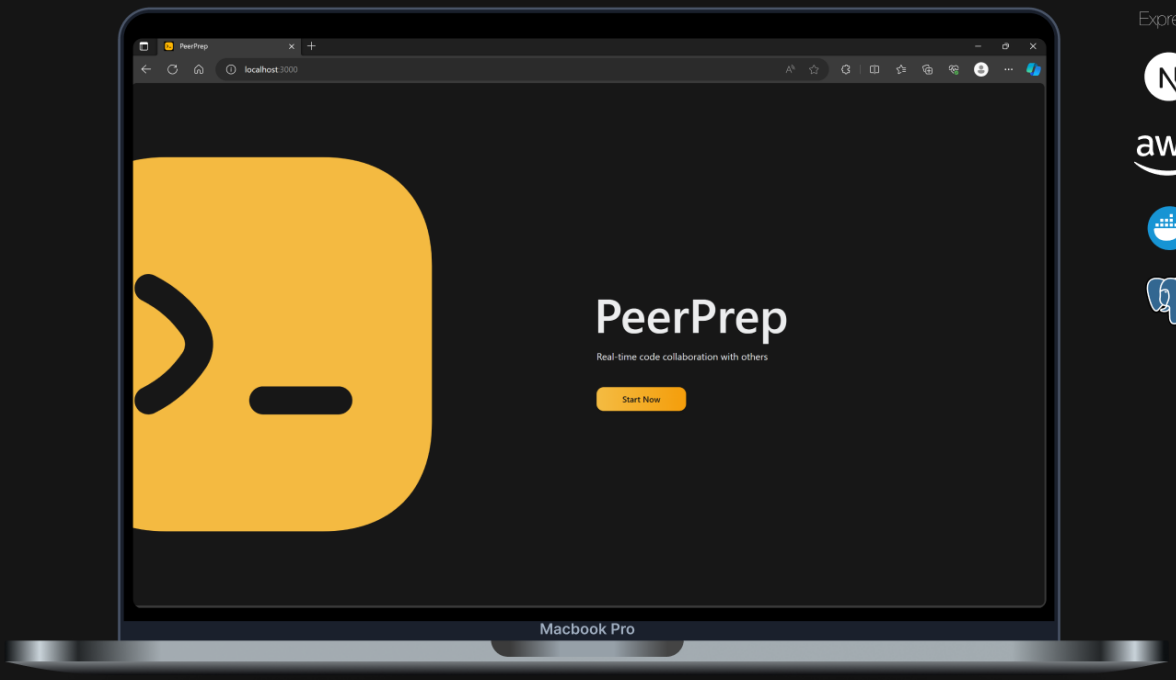
PeerPrep - G05 Project Report

CS3219 Software Engineering Principles and Patterns



A0234940H	Chen Junsheng
A0219814B	Tan Rui Yang
A0225644E	Tan Le Yi
A0217711M	Tan Xing Jie
A0217991U	Tay Weida

Source Code: https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g05

Deployed To: https://master.d2wfcislijzove.amplifyapp.com/

Table of Contents

Introduction

Background

Increasingly, companies are incorporating technical interview platforms like LeetCode and HackerRank into their hiring processes to identify suitable candidates for their job positions. There is a growing need for problem-solving platforms that enable individuals to practice their coding skills before technical interviews. However, most existing platforms for practice primarily emphasize solitary problem-solving, lacking a systematic approach to guide users in articulating their problem-solving approaches. This skill is crucial in real-life software engineering, requiring explanations and justifications for one's work and solutions. Peerprep aims to bridge this gap.

Product Overview

Peerprep aims to offer a collaborative platform where users can prepare for technical interviews in real time with an unknown partner they're matched with. The matching process considers shared preferences, and redirects pairs to a browser-based code editor where they collaboratively solve coding questions. Additional support from real-time chat and AI fosters a conducive environment for collaborative learning, effectively simulating the interview environment. After practice sessions, users gain a summary of the level and topics of problems attempted, providing a progress tracker for their interview preparation.

Contributions

Feature Track	Member	Role	Deliverables
User	Rui Yang	Backend, QA	Implemented RESTful API endpoints, integrated backend with cloud SQL database and set up unit and integration tests.
	Weida	Frontend	UI components that support CRUD of users.
Question	Junsheng (N4)[1]	Frontend	UI components to support CRUD of questions.
	Xing Jie	Backend	Implemented RESTful API endpoints and integrated backend with No-SQL database.
	Rui Yang	Full Stack	(Frontend) Implemented UI design for question detail page (Backend) Implemented RESTful API endpoints, integrated backend with SQL database, and set up unit and integration tests.
Matching	Junsheng (N4)	Full Stack	(Frontend) Created the Matching Service UI components and logic for frontend. (Frontend) Created UI to retrieve questions during session initiation. (Backend) Implemented the events & event processing, set up unit tests.
	Weida	Backend	Implemented event bus for Matching Service to create rooms for Collaboration Service.
Authentication	Le Yi	Full Stack	(Frontend) Refined the UI components related to login/signup, implemented authentication in the frontend middleware for controlled routing (Backend) Implemented the RESTful API endpoints, developed the authentication logic and implemented it as a middleware for User, Question and History Services.
	Xing Jie	Full Stack	(Frontend) Created the UI components for password resetting and email verification (Backend) Implemented the RESTful API endpoints related to password resetting and email verification, integrated email sending service.
	Rui Yang	QA	Added error handling for Auth Service, unit and integration tests for each API endpoint.

[1] N* refers to nice-to-have features

Feature Track	Member	Role	Deliverables
History (N2)	Rui Yang (N2)[2]	Full Stack	(Frontend) Implemented history-related UI components and pages (Backend) Implemented RESTful API endpoints, integrated backend with SQL database, and set up unit and integration tests.
Collaboration	Rui Yang (N5)	Frontend	Implemented the main UI components that support the collaborative workspace, including an improved code editor.
	Weida	Full stack	(Frontend) UI updates to the frontend Collaboration Service and sockets (Backend) Implemented the structure and main logic for Collaboration Service in the backend.
	Le Yi (N1, N3)	Full Stack	(Frontend) Created UI design for chat space, test cases and code execution panels (Backend) Implemented communication means as well as code execution functionality with Judge0 api.
	Xing Jie (N7)	Full Stack	(Frontend) Update chat space UI for chatting with AI (Backend) Implement generative AI within the chat with OpenAI API.
Continuous Deployment	Junsheng (N8, N9, N10, N11, N12)	DevOps	Set up the full framework of AWS deployment with cloudformation stack and GH actions successfully and local deployment. Containerisation of backend services.
	Weida (N8, N9, N10, N11, N12)	DevOps	Deploy Collaboration Service and compiled applications for local deployments.
Continuous Integration	Rui Yang (N8)	DevOps	Set up GH actions for every PR to run unit and integration tests.
Report (Non-technical)	All	All	Final report.

[2] N* refers to nice-to-have features

Project Management

In the dynamic landscape of web application development, effective project management plays a crucial role for successful outcomes. We carefully designed the project management methodologies for Peerprep to ensure we can deliver the best web application in time. In this section, we will delve into the approaches, tools, and practices that define our project management framework.

Project Management Methodologies

Our approach to project management is anchored in the agile software engineering principles of Scrum and Kanban, combining structured sprint cycles with continuous delivery flexibility. The combination of Scrum and Kanban framework allows greater flexibility for the project management and planning, enabling us to adapt and respond to evolving project requirements efficiently.

On a weekly basis, our team engages in Sprint Planning sessions where members share about the progress, discuss ongoing tasks, and collaboratively plan the next tasks and objectives for the upcoming sprint. This regular meeting provides a platform for open communication, enabling the team to address challenges, share insights, and collectively devise solutions.

Collaboration Tools

The collaboration tools we used include GitHub and Notion. GitHub stands as the primary version control system and the codebase of Peerprep. In tandem with GitHub, we utilize the GitHub Kanban board for project tracking and organization. We create new issues as the work to do, which is equivalent to the concept of raising tickets in the professional software engineering industry.

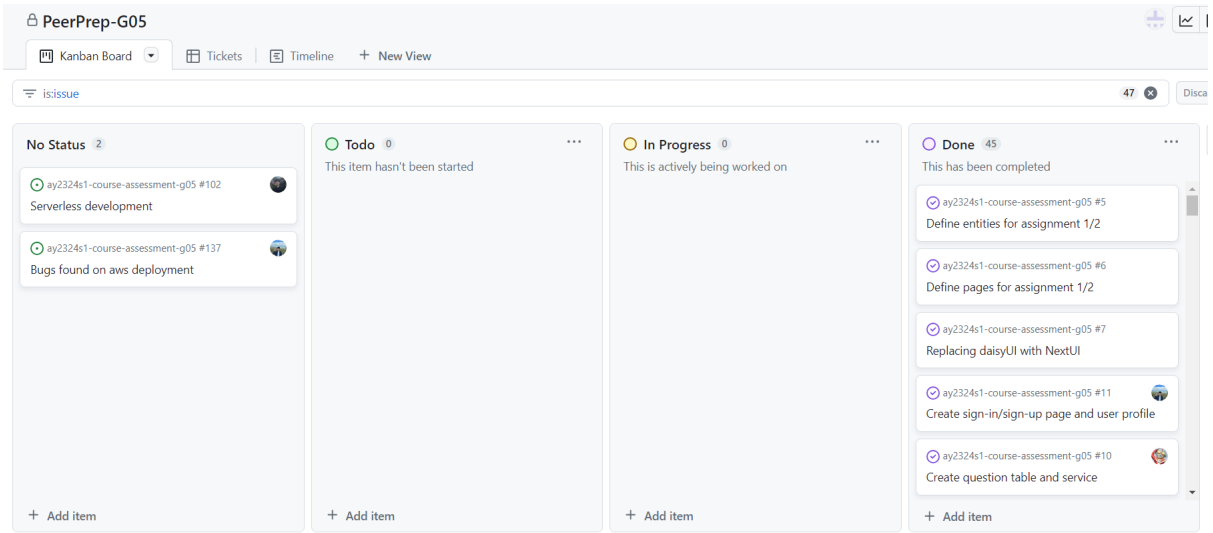


Figure 1. GitHub Kanban board which organizes our tasks by their progress statuses.

Additionally, Notion plays an essential role as our shared workspace for documenting coding standards, best practices, as well as software requirements, enhancing knowledge sharing within the team.

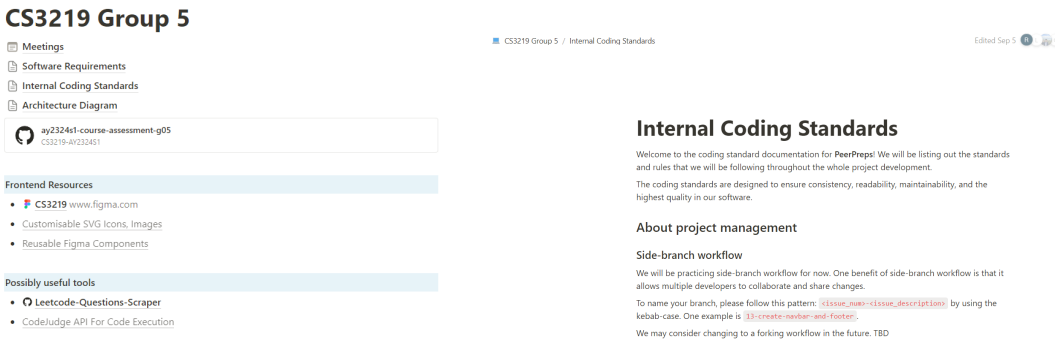


Figure 2. Notion Space and Internal Team Standards Documentation.

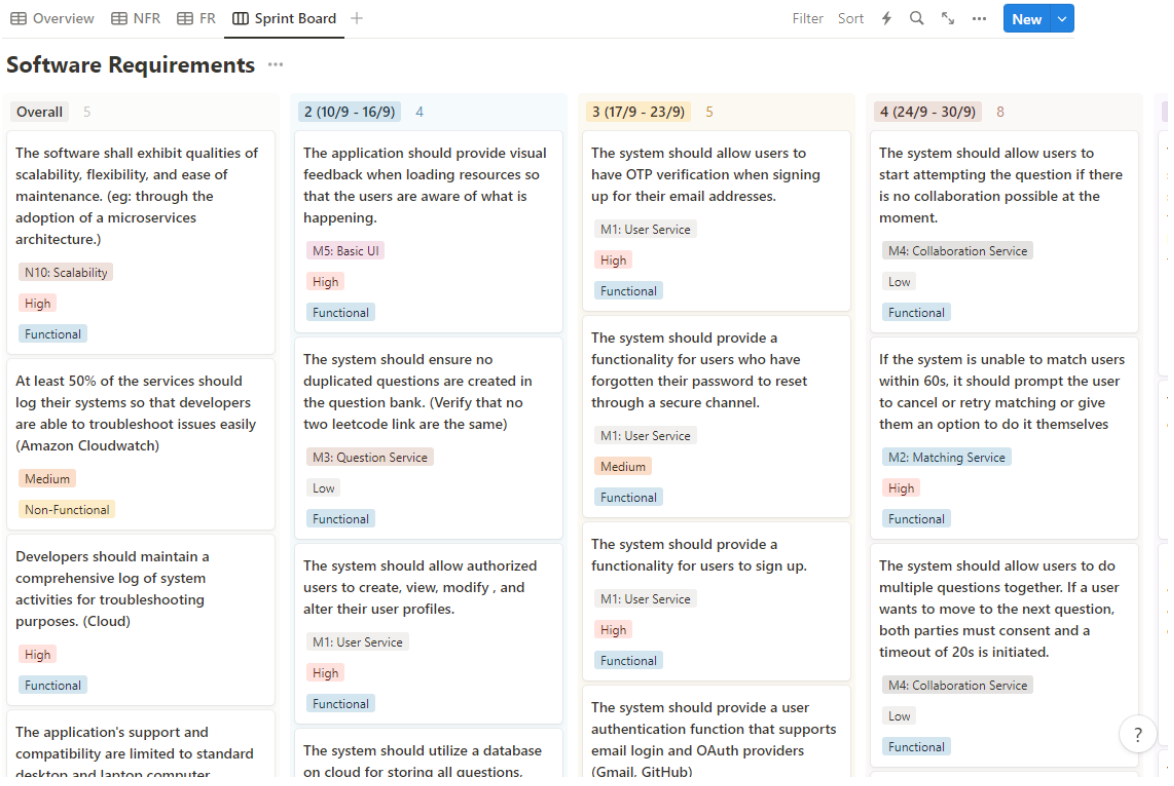


Figure 3. Software Requirements Documentation.

Maintaining High Code Quality

Ensuring the integrity and quality of our codebase is one of the top priorities for our team. For every new Pull Request (PR) created, we mandate at least one team member who is not directly involved in the changes to conduct a thorough review before merging to the existing codebase. This comprehensive review includes testing and providing constructive feedback, facilitating a collaborative approach to code quality assurance.

To streamline the review process and enhance documentation, we created a PR template on GitHub. This template acts as a guide for developers to provide essential information about their changes, such as the purpose of the PR, the related issues, and steps for testing. This

practice not only ensures completeness in documentation but also contributes to a more efficient and effective review process.

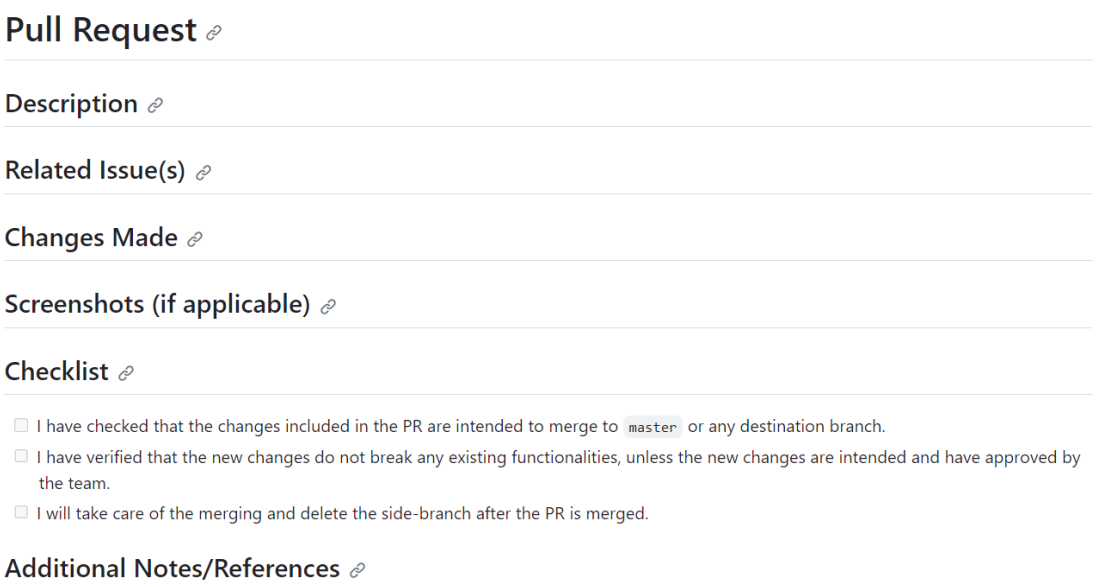


Figure 4. GitHub Pull Request Template.

Furthermore, we also have an automated pipeline for running tests for the new code changes. This is to ensure that they do not introduce any issues or regressions to existing functionalities. This automated testing framework serves as a robust safety net for maintaining the stability of our application. For more details about the automated tests, please take a look at the Continuous Integration section.

Project Progress and Flexibility

Our development philosophy revolves around delivering Minimum Viable Products (MVPs) within one or more sprints. We do not enforce strict deadlines for project delivery, and instead we rely on the self-discipline of each team member. With progress tracked through Kanban workflows and Sprint Planning meetings, our flexible approach to project timelines accommodates dynamic development requirements.

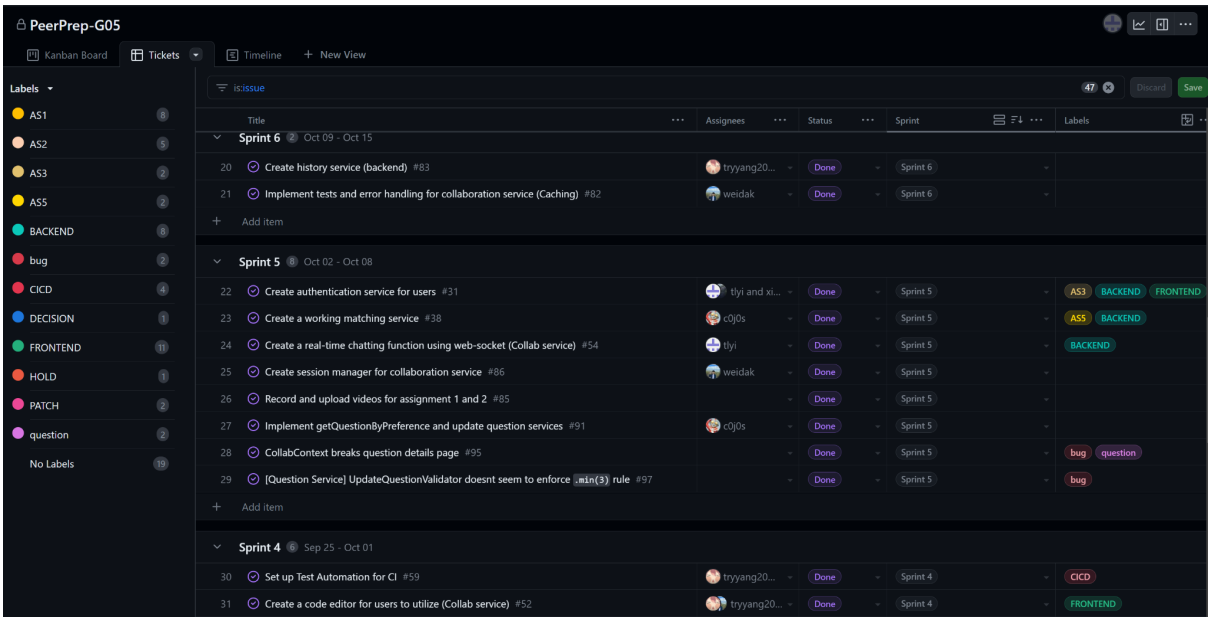


Figure 5. GitHub Kanban Board.

Ensuring Consistent UI

Since our workflow involves members working independently on both frontend and backend, to ensure that everyone is on the same page about the UI design, we did basic wireframing on Figma before we started work on the code.

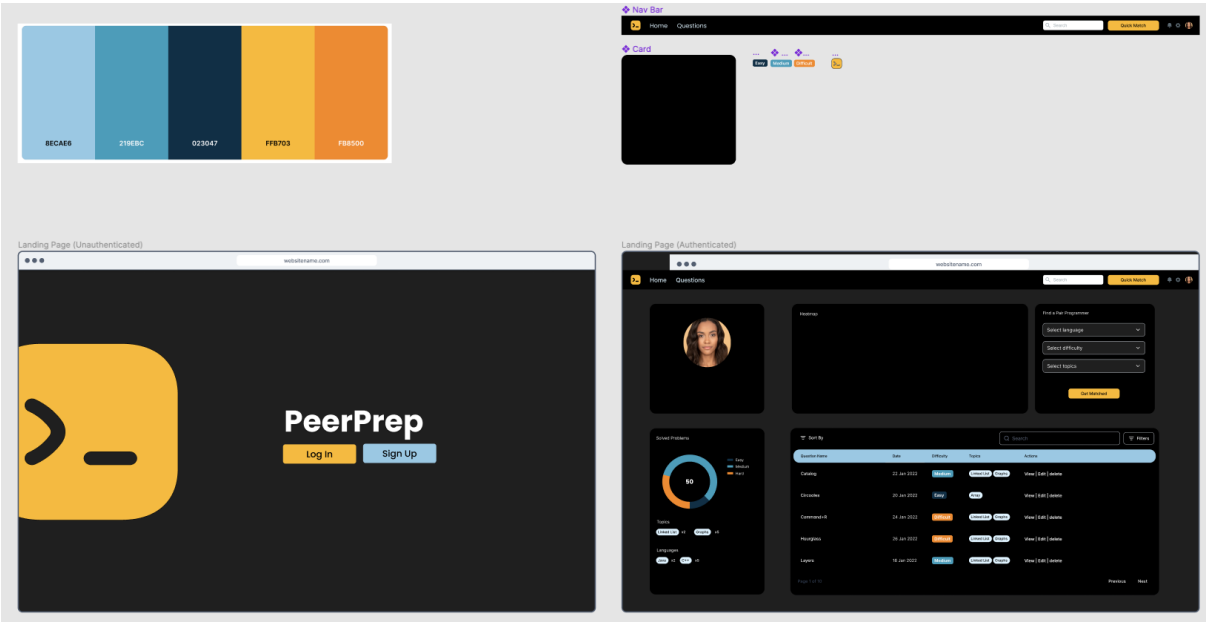


Figure 6. Figma Wireframing.

This served as a good guideline for future frontend development, ensuring that our final product provides a coherent experience for the user. In the end, we can see that our final product adhered closely to our initial design prototype.

Project Requirements

The project requirements serve as the foundation upon which successful solutions are built. In the context of Peerprep, we have categorized these requirements into Functional Requirements (FR) and Non-Functional Requirements (NFR), articulating the essential functionalities and characteristics that the software must possess.

Functional

Functional Requirements outline the specific functionalities and features expected from the software. Each requirement undergoes a meticulous ranking process, wherein the team assigns priority levels—High (H), Medium (M), and Low (L). This prioritization system dictates the development sequence of services, with High priority denoting features accorded top precedence during the implementation phase. Conversely, Low priority designates features deemed as advantageous but not essential, serving as non-blocking and supplementary elements in the software development process. Each functional requirement is also categorized into 6 different services within our software architecture.

