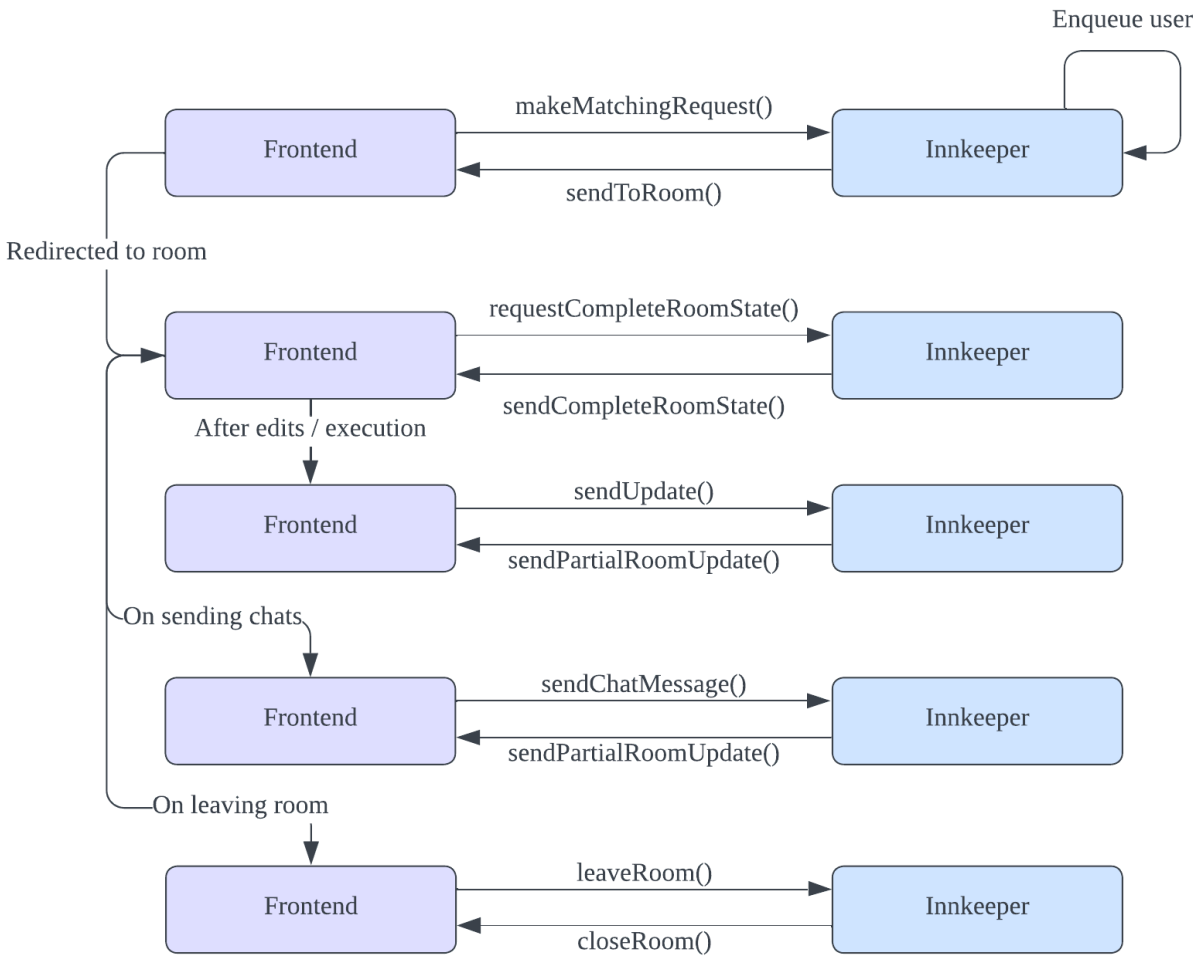
In keeping tethered to these principles, the Innkeeper acts as a central authority, processing all of client requests and being fundamental in maintaining the state and order of events, and thus the structure and flow of the engagement.

Since all actions in the room, and the queue state is entirely short-lived (does not exist outside of an active session or once the user leaves a queue), there was no reason to support data persistence.

8.4.2 API Calls



To initiate a connection, first the client (frontend), initiates a Socket.IO connection. These calls are abstracted away by the Socket.IO library, but it makes a HTTPS `connect` request, that on success is `upgraded` to a WebSocket connection. This upgrade is predicated by authentication middleware logic, where we call Firebase's Authentication-as-a-Service to validate the user JWT.



Once connected, the user begins in the lobby. If the user is already present, they will be sent to their room immediately. Otherwise, they may make a matching request with a difficulty parameter of their choice. They will be enqueued, and once a pair has been found, they will be sent to their room.

From this point onwards, the clients initiate all events by making edits or triggering an execution, sending chats or leaving the room. In all of these scenarios, they make a request to the Innkeeper, which acts as the mediator/broker. The innkeeper then proceeds to publish messages to the right topics by notifying the relevant subscribers (both users in the room). This pattern continues throughout the users stay in the room, until they leave. Once a user leaves the room, both clients will be exited from the Innkeeper room and the topic will be closed.

