NUS
National University
of Singapore

CS3219 AY22/23 Semester 1
Team 55: Leet Warriors (PeerPrep)

Name	Matriculation Number	NUS Email
Ezekiel Toh Fun Kai	A0218061N	e0544097@u.nus.edu
Glenn Lim Jun Wei	A0217656X	e0543692@u.nus.edu
Wang Hong Yong	A0217842A	e0543878@u.nus.edu
Zhang Zhikai	A0217567W	e0543603@u.nus.edu

LEET WARRIORS

GitHub Repository: https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g55

Motivation and Inspiration

In recent years, the tech industry has become increasingly saturated. To reflect the increasing pool of applicants and the concomitant increase in competitiveness amongst them, Tech companies have made their interviews exponentially more complex and difficult. This has caused many students to flounder during their technical interviews, as they face difficulty in verbally expressing their thought processes or understanding higher level problems posed to them.

The common approach to interview preparation thus far was to practise questions alone. However, in the face of the increasing difficulty and demands of interview processes, this approach has proved to be archaic and insufficient.

Our team members felt strongly for the need for a platform where students could engage in collaborative learning, and build upon each other's ideas and thoughts. Drawing inspiration from PeerPrep, we have created a novel collaborative learning platform that will revolutionise the way students prepare for interviews - leetWarriors. In addition to the basic features of a collaborative learning platform which allows for asynchronous collaboration online, we have also included communication features such as a comment section under each question post. Our team has also included meaningful data to allow students to track their progress, in order for them to remain motivated in their work.

Project Team

	Technical Contributions	**Non-Technical Contributions**
Ezekiel Toh Fun Kai	• Implemented History Service used by Learning Pathway feature • Implemented JWT token and login for User Service • Contributed to deletion of accounts for User Service • Implemented GitHub Actions CI/CD pipeline for microservices • Setup deployment for node servers to Google Cloud Run	Contributed to final report

	• Setup deployment for NextJS to Vercel • Handled app migration from ReactJS to NextJS • Set up pre-commit hooks for Eslint • Setup facade pattern for frontend • Wrote tests	
Glenn Lim Jun Wei	• Implemented logout for User Service • Implemented Question Service • Implemented Settings Page • Implemented Question List • Implemented Question Page • Implemented Comment/Discussion section • Integrated flow for question and matching services • Contributed to deletion of accounts for User Service • Setup Redis cache for blacklisting of JWT tokens • Setup initial Dashboard Page • Setup Zustand as a global state management tool to manage User state across application • Wrote tests	Contributed to final report
Wang Hong Yong	• Implemented Editor Service • Implemented changing of passwords for User Service • Implemented change password page • Implemented flow for matching, editor and communication service • Implemented sessions page • Contributed to implementation of matching service • Implemented graphs to represent history data on dashboards • Wrote tests	Contributed to final report
Zhang Zhikai	• Set up initial Matching Room Page and improve UI for final application • Set up socket for Matching Service • Set up middleware for services • Setup ESLint and Prettier for entire project • Implemented hashing of password for User Service • Implemented Matching Service • Implemented Communication Service • Implemented Chat Window UI for the frontend • Wrote tests	Contributed to final report

Project Scope & Schedule

Our project aims to tackle the issues faced by students preparing for their technical interview, allowing students to collaborate and solve questions commonly tested in live interviews or technical assessments.

Week	Date	**Discussion**
3	25 Aug	- Set up team repo - Set up project notes and board - Discuss plan for upcoming weeks - Discuss MVP implementation

4	1 Sep	- Setting up User Service
		  - Set up backend for sign up
		  - Set up backend for login
		  - Set up backend for username check
		  - Set up backend for change password
		- Generate FRs and NFRs
		- Set up Zustand

5	8 Sep	- Matching Service - Migrate to NextJS from ReactJS and deploy - Set up ESLint and Prettier, Vercel pre-commit hook
6	15 Sep	- Matching Service (cont.) - Work on project document
7	29 Sep	- Refactor matching service and integrate with frontend - Set up question service (only api call/primitive UI) - Set up collaboration service (only api call/primitive UI) - Write test for user service
8	6 Oct	- Integrate frontend UI with question and collaboration services respectively - Deploy all microservices
9	13 Oct	- Milestone 2 preparation
10	20 Oct	- Add tests for all services and frontend - Set up history service - Set up communication service - Scrape questions for question service and dump into database - Build view questions on frontend
11	27 Oct	- Improve Frontend UI - Integrate tests into CI/CD pipeline - Improve matching logic
12	3 Nov	- Complete all diagrams and documentation - Improve session page layout and UX - Improve API authentication
13	9 Nov	- Submission

Technical Stack

	Technologies
Frontend	Next.js
Backend	Express.js/Node.js
Database	MongoDB, Redis
Deployment	Vercel/GCP Cloud Run, MongoDB Atlas, Redis Lab
Pub-Sub Messaging	Socket.io
Cloud Providers	GCP
CI/CD	Vercel, GitHub Actions
Project Management Tools	Github Issues, Github Projects, Google Drive

Architecture

LeetWarriors has 3 major components that are fulfilled by an overlap of the microservices we designed – an authentication system, peer code system, and learning pathway system. The primary system that drives the app is the peer code system, which will receive high volumes of requests and requires low latencies to ensure that the process is smooth.