User Service

S/N	Requirement	Priority
FR-U1	The system should allow authorized users to create, view, modify, and delete their user profiles.	H
FR-U2	The system should let users change their username and password.	H
FR-U3	The system should ensure that no two users have the same email.	H
FR-U4	The system should store the user data permanently in a cloud database unless a valid delete request is made.	H
FR-U5	The system should allow users to change the profile picture, and ensure the picture is stored permanently unless a deletion request is made.	M

Auth Service

S/N	Requirement	Priority
FR-A1	The system should provide functionality for users to sign up.	H
FR-A2	The system should allow users to verify their emails after signing up through a one-time link sent to their emails.	H
FR-A3	The system should provide the functionality for users who have forgotten their password to reset through a secure channel.	H
FR-A4	The system should block logins when the wrong credentials are provided.	H
FR-A5	The system ensures that the user is logged in even when the browser is refreshed (session persistence).	H

Question Service

S/N	Requirement	Priority
FR-Q1	The system should allow users with ADMIN roles to manage (create, update, delete) questions in the repository.	H
FR-Q2	The system should provide non-admin users to modify the question bank.	H
FR-Q3	The system should allow all users to view all questions available in the question bank without showing individual question details.	H
FR-Q4	The system should utilize a database on cloud for storing all questions, and it must ensure that the data is retained indefinitely unless a database administrator chooses to initiate deletion.	H
FR-Q5	The system should ensure no duplicate questions (questions with the same title) in the question bank.	M
FR-Q6	The system should ensure each question has necessary data such as the question title, description, topics, complexity, etc.	H

History Service

S/N	Requirement	Priority
FR-H1	The system should provide a way to keep track of completed questions. (e.g.: display a list of completed questions on the user profile page)	H
FR-H2	The system should provide visualization of attempted questions statistics. (eg.: display a heatmap of past submissions, display charts to show the number of attempted questions by different categories such as topics, complexities, and programming languages)	H
FR-H3	The system should be able to show code submissions made by a user for different questions and different programming languages.	H
FR-H4	The system should allow users to store the code submission and mark the question as attempted on terminating a collaboration workspace.	H
FR-H5	The system should ensure the code submission and question data is stored in a persistent database.	H
FR-H6	The system should ensure no duplicate history is stored for the same user, question and programming language.	M

Matching Service

S/N	Requirement	Priority
FR-M1	The system should match users with the same criteria (topic, language, difficulty) together.	H
FR-M2	If the system is unable to match users within 60s, it should prompt the user to cancel or retry matching.	H
FR-M3	The system should display a timer to indicate the waiting time for the matching process.	H
FR-M4	The system should allow users to choose multiple topics, language and difficulty for matching.	M
FR-M5	The system should allow users to cancel matching request.	M
FR-M6	The system should provide users to select questions on the fly during a session initiation.	L
FR-M7	The system should allow matching of users with common criteria (topic, language, difficulty).	L
FR-M8	The system should display the common criteria used for pairing users.	L

Collaboration Service

S/N	Requirement	Priority
FR-C1	The application should provide a collaboration workspace for two users to code.	H
FR-C2	The system should allow users to edit on a code editor that has support for code formatting and syntax highlighting in multiple coding languages.	H
FR-C3	The system should allow users to communicate to each other in real-time through chat.	H
FR-C4	The application should allow each other to view their partner's current cursor movements such as location and selection at any given time.	L
FR-C5	The system should allow users to be able to execute code on submission and simulate their own inputs for the given question.	M
FR-C6	Once matched, the system should provide a time limit (60 minutes) for users to complete the question.	H

Non-Functional

In contrast, Non-Functional Requirements (NFR) encapsulate the quality attributes and characteristics that are critical to the software's performance, usability, and overall user experience. We have crafted a set of NFRs to guarantee that our product not only meets the needs of our target users, but also aligns with the established standards of software engineering.

S/N	Requirement	Priority
NFR-1	Response times for real-time collaboration code updates shall be less than 1s to ensure a seamless coding experience.	H
NFR-2	The software testing activities and ongoing development work must not adversely impact the stability or functionality of the existing codebase in any way.	H
NFR-3	At least 50% of the services should have logging to capture events, incoming and outgoing traffic so that developers are able to troubleshoot issues easily. (Amazon Cloudwatch)	L
NFR-4	The application's support and compatibility are tailored for standard desktop and laptop computer devices, irrespective of screen size. However, the application is not intended for use on smaller devices such as mobile phones, iPads, and similar platforms.	M
NFR-5	The application should support the latest versions of major browsers. (Chrome, Firefox, Edge, Arc)	M
NFR-6	The application is mandated to safeguard user confidential data, including profile images and passwords, ensuring the prevention of any unauthorized exposure of secrets.	H
NFR-7	The application should ensure data validity (eg.: the registered email, the file type uploaded) to prevent bot abuse and safeguard data integrity.	H

Compliance with NFRs

Response times for real-time collaboration code updates shall be less than 1s to ensure a seamless coding experience.

- In designing our system, we aim to reduce latency to increase usability. Thereby, we only emit changes when we need to, such as emitting an insertion of a character in the exact location instead of emitting the whole library/editor content. This streamlines the number of bytes exchanged.
- When we migrated to AWS, we chose the region closest to the users. This can be further upgraded in the future if our product were to receive global recognition, and allocate the availability zones based on the location of the user. Thereby these development choices allow us to achieve a low latency during collaboration.

The software testing activities and ongoing development work must not adversely impact the stability or functionality of the existing codebase in any way.

- In our software testing framework, we employ two distinct forms of tests: unit tests and integration tests. To safeguard the production database from unintended data modification, we implement the mocking technique during unit testing. For integration tests, a dedicated testing and development database is utilized, preventing any impact on the production environment. For details about software testing, please refer to the Continuous Integration section.
- On the development side, we prioritize the integrity of the production database schemas. To explore proof of concepts and potential modifications, we exclusively use a separate test database. Only after thorough testing and assurance that the changes won't disrupt existing functionalities do we proceed to push these modifications into the production environment.

At least 50% of the services should have logging to capture events, incoming and outgoing traffic so that developers are able to troubleshoot issues easily. (Amazon Cloudwatch)

- AWS Cloudwatch log groups were created for every microservices to capture their internal activities and events for troubleshooting. An additional log group was created for AWS API Gateway to capture all incoming requests. Thus, we have achieved 100% coverage for logging of services.

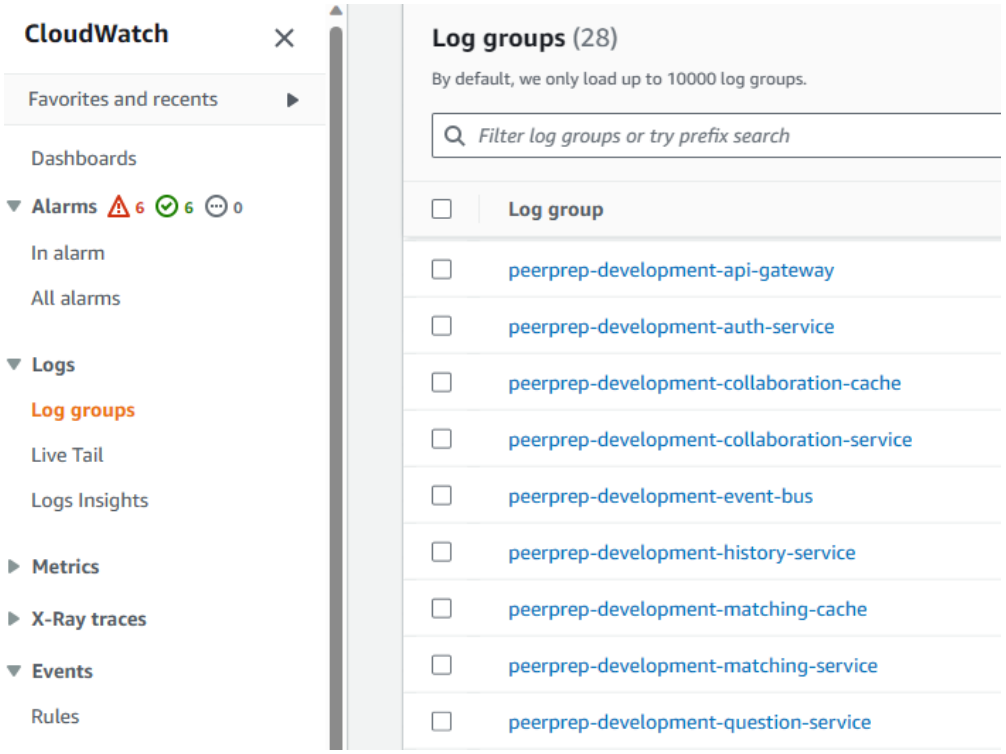


Figure 7. Cloudwatch Log Groups.

The application's support and compatibility are tailored for standard desktop and laptop computer devices, irrespective of screen size. However, the application is not intended for use on smaller devices such as mobile phones, iPads, and similar platforms.

- To guarantee application compatibility with desktop and laptop computer devices, we prioritize the responsiveness of user interfaces (UI). The application is designed to adeptly render UI elements for various screen sizes, accommodating the dimensions of standard laptop screens.

- Given our emphasis on fostering a collaborative coding experience, resources are not allocated to optimizing rendering for small devices such as mobile phones. The application focuses on delivering an optimal user experience on desktop and laptop platforms.

The application should support the latest versions of major browsers. (Chrome, Firefox, Edge, Arc)

- To meet this requirement, we guarantee that all packages utilized in the application are compatible with a wide range of browsers. Additionally, we ensure that the CSS styling adheres to compatibility standards across various browser environments.

The application is mandated to safeguard user confidential data, including profile images and passwords, ensuring the prevention of any unauthorized exposure of secrets.

- We focus on the security of the application. To ensure data confidentiality, we only stored the hashed passwords in the database. For profile images, we utilize Amazon S3 Bucket as the cloud storage of user profile images. The S3 Bucket provides security services such as access control, server-side encryption, and HTTPS support. Only authorized hosts with additional AWS secrets can access the cloud database.

The application should ensure data validity (eg.: the registered email, the file type uploaded) to prevent bot abuse and safeguard data integrity.

- This requirement is fulfilled through the implementation of diverse constraints at various levels, coupled with meticulous validation procedures. Upon new user registration, the system initiates a verification process by sending a confirmation email to the provided email address. Users must confirm this email to activate and access application functionalities.
- To prevent misuse of data storage, stringent constraints are imposed at different application layers. At the frontend, file uploads are checked to ensure they adhere to the designated image file format. Additionally, email and password strength requirements are enforced. In the backend APIs, comprehensive validation is performed, surpassing frontend constraints, and includes checks for data validity, question complexity, and adherence to specified input lengths. At the database level, NOT-NULL constraints are implemented to guarantee the presence of essential data, fortifying the overall data integrity measures.

Quality Attributes

We have shortlisted 8 quality attributes that we will use to guide our design choices and have ranked them as a team in order of priority.

Attribute	Score	Scalability	Integrity	Performance	Reliability	Robustness	Security	Usability	Maintainability
Scalability	2		^	^	^	<	<	^	^
Integrity	5			<	<	<	<	^	^
Performance	1				^	^	^	^	^
Reliability	3					<	<	^	^
Robustness	2						<	^	^
Security	1							^	^
Usability	7								<
Maintainability	6								

As observed from the table, our priorities lie in **Usability**, **Maintainability** and **Integrity**. These quality attributes served as strong guidelines for the design choices we have made throughout the project.

System Design

Overall Architecture

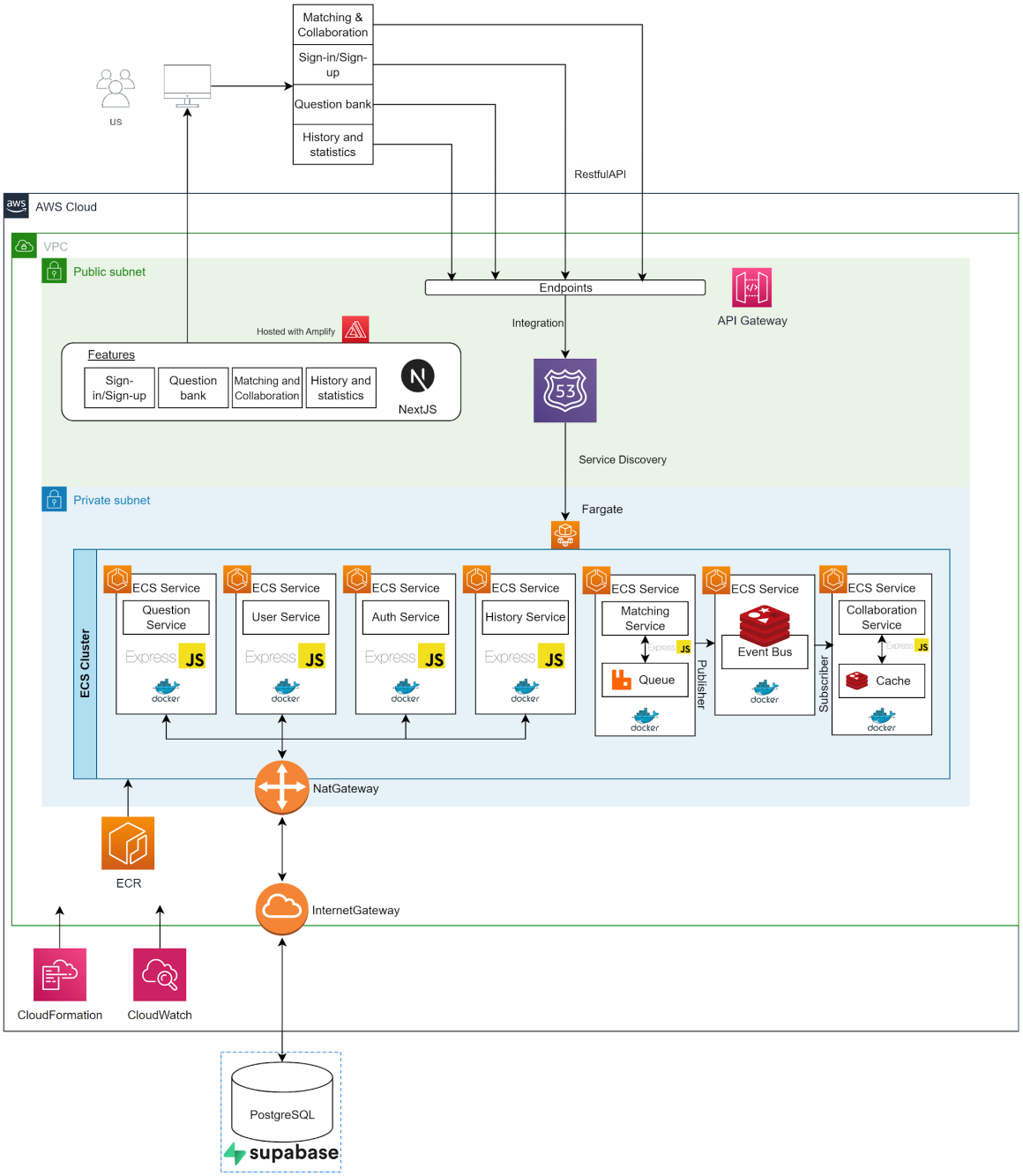


Figure 8. PeerPrep Overall Architecture Diagram.

The diagram above provides a high level overview of our system architecture design for Peerprep in the production environment. The frontend will be hosted on AWS amplify while the backend microservices will be deployed to AWS ECS cluster. Communicate between frontend and backend is facilitated using Restful APIs and Sockets though AWS API Gateway. The database solution is hosted PostgreSQL in Supabase.

Refer to this for a detailed description of our frontend architecture.

Refer to this for a detailed description of our backend architecture.

Refer to this for a detailed description of our frontend and backend architecture deployed in AWS.

Technology Stack

Tech Stack	Rationale
NEXT.JS (Frontend)	• In-built file-based routing protocol that simplifies page routing • In-built server-side rendering (SSR) that greatly enhances performance as well as improving search-engine optimization (SEO) • Seamless integration support with React, provides strong foundation for building dynamic user interfaces
Express.js (Backend)	• Minimalistic and opinionated framework that provides high flexibility for developers • Robust middleware supports for routing • Smooth learning curve for beginners
PostgreSQL (Auth, User, Question, and History Services)	• Open source relational database management system that supports establishing relationships between entities
Prisma	• Beginner-friendly ORM tool that strongly support Typescript and PostgreSQL • Allows intuitive declarative data modeling • Highly performant automated real-time data synchronization
Redis	• Performance • Native support for socket-io
RabbitMQ	• An open source message broker that provides low-latency messaging • Ideal for handling time sensitive information like matching requests • Support various programming languages officially with community plug-ins • Easy to deploy

Frontend Architecture

We will cover the organization, structure, and design principles applied to the user interfaces of Peerprep. Our approach emphasizes a clear distinction between the frontend and backend components, aligning closely with fundamental software engineering principles such as the separation of concerns and the single responsibility principle.

Architecture Style

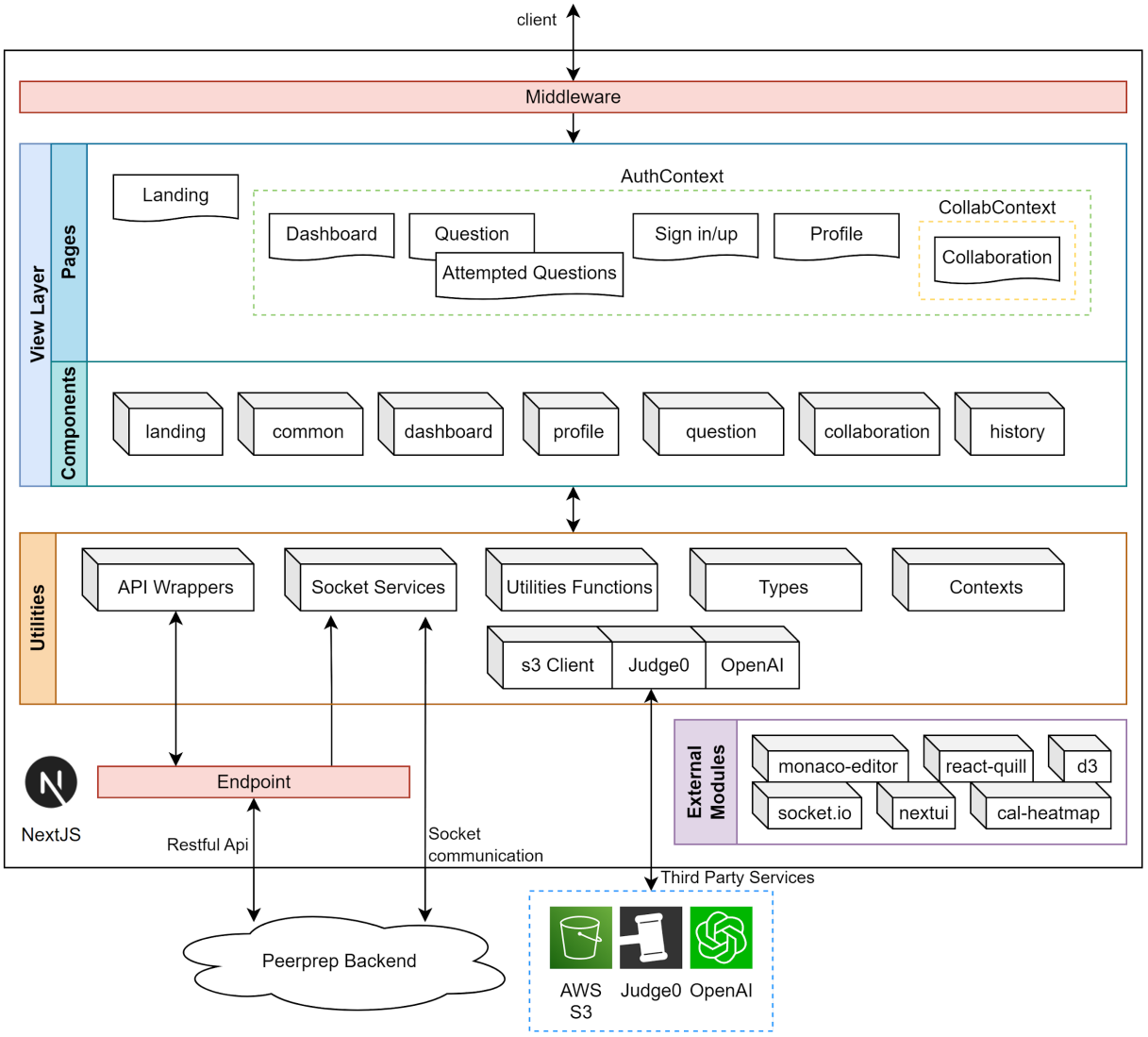


Figure 9. Frontend Architecture Diagram.

The diagram above provides a high-level overview of our frontend, built using Next.js 13 framework with external node modules and libraries.

The middleware layer, a mechanism supported by Next.js natively, serves as a crucial gateway for processing incoming client requests before they reach their intended pages. Within this layer, we have implemented authorization logic to facilitate controlled access to various frontend views.

The presentation/view layer is responsible for housing view declarations and reusable UI components, organized within their respective domains folders. Page declaration files define the structure and layout of our page views, while UI components focus on user interaction. Some pages are nested within specific contexts, enabling the sharing of data and methods across views. In the diagram, we highlight the usage of AuthContext and CollabContext for authentication and collaboration purposes. Additionally, we have other supporting contexts for styling and History Services, among others.

To ensure a clean separation of concerns and modularity, all essential business logic for the view layer entities is extracted to the utility layer. This design pattern ensures that the view layer entities remain separate from their underlying business logic, allowing for a clear division of responsibilities and feature scopes. The utility layer includes API Wrappers, which provides an abstract of interactions with backend service, Socket Services for managing Socket.io communication, third-party clients, and various reusable helper functions for the frontend.

Finally, the endpoint layer is designed to serve as a centralized gateway for communication with backend microservices. It allows for standardization and centralized monitoring of backend API calls, and supports the adaptation of endpoint calls for different deployment environments, such as switching between localhost and API gateway addresses based on the environment configurations.

Authentication and Account Services

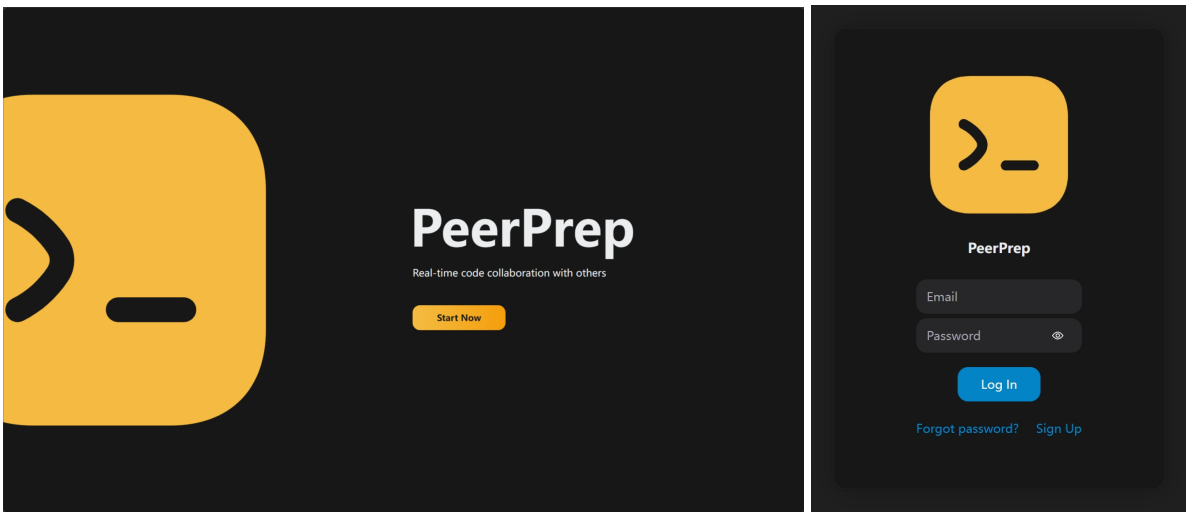


Figure 10, 11. Landing page (left), Login (right).