9 Third Party Libraries

9.1 Firebase

9.1.1 OAuth

We employed GitHub and Google OAuth for user authentication in our system. When users sign up or log in, the frontend client securely interacts with Firebase Authentication, enabling them to register or sign in using their Google or GitHub credentials. Subsequently, we store the obtained Firebase token in a singleton class, FetchAuth, and include this token in the header of each request. The backend rigorously verifies the token's validity for every incoming request. In the process of creating user profiles, our application communicates with the User Service by sending the Firebase token. This interaction facilitates the creation of user profiles, including essential details such as name, preferred language, and profile picture. By leveraging Firebase and OAuth providers, we ensure a robust and seamless authentication and user profile management system.

9.1.2 Storage

We stored images in Firebase Storage. The process starts by the user sending the image from our frontend application as FormData to our backend server. Our User Service, equipped with Firebase Admin SDK, receives the image and uploads it to Firebase Storage, converting the image file into a format compatible with Firebase. Upon successful upload, Firebase Storage generates a unique URL for the image. This URL can be retrieved from the profile route from the User Service. The frontend can then use this URL to fetch and render the image in the user interface, displaying the user's profile picture. This process effectively leverages the Firebase ecosystem to securely store and retrieve images with unique URLs for easy access and display in web or mobile applications.

9.2 YJS

9.2.1 How is it used

In our application, Yjs is employed to facilitate real-time collaboration features during mock interview sessions between two users. Specifically, we utilize Yjs's socket and it's Docs functionality, which can be seamlessly integrated with Codemirror, our code editor component.

9.2.1 Why is it used

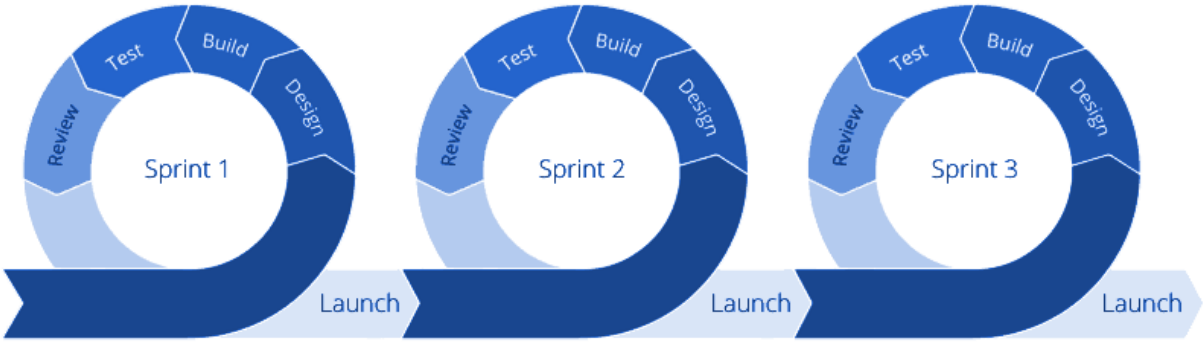
Y-Docs, powered by Y-JS, are essential because they ensure that students practicing mock interviews and collaborative coding sessions can work together effectively and see each other's progress instantly . This capability enhances the learning and practicing experience, making it more engaging and valuable for our users.

Additionally, Y-Docs address library compatibility issues with Codemirror. Initially, our team attempted to use basic sockets to facilitate real-time code editing, but encountered numerous reconciliation issues. Without a robust solution like Y-Docs, achieving a smooth and reliable real-time collaboration functionality would have been challenging. This revised version all in all ensures a smoother and more reliable user experience during real-time collaboration.

10 DevOps

10.1 Sprints

Our team has primarily adopted the Agile software development methodology, whereby our entire development process is split up into 4 major sprints, with each sprint lasting for 2 weeks. During each sprint, we complete a round of planning, building, testing and review. We have also ensured that at the end of each sprint, we have a working product that comes with the new features developed during the sprint. We conduct weekly sprint meetings to review and plan for one week's deliverables, as well as to conduct regression testing and deployments.



Agile Software Development Methodology

	Sprint 1	**Sprint 2**	**Sprint 3**	**Sprint 4**
Bharath Chandra Sudheer	Implement Basic websocket	Integrate innkeeper with frontend	Test innkeeper and resolve bugs	Buffer Finalize deployment Prepare for Assignment 5 Demo
Elroy Goh Jun Ying	Implement landing page without integration with backend	Integrate all frontend components with backend	Test integration and fix bugs Implement user activity table	

	Implement code editor without integration with backend			Prepare for final submission Prepare for project presentation
Kevin Chua Kian Chun	Set up firebase Implement table component	Integrate firebase with frontend	Test integration and fix bugs Implement user activity table	
Manoharan Ajay Anand	Implement user service	Implement executor service	Support changes to backend when bugs arises	
Ruppa Nagarajan Sivayoga Subramanian	Implement question service	Set up API gateway Set up containerisation	Support changes to backend when bugs arises Deployment on DigitalOcean	

10.2 Continuous Integration

We have two distinct CI pipelines:

1. The first pipeline builds and lints the project files. This allows developers to ensure that the code they have written can be built without any errors in an environment that mirrors production while also ensuring consistent code style.
2. The second pipeline builds the docker images for each microservices and pushes it to the GitHub Container Registry. Similar to Docker Hub, this is where we can store and pull custom docker images. This speeds up our docker build process as each

merge to master branch will trigger an update to the docker images, allowing for easy deployment in production.