As such, we need our core services to be scalable to meet the high volumes of requests while having sufficient capacity to respond to requests quickly.

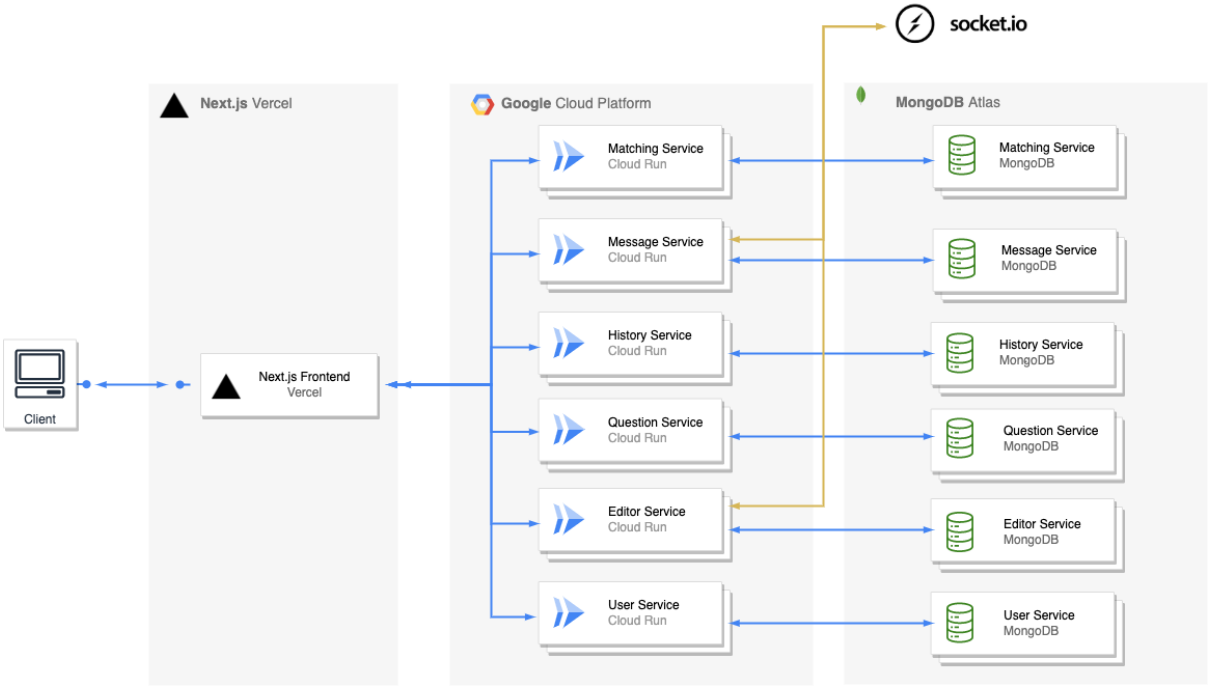


Fig: App Architecture Diagram

Design Decisions

We decided to design our system using microservices architecture for its scalability, reliability and performance. In order to meet these requirements, we decided to host our microservices on Google Cloud Run, which meets each of these requirements, while being affordable and easy to implement.

Each of our services is hosted on MongoDB for its scalability and ease of implementation for our specific use case. In particular, it is easy to create multiple DBs for each microservices and connect to any of them through the cluster.

Architecture		
	Monolithic	**Microservices**
Development Process	Monolithic allows for a fast and simple development process at the beginning on a single codebase. However, as the codebase increases in size, the nature of the monolithic application being tightly coupled result in more frequent downtime and slower debugging as the tightly coupled code introduces more likelihood of a single point failure and finding problems in a large monolithic codebase is often more complex and makes developers less productive. Simple to deploy but as any new changes would require the entire application to be redeployed, the risk associated with redeployment increases.	Microservices are loosely coupled. Each service can be independently developed and deployed, allowing developers to explore using different technologies best suited for each microservice's needs. Development may be slower at the beginning and can introduce unnecessary complexity to a less complex system. However as more features are developed, the development process becomes relatively smoother as the reduced coupling increases productivity. Deployment is complex as there are many options for deployment. However, redeployment will be less risky and less prone to a single point of failure problem as each microservice is independently deployed.
Scalability	Simple to scale as it only involves scaling an entire application by itself by creating multiple instances of it. However, this is not resource-efficient as there is no control over each component's resource allocation which can greatly differ depending on the purpose. In the case where a single component requires more resources relative to the rest of the system, there is a need to allocate more resources to the entire system instead, which introduces redundancy.	Microservices are very scalable and resource-efficient. As each microservice is deployed and developed independently, developers can monitor the usage of resources by these services and efficiently allocate resources like CPU and storage based on expected demand or monitored demand.