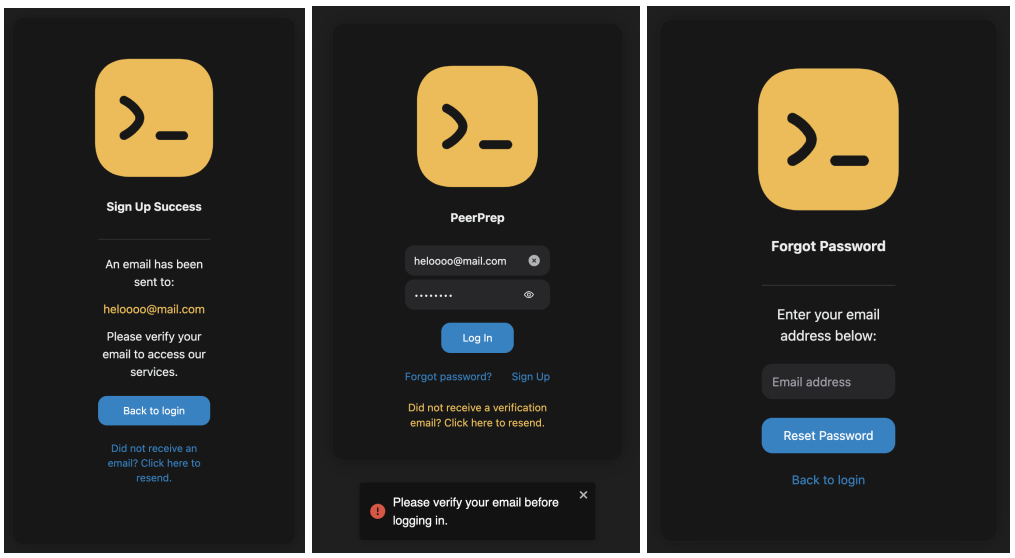


Figure 12, 13, 14. Sign up success (left), Unverified login (middle), Forgot password (right).

Users are able to sign up for an account using email and password. Before logging in, this email would need to be verified through a link sent to their email. To ensure that users are not locked out of their account if they did not receive or could not find the verification email, we make sure to include resend links both after a successful sign up and an unverified login. Users are also able to request for a password reset email should they forget their password. This makes sure that our Account Services are easy to use and reliable.

Dashboard

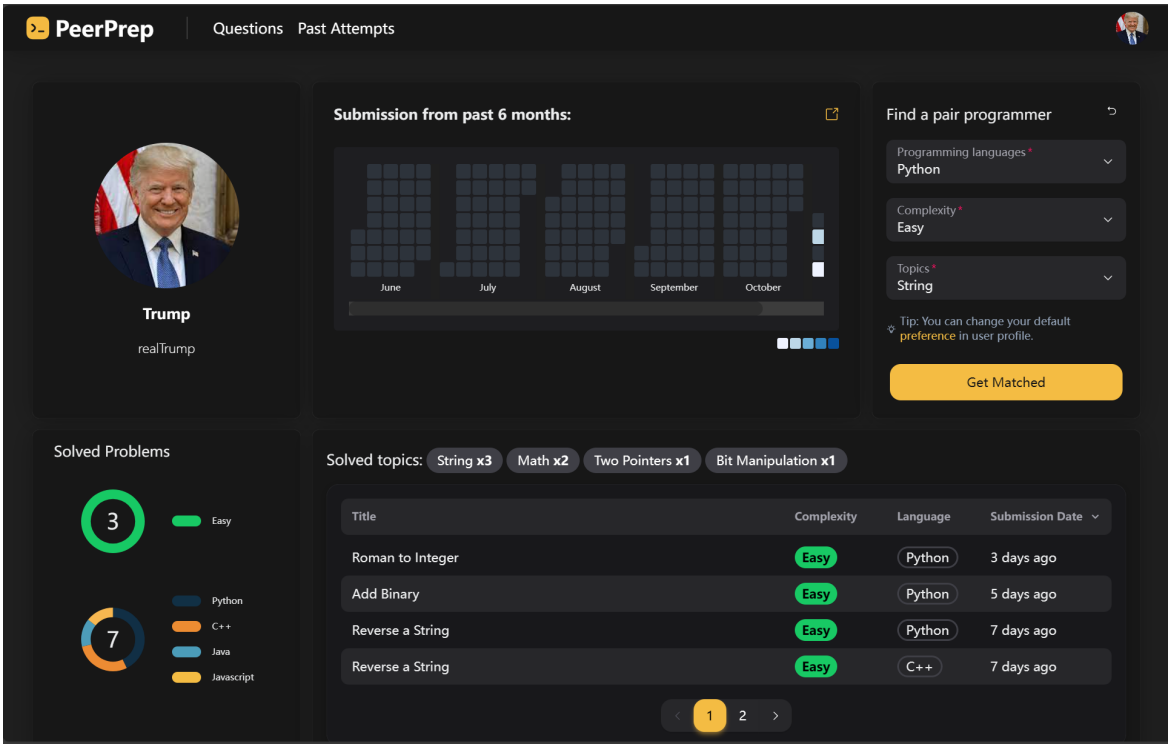


Figure 15. The dashboard shows an overview of the various features in our app, including the profile statistics and Matching Service.

User Service

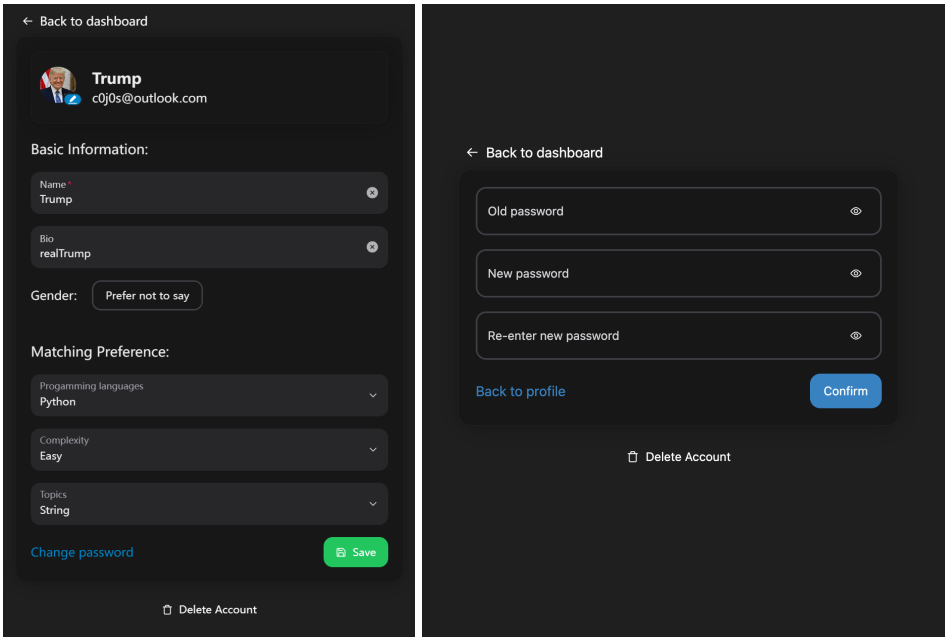


Figure 16, 17. Edit profile page (left), which includes the option to change password (right).

User Service allows users to be created in the sign up page. After email verification, users are successfully signed up and they are allowed to log in to their account to modify some of their personal details, which includes their preferences for their programming languages, complexity and topics. For this project, we allowed users to only select their preferred programming language from Python, Java, C++ and Javascript as these are the languages that are supported for this project. From the profile page, users can also change their password.

Question Service

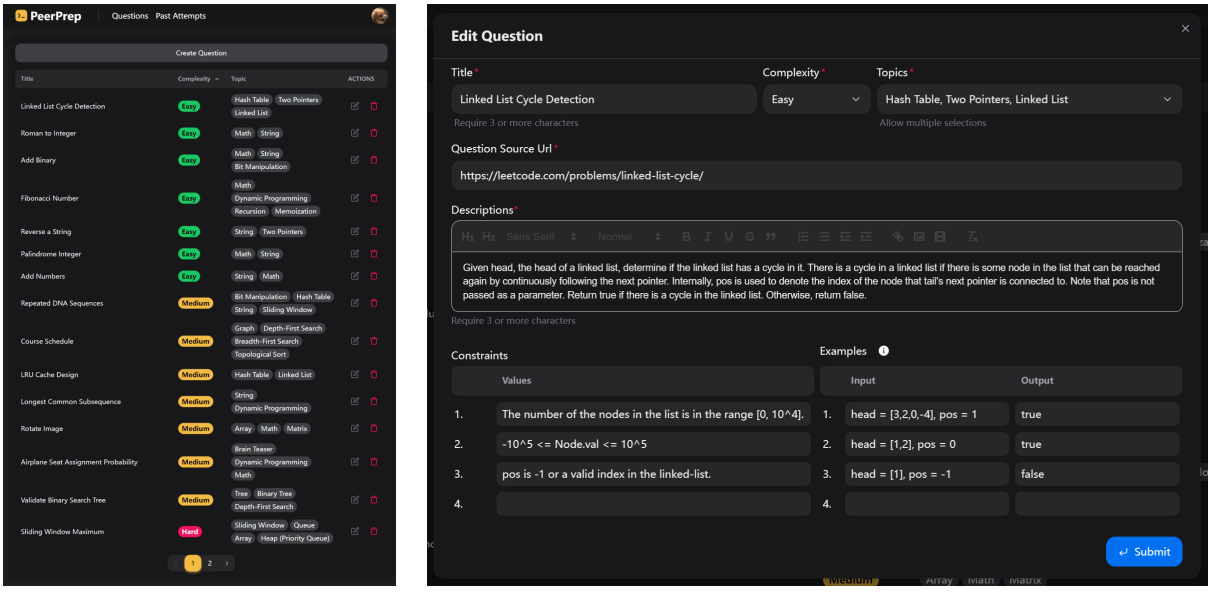


Figure 18, 19. Question Table and Modify Question Modal.

The Question Service offers various UI interfaces for authorized users to access, create, update, or delete questions within Peerprep's question bank. Accessible through the main navigation bar, the question page features a table displaying all available questions in the bank. This table includes additional metadata such as complexity and topics for user reference. Users have the ability to sort the list by clicking on column headers, and pagination is supported to reduce the need for excessive scrolling.

For normal users with limited permissions, the question table hides management features for the question bank. However, for admin users, a "Create Question" button and an additional actions column are visible, allowing them to create, update, and delete questions if necessary.

When an admin clicks on the "Create Question" button or the edit icon, a modal with the same layout appears, enabling admins to fill in and create or edit a question. This ensures a standardized approach to managing questions. The following table provides a detailed description of the input fields in the presented form.

Field	Required	Description	Validation
Title	Y	(Text) Title of the question.	>= 3 Characters
Complexity	Y	(Single selection) Complexity of the question. Ranging from Easy, Medium and Hard.	Valid Complexity
Topics	Y	(Multiple selection) Topics of the question.	Valid Topic
Question Source Url	Y	(Url) Url of question source.	Valid Url
Descriptions	Y	(Rich Text) Description of the question. Note: Images are allowed, but refrain from inserting large images.	>= 3 Characters
Constraints	N	(List) Constraints defined for the question.	Non-empty if filled
Examples	N	(List) Example inputs and outputs for the question.	Non-empty if filled

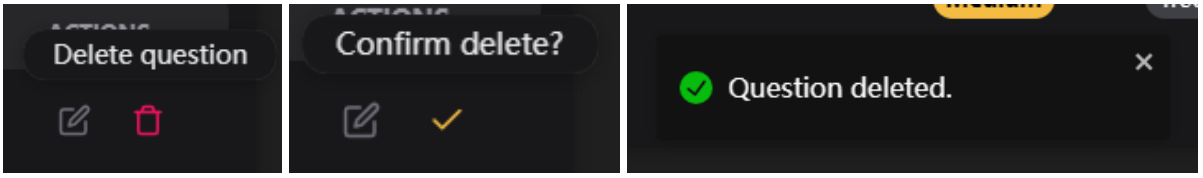


Figure 20. Delete Question UI Components.

The admin has the capability to delete a question by clicking on the trash bin icon located in the action column of the respective row. Upon clicking, the icon transforms into a tick icon to

confirm the deletion. Once confirmed, the question is removed, and the user receives a notification (toast) regarding the action's result. This process ensures a deliberate confirmation before permanently removing the question and provides immediate feedback to the user.

History Service

The application's frontend history feature will present users with comprehensive statistics regarding their past attempted questions. This includes details such as the count of attempted questions categorized by topics, difficulties, and programming languages. Additionally, users can view a table listing all questions attempted and a heatmap illustrating submission history over the past six months. This feature aims to provide users with valuable insights into their performance and engagement with different aspects of the question sets.

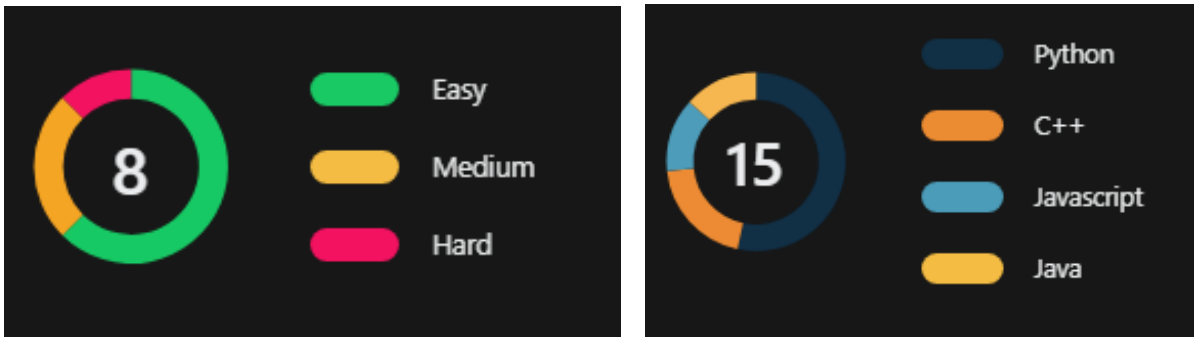


Figure 21, 22. Attempted Questions Statistics Donut Chart.

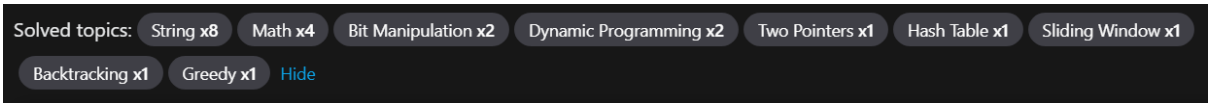


Figure 23. List of Attempted Questions By Topics.

To visualize a heatmap, we employ an external library known as *cal-heatmap*. Importantly, we leverage the useMemo hook during the rendering of the heatmap to avoid the re-computation of resource-intensive data when users refresh the screen. This contributes to the app performance by reducing the loads.

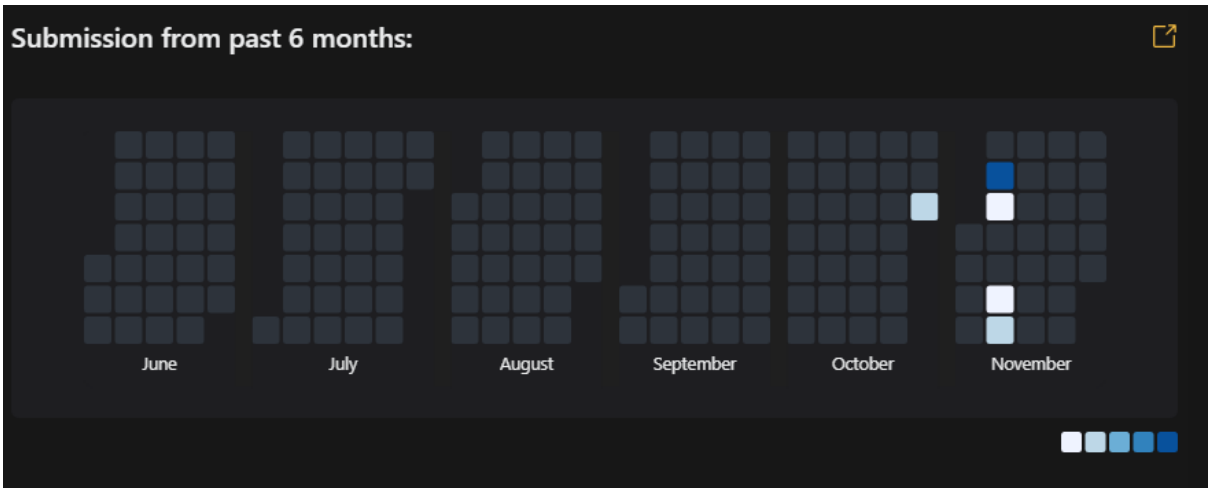


Figure 24. Heatmap of Attempted Code Submissions.

In rendering the attempted question table, we adhere to the Don't Repeat Yourself (DRY) principle. This is particularly important as the history table is presented both on the dashboard page and within the dedicated history page. By default, the table is sorted based on the submission time of each question in descending order, ensuring that the latest submission is consistently displayed at the top row. It's worth noting that users also have the flexibility to sort the table by clicking on the headers for title, complexity, and language. To enhance code efficiency, we also employ the useMemo hook during the table rendering process, improving the app performance.

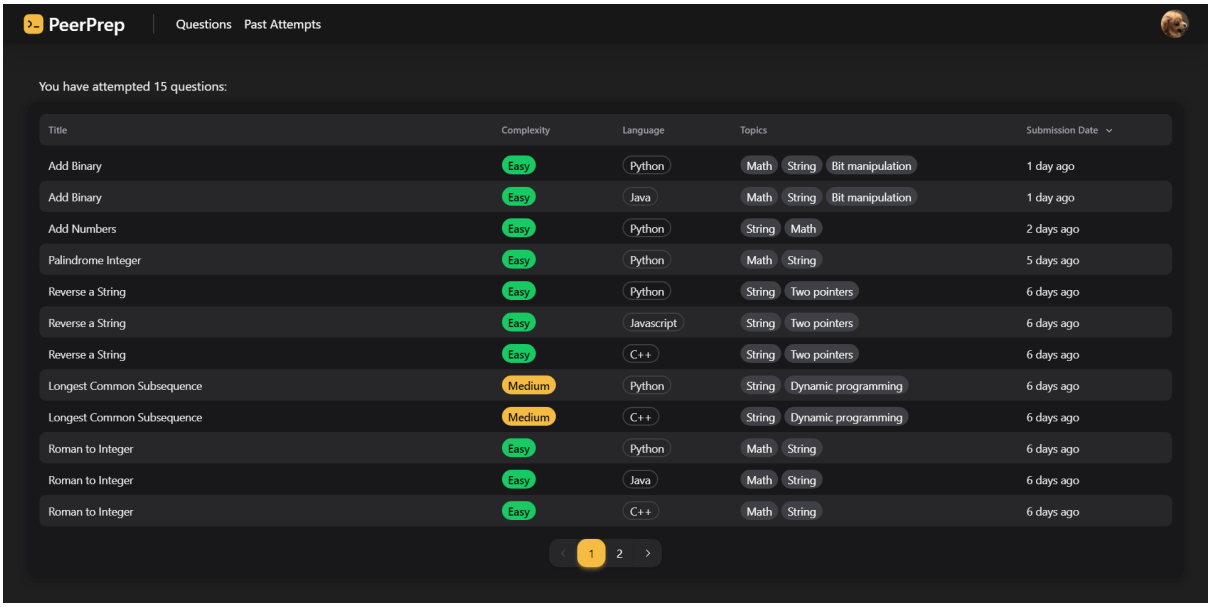


Figure 25. Attempted Questions Table.

Each question in the history table is clickable, and it will navigate the user to the detailed code submission page. The frontend component internally communicates with the History Service to fetch the submitted code as well as the question details.

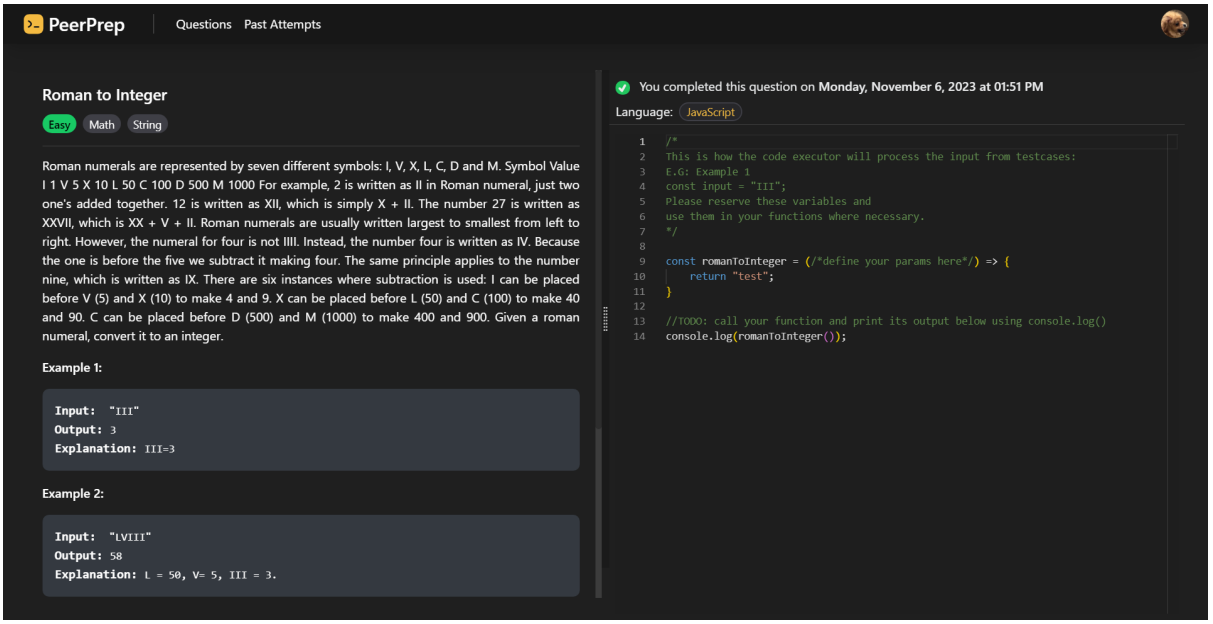


Figure 26. Code Submission Details.

Matching Service

The matching feature comprises two view components: the Matching Card and the Matching Lobby, along with a utility component called Socket Service for facilitating socket communication with the backend.

Matching Process

The following is a state diagram of the Matching Service for the frontend.

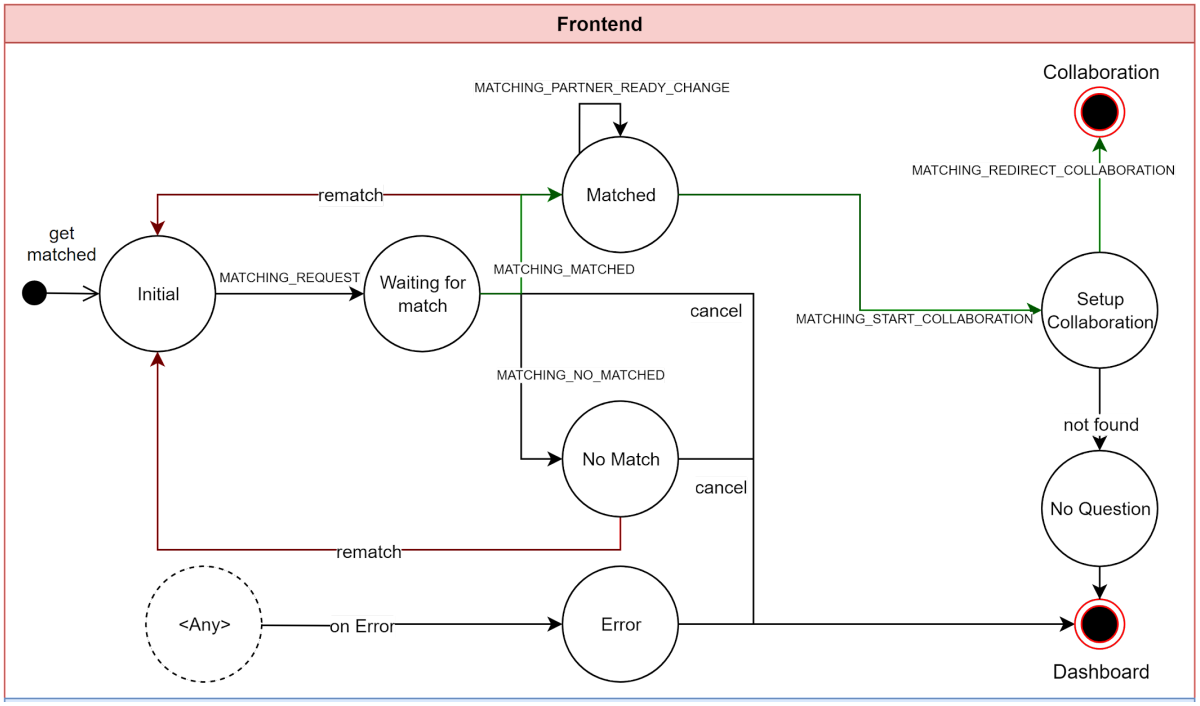


Figure 27. State Diagram for Matching Service Frontend.