10.3 Manual Deployment

Our team has used DigitalOcean to deploy the application. We could have gone with an AWS EC2 instance or a GCP Virtual Machine but decided to use DigitalOcean as our team members were more familiar with the DigitalOcean interface the most. However, using AWS or GCP would have been just as easy.

We have used a NGINX API gateway that listens to the HTTP/HTTPS ports for incoming requests and forwards them to the relevant microservice to handle the requests. This type of an API gateway is also known as a reverse proxy.

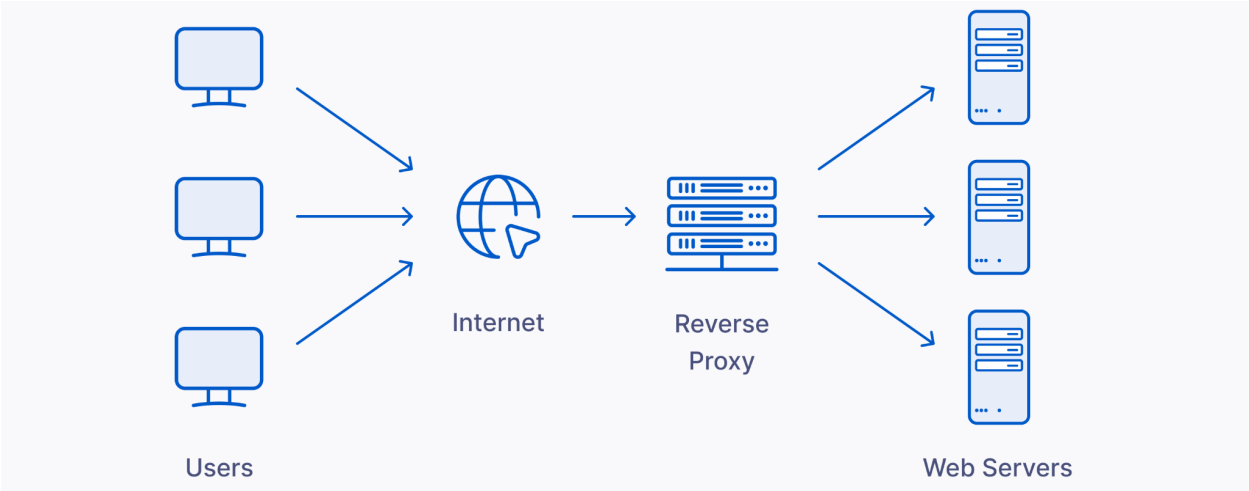


Diagram demonstrating a reverse proxy

10.3 Deployment Process

1. SSH into the production server instance.
2. Create the .env file and firebase files with all the production secrets.
3. Run the deployment on the server using:
    a. `docker compose -f docker-compose.yml up -d`

11 Learning Points

11.1 Conflict-Free Replicated Data Type (CRDT)

Initially, we tried to build a simple central-authority that manages state for the two users in a room by merely updating the value stored. However, even with the speed of in-memory storage, we encountered serious problems immediately with all of our attempts at managing delayed or simultaneous attempts.

After further research we realized that this problem was already addressed in distributed systems through the concept of Conflict-Free Replicated Data Types (CRDTs). CRDTs are a class of data types that achieve strong eventual consistency and can be reliably replicated across different systems with optionally a central authority to keep a store for clients that lose state.

This was a far more powerful and effective solution, and on reflection it was obvious to us that we should have gone to it from the start, considering the difficulties involved in ensuring a smooth collaborative experience.

11.2 Stability of API (Next.JS 13)

We faced numerous issues with integrating existing libraries like react-icons and tan stack query. We should have done our due diligence in researching whether libraries that are compatible with React are also compatible with Next.js. All in all, this serves as a good experience for us to realize that fairly often we need to stick with stable, well-established APIs.

In our case, based on past experience, using Next.JS 12 would have certainly worked. Being a stable release for far longer, all issues with AntDesign and other libraries would not have cropped up.

11.4 Teamwork

Communication and consideration for others is important. Our thought process, intentions and plans have to be spelled out to help everyone be on the same page and to move forward efficiently. It is also important to be tactful for peace and so that everyone will be encouraged to contribute good ideas to make our project the best it can be.

12 Possible Further improvements

12.1 Innkeeper

12.1.1 Additional Features

Given our close implementation of event-driven architecture principles, adding new types of events or actions should be relatively straightforward — we introduce a new event type, define the effect it has on the system, and ensure the server notifies clients properly.

Additionally, with the CRDT system (Yjs), the system is quite adaptable for introducing more real-time interactive features. CRDTs make it easy to add replica-edited functionality without worrying about synchronization conflicts.

Some examples of these features can be supporting the ability to collaboratively interact with AI assistants or to engage in visual whiteboard sessions.

12.1.2 Horizontal Scalability

Socket.IO lends itself better to enterprise features in general, especially for horizontal scaling or load balancing. Through the use of adapters to Redis Pub/Sub mechanism and Socket.IO's inter-server communication, we can transparently scale this product to any number of Innkeeper instances with load balancing. This will facilitate real time communication and uninterrupted access even for tens of millions of users at a time.

Coupled with moderate changes to how the system is arranged, it's very feasible to even adopt a distributed approach and even support adaptive load balancing.