Rationale & Conclusion	Microservices Architecture provides us with a better development experience over time due to the fact that it is loosely coupled. The loosely coupled nature of microservices has reduced the time needed to find and isolate bugs. Also, there is less chance of a single point of failure, as even when a microservice is down, our application is still able to function with the other microservices. We are able to be more productive as each microservice can be developed, updated, deployed, and scaled without affecting the other services. Microservices allow each service to be more adaptable and resource-efficient which is beneficial to us.

Database		
	SQL	**NoSQL**
Scalability	Most SQL databases are vertically scalable. By being vertically scalable, it means having to increase the load on one specific server, incurring a higher cost of increasing components such as RAM, SSD or CPU.	NoSQL databases are usually horizontally scalable. This means being able to add more servers to handle increase in traffic, allowing them to scale better to constantly evolving data.
Storage & Schema	Table-based store. Less suitable for our use case as schemas are more rigid.	Document-based store. More suitable for our use case where our application is growing and requires more flexibility about the way we store our data.
Rationale & Conclusion	MongoDB (NoSQL) provides a more meaningful way for us to store our data as compared to relational databases (SQL), mainly due to our small-scale application which is ever-changing and constantly growing at a fast pace. We require the flexibility that NoSQL can provide, and leveraging the cheaper costs and ease of scalability that NoSQL brings about as our application grows.	

Microservices Deployment		
	Cloud Run	**App Engine**
Load Balancing	Auto-scaling. Container instances are created or destroyed based on resources required.	Auto-scaling. Instances are created as more resources are required.
Socket	Works out-of-the-box	Does not work well with sockets. Requires to set up App Engine Flex with proper configurations for it to work.
Deployment	Builds and runs a container based on Dockerfile provided in source files. Accepts a configuration file to set-up the environment within Cloud Run.	Runs all microservices in a single app. Requests are received by the main service and reverse proxied to the respective microservices. Serving requests have to be set up.
Rationale & Conclusion	Cloud Run gives us flexibility in terms of writing new services to meet our evolving demands. Once we are ready to deploy a new service, we simply need to copy the existing Dockerfile as is, and we are able to deploy the application. While App Engine is almost as good in this aspect (though, setting up the relevant folders to serve requests is troublesome), a **major flaw** is the inability to work with sockets out of the box.	

Frontend Framework		
	NextJS	**Vanilla ReactJS**
Performance	Because of the availability of server-side rendering and static generation of websites, pages are pre-rendered on build, cached and served over a CDN. Thus loading of pages are extremely fast	Performance speed is slow compared to NextJS as it uses client-side rendering. And pages are rendered on demand causing it to be slower than NextJS.
Search Engine Optimization (SEO)	NextJS allows one to build SEO-friendly web applications as statically generated web pages are	Since HTML is rendered on the client-side, the keywords might be hard to find on search engines and possibly

	said to be SEO optimized and results in top searches.	result in poor SEO.
Built-In features	NextJS comes with many built in features and capability. For example in-built routing and enhanced development compilation.	React on the other hand is extensible and explicit installation of modules are needed to customize your application.
Rationale & Conclusion	NextJS offers us various tools which provides ease of setting up and accelerates our development process. Its deployment is also fairly simple with just a few clicks. On top of all that, NextJS also has a better performance, therefore our team has decided to proceed with **NextJS**	

Design Patterns

Facade Pattern

We hide the work and details required for the frontend to make API calls. Each request method retrieves and inserts the authorization token into the header of the request. For URLs which require dynamic slugs, such as `/api/history/completed/:username`, passing in the ‘username’ as one of the options of the function calls is all that is required for the facade methods to parse and replace the `:username` with the specified username in the function arguments.