The matching process begins when the user triggers the "Get Matched" option in the Matching Card. The Matching Lobby, which initializes a Socket Service, will be presented to the user as the view component for the entire matching process.

Upon instantiation, the Socket Service will establish a connection with the Matching Service. If successful, the lobby will switch to the waiting view while waiting for a match. The Socket Service will send the chosen preferences to the backend through the "REQUEST_MATCH" event.

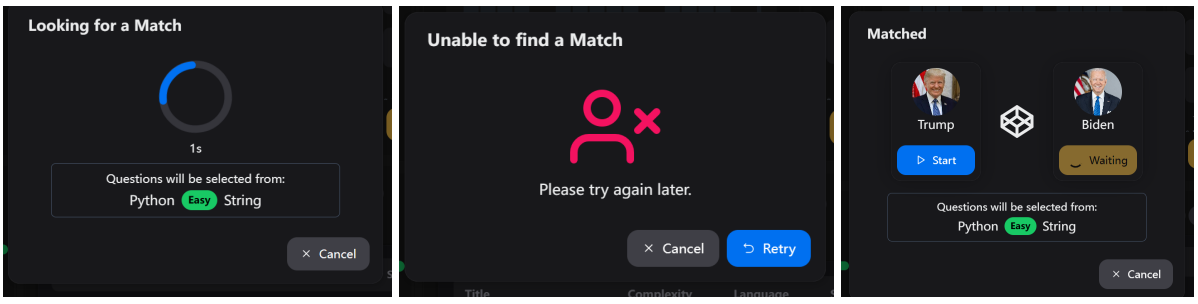


Figure 28. Matching Service UI Components.

In the matched state, the lobby will display the profile of the paired partner and the common preferences used for matching. To start a collaborative session, both users will have to indicate their readiness by pressing the start button in their respective profile sections. Internally, the Socket Service will trigger the "MATCHING_USER_READY_CHANGE" event to notify the backend of the partner's status.

Once both users are in the ready state, the host will be able to initiate the collaboration session. The session will start automatically after a 10-second countdown. Depending on the common preferences used for matching the users, the host will either navigate directly to the collaboration page or be given a setup or peer preparation prompt with one or two options to choose the language and questions for the collaboration session.

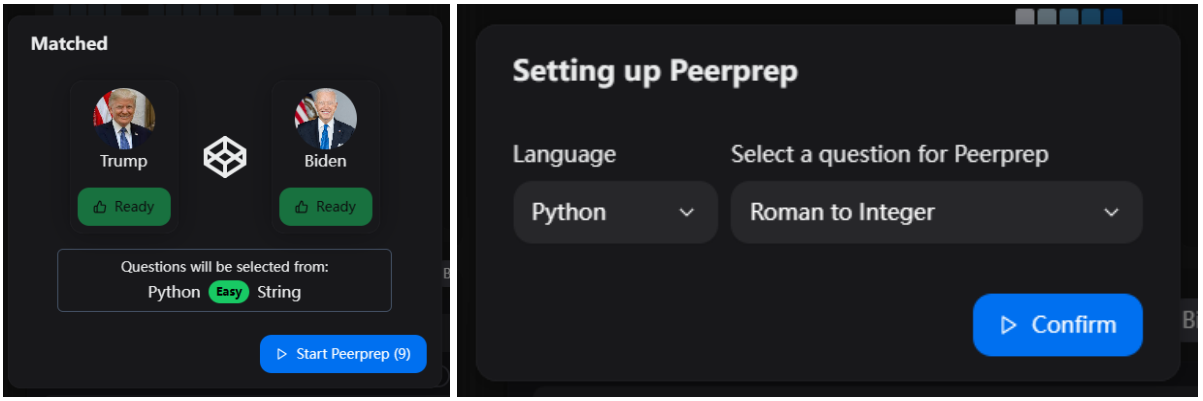


Figure 29, 30. Upon successful match (left), the host is able to select the language and question (right).

If the partner leaves the room for any reason (cancellation, network issues, or page refresh) before starting the collaborative session, the matched view will reflect the status and allow users to either cancel or request a rematch to find a new partner.

In this case, rematching will reuse the same preferences selected for the previous matching request, and users can cancel the process to change the options in the Matching Card.

Throughout the matching process, the Matching Lobby would switch into error state for any disruption (application error, network error).

Socket Events

Receiving Event	Description
MATCHING_MATCHED	On success of matching requests. Partner detail will be visible in the matched view.
MATCHING_NO_MATCHED	On timeout(dev:15s, prod:60s) of matching requests.
MATCHING_PARTNER_READY_CHANGE	On partner ready state change in matching view. Owner will be able to start a collaboration session when both users are in ready state.
MATCHING_REDIRECT_COLLABORAT	On completing the collaboration session setup (if any), user will be

ION	redirected to the collaboration page.

EmitingEvent	Description
MATCHING_REQUEST	Client sends a request containing preferences to start the matching process.
MATCHING_USER_READY_CHANGE	Client change its ready state to “ready”.
MATCHING_START_COLLABORATION	Client(owner) request to start the collaboration session when both users were in ready state.

Collaboration Space

The collaboration space only consists of 1 view that displays the problem statement, the code editor, the chat bubble as well as the code compilation and execution functionality. For the domain of this project, we only aim to support Python, Javascript, C++ and Java. This session will be created for 60 minutes, where users can work together in tandem to solve the problem together.

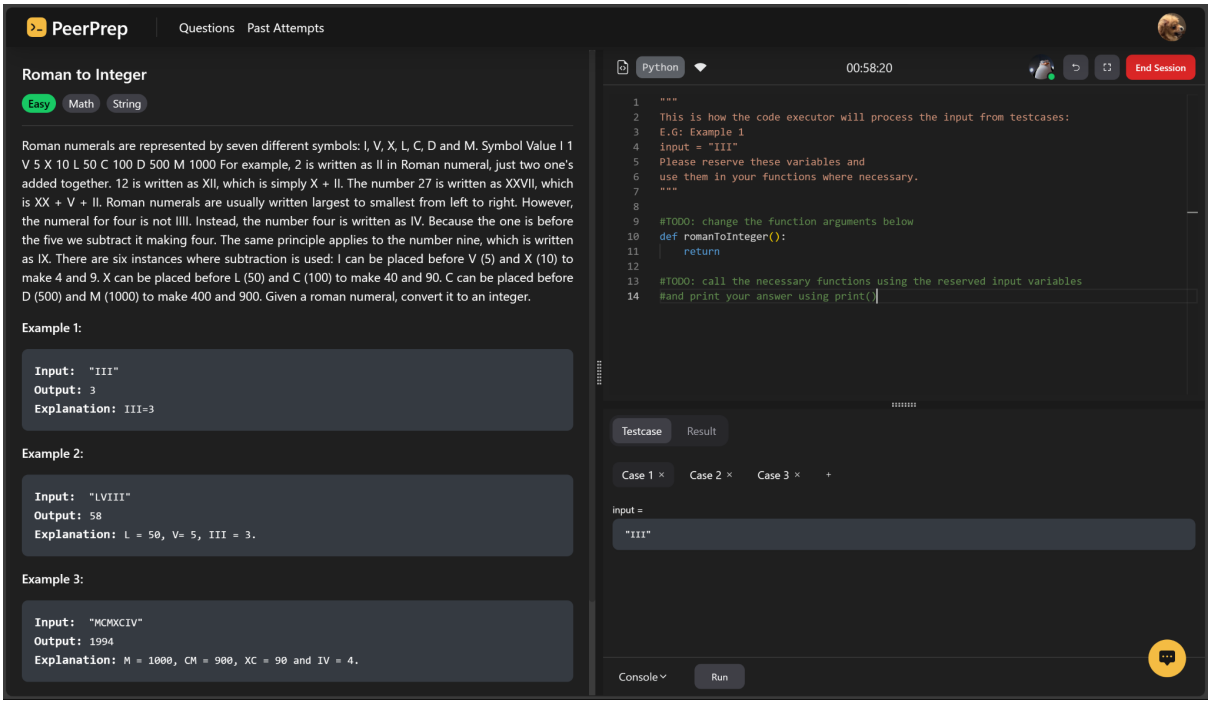


Figure 31. View of collaboration frontend.

The above is the view of the collaboration space, and has 3 main panels. The left shows the problem statement while the right shows the collaborative code editor with code execution functionality on the lower space. All the views are separated by an adjustable split view, thereby a user can simply drag the screen sizes to their own preference. The console can

also be toggled to be hidden away. All these attributes concur to our focus on achieving a high degree of usability.

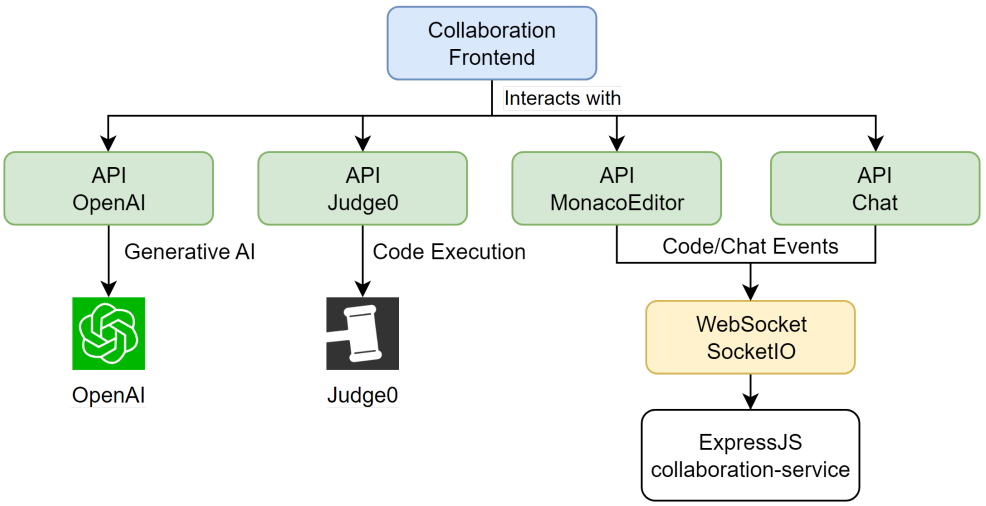


Figure 32. APIs that collaboration interfaces with.

The diagram above shows some external libraries that we used (OpenAI, Judge0, and MonacoEditor) and implemented ourselves (Chat).

Editor

We employed MonacoEditor API that interacts with the socket-io library to provide us with a sense of real-time collaboration. The MonacoEditor library provides us with an improved code editor that has IntelliSense and Validation built in, and has basic syntax colorization, overall providing an experience similar to Visual Studio Code. This library also allows us to subscribe to events made to the editor, such as selection events and content changes, which is what our collaborative space is built on.

:IModelContentChange
range: IRange rangeOffset: number rangeLength: number text: string

The way the collaboration space publishes code content changes are by the location of where these changes are made. It uses the IModelContentChange[] array and emits them to the server using the socketService. It depicts the range of the edit, the text that is supposed to be inserted/deleted in that space, as well as the offset and length of changes.

Figure 33, 34. Highlighting and cursor tracking.

The highlighting and selection cursor also works on the same methodology, except they only emit a range that tells the other user the location or selection of the user's cursor based on the ICursorSelectionEvent.

Chat

To communicate with your partner, we have set up a Chat functionality where users are able to open and dismiss it. Upon receiving a chat message while the bubble is unopened, they will receive a notification. Users are also able to align the chat to the left or right of the window for improved usability, based on personal preference. Both the chatting and editor changes are emitted as socket events to the backend and can be further elaborated in this diagram here.

For generative AI, we used OpenAI's GPT4 API that generates a message upon the command "/ai" within the chat. This calls the OpenAI API instead of the chatting API and allows us to ask questions related to the puzzle with the help of GPT4.

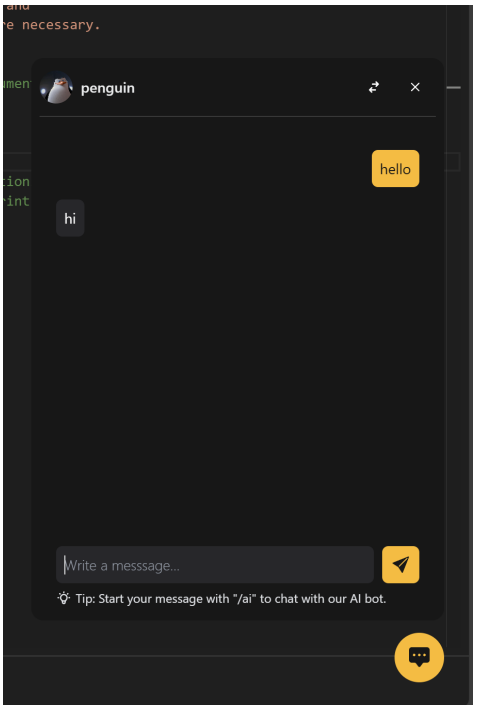


Figure 35. Chatspace.

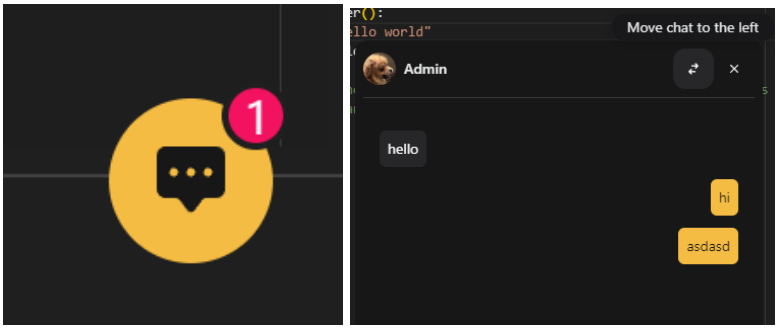


Figure 36, 37. Details - Notification counter if message is sent while the chatspace is not open (left), Option to toggle chatspace to the left or right of the screen (right).

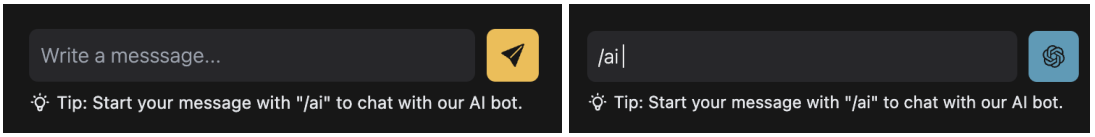


Figure 38, 39. The send button is usually in yellow (left). Once /ai is typed at the start of the message, the button will turn blue with the ChatGPT logo, indicating to the user that this message will be sent to ChatGPT (right).

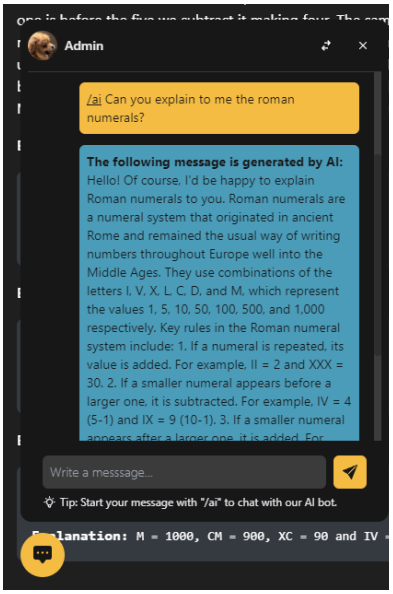


Figure 40. A reply from ChatGPT will be in blue. The same message, also in blue, will be seen on the other party's chatspace.

Code Execution/Compilation

Using Judge0 API, we have guaranteed code compilation and automated test case execution for Python and Javascript. We have chosen to implement these two languages due to their increasing popularity and demand in the market, and have decided to only ensure code compilation for C++ and Java.

For Python and Javascript, we support automated test case execution by parsing the input from the test cases. After the user presses "Run", the input will be dynamically inserted as variables into the users' code before sending it to the Judge0 API. This way, users can simply use the variables in their code and do not have to handle reading of inputs by themselves. This is common practice in programming exercise websites like LeetCode, and we believe that this feature will greatly enhance the usability of our code collaboration feature.

To make sure that the instructions are clear to the users, we provide them with an auto-generated example of how the input will be parsed with respect to the first example in the question.

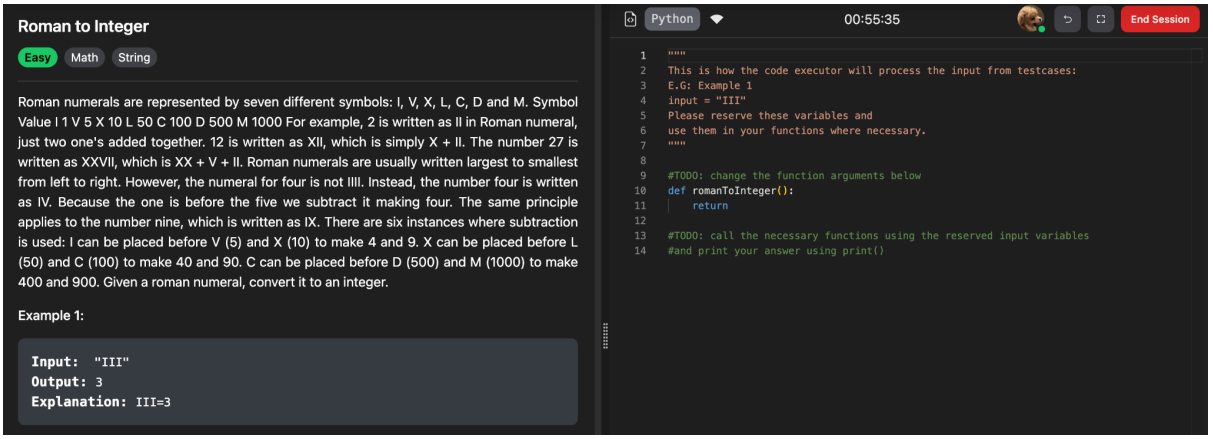


Figure 41. Instructions for Python.

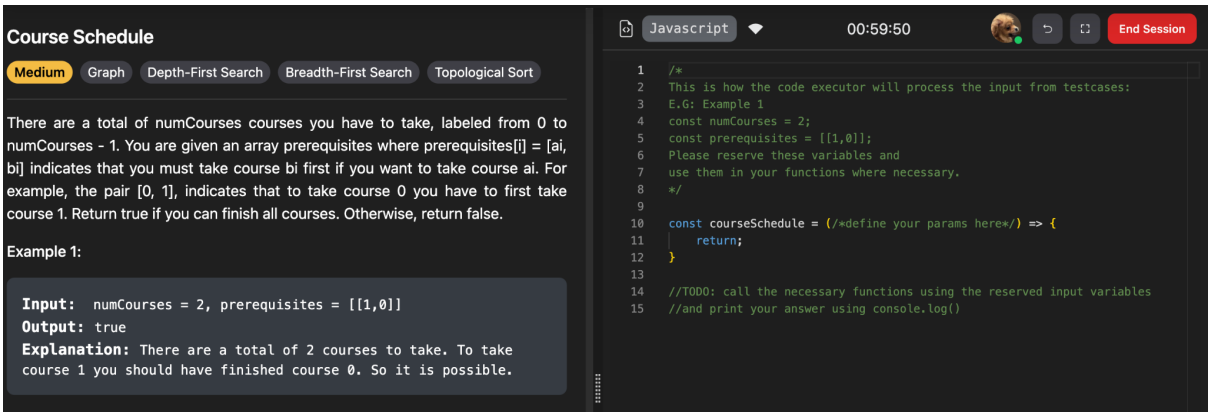


Figure 42. Instructions for Javascript.

All the examples given in the question will be parsed into test cases. Users can duplicate these test cases, edit or delete them as they wish.

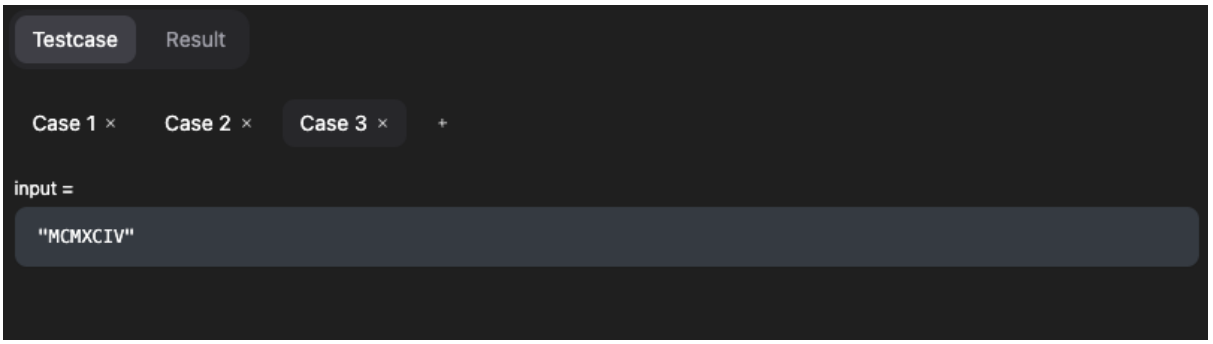


Figure 43 . UI for test cases.

Finally, the user can press “Run” to execute all the test cases at once. For test cases which are extracted from the examples in the question, since we are able to determine the correct answer from the example, we will also check for the correctness of the users’ output. For custom test cases, since we do not have a solution code, we will simply provide the execution result of their code.

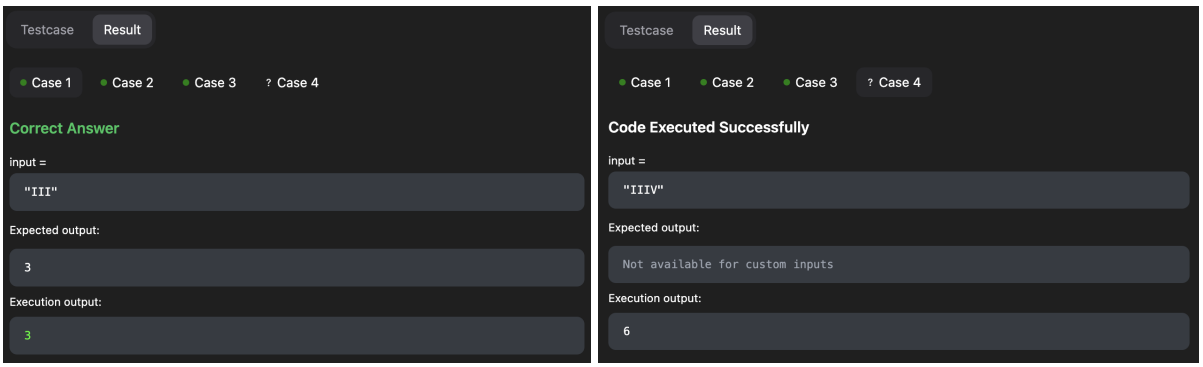


Figure 44, 45. Execution result for a given test case (left) and custom test case (right).