12.3 Frontend

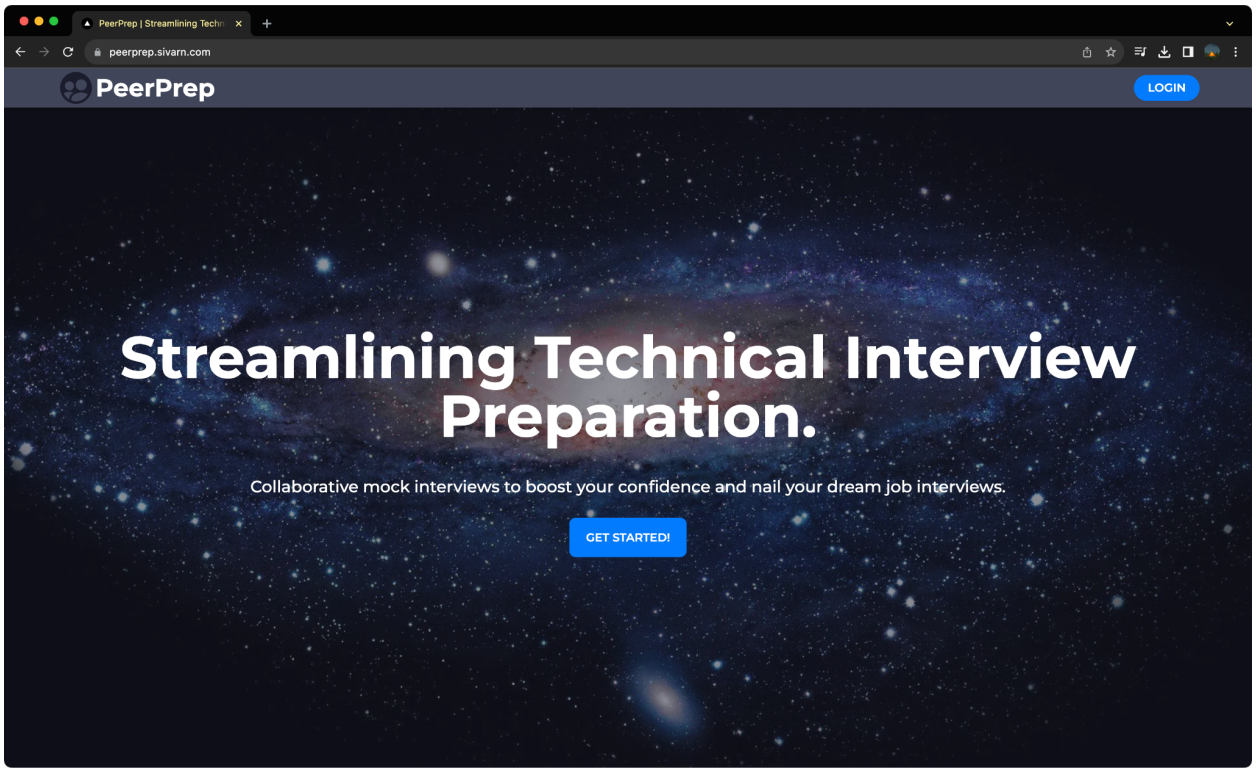
12.3.1 Activity Table in Lobby

Our current activity table only displays questions attempted by the user. We could actually integrate information about who the user matched with and the code written by the user.