This keeps the fetching logic abstracted from the main work.

```
/**
 * URL can be in the format of `/api/:username`, as long as a 'username' entry is provided in the options.params object.
 */
export const get = async <T>(url: string, options?: IGetOptions) => {
  const parsedUrl = parseUrlParams(url, options?.urlParams);

  const resp = await axios.get<T>(parsedUrl, {
    headers: getAuthorizationHeader(),
    params: options?.queryParams,
  });
  return resp.data;
};

export const post = async <T>(url: string, data?: any, options?: AxiosRequestConfig<any>) => {
  const resp = await axios.post<T>(url, data, {
    ...options,
    headers: getAuthorizationHeader(),
  });
  return resp.data;
};

export const put = async <T>(url: string, data?: any, options?: AxiosRequestConfig<any>) => {
  const resp = await axios.put<T>(url, data, {
    ...options,
    headers: getAuthorizationHeader(),
  });
  return resp.data;
};

export const _delete = async <T>(url: string, options?: AxiosRequestConfig<any>) => {
  const resp = await axios.delete<T>(url, {
    ...options,
    headers: getAuthorizationHeader(),
  });
  return resp.data;
};
```

Fig: Facade interface

Publish-Subscribe Pattern

Our communication-service as well as editor-service makes use of Socket.io, which is an implementation of the publish-subscribe design pattern in order to create real-time communication through the browser.

Microservices Pattern

Microservices is an architectural style that structures applications as a collection of loosely connected services, making it easier for us to scale and build applications. With microservices, we divided our application into 6 different microservices; user service, communication service, editor service, history service, matching service and question

service. Each service uses a dedicated database, separating the database between services. This way we can reduce the amount of functional requirements each microservice is responsible for, and decouple the data between services.

Responsibilities	Microservice
User Management	User Service
Questions Management	Question Service
Handle Matching and Session Management	Matching Service
Manage Editor based on Session	Editor Service
Manage Chat based on Session	Communication Service
Manage history of all sessions created	History Service

Table: Mapping of responsibilities to microservices

Each of our services are implemented with Nodejs and Express. They also follow a Model View Controller (MVC) architecture.

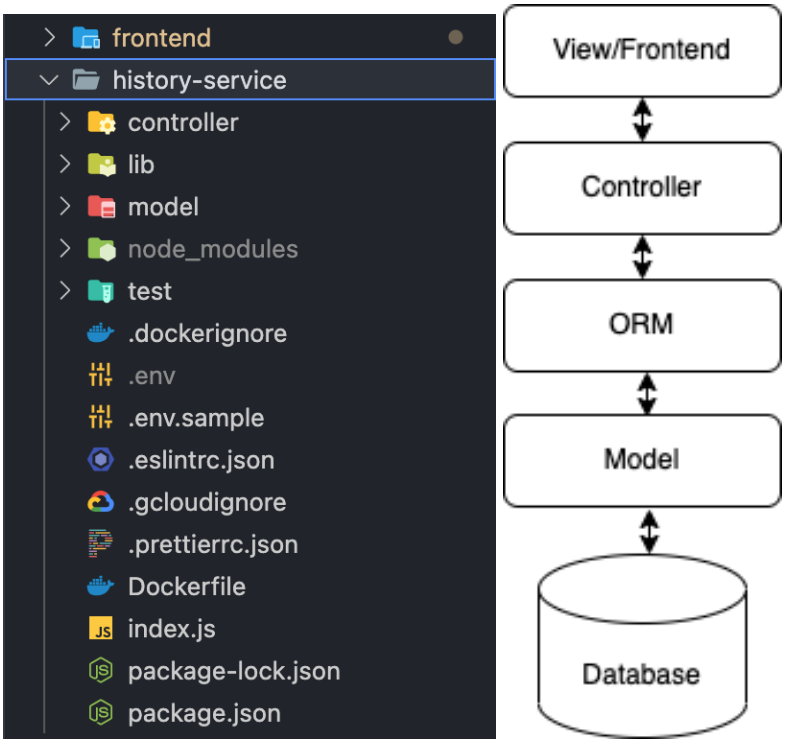


Fig: MVC pattern within microservices pattern

Specifically, we implemented the microservices with MVC and with a ORM layer. This ORM layer will allow us to reduce the repetitive code for a particular multiple data access layer, saving, deleting and reading object from the database. This mapping of program objects to

database data also means that should we decide to change the database in the future, we only have to change the repository file.

This allows for better separation of concerns and makes each component of the service easily testable and extensible. Each service provides several REST APIs endpoints that the frontend client can interact with.

Development Process

The entire development process was planned before we started development. Before the first sprint, the team had decided on our SDLC model, coding standards, and CI/CD pipeline which will be elaborated below.

Scrum

Structure

The team adopted the Scrum methodology which is a sub-model of the popular Agile methodology. Scrum allows the team to deliver increments of valuable work in short cycles called sprints in which we hold weekly. Through these sprints, ongoing feedback can be given, allowing for inspection and adaptation.