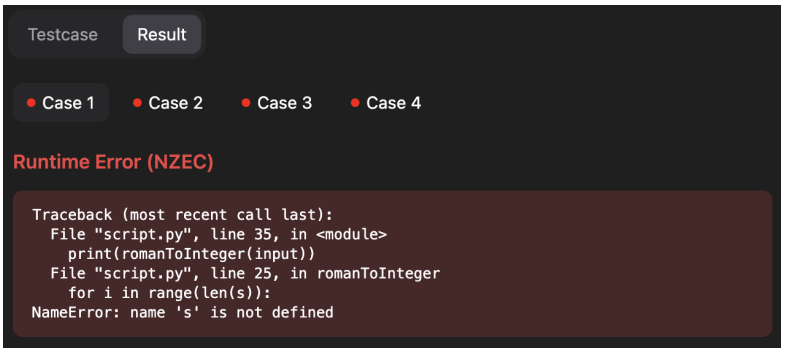


Figure 46. Compilation and runtime errors are shown as well.

The inputs are carefully parsed to account for differences in how certain variables are declared in the two languages, for example null is “None” in Python and “null” in Javascript. This ensures that the code that we initialize for them is error free.

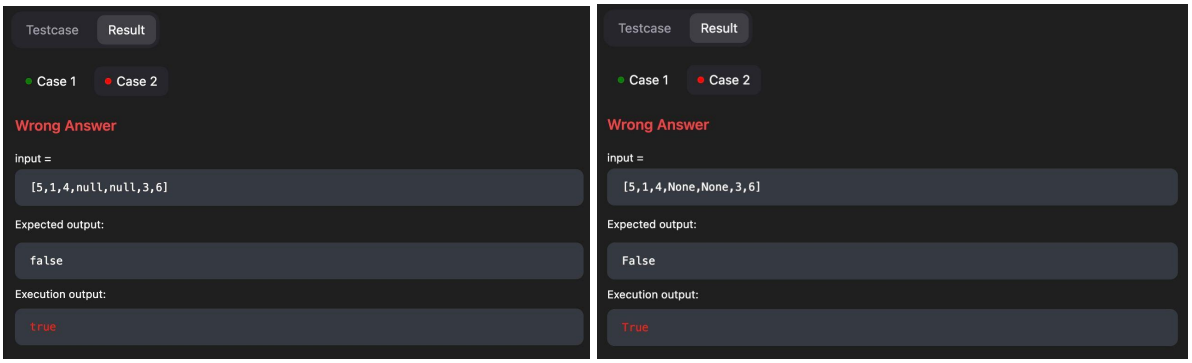


Figure 47, 48. The same test case is parsed differently for the two languages, Javascript (left), Python (right).

For C++ and Java, we decided to omit automated test case execution as the way that these two languages handle different data structures is much more complex. To ensure that the parsing is done accurately, it is ideal to have a manually written driver code for each question. Due to the limited time frame for the project, we decided to channel our time and effort into making sure the other features run perfectly instead. However, users are still able to write, compile and view the compilation results in these languages.

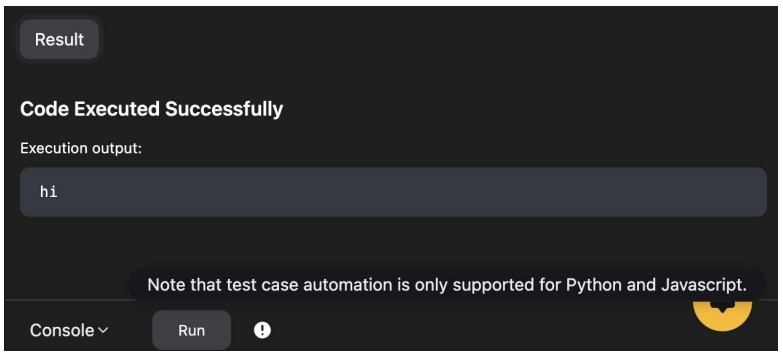


Figure 49. For C++ and Java, a tooltip beside the "Run" button tells users that test case automation is not supported when it is hovered. However, the console still shows the execution output.

When a user decides to terminate or when the timer has run out, he/she will receive a modal that prompts him/her to confirm the termination or return to the dashboard since the session has already ended. Clicking on "Terminate" or "Back to dashboard" creates a post request to the History Service to save the attempt by the user.

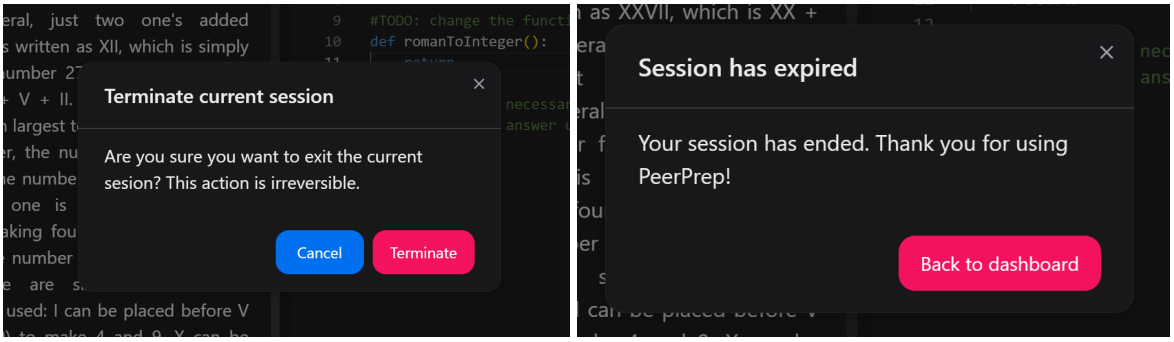


Figure 50, 51. Terminating the sessions from the frontend.

Backend Architecture

Design Pattern - Microservice Architecture

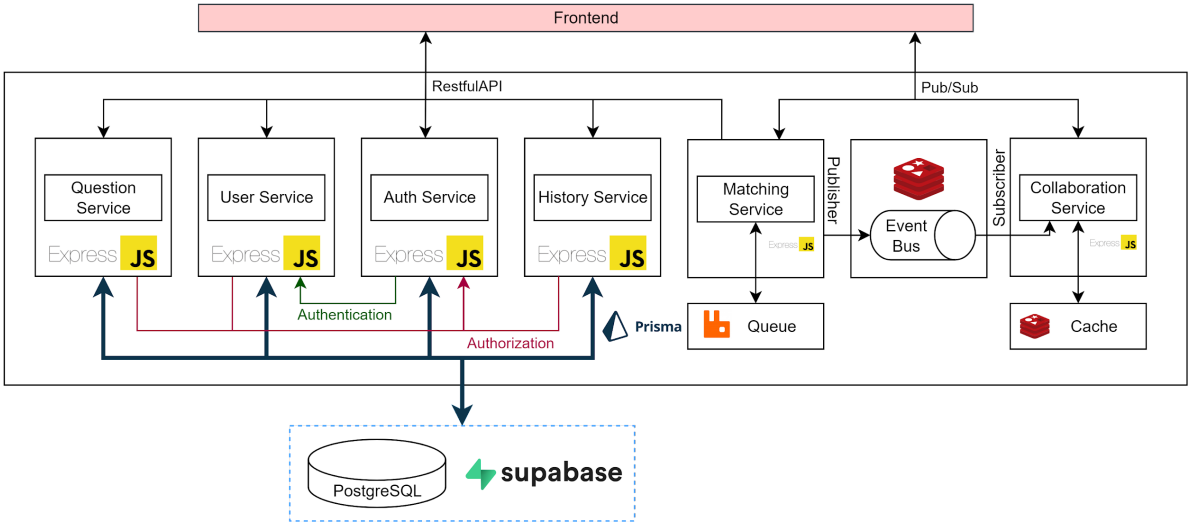


Figure 52. Backend Architecture Diagram.

Peerprep's backend architecture comprises six different services: User, Question, Auth (Authentication + Authorisation), History, Matching, and Collaboration. This microservices approach is adopted to enhance modularity, scalability, and separation of concerns. Adhering to the single responsibility principle, the development process is significantly expedited as this architectural design approach enables concurrent work, and ultimately contributes to improved code maintainability, complying with our primary quality attribute.

In terms of the database, the backend utilizes two Redis instances for caching and relies on an external PostgreSQL database hosted on Supabase as the primary data layer solution. Despite introducing some degree of coupling, the centralized database streamlines data management and eliminates the need for intricate logic to consolidate data from various microservices. Additionally, it greatly simplifies the work needed to ensure data integrity by establishing and enforcing relationships and constraints systematically. This centralized approach contrasts with the complexities associated with data synchronization when utilizing separate databases.

Each microservice is developed using Express, a NodeJS web application framework known for its speed, minimalism, and flexibility in constructing RESTful web APIs and servers. The Question, User, Auth, and History Services each have their respective APIs, while Matching and Collaboration Services incorporate a socket protocol to facilitate real-time communication with external services. Further details on the usage of the cache layer are available in the Matching and Collaboration services documentation.

In ensuring the security and authenticity of our APIs, each microservice internally communicates to authenticate incoming requests. For a more comprehensive explanation of the authentication flow, please refer to the Auth Service documentation.

Furthermore, to streamline database communication, Prisma, an Object-Relational Mapping (ORM) tool, is integrated into the microservices. This incorporation facilitates the management of entities, relationships, and data queries with the external PostgreSQL database.

Design Pattern - Model-View-Controller (MVC)

For each microservice design, we adhere closely to the Model-View-Controller (MVC) architecture when designing the interfaces of the server. The MVC is a design pattern that provides better separation-of-concern, maintainability, and testability by modularising the application into three interconnected components, namely the model, view, and controller.

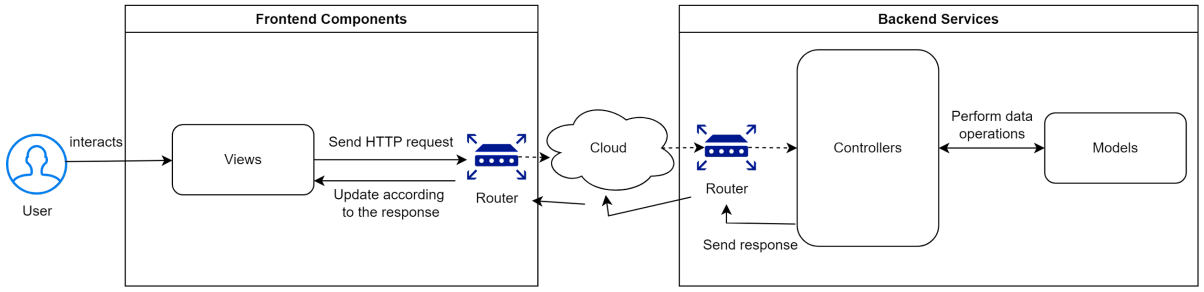


Figure 53. MVC framework in Peerprep Backend Services.

Model

The model represents the application data and business logic. In our design, the model corresponds to the database schema, engaging in interactions with the database to execute operations, including querying, updating, and deleting records. Comprehensive details regarding the model design for each service can be found in their respective microservice sections.

View

The view encapsulates the presentation of data, commonly encompassing layout components and user interfaces directly visible to the user. In our microservices architecture, the view of each service is manifested in the frontend layer instead of each backend service.

Controller

The controller, positioned between the model and the view, controls the flow of application execution. In our backend services, controllers can be viewed as the endpoints available in each microservice API or event handlers. They play a crucial role in managing the data flow within the application's (models) and ensure the appropriate rendering of user interfaces (views).

Auth Service

The Auth Service handles all the logic that has to do with authentication as well as authorisation (to be termed collectively as “Auth”).

Since the nature of Auth requires information of the users, coupling between Auth and User logic is unavoidable. To reduce the dependency of Auth on the User Service, we decided to connect Auth directly to the user database. For read operations such as checking if a certain user id exists, Auth will directly retrieve the information from the database. However, for write operations such as creating or updating of user information, Auth will still call on the User Service endpoints to make these changes. This ensures that there is still a single source of truth for all user data.

Endpoints

Below is a table that lists out all the endpoints available in Auth and their functionality.

Method	Endpoint	Description
GET	/health	Check if the API is in good condition and is connected to the database.
	/verifyResetPasswordLinkValidity/:userId/:token	Verifies that the reset password link is valid for the certain user.
POST	/registerByEmail	Used to register a new user by email. Internally, it calls on the User Service to create a new user.
	/loginByEmail	Used to log in a user by email.
	/logout	Used to log out a user and clear the HTTP-Only cookie containing the JWT token.
	/validate	Used to authenticate a user. A user is considered authenticated if the request header contains a HTTP-Only cookie with a valid JWT token.
	/validateAdmin	Used to authenticate an admin. An admin is considered authenticated if the request header contains a HTTP-Only cookie with a valid JWT token and the user it corresponds to is an admin.
PUT	/resendVerificationEmail/:email	Used to resend a verification email to the user's email. Internally, it calls on the User Service to update the user's record in the database with a verification token that will be used to verify the validity of this link.
	/sendPasswordResetEmail/:email	Used to send a password reset link to the user's email. Internally, it calls on the User Service to update the user's record in the database with a password reset token that will be used to verify the validity of this link.
	/changePassword/:id	Used to change the password of the user. There are 2 use cases for this endpoint: Case 1. User who is logged in is trying to change their password Case 2. User who is not logged in is trying to change their password by using the

		password reset link. In either cases, this endpoint will handle verifying the necessary information. Internally, it calls on the user service to update the user's record in the database with the new password.

Auth is implemented to be used either independently, or as a middleware for other services.

As an independent service, Auth handles the registration, email verification, login, logout and password reset flows. Its responsibility is to issue and validate encrypted JWT tokens for such requests in order to ensure that they come from an authorized source.

registerByEmail Flow Illustration

Here, we illustrate the *registerByEmail* flow in detail.

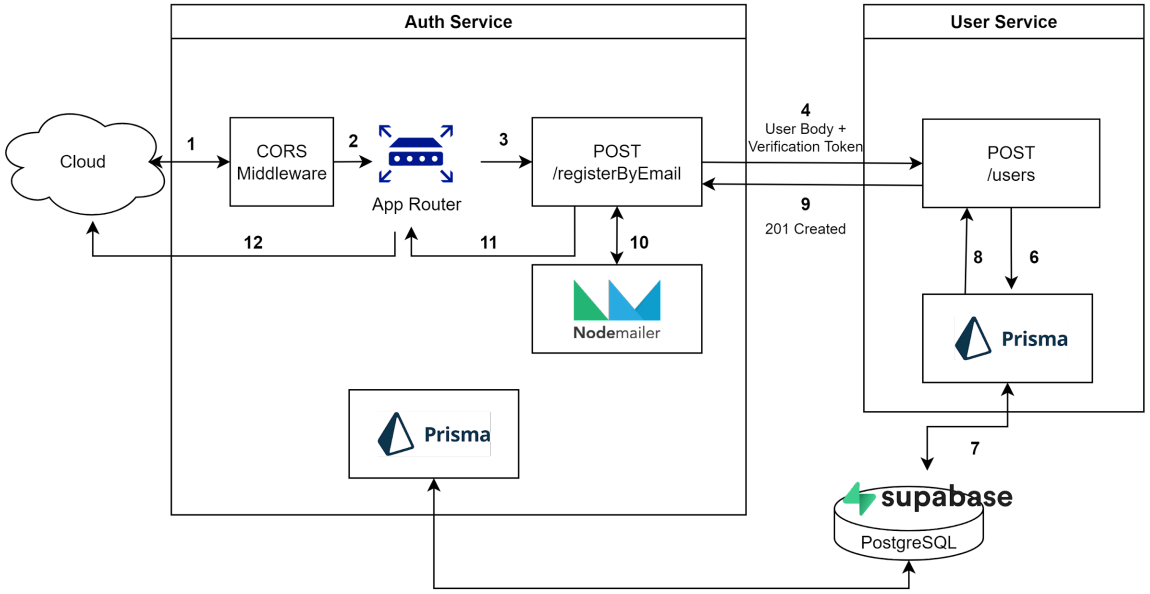


Figure 54. Flow for the registerByEmail endpoint in the Auth Service.

Steps

1. Incoming HTTP requests reach the Auth Service API, intercepted by the Cross-Origin Resource Sharing (CORS) middleware.
2. Only forward requests sent from authorized hosts/origins to the app router.
3. Forward authorized request to the *registerByEmail* handler.
4. Parse the request body containing the new user information and generate a verification JWT token. Send the user body with the verification token to the *users* handler in the User Service.
5. (User Service) User Service does the necessary validation checks to ensure that the request body is valid.
6. (User Service) Interact with the Prisma ORM tool to access the models and send the database query to update user information.
7. (User Service) Handle data query with external database.

8. (User Service) Send the query result to the handler.
9. (User Service) Construct the HTTP response according to the query result. In this case, we illustrate a successful request.
10. The *registerByEmail* handler receives the successful response and calls on the Nodemailer library to send out a verification email to the registered email.
11. Construct the HTTP response based on the result.
12. Send the response back to the caller.

As a middleware, Auth is implemented as an app-level ExpressJS middleware for other microservices which require access control, namely the User, Question and History Service. All incoming requests to these services will first be intercepted by this middleware, where a call will be made to either one of these two endpoints in Auth for user validation - */validate* and */validateAdmin*. */validate* will allow all registered users access, while */validateAdmin* will only allow registered admins access. These two endpoints allow the various microservices to fine-tune the level of access on certain routes. If Auth rejects the validation, the request will not go through and will return Unauthorized/Forbidden. Else, the request will be forwarded to the original intended route handler. A detailed example of how the Auth middleware plays a part in the logic flow of User Service is illustrated in Figure 58. Note that since Auth Service internally calls on User Service endpoints, the middleware in User Service is implemented such that calls from the Auth Service bypasses its own middleware, else it will result in an infinite loop.

The core logic for authentication involves checking for the validity of a JWT token embedded in a HTTP-Only cookie in the request headers. We chose to embed the token in a HTTP-Only cookie as it provides additional security by preventing client-side scripts, such as Javascript, from accessing and modifying the cookie's value, mitigating the risk of malicious attacks such as Cross-Site Request Forgery. Once a user is logged in, a cookie will be issued and attached to the response headers. This cookie will contain a JWT token that has been encoded with the user id and an expiry date. On the backend, this cookie is automatically forwarded with all subsequent requests. On the frontend, since Next.js has its own server, the cookie will need to be manually extracted and attached as a client-side cookie, so that all subsequent requests from the client will contain this cookie. In the two endpoints for validation, we check for the validity of the JWT token. This involves two steps. First, we utilize the library "passport-jwt", which will handle extraction, decoding and validation of the token. After it passes the validation check by "passport-jwt", we will then query the database to see if the user id that the JWT token encodes actually exists. This is an additional layer of security to ensure that the JWT token belongs to a valid and existing user.

Validate Flow Illustration

Here, we illustrate the *validate* flow in detail.

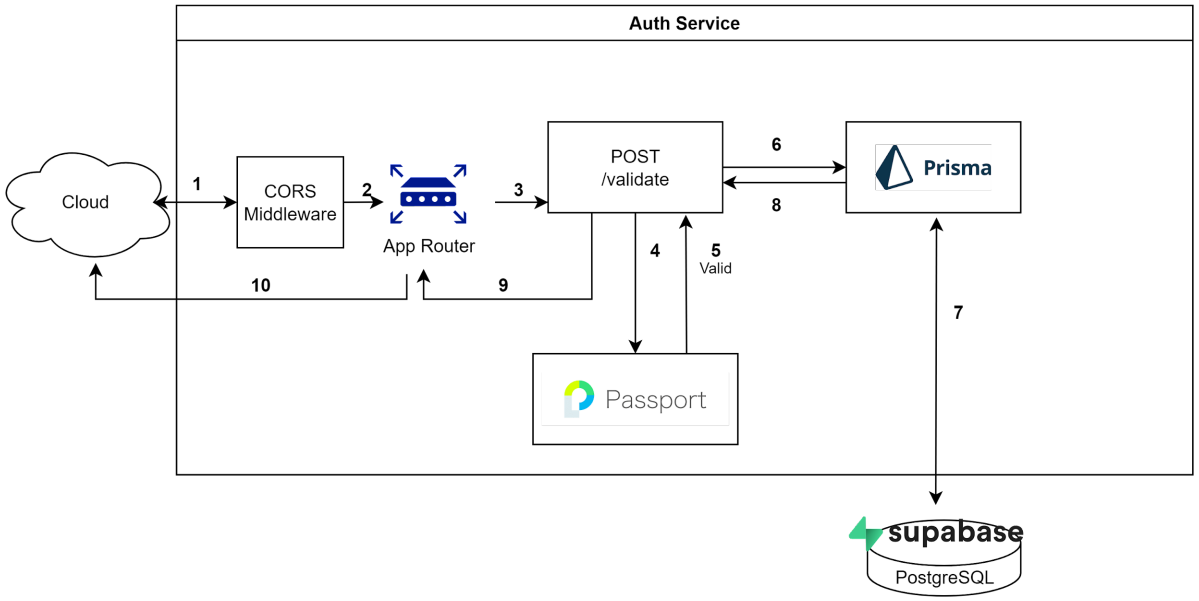


Figure 55. Flow for the validate endpoint in the Auth Service.

Steps

1. Incoming HTTP requests reach the Auth Service API, intercepted by the Cross-Origin Resource Sharing (CORS) middleware.
2. Only forward requests sent from authorized hosts/origins to the app router.
3. Forward authorized request to the *validate* handler.
4. Send the request to passport-jwt, a strategy under passport-js. Passport-jwt will extract the token, validate it and allow the request to go through if valid. If invalid, it will return 401 Unauthorized directly to the router. Here, we demonstrate the flow for a valid token.
5. The validate handler implements a custom validity check. Here, it will query the user database to ensure that the user id encoded in the JWT token exists.
6. Interact with the Prisma ORM tool to access the models and send the database query to update user information.
7. Handle data query with external database.
8. Send the query result to the handler.
9. Construct the HTTP response based on the result.
10. Send the response back to the caller.

The benefit of implementing Auth as an individual microservice, instead of having each microservice handle their own access control, is that there is clear separation of concerns. Since other microservices just need to call the relevant endpoints from Auth to validate the access, they do not have to be concerned with the authentication flow. Consequently, this makes it much easier to extend and alter the authentication flow in the future with minimal changes required across services, for example by adding more authentication providers such as OAuth.

User Service

The User Service is one of the microservices in the application backbone. The primary goal of this service is to manage all the user-related data operations, including read, write, update, and delete operations. Please note that User Service does not include authentication functionality as the latter involves complex logic in handling different situations and has been separated from User Service to achieve better code modularity.

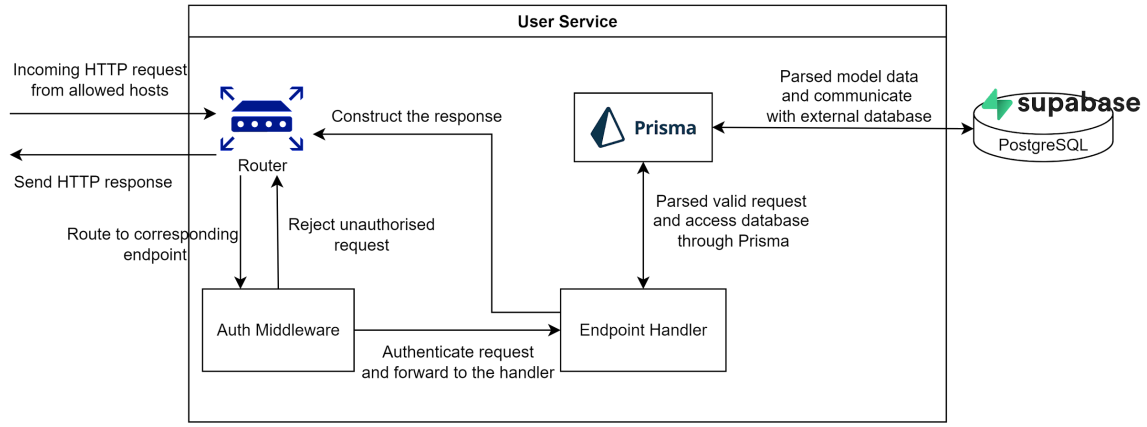


Figure 56. Architectural Diagram of User Service.