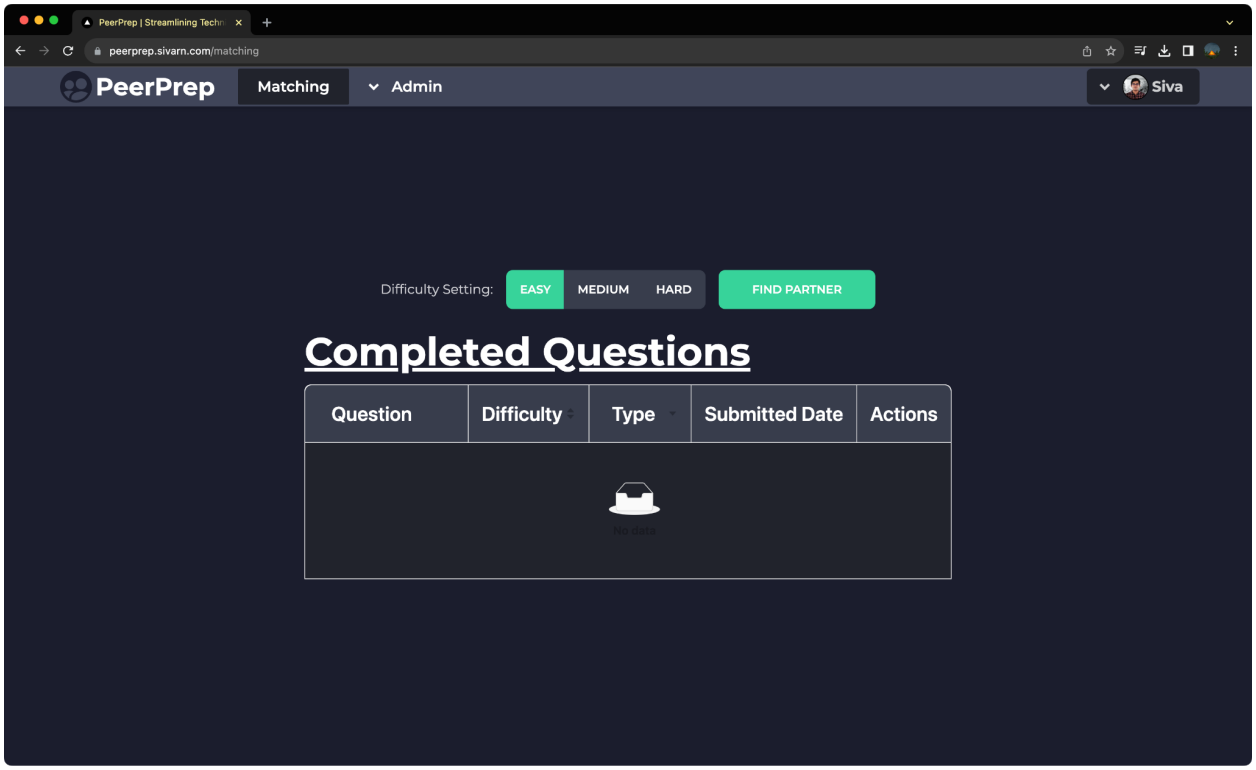
12.4 Deployment

As of now, our project does not utilize Kubernetes for deployment purposes. This decision was grounded in the assessment that, given the current scale of our project, a Kubernetes-based deployment strategy is not necessary. Our rationale hinges on the belief that the complexity and resources required for Kubernetes are more justifiable for larger-scale operations. However, with foresight and scalability in mind, we have meticulously established a robust containerization framework. This strategic setup ensures that our system is Kubernetes-ready, facilitating a smoother and more efficient transition to a Kubernetes-based deployment model in the future. This approach positions us well to scale up our deployment infrastructure seamlessly as our user base expands, ensuring that we can maintain efficiency and reliability even as our needs evolve.