The team's weekly sprints are set to be conducted every Thursday evening.
Structure of our sprint are as follows:

1) Every team member will update the others on what they have managed to complete and whether they are on track with respect to the timeline given last sprint.
2) Blockers are brought up and discussed, specific technical details will be discussed with relevant teammates later on if not all.
3) Go through what is completed as a whole and what can be released/deployed
4) Go through the backlog if any and assign tasks for this sprint until the next sprint after discussing individual workload and capacity.

Overview of our Sprints		
Sprint(Week)	Date	**Key Deliverables**
3	25 Aug	- Set up team repo - Set up project notes and board - plan for upcoming weeks - MVP implementation

4	1 Sep	- Setting up User Service
		  - Set up backend for sign up
		  - Set up backend for login
		  - Set up backend for username check
		  - Set up backend for change password
		- Generate FRs and NFRs
		- Set up Zustand

5	8 Sep	- Matching Service - Migrate to NextJS from ReactJS and deploy - Set up ESLint and Prettier, Vercel pre-commit hook
6	15 Sep	- Matching Service (cont.) - Work on project document
7	29 Sep	- Refactor matching service and integrate with frontend - Set up question service (only api call/primitive UI) - Set up collaboration service (only api call/primitive UI) - Write test for user service
8	6 Oct	- Integrate frontend UI with question and collaboration services respectively - Deploy question service - Deploy collaboration service - Integrate user-service tests with deployment (CI/CD)
9	13 Oct	- Milestone 2 preparation
10	20 Oct	- Integrate other services tests with deployment (CI/CD)
11	27 Oct	- Improve Frontend UI - Add tests for all services and frontend
12	3 Nov	- Complete all diagrams and documentation

Github Projects

In addition, the team also utilizes github projects to keep track of each other's progress, backlog and the sprint itself.

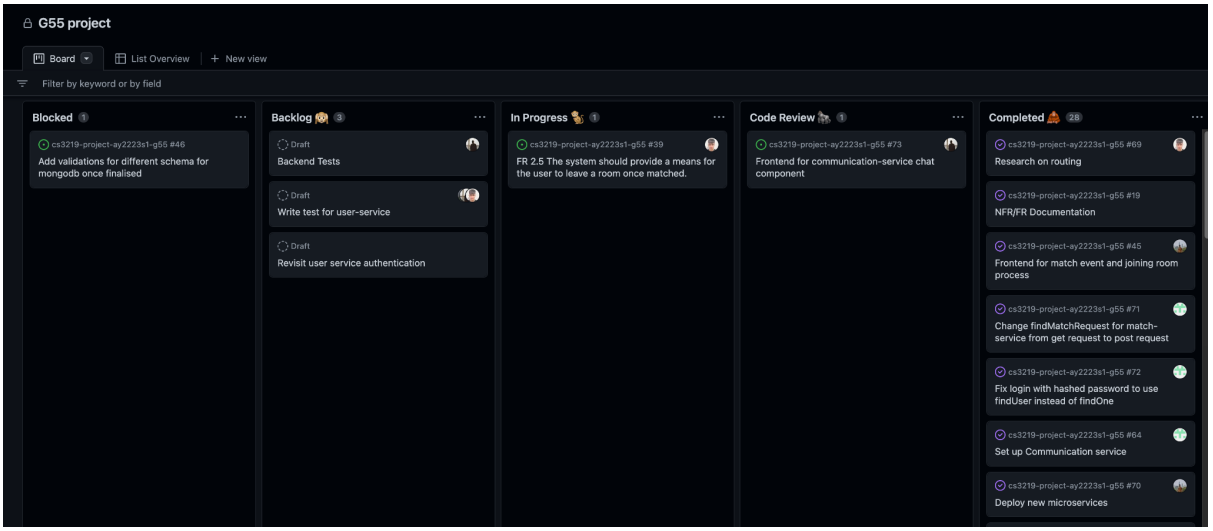


Fig: Example of what our github projects looks like during a sprint

Coding Standards

TypeScript

Use of TypeScript in frontend Next.js Project. TypeScript allows us to set types on variables and functions, and type-check our code statically which allows us to catch errors earlier at compile time.
The inclusion of types allows every developer on the team to easily collaborate and share features with each other. TypeScript reduces the chances that another developer introduces code that does not work with existing code, as developers can rely on type definition to understand what a part of the code requires as input or returns.

ESLint & Prettier

The use of ESLint reflects the team's decision to have a consistent and clean code. ESLint is used to enforce code standards for both style and syntax. The team adopts the use of Airbnb's JavaScript Style Guide, which includes a list of well-developed rules configured for ESLint developed in-house by Airbnb and is well-adopted by the developer community for enforcing standards in JavaScript and React. The team also includes typescript-eslint which is TypeScript support for ESLint which includes over 100 rules that detects code violations, bugs and stylistic issues specifically for typescript code on top of existing rules in ESLint. We also use Prettier to aid on top of ESLint in code format standardization.