Above is a high-level architectural diagram of User Service. The User Service comprises a RESTful web API integrated with a cloud PostgreSQL database hosted in Supabase that supports multiple operations. The API is composed of a router, middlewares, handlers, and the Prisma ORM tool. The endpoint handler is the MVC controller under this design.

Endpoints

There are multiple operations supported by the API. Below is a table that lists out all the endpoints available and their functionality.

Method	Endpoint	Description
GET	/health	Check if the API is in good condition and is connected to the database.
	/users/:userId	Retrieve information related to the user with the *userId*. This information includes the name, email, gender, role, preferences, etc.
	/users/email	Similar to the above endpoint, but filter users with the *email* instead.
	/users/:userId/preferences	Retrieve the preferences set by the user. The preferences include preferred programming languages, as well as the question difficulty and topics.

POST	/users	Create a new user with necessary data and store the information into the database.
PUT	/users/:userId	Update the user information for the user with id *userId*.
	/users/:userId/preferences	Update the preferences for the user with id *userId*.
DELETE	/users/:userId	Remove the user details from the database.

Models

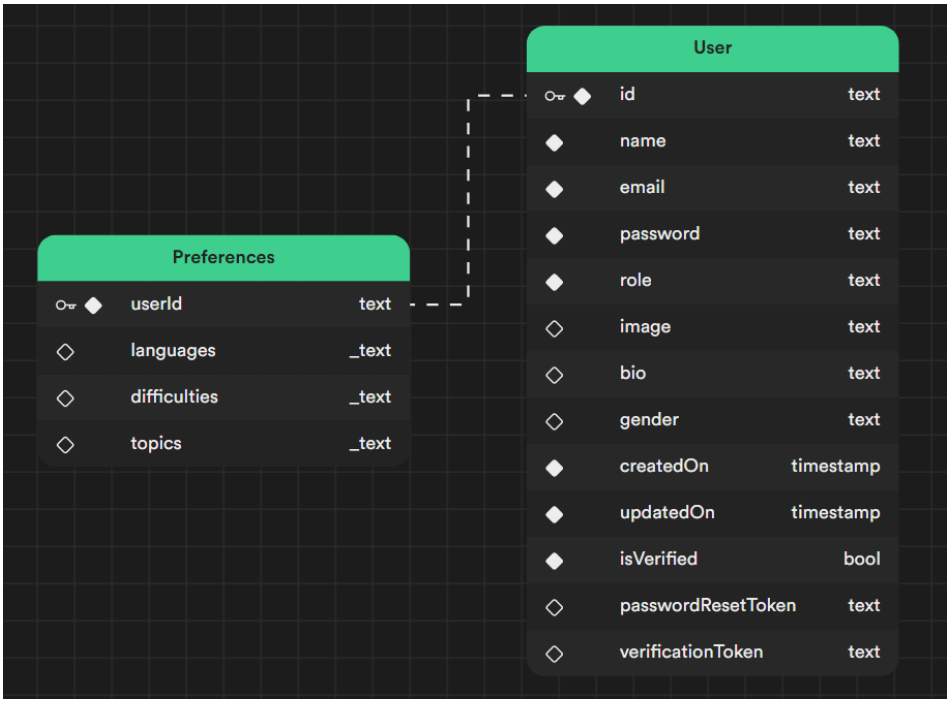


Figure 57. User Service Schemas Visualization.

The depicted schema design provides an overview of the models within the User Service. Notably, the Preferences schema establishes a foreign key constraint relationship with the User schema. Additionally, various constraints, including the NOT NULL constraint for each attribute, are denoted by a distinctive white diamond shape.

getUserByEmail Flow Illustration

We will explain the *getUserByEmail* flow execution in detail to better illustrate the structure of the User Service and the logic flow.

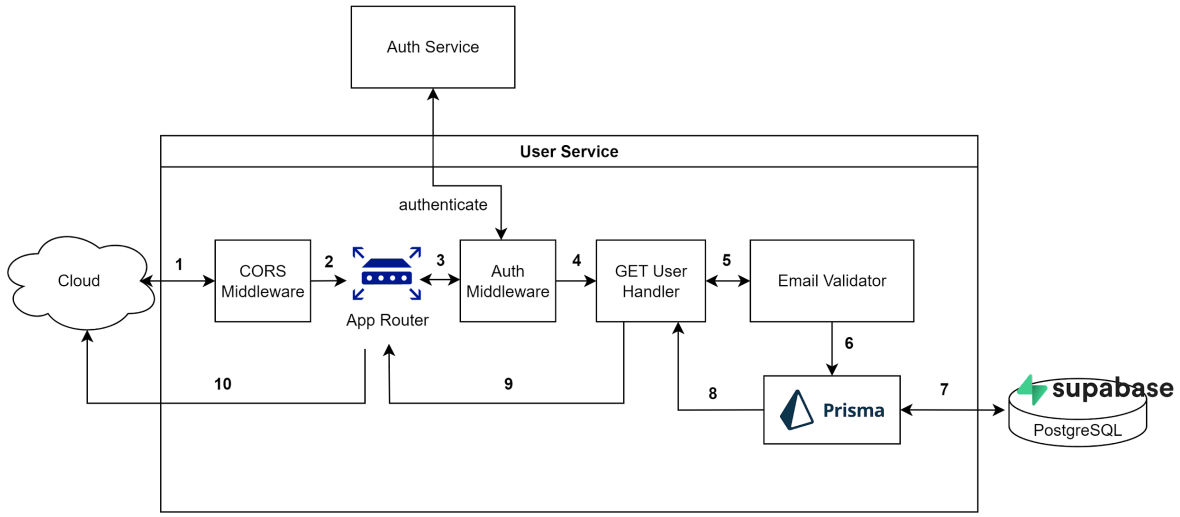


Figure 58. Flow Diagram of getUserByEmail.

Steps

1. Incoming HTTP requests reach the User Service API, intercepted by the Cross-Origin Resource Sharing (CORS) middleware.
2. Only forward requests sent from authorized hosts/origins to the app router.
3. The request is intercepted by the Auth middleware to ensure the requests are sent from authorized users registered in Peerprep.
4. Forward authenticated requests to the *getUserByEmail* handler.
5. Parse the request to check for email validity in the request query parameters.
6. Interact with the Prisma ORM tool to access the models and send the database query to retrieve user information.
7. Handle data query with external database.
8. Send the query result to the handler.
9. Construct the HTTP response according to the query result.
10. Send the response back to the caller.

Error handling

In order to enhance the application's resilience and robustness, considerable effort has been dedicated to addressing various error scenarios that may arise during client communication with the API.

In case the CORS check fails, the request is immediately blocked and will not reach the app router. For auth check failure, the HTTP response will have a status code of 401 (Unauthorized), and be sent back to the app router. For validator check failure, the HTTP response will have a status code of 400 (Bad Request), and be sent back to the app router.

Question Service

The Question Service maintains questions storage, and provides questions from the question bank to matched users according to the filters set by their preferences or matched options. Similar to User and Auth Service, the Question Service contains a RESTful web API for synchronous request and reply communication.

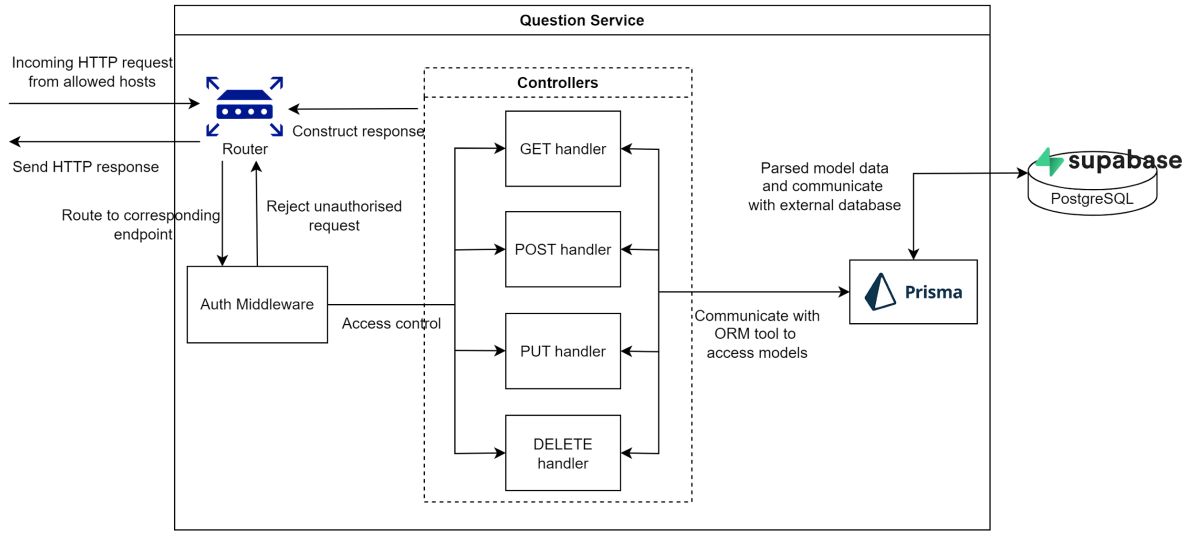


Figure 59. Architectural Diagram of Question Service.

Unlike User Service which allows the authenticated caller to access all endpoints, Question Service provides stricter access control to the handlers. Without having a match, a non-admin caller can only access the GET handler, where he or she can see how many questions and get individual question details. However, the caller has no access to create new questions, nor update existing questions and delete questions. Only the ADMIN role user can perform these operations. The access control is done in the Auth middleware, which interacts with only the Auth Service for role checks, to prevent deep coupling with the User Service.

Endpoints

Below is a table of all the API endpoints available in Question Service.

Method	Endpoint	Description
GET	/health	Check if the API is in good condition and is connected to the database.
	/questions	Retrieve a list of questions available in the question bank, omitting the question details such as the description, examples, and constraints of the question to save bandwidth when transmitting data.
	/questions/:questionId	Retrieve specific question information with id *questionId*, with all the details available.

	/topics	Retrieve all distinct topics available for all questions in the question bank.
POST	/questions	**(Restricted to ADMIN)** Create a new question and save it to the question database.
PUT	/questions/:questionId	**(Restricted to ADMIN)** Update the question details for the question with id *questionId*.
DELETE	/questions/:questionId	**(Restricted to ADMIN)** Remove the question from the database.

Models

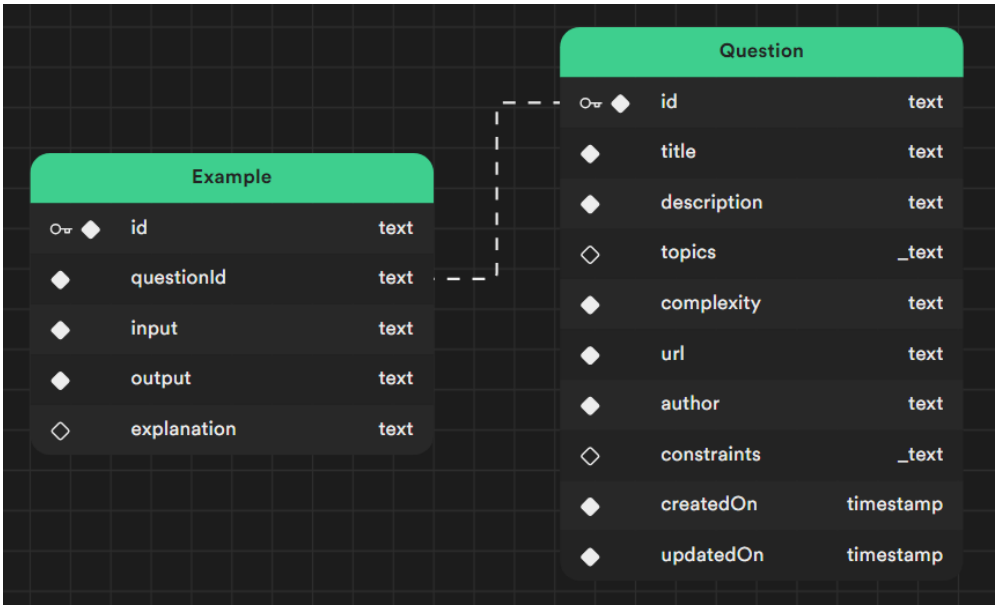


Figure 60. Question Service Schemas Visualization.

The Question models consist of two schemas, the Question schema and the Example schema. Similar to User Service, the Example schema has a foreign key constraint to the Question id, and the NOT NULL constraint is represented using the white diamond shape in front of each attribute.

Flow Illustration

To better describe the request-reply pattern in Question Service, we will use the *createQuestion* API call as an example to illustrate the execution flow.

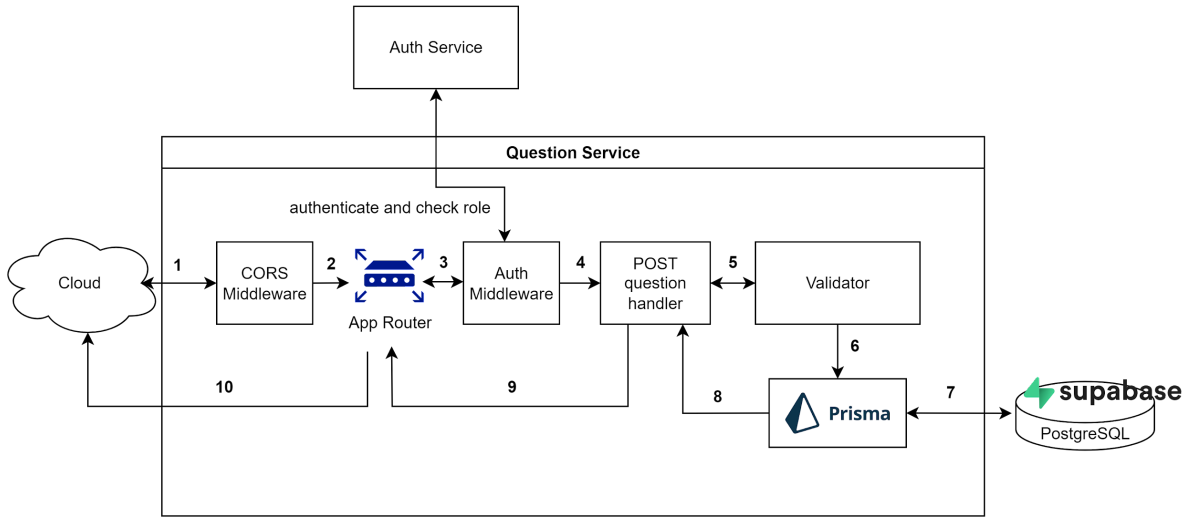


Figure 61. Flow Diagram of createQuestion.

Steps

1. Incoming HTTP requests reach the Question Service API, intercepted by the Cross-Origin Resource Sharing (CORS) middleware.
2. Only forward requests sent from authorized hosts/origins to the app router.
3. The request is intercepted by the Auth middleware to ensure the request is authorized, and the caller has an **ADMIN** role.
4. Forward authenticated requests to the *postQuestion* handler.
5. Parse the request to check for request body validity.
6. Interact with the Prisma ORM tool to access the models and send the database query to store the new question.
7. Handle data query with external database.
8. Send the query result to the handler.
9. Construct the HTTP response according to the query result.
10. Send the response back to the caller.

Error Handling

The error-handling logic within the Question Service closely mirrors that of the User Service, with the notable distinction of an advanced access control mechanism integrated into the Auth middleware. This measure is implemented to prevent unauthorized access to the API.

In case the CORS check fails, the request is immediately blocked and will not reach the app router. For auth check failure, the HTTP response will have a status code of 401 (Unauthorized) or 403 (Forbidden), and be sent back to the app router. For validator check failure, the HTTP response will have a status code of 400 (Bad Request), and be sent back to the app router.

History Service

As Peerprep allows users to attempt different questions by matching with a partner and collaborating, when the session terminates, the attempt or code submission should be stored. This is what the History Service provides. It enables persistent storage of past attempted questions, with the corresponding code submissions.

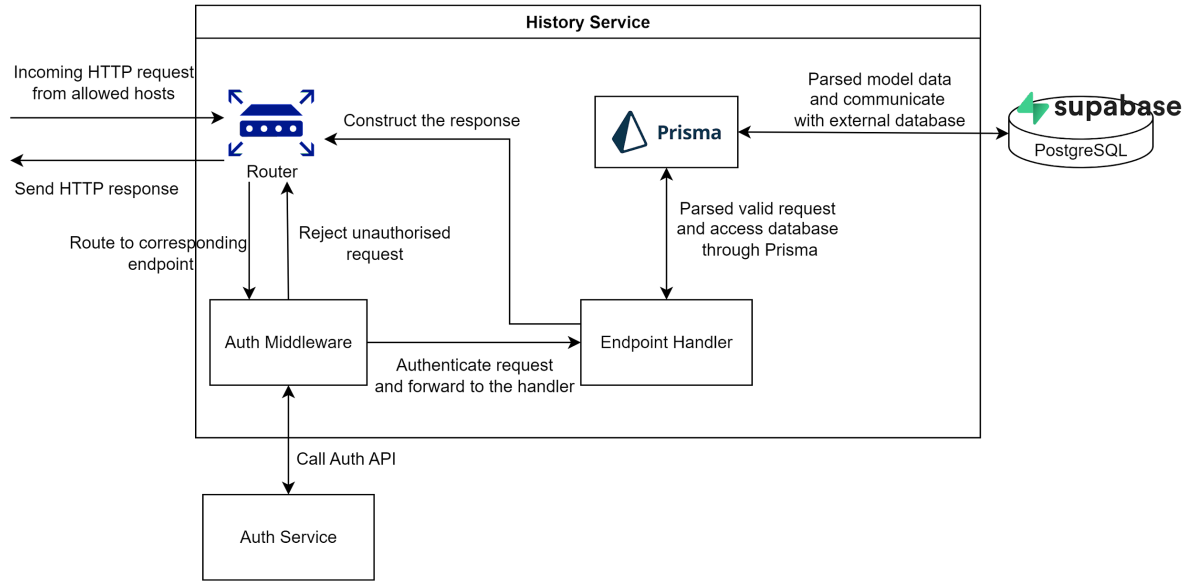


Figure 62. Architectural Diagram of History Service.

The architecture of the History Service is almost the same as the User Service, which consists of a RESTful web API that provides operations such as retrieving past attempted questions list, code submissions, as well as creating new history, updating code submissions, and deleting history. Take note that there is no access control for calling the APIs, and it only requires the request to be authenticated.

Endpoints

The History API provides 6 endpoints for history management. Below is a list of all the endpoints available:

Method	Endpoint	Description
GET	/health	Check if the API is in good condition and is connected to the database.
	/history	Retrieve a list of history available specific to a user, a question, or both. Query parameters for *userId* and/or *questionId* are required.
	/history/user/:userId/question/:questionId/code	Retrieve the code submission for the specific question attempted by the user, together with the programming language used. Query parameters for languages are supported.
POST	/history	Create new history for the attempted question,

		as well as the related code submission for a user.
PUT	/history/user/:userId/question/:questionId/code	Update the existing code submission for question with *questionId* made by the user with *userId*.
DELETE	/history/user/:userId/question/:questionId	Delete the existing history for the specific question made by the user.

Models

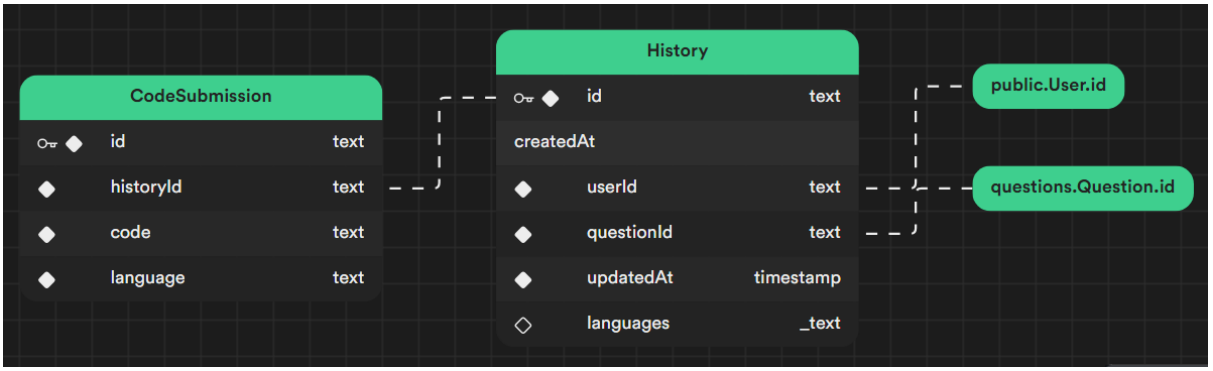


Figure 63. History Service Schemas Visualization.

The schemas for History Service are slightly more complicated. The History schema has 2 foreign key constraints to the User schema and Question schema. If the user or question is deleted, the deletion will cascade to the history storage as well. Besides, there is a strict unique constraint for any pair of user id and question id. This means that there will be no 2 history records that have the same user id and question id.

For CodeSubmission schema, the *historyId* is referencing the History schema (foreign key constraint). Any pair of *historyId* and *language* in the CodeSubmission schema must be unique. This means that the history database will not store multiple code submissions for the same question that uses the same language. Furthermore, there is a NOT NULL constraint set for the *code* and *language* to ensure data integrity.

The design choice of enforcing uniqueness for (*historyId*, *language*) provides several benefits:

1. It prevents spam submissions and potential database strain.
2. It simplifies query logic for computing history statistics.
3. It prevents storage of redundant data.

createHistory Flow Illustration

We will demonstrate an example of the *createHistory* API execution flow to better describe the backend structure and logic flow.

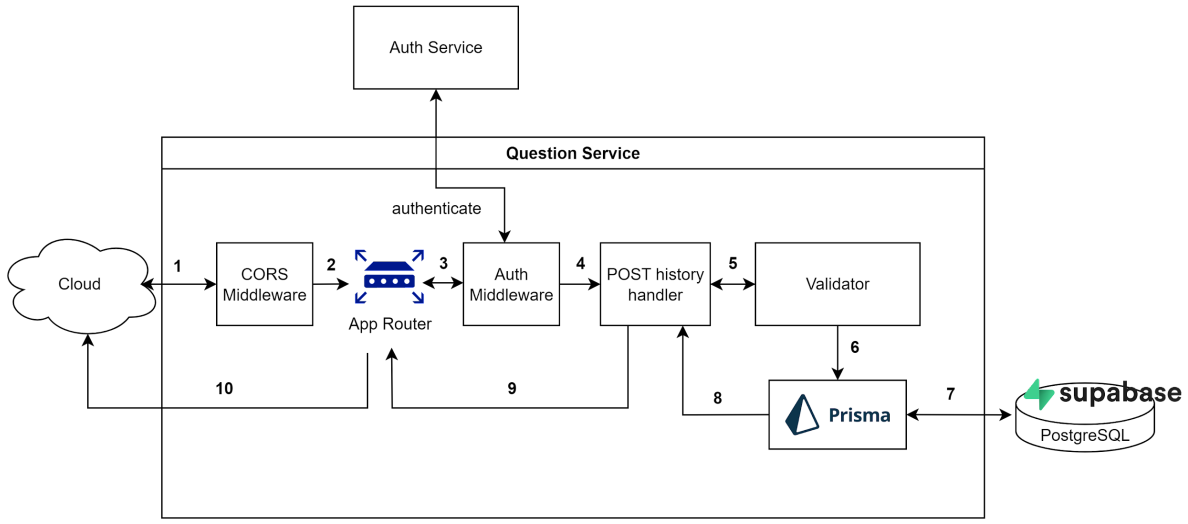


Figure 64. Flow Diagram of createHistory.

Steps

1. Incoming HTTP requests reach the History Service API, intercepted by the Cross-Origin Resource Sharing (CORS) middleware.
2. Only forward requests sent from authorized hosts/origins to the app router.
3. The request is intercepted by the Auth middleware for authentication.
4. Forward authenticated requests to the *postHistory* handler.
5. Parse the request to check for request body validity.
6. Interact with the Prisma ORM tool to access the history and code submission models and send the database query to store the new history record.
7. Handle data query with external database.
8. Send the query result to the handler.
9. Construct the HTTP response according to the query result.
10. Send the response back to the caller.