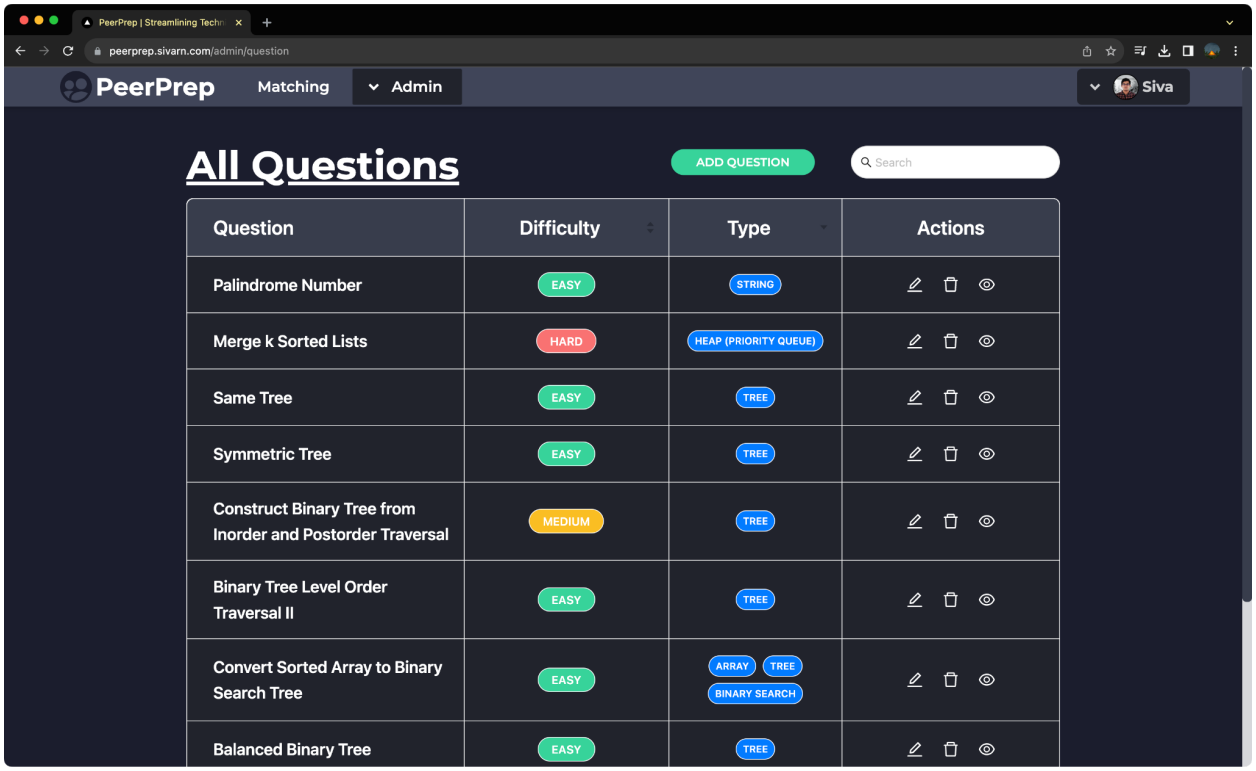
13 Product Screenshots



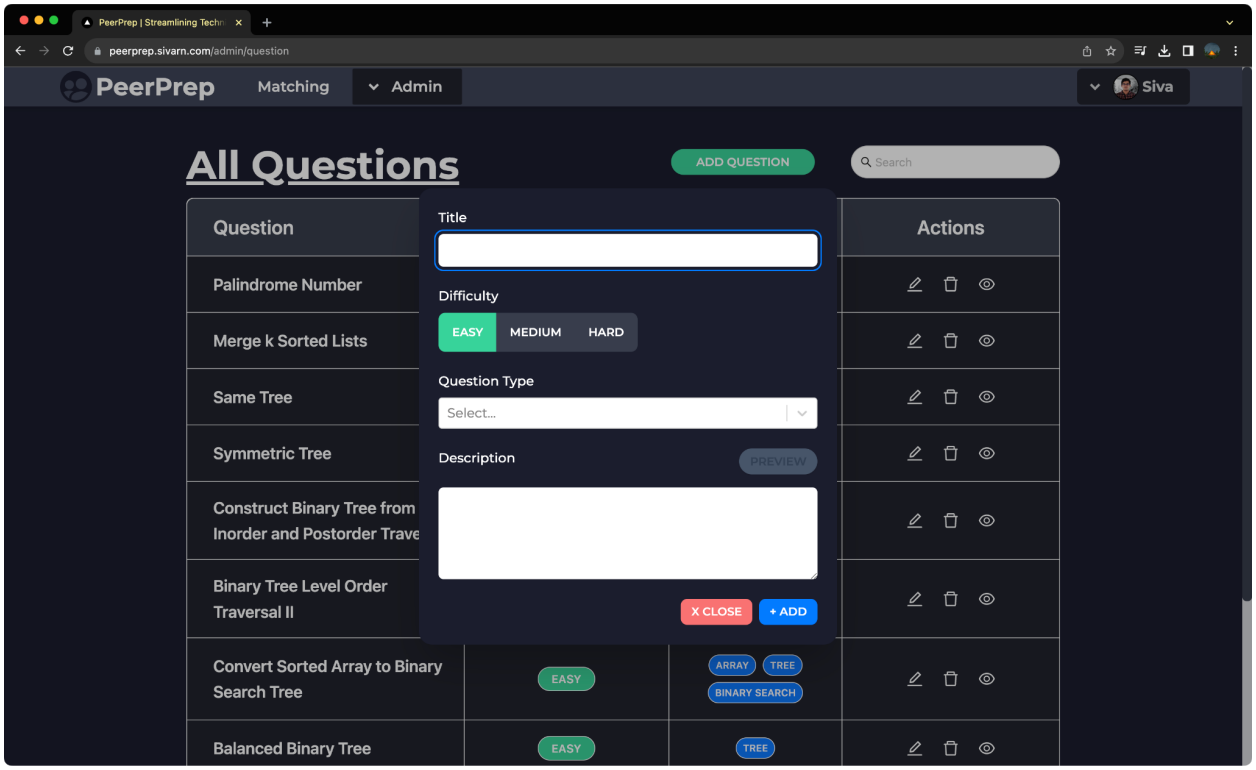
Landing Page



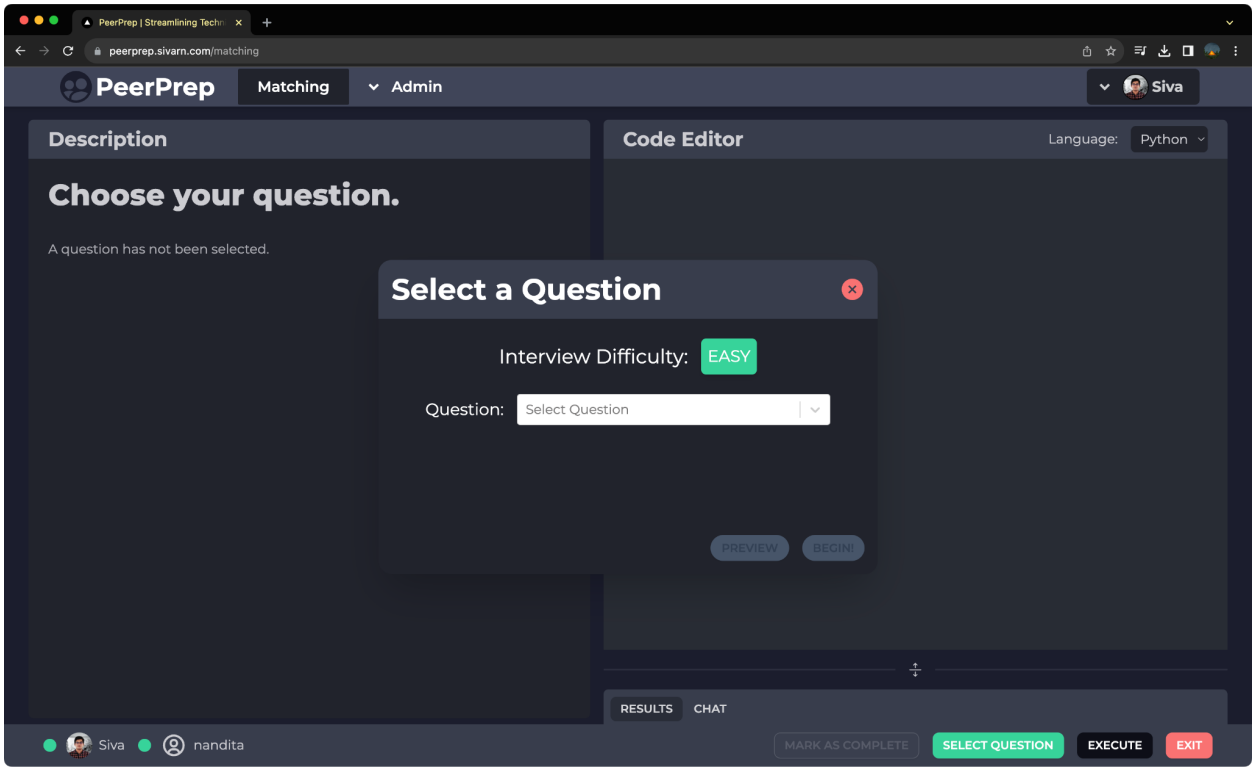
Matching Page



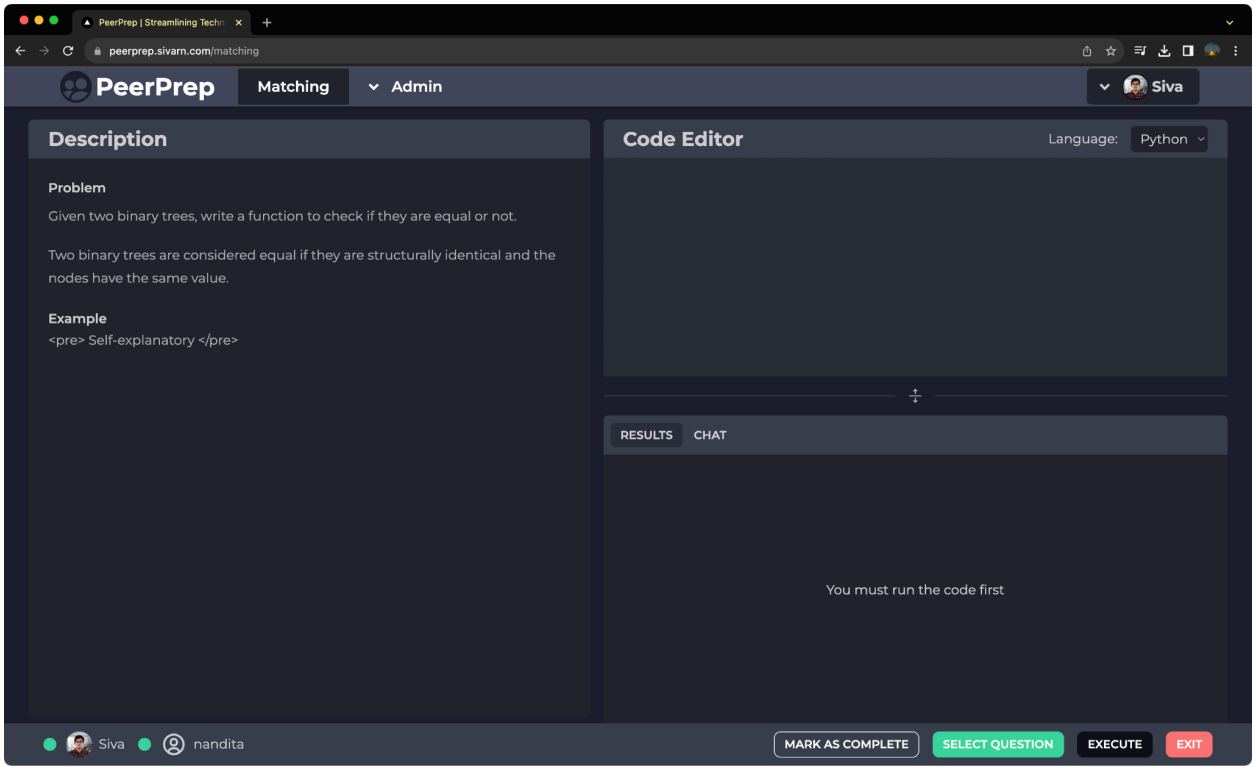
Admin Question Portal



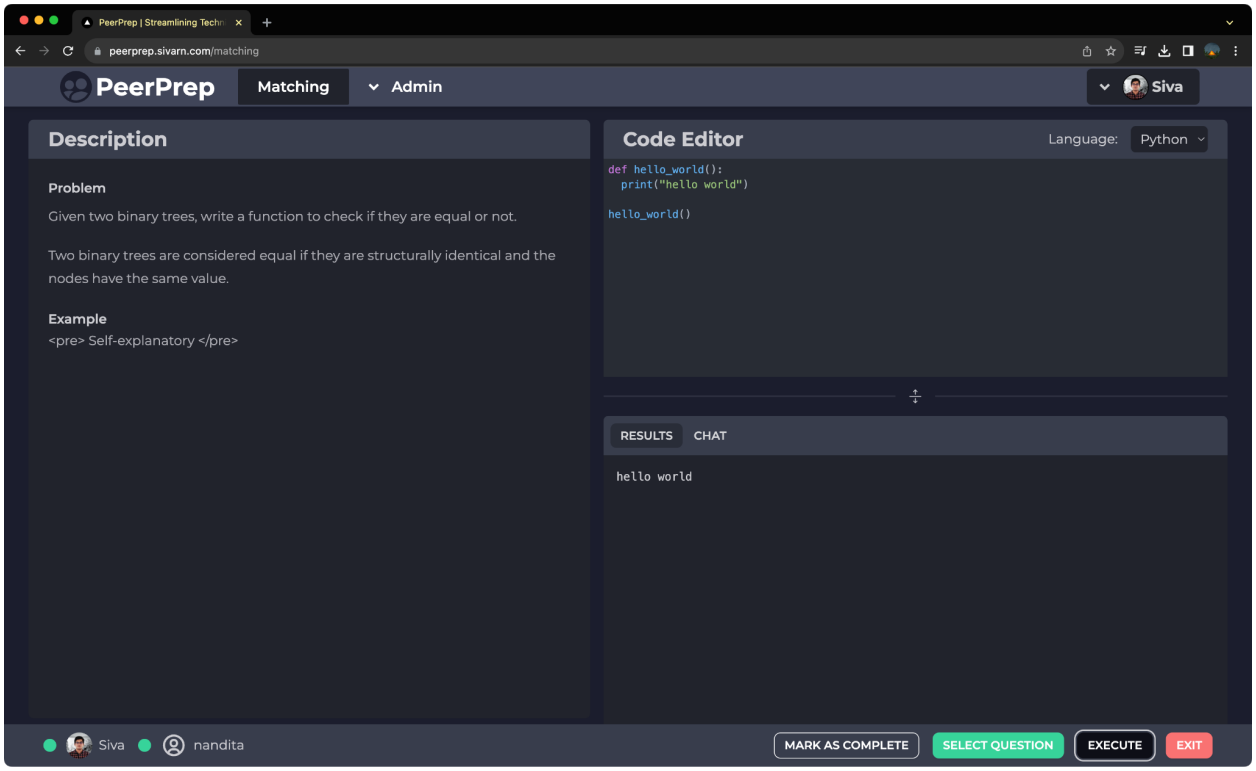
Create Question Modal