Fig: ESLint Configurations

```
{
  "tabWidth": 2,
  "singleQuote": true,
  "jsxSingleQuote": true,
  "printWidth": 100,
  "semi": true,
  "endOfLine": "auto",
  "trailingComma": "es5"
}
```

Fig: Prettier Configurations

Pre-commit Hook

We implemented a pre-commit hook to run the lint checks whenever we make a commit. This is for the developer to statically test the code before it gets pushed up to the PR, increasing standardization across the codebase and reducing style bugs.

CI/CD Pipeline

Frontend Preview Deployment

Whenever new commits are pushed to a new branch, a preview build will automatically be generated and deployed by the Vercel integration. Frontend preview builds interact directly with the production API points, allowing us to effectively and rapidly test changes before merging into production.

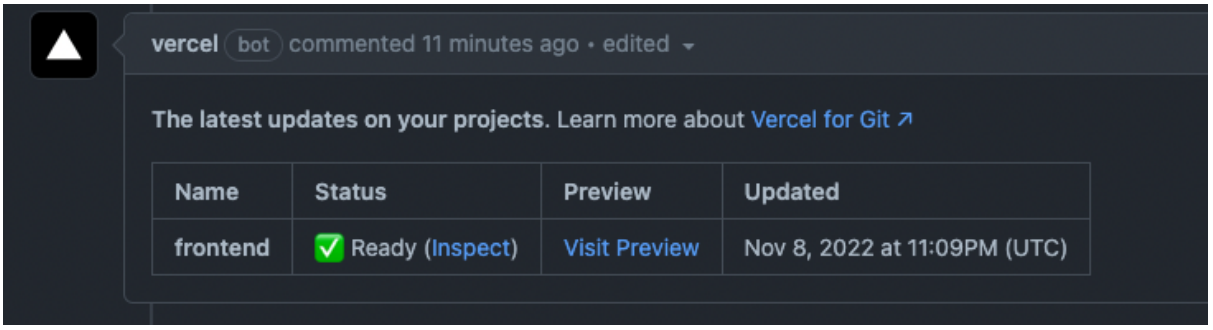


Fig: Frontend preview build success message and preview link

Frontend Deployment

New commits merged into our main branch will automatically be built and deployed by the Vercel integration.

Backend Test (PR)

Backend tests are run when commits are made to PRs which target the master branch. This facilitates code reviews, reducing human error while improving productivity as part of the CI pipeline.