Error handling

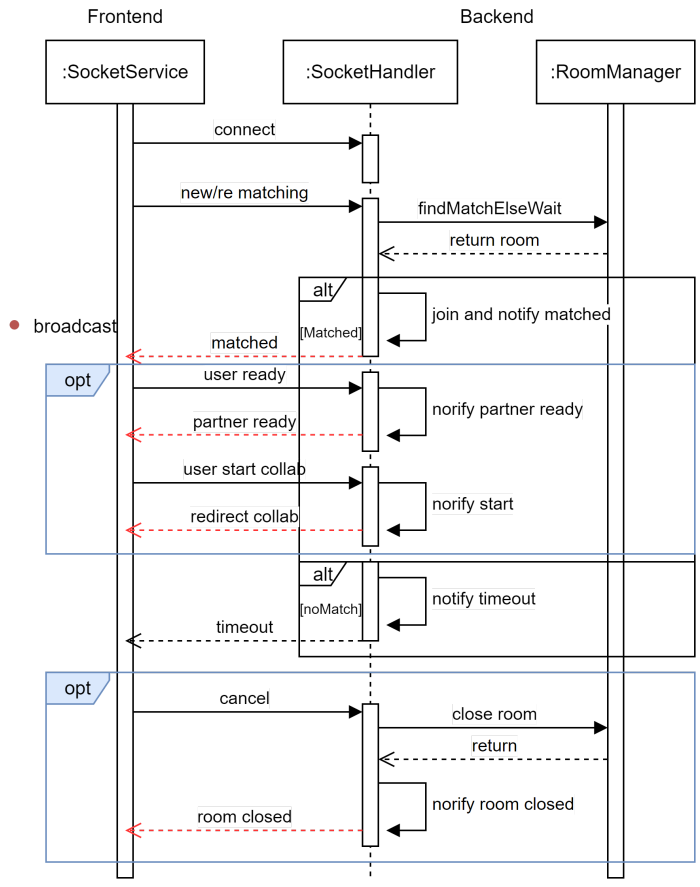
In case the CORS check fails, the request is immediately blocked and will not reach the app router. For auth check failure, the HTTP response will have a status code of 401 (Unauthorized), and be sent back to the app router. For validator check failure, the HTTP response will have a status code of 400, and be sent back to the app router.

Matching Service

The Matching Service is an independent microservice designed to support the matching feature. At its core, the microservice comprises three distinct components to facilitate the matching mechanism, namely a SocketHandler, RoomManager, and RabbitMQManager.

The SocketHandler is designed as a client-facing layer focused on handling socket events from clients. The RoomManager is a middle layer that helps preprocess both incoming matching requests from SocketHandler and replies from RabbitMQManager into a proper Room Entity to retain the matching context throughout the exchange. RabbitMQManager is an adapter for interactions with the RabbitMQ system, which is used as the message broker for matching requests. Additionally, the microservice communicates with the Collaboration Service via an event bus to enable the establishment of a collaboration session for pairs.

SocketHandler with RoomManager



When the backend's Socket Handler receives a "REQUEST_MATCH" message, it triggers the "findMatchElseWait" function in the Room Manager. This function is responsible for interacting with the message broker to find a room for the matching request.

If no match is found within a specific time limit, the matching request is closed, and a "NO_MATCH" event is sent to the client. This would prompt the lobby to switch to the no-match view. On the other hand, if a match is found, a "MATCHED" event is emitted to the client.

As described in the frontend, upon success, the client is presented with two options: updating their user's ready state or initiating the collaboration session. Choosing either option triggers a "notify" subroutine, which broadcasts the change to all members in the same room.

Throughout the matching process, the Socket Handler also handles cancellation requests. In the event of a cancellation request, it instructs the Room Manager to close the room and broadcasts this change to the partner. This ensures that all relevant parties are informed and the room is properly closed.

Socket Events

The following is a state diagram of the Matching Service for the backend.

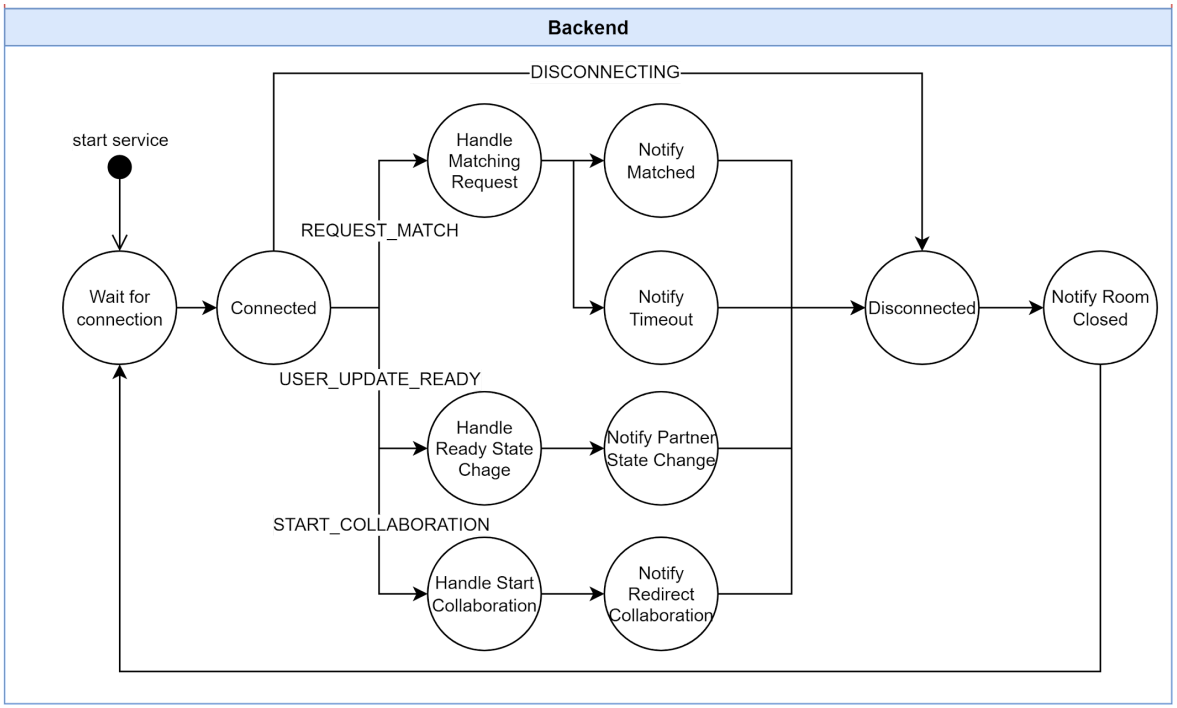


Figure 65. State Diagram for Matching Service Backend.

Receiving Event	Description
CONNECTED	On socket connected with client.
CONNECTION_ERROR	On error during socket communication with client.
REQUEST_MATCH	Client request for a match. Server will try to find a room based on the preference in the payload, or create one for waiting.
USER_UPDATE_READY	Client indicate readiness to start the collaboration session. Server will notify partner about the ready state change through MATCHING_PARTNER_READY_CHANGE event.
START_COLLABORATION	Client(owner) initiate the collaboration session. A room id will be generated and broadcasted to Collaboration Service to create the session for collaboration.
DISCONNECTING	On client disconnecting. Server will close and notify other user in the room about the status.
DISCONNECT	On client disconnected. End of matching process.

EmitingEvent	Description
MATCHED	Server notifies client about of a successful match, partner detail will be attached.
NO_MATCH	Server notifies client about of a unsuccessful match due to timeout.
PARTNER_READY_CHANGE	Server notifies client about change in partner ready state.
REDIRECT_COLLABORATION	Server notifies client to redirect to navigate to collaboration page of the collaboration session.
ROOM_CLOSED	Server notifies client(any) about the closure fo matching room due to complications (cancellation, connection dropped, network issue)

RoomManager with RabbitMQManager

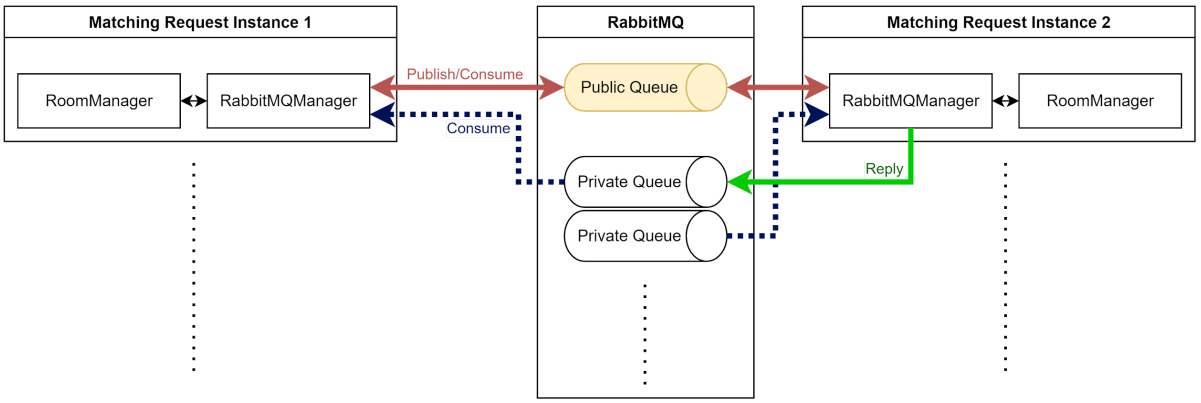


Figure 66. RabbitMQ interaction with Matching Service.

For each matching request, the RoomManager either reuses an existing RabbitMQManager or creates a new one to interact with RabbitMQ. The RoomManager then initiates the "publishPublicAndConsumeMatched" function in the RabbitMQManager to commence the matching process. This process relies on a permanent public queue for matching requests and variable private queues for direct exchanges between instances.

Upon the arrival of a new matching request in the RabbitMQManager, the manager subscribes the instance to the public queue, creates a unique private queue for the instance, and subscribes the instance to it. Subsequently, the matching request, containing preferences along with an additional reply ID, is published to the public queue. The instance enters a waiting state, monitoring both the public and its private queues for incoming messages.

Whenever a new message is published to the public queue, all subscribed instances process the message to assess the suitability of a match using a matching predicate, comparing the incoming preference with their own. If a match is found suitable, each instance attempts to contact the owner of the matching request via the private queue using the ID attached in the incoming message. Those unsatisfied simply disregard the message.

Upon receiving a message from its private queue, indicating a match, the owner instance selects a partner purely on a first-come, first-served basis since all replies are of valid match. Subsequently, the instance relinquishes control back to the RoomManager for further interactions with the selected partner.

Matching Criteria

The core matching logic is defined within the "ifPreferenceOverlapped" function. This function takes two preference objects as input to determine their suitability for a match, yielding a boolean value. These preference objects contain selected programming languages, problem complexities, and topics. To streamline the comparison logic with other preferences, the attributes within the preference object are encoded into binary values..

The encoder logic is as followed:

- Consider the option space, create a binary string of 0s with length of the option space. E.g. 4 language options is translated to “0000”.
- For every chosen option, set the corresponding position of the binary string to “1”. E.g. Python and JavaScript selected is translated to “1001”.

The comparison of binary strings is performed using an AND logic operator to determine the compatibility of matching requests. In the context of collaborative sessions, the programming language holds primary importance, followed by complexity and topics.

	language	**complexity**	**topics**
Preference 1	1001	100	10001000 …
Preference 2	1000	101	10000000 …
Final	1000	100	10000000 …

The resulting matched preference is decoded using the final preference binary string from the AND logic. The decoder operation reverses the process established in the encoder function.

Collaboration Service

Session Initiation

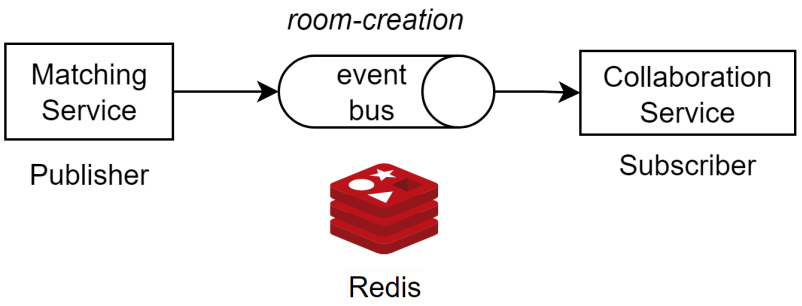
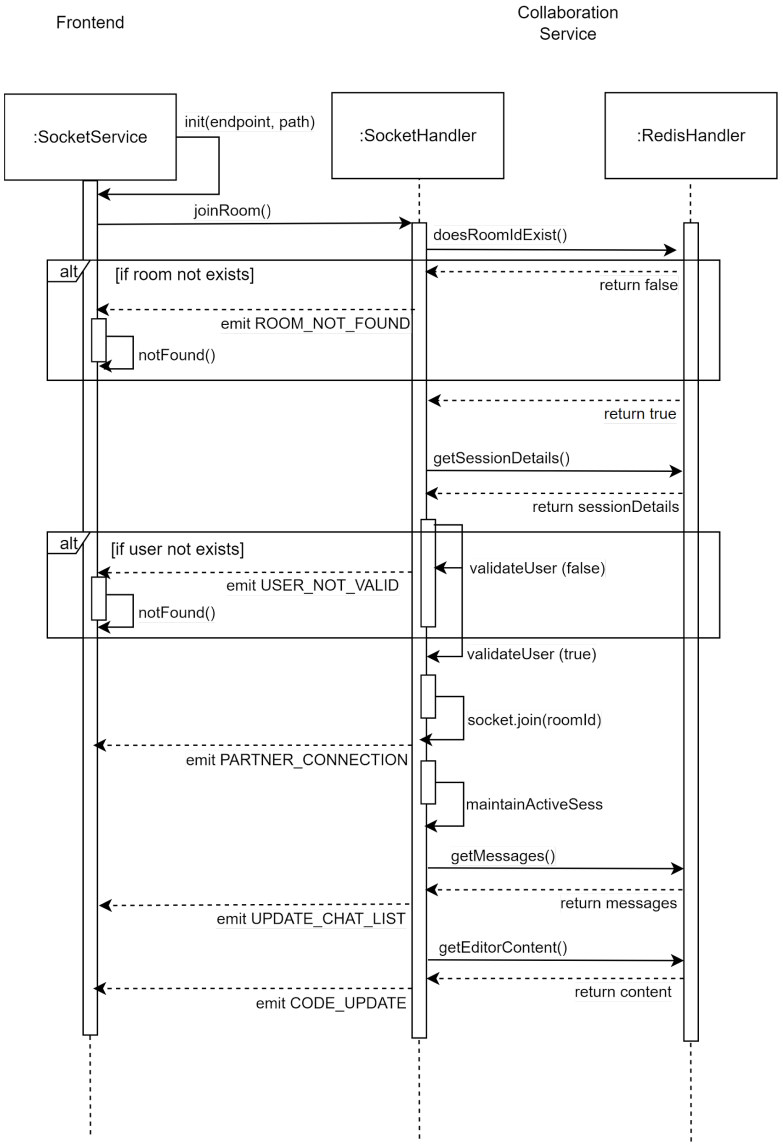


Figure 67. PubSub relationship between Matching and Collaboration.

Upon successful match in Matching Service, Matching Service produces a room creation message for Collaboration Service to consume. The Collaboration Service then adds these details to cache, indicating a session is live. The frontend will push both users to the new collaboration room and allow both users to subscribe their sockets to the backend.



This diagram describes the joining of a collaboration session. It checks the validity of a roomId in the event where a user tries to join a room that has already closed or does not exist where failing this prompts the frontend to return a notFound() error and denies access to the collaboration view. The backend then retrieves the session detail from the cache and attempts to validate the user, failing which also denies access to the collaborative view (another user tries to access the room).

If all is successful, it will allow the socket to join the roomId and notify the partner that the other user has connected, and emit chat list and code content if there are any that exist within the session cache.

Live Sessions

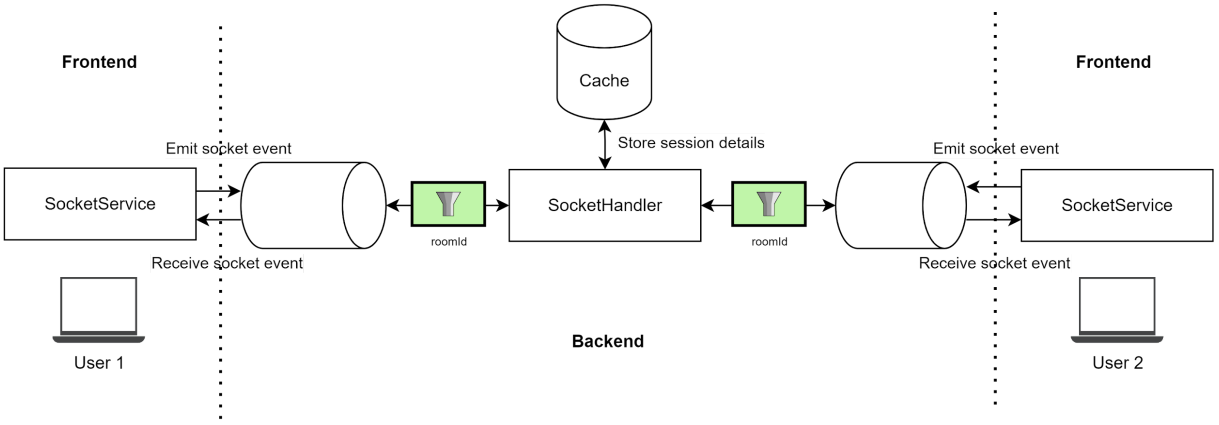


Figure 68. Overview of live sessions between two clients.

The above shows the overall diagram of how the frontend interacts with the backend in an active collaboration session. The socket events are only fired through a filter with their roomIds so other users will only be subscribed to their room's events. Redis is used as an on-session cache due to its quick, in-memory solution and is used to store the session end time, the users validity, the roomId and the code of the active sessions.

Socket Events

Receiving Events	Description
CONNECTION	On client connection
DISCONNECT	On client disconnection
JOIN_ROOM	Join the client's socket to the room with the given roomId
CODE_CHANGE	Receives the entire code editor content from client
CODE_EVENT	Client emits changes on editor with a specific position on the editor Eg. Event(char c: 'c', line: '12', column: '3') indicates a character of 'c' has been inserted in line 12 column 3 to the other user
SEND_CHAT_MESSAGE	Client emits a chat message to server
END_SESSION	Client indicates he is ending session
CONFIRM_END_SESSION	Client confirming he is leaving the session
GET_SESSION_TIMER	Client obtains session timer from server's session cache

PARTNER_CONNECTION	Client obtains partner's connection
SEND_CURSOR_CHANGE	Client sends cursor location
SEND_HIGHLIGHT_CHANGE	Client sends selected text

Emitting Events	**Description**
SEND_CODE_EVENT	Server emits the code change event to the frontend of the other user
UPDATE_CHAT_MESSAGE	Emits chat message to the other client
UPDATE_CHAT_LIST	Emits the room's previous chat messages when a client rejoins the same roomId
CODE_EVENT	Emit code change event to the other client
HIGHLIGHT_CHANGE	Emit the highlighted area of the other client
CURSOR_CHANGE	Emit the cursor change of the other client
END_SESSION	Emits the code content that was stored to the user terminating the session
SESSION_TIMER	Emits the session's terminating time to the user
PARTNER_LEFT	Emit and inform the other client his partner terminated the session
ROOM_NOT_FOUND	Emits when room sent by user is not found within the cache
USER_NOT_VALID	Emits when the user is not a valid user within the cache

Error Handling

To maintain strict room validity, two primary verifications were established before a user can access a collaboration room.

1. Validate roomId

   Initially, our approach to prevent a user's ability to create rooms was focused on the frontend (performing <peerprep_url>/collaboration/[roomId]/) but pivoted our strategy to relying on Matching Service to create the room. This ensures that rooms are exclusively generated through the Matching Service, guaranteeing creation only when two users are involved. Thus, sessions are only loaded through the cache and

if the roomId does not exist, the server will emit ROOM_NOT_FOUND indicating that there is a 404 error to the frontend of that client.

2. Validate user

   To prevent unauthorized users from joining the same room, a data structure maintains session IDs and permitted users within each session. Therefore, everytime a user joins, their user.id is checked against the memory before emitting an USER_NOT_VALID event to the client if he is not valid. This prompts a notFound() page and displays 404 for the user on the frontend. In the event that a user manually terminates his session, his user.id will be revoked from this data structure and be prevented from re-joining the session.

In the scenario where a user has left the session but has not manually terminated it/session timer ended, he can still rejoin the session and load the code from the cache. This could be due to varying factors, such as browser crashing or losing of their internet connectivity temporarily.

Session Termination

Users can terminate an active session in two ways: either by manually ending it or by the session timing out after the full 60 minutes. In either case, the cache transmits the saved state to the client. The client then posts this state to the History Service for the storage of attempted questions. This ensures that whether a user ends the session manually or if it times out, the progress and attempted questions are securely saved for future reference in the History Service. After the last user has left or the session has timed out, the service will delete the cached details from its data structure and session storage completely.

DevOps

Local Deployment

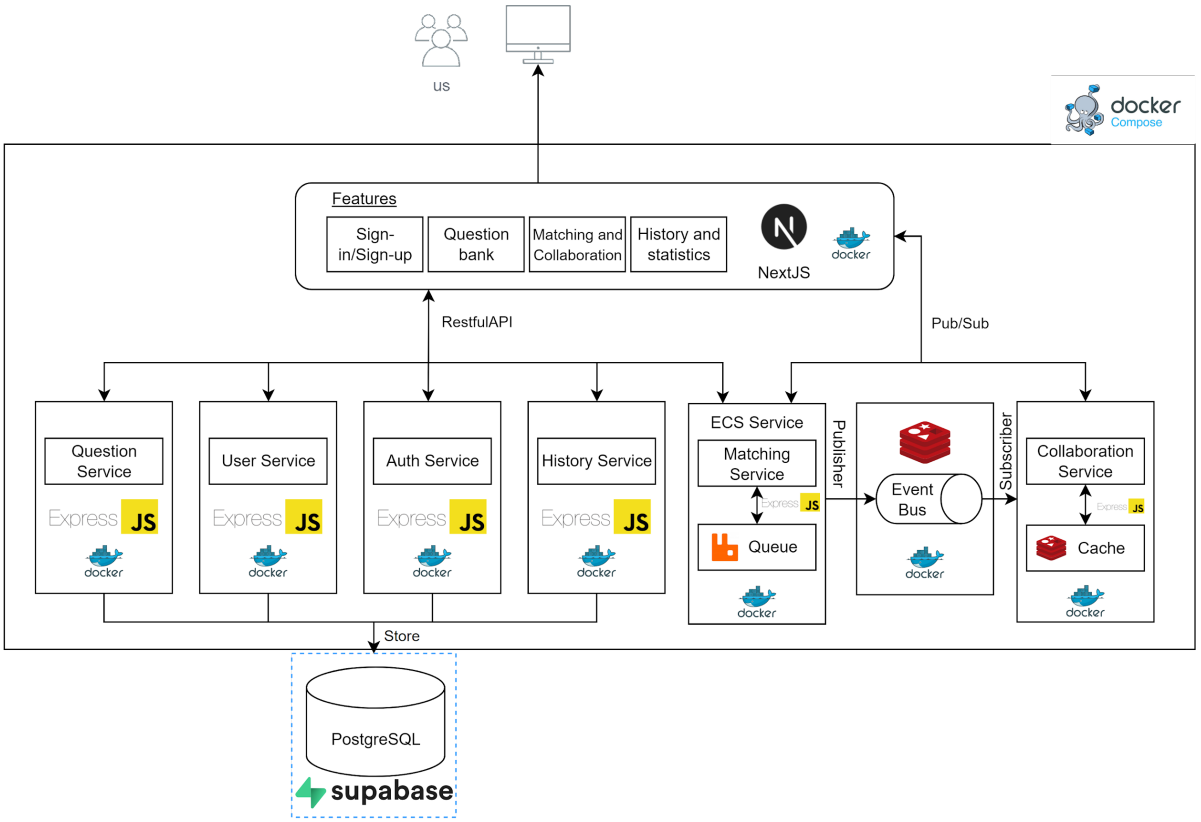


Figure 69. Local Deployment of PeerPrep.