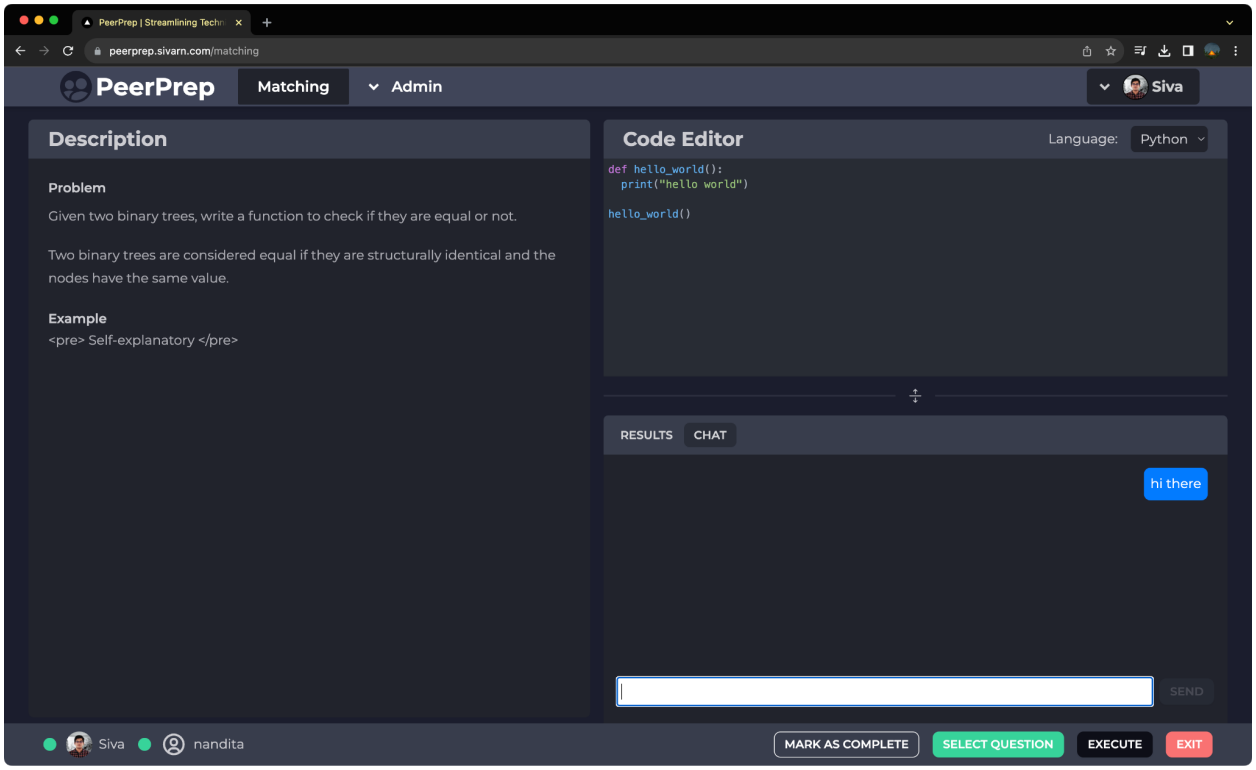
Match Success Page



Code editor Page



Code execution


PeerPrep | Streamlining Techr
peerprep.sivarn.com/matching
PeerPrep
Matching
Admin
Siva
Description
Problem
Given two binary trees, write a function to check if they are equal or not.
Two binary trees are considered equal if they are structurally identical and the nodes have the same value.
Example
<pre> Self-explanatory </pre>
Code Editor
Language:
Python
def hello_world():
print("hello world")
hello_world()
RESULTS
CHAT
hi there
SEND
Siva
nandita
MARK AS COMPLETE
SELECT QUESTION
EXECUTE
EXIT

Chat