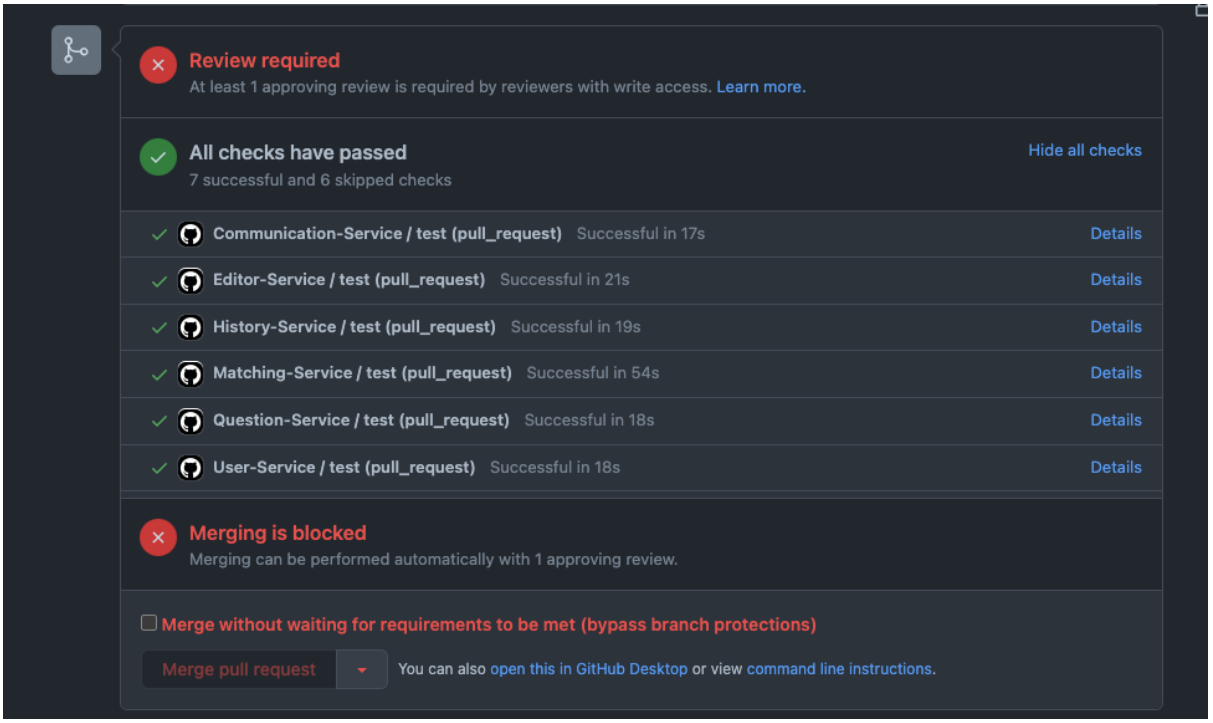


Fig: CI tests for a pull request

Backend Test and Deployment

Servers are automatically deployed whenever a new commit with changes are pushed to master. Tests are run before deployment, as a last measure to ensure that endpoints are checked before being pushed into production.

Features

User System

Since LeetWarriors does not contain extremely sensitive data, and it is likely that a user will have to log in regularly to work on practice problems, it is beneficial to have the user's credentials cached in their browser so that they do not have to repeatedly log in every time they access LeetWarriors. As such, we generate a JWT for the user on their first login, and cache them in the browser's cookies.

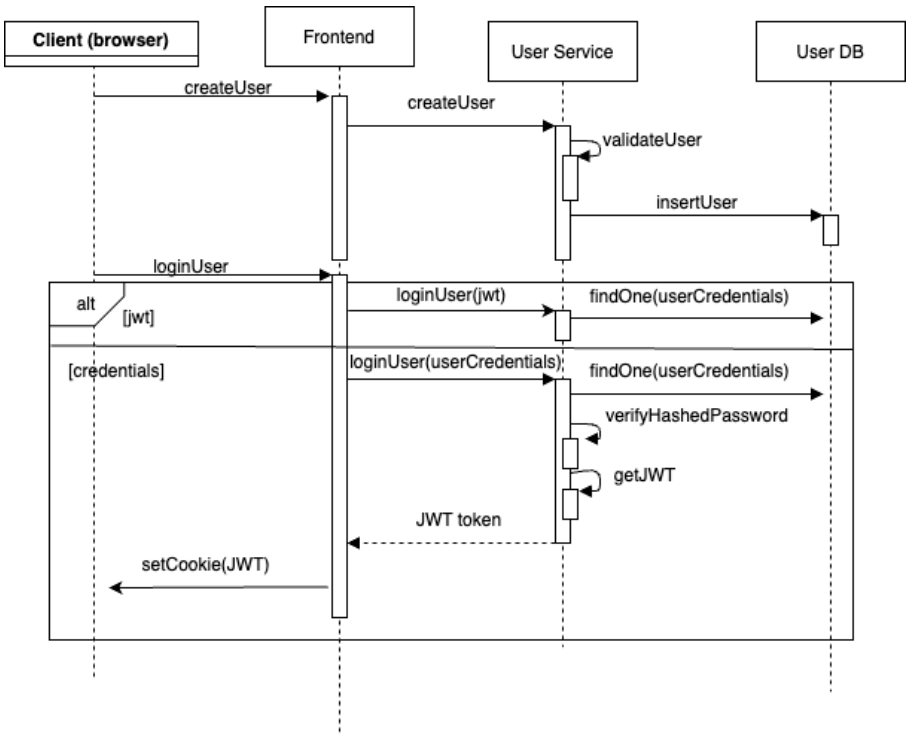


Fig: User Service Sequence Diagram for creating and logging in user

HTTP Request Method	Endpoint	Body	Response	Purpose
POST	/api/user	{ **username**: String; **password**: String; }	**201** { **successMessage**: string } **400** Invalid or missing fields **409** Username already exists	Creates a new user
PUT	/api/user	{ **oldPassword**: string; **newPassword**: string; }	**200** { successMessage: string } **401** Unauthorized **404** User not found **409** oldPassword does not match existing password	Updates a user's credentials
DELETE	/api/user		**200** { **message**: String; } **401** Unauthorized **404** User not found	Deletes the user
POST	/api/user /login	{ **username**: string; **password**: string; **currToken**: string; }	**200** { **questions**:String[]; } **401** Invalid or missing credentials	Authenticates a user

GET	/api/user /session		**200** { **successMessage**: string; **_id**: string; **username**: string; } **401** Invalid or missing credentials **404** User not found	Checks if authorization token is valid
POST	/api/user /logout		**200** { **successMessage**: string; } **401** Invalid or missing credentials	Logs out a user

Question System

The Question System is an indispensable feature used throughout the app, providing end-users with an ample number of questions. It leverages the Question Service and the developed UI to display a list of questions which are stored in our database.