Docker Compose was implemented to streamline the local deployment of both frontend and backend services. Users can initiate all applications simultaneously by executing "docker-compose up" while providing the necessary ".env" file. All containers operate within the same "peerprep" network, allowing the frontend and backend services to communicate with each other via a private DNS name managed by the Docker network. This configuration is specified in the "docker-compose.yaml" file.

Cloud Deployment on AWS

Essential AWS Resources

Terminology	Description
Elastic Container Service	A fully managed container orchestration service in AWS to run, manage, and scale containerized applications using Docker containers.
Elastic Container Registry	A fully managed Docker container registry service in AWS.

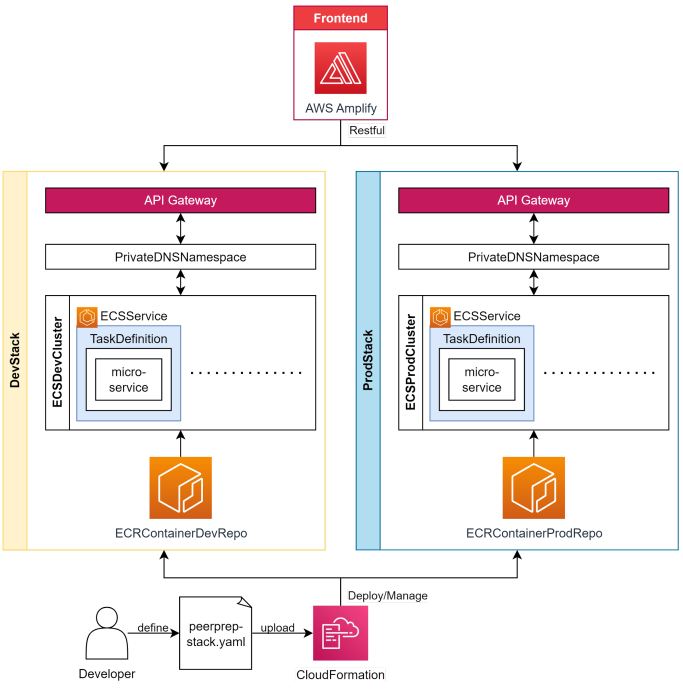
ECS Service	A logical group of tasks defined by a task definition to run in a cluster. It is responsible for ensuring that a specified number of tasks/replica are running and maintained at a desired state. Services can be configured with Service Discovery and Service Connect to allow for integration and communication with external resources.
ECS Task Definition	A blueprint for a containerized application with various parameters for how containers should be launched in an ECS service.
Amplify	A hosting platform in AWS that is focused on deploying full-stack web applications.
API Gateway	A fully managed service that enables user to create and publish RESTful APIs for applications.
CloudFormation	A service that allows admins to define and provision AWS infrastructure and resources in a declarative and automated way.

Deployment Overview

The project consists of a frontend application with six microservice applications for the backend. The frontend is hosted in AWS amplify, while the microservices are deployed in the ECS cluster within their dedicated ECS service. Communication between Amplify frontend and backend services is facilitated through AWS API gateway, which incorporates CouldMap integration with Service Discovery enabled ECS services. Additionally, ECS Service Connect was configured for ECS services to support inter-service communication.

The deployment process to AWS is fully automated, as described in the CI/CD pipeline.

Deployment Strategy for AWS



AWS CloudFormation provides a straightforward way to manage our AWS infrastructure. Allowing us to employ a two-stack approach to deploy backend services for development and production[3] environments.

The definitions for resources for the AWS infrastructure are outlined in the "peerprep-stack.yaml" file. This approach allows our team to effortlessly replicate and deploy an entire stack through CloudFormation, simplifying the management and provisioning of resources for different environments.

[3] The production stack will be inactive to minimize costs associated with AWS resources.

CI/CD

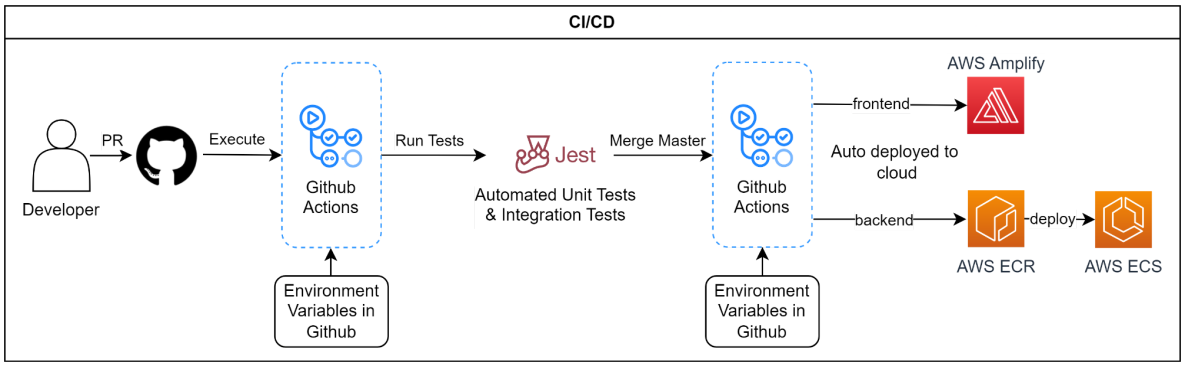


Figure 70. CI/CD Pipeline.

We employ GitHub Actions to support our CI/CD process. The following table below is a breakdown of the declared Actions, their trigger hooks, and a brief description of the tasks and outcomes they entail.

GitHub Action	Trigger	Task
Backend Services Tests	Pull request on master.	Perform unit testing on all 6 backend microservices.
Run Question Service Integration Tests	Pull request on master.	Perform integration testing on question service.
Run User Service Integration Tests	Pull request on master.	Perform integration testing on user service.
Run Auth Service Integration Tests	Pull request on master.	Perform integration testing on auth service.
Run History Service Integration Tests	Pull request on master.	Perform integration testing on history service.
Build and deploy image to AWS ECS (Dev)	Push on master and development branches.	Build and push docker images of backend microservice to the development repo in AWS ECR. Trigger update to ECS Services in development ECS cluster to deploy the latest images.
Build and deploy image to AWS ECS (Prod)	Push on production branch.	Build and push docker images of backend microservice to the production repo in AWS ECR. Trigger update to ECS Services in production ECS cluster to deploy the latest images.
Frontend Build Check	Pull request on master, push on	Perform production build on frontend application to detect blocking errors before deployment to public hosting platform.

	production branch.	

The following table highlights third party services that support our CI/CD workflow.

Integration	Trigger	Task
Development	Commit on development branch.	Amplify deploys the latest frontend in development branch to “development” hosting automatically.
Amplify	Commit on amplify branch.	Amplify deploys the latest frontend in amplify branch to “amplify” hosting automatically.
Master	Commit on master branch.	Amplify deploys the latest frontend in master branch to “master” hosting automatically.
Production	Commit on production branch.	Amplify deploys the latest frontend in production branch to “production” hosting automatically.

Continuous Integration

Continuous Integration (CI) is a foundational DevOps practice that establishes an automated pipeline for seamlessly integrating new code changes with existing source code. This process is facilitated through the adoption of various automation tools. CI plays a crucial role in fostering team collaboration by continuously checking the correctness of new code, identifying integration issues, and maintaining a consistently working codebase.

At Peerprep, we leverage GitHub as our version control system, utilizing a CI pipeline that conducts three essential checks—unit tests, integration tests, and frontend build checks. These checks are triggered with every pull request (PR) aimed at merging into the master branch. This proactive approach ensures the correctness of new changes and upholds a high standard of code quality within our project.

Unit Testing

Unit testing is a fundamental methodology in software testing where individual components or functions are tested independently to ensure their behavior aligns with expectations. In our backend APIs and services, we have implemented numerous unit tests for various handlers. We used the Jest library to craft test cases. To isolate API handlers from their dependencies, we employ mocking for operations such as database interactions and external API calls.

Adhering to the 3A framework for each test, we follow a structured workflow. First, we initialize the necessary data, including mocking for dependent operations (Arrange). Next, we invoke the target function or component to be tested (Act). Finally, we scrutinize the outcome and code behavior to ensure they align with the expected results (Assert).

To elucidate this process, consider our unit testing workflow for the *sendPasswordResetEmail* handler in the Auth Service API as an example.

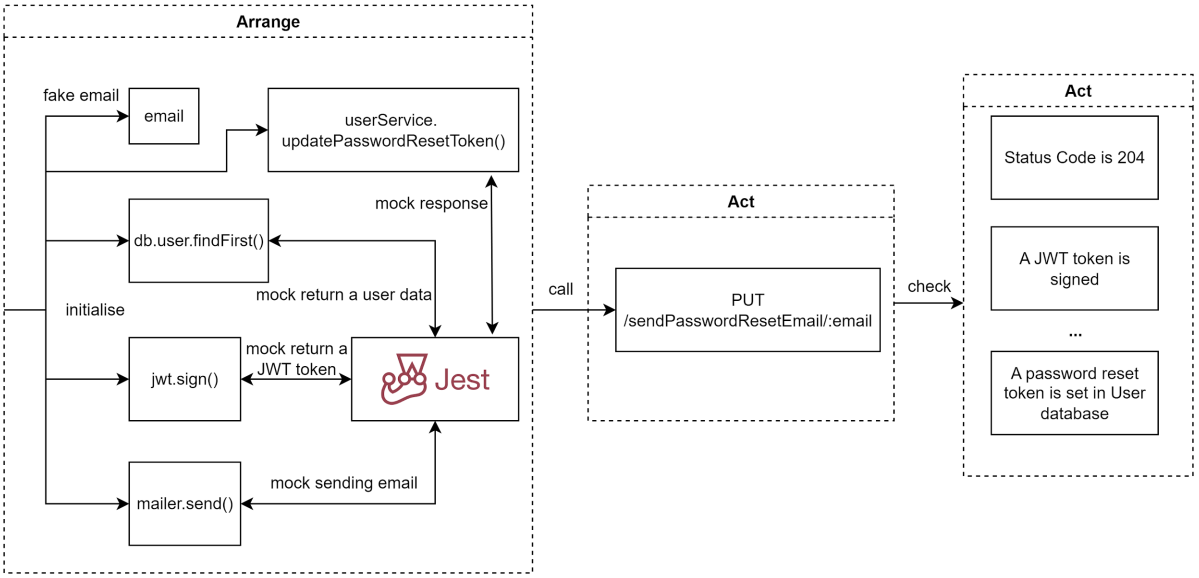


Figure 71. Unit Test Flow Diagram for sendPasswordResetEmail handler.

In the outlined process, we employ mocking for four dependent functions:

1. *db.user.findFirst()*. This function, sourced from the Prisma API, executes a query to the database. We mock its returned value with self-created user information, which may not exist in the actual database. This ensures a valid user context for testing.
2. *jwt.sign()*. Originating from the *jsonwebtoken* library, this function signs a JWT token using the user's email and an app secret. To simplify testing, we mock the return value to be a deterministic token, providing predictability in our test scenarios.
3. *mailer.send()*. Derived from the *nodemailer* library, this function handles the sending of emails. Given that the email used in testing is fake, there's no need to send an actual email. Therefore, we mock the implementation of this function to perform no actions, preventing unintended email transmissions during testing.
4. *userService.updatePasswordResetToken()*. This function, a helper method internally calling an API from the User Service, is isolated for testing by mocking its behavior and response. This ensures that the test focuses on the specific functionality of the target function without involving the actual User Service API.

By strategically employing mocking for these functions, we create a controlled testing environment that allows us to assess the isolated behavior of the *sendPasswordResetEmail* handler in the Auth Service API.

Subsequently, we proceed by calling the target test function, specifically the *sendPasswordResetEmail* API handler. Following this execution, we perform multiple assertions on both the response generated by this API call and its associated behavior.

It's essential to highlight that all unit tests are conducted using a white-box testing approach. This implies that we possess a comprehensive understanding of the expected outcomes of the test function. This knowledge allows us to assert not only the results but also the internal behavior, ensuring a thorough assessment of the test cases.

In total, we have meticulously crafted 157 unit tests across our services. We have orchestrated a streamlined process by preparing a script that executes each unit test. This

script is integrated into GitHub Actions, enabling us to run the tests concurrently on two distinct Node versions—Node v18 and Node v20. Furthermore, the test environment is orchestrated on a Linux operating system, ensuring our code's robustness across diverse OS platforms. Below encapsulates a summary of the tests distributed across each service:

Backend Service	Number of Unit Tests	Code Coverage (%)		
		Lines Coverage	Branches Coverage	Functions Coverage
Auth	45	83.58	69.14	64.91
User	23	79.39	55.23	79.54
Question	24	84.55	62	86
History	54	87.69	80.15	84.21
Matching	9	62.82	51.16	47.61
Collab	5	53.81	22	34.61

The test coverage for the Matching and Collaboration Services is notably limited. This is attributed to the absence of APIs within these two services. Currently, our testing efforts are focused on specific utility functions and validating the functionality of communication logic with the queue and cache systems.

Integration Testing

The unit tests focus on individual components, ensuring their functionality in isolation. However, to validate the seamless interaction of multiple components, we conduct integration tests. These tests involve the actual execution flow of API handlers without any mocking, mimicking real API calls, database operations, and JWT token processing. We use a separate test database to avoid impacting the production database.

Following the 3A framework, we primarily perform integration tests on backend APIs due to time constraints and their susceptibility to logic flaws. We will continue using the *sendPasswordResetEmail* handler in the Auth Service API integration test as an example, encompassing the arrange, act, and assert phases.

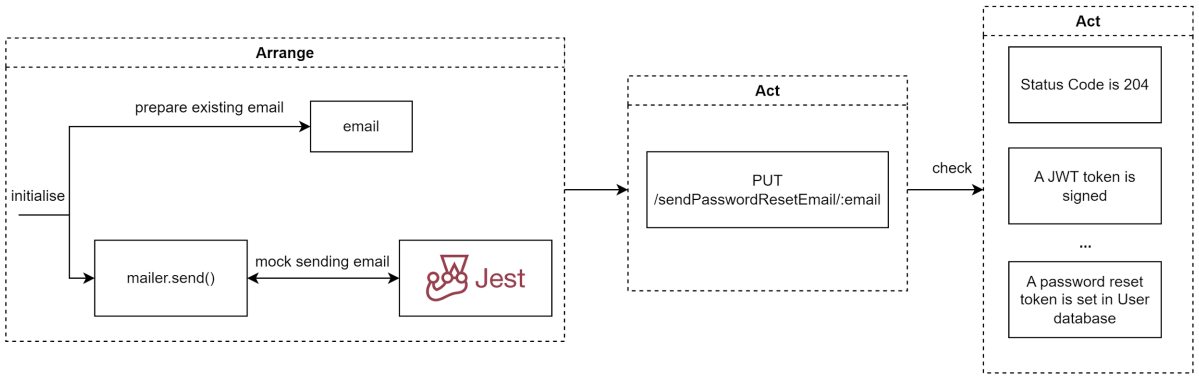


Figure 72. Integration Test Flow Diagram for sendPasswordResetEmail handler.

Compared to unit tests, there are notable distinctions in our integration tests:

1. We utilize actual emails from the test database instead of generating fake ones.
2. We refrain from mocking functions or API calls for most dependencies, including the User Service, database queries, and JWT token signing. However, we continue to mock the email sending function to avoid unwanted email transmissions during testing.

Following the 3A framework, we initiate data preparation, invoke the test function (e.g., *sendPasswordResetEmail* API handler in Auth Service), and subsequently make assertions. These integration tests maintain a white-box testing approach. Notably, we've designed the data to ensure independence between tests—modifications in one test function do not impact results in others. This practice emphasizes a separation of concerns, even within the testing phase.

In total, we've developed 93 integration tests across backend APIs, encompassing Auth, User, Question, and History Services. Jest continues to be our testing facilitator. To uphold security, we safeguard sensitive information by utilizing multiple environmental variables (e.g., DATABASE_URL, TEST_USER_ID) stored as repository secrets in our shared GitHub repository. Following is a summary of the code coverage achieved in our integration tests.:

Backend Service	Number of Tests	Code Coverage (%)		
		Lines Coverage	Branches Coverage	Functions Coverage
Auth	20	80.79	59.37	87.32
User	22	72.13	48.64	70.83
Question	25	83.53	71.96	80.35
History	26	77.5	46.09	72.34

It's evident that the code coverage for integration tests is typically lower than that of unit tests. This is an anticipated outcome since in integration tests, we prioritize scenarios that may alter the responses of dependent functions rather than extensively testing invalid inputs in request query parameters and bodies. One example is modifying database data to observe corresponding changes in query responses.

Frontend Build Checks

The frontend build check involves executing the production build command for the frontend application. This process plays a key role in pinpointing issues and errors within the code that are visible during development. It serves as a final checkpoint to ensure there are no critical errors that could hinder the smooth deployment of the frontend to AWS Amplify.

Branch	Use case	Description
Development	Frontend and backend	This branch is hooked to both frontend and backend deployment actions and integrations. It is useful for development and testing of frontend, backend and CI/CD related changes.
Amplify	Frontend only	This branch is hooked to frontend deployment actions and integrations. It is useful for development and testing of frontend related changes.
Master	Frontend and backend	This is a controlled branch that serves as an staging environment for testing of the latest frontend and backend applications in the master branch.
Production	Frontend and backend	This is a controlled branch that is used mainly for deployment of frontend and backend applications to the production environment.

Improvements and Enhancements

While our Minimum Viable Product (MVP) currently meets the essential requirements of the application, there exists significant potential for further expansion. We have identified various avenues for improvement across usability, scalability, performance, availability, and security in order to elevate the overall functionality and user experience of Peerprep. These enhancements aim to propel the application to greater heights, ensuring its continued growth and success.

Usability

Improvements	Description
Automated test case execution for all supported languages	Currently, we only support automated test case execution for Python and Javascript, which might be disadvantageous for users who are used to coding in the other languages (Java, Python).
Coding within different time frames	In some companies, we are placed into a pair coding assessment where we take turns to code on a timed basis. We could possibly incorporate this such that one user is read only while the other user is able to type for a fixed amount of time and reverse the role every 2 minutes.
Video/voice communication	Due to the nature of coding in pairs, we could incorporate video or voice communication over WebRTC to provide a more intuitive communication experience for the users.
Add friends feature	Introducing an "Add Friends" feature is a potential enhancement that would empower users to expand their network connections. This feature enables users to establish connections with each other, fostering a sense of community and facilitating future collaborations without relying on random matching.
Forum and discussions	Along with the “Add Friends”, introducing a forum and discussion section will further create a community space for discussions of coding questions and interview tips.
Helper whiteboard and drawing tools for brainstorming and idea illustration	To enhance collaboration experience, one possible way is to implement a feature that enables participants in the same workspace to create and share diagrams in real-time. This functionality can enrich discussions by providing a visual aid for brainstorming and idea illustration, promoting a more engaging and interactive collaboration experience.
Support OAuth login	To enhance user convenience and eliminate the need for users to remember specific credentials, consider implementing OAuth Login functionality. By integrating with third-party services like "Google" and "GitHub," users can log in

	seamlessly using their existing accounts, improving the overall user experience.

Scalability & Performance

Improvements	Description
Migrate Database to AWS	The database used currently is a hosted service on a third party platform. We are lacking the ability to upscale the data traffic if the number of users in Peerprep increased. It is best to migrate over to AWS cloud database solutions like RDS to provide better flexibility on controlling the database instances, as well as mitigating potential networking and latency issues.
Caching for backend APIs	The APIs in Peerprep backend are lacking the caching ability. This will increase the traffic sending to the database as well as slowing down the backend servers. We can mitigate this by utilizing a cache system like Redis for each of the API. This will have significant performance improvement.

Security & Availability

Improvements	Description
Implement Rate Limit for backend APIs	In order to promote fair API usage and safeguard our endpoints from potential Denial of Service (DoS) attacks, we are contemplating the implementation of an endpoint-specific rate limit. This measure aims to prevent any single user from consuming excessive resources through the APIs, fostering a fair and secure environment for all users.
Region selections based on the user's location	Currently, all our resources are provisioned under the region of ap-southeast-1 (Singapore). As our user base grows in the future, we must be able to provide different locations with good latency for better availability and performance across the different regions.
Improve CI/CD pipeline with Staging environment	Having a Staging environment before deploying the changes to production will greatly reduce the down time of the application. We can also perform security tests such as the penetration tests in the STG environment before deployment.

Reflections and Learning Points

The past few weeks working on Peerprep have been a challenging but fruitful journey. As a team, we managed to bring Peerprep from an ideation to fruition through countless hours of hard work, collaboration and applying what we have learnt in lectures. Here are some learning points we want to elaborate on:

1. Choice of technology stack

   Our group views that selecting the appropriate technology stack is crucial in our project's development. We delved into evaluating various technologies, considering factors like compatibility, scalability and ease of integration. Our eventual choices were guided by our quality attributes that we have selected and resulted in a smoother development of the project.

   One thing we focused a lot on was to invest in time to research on structuring our frontend/backend, discussing which tools to use (libraries vs custom implementations), dependencies that we could lean on and focusing on building a foundation of our project. By week 4, we were quite certain of the utilities to use and had a good framework in mind before we embarked on developing the project. This procedure saved us countless hours from re-doing components/services that may depend on these APIs.

   For our project, our group faced some difficulties when using Next.js 13. As the framework is relatively new, there are limited resources available online and debugging can be challenging. Despite that, our group found effective workarounds to overcome such challenges. Overall, we recognise that the strategic choice of technology is paramount, and equally important is the ability to adeptly address and overcome challenges faced.

2. Implementation of microservice architecture

   As everyone was new to the microservices architecture, trying to initialize the project and splitting the features was confusing for us all. We initially assumed we should develop a monolithic code base before we could break it into smaller services, but in the end we managed to kick start the base application with a rough framework that uses the microservice architecture. After developing the user and question service, the concept became much clearer and we managed to grasp deeper understanding in microservices. Reflecting on the development process, we are now able to clearly identify the benefits of the microservice architecture Since the services are independent, different people were able to work on multiple services concurrently, which really helped to improve the overall team productivity. Such an architecture also made a lot more sense once we moved to cloud deployment, where we saw that each service could be scaled differently by demand.

3. Project management

   As all of us were inexperienced in web development, a huge concern was trying to project our workload based on a given ticket. Though tasks looked deceptively straightforward to implement, we always tend to uncover pervasive bugs within the implementations, leading to setbacks on our sprints as we have other subjects to focus on. However, all turned out well and we managed to put forth the product.

Lastly, our team would like to express our utmost gratitude to the teaching team and our project mentor, Mr Gaurav Gandhi, for all the hard work put into making this module an enjoyable one. Their guidance, support, and valuable feedback have been instrumental in our growth throughout this project, and we appreciate the commitment they've shown to our learning and development throughout this course.

References

11 most in-demand programming languages in 2023. Berkeley Boot Camps. (2023, January 5). https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

Atlassian. (n.d.). *What is continuous integration*. https://www.atlassian.com/continuous-delivery/continuous-integration

MozDevNet. (n.d.). *MVC - MDN Web Docs Glossary: Definitions of Web-related terms: MDN*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Glossary/MVC

What are microservices?. microservices.io. (n.d.). https://microservices.io/

Documentation: Next.js. Docs | Next.js. https://nextjs.org/docs

Jrgarciadev. (n.d.). *Introduction*. NextUI RSS. https://nextui.org/docs/guide/introduction

Installation - tailwind CSS. Installation - Tailwind CSS. (n.d.). https://tailwindcss.com/docs/installation

Docker Overview. Docker Documentation. (2023, September 8). https://docs.docker.com/get-started/overview/

Documentation: Table of contents. RabbitMQ. (n.d.). https://www.rabbitmq.com/documentation.html

Documentation. Redis. (n.d.). https://redis.io/docs/