Users are able to filter through the 3 different difficulties that are used to categorize questions — **Easy**, **Medium**, and **Hard**. The Question System provides a description of the problem to solve, as well as adequate examples for users to test against their solutions. For each question, the system also provides a comment section which can be used for discussion or other purposes.

The questions were taken from here, using a Python script to scrape the HTML page.

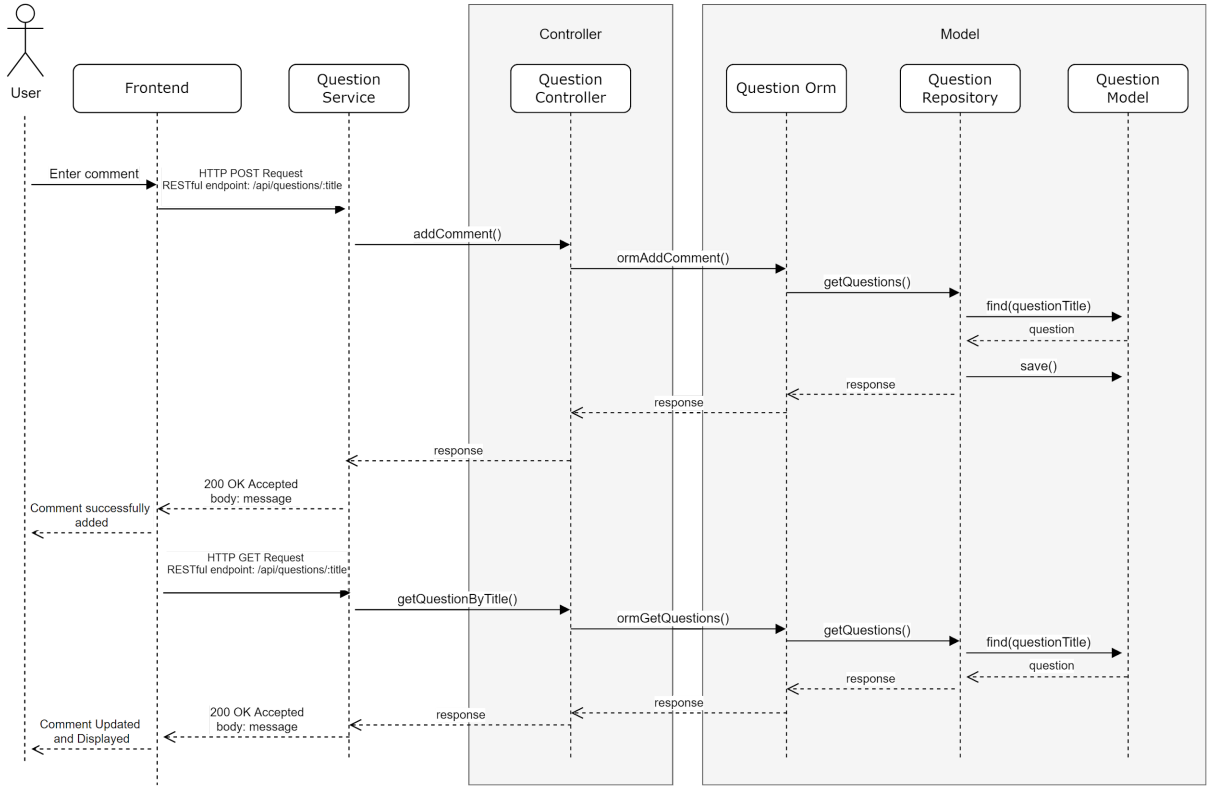


Fig: Sequence Diagram demonstrating the process of adding a comment to the comment section.

HTTP Request Method	Endpoint	Body	Response	Purpose
GET	/api/question/question		**200** { **questions**: QuestionModel[] }	Retrieves all questions
POST	/api/question/question	{ **title**: String, **description**: String, **difficulty**: enum: [‘Easy’, ‘Medium’, ‘Hard’] }	**200** { **message**: String } **400** Invalid or missing title/difficulty/description { **message**: String }	Adds a question

GET	/api/question/question/:title		**200** { **question**: QuestionModel[] } **400** Invalid or missing title { **message**: String }	Retrieves question based on title
POST	/api/question/question/:title	{ **user**: String, **comment**: String }	**200** { **message**: String } **400** Missing title/user/comment { **message**: String }	Adds a comment to the comment section for a particular question

Communication System

The Communication System is a feature that makes use of the message service of our communication service. The message service allows users to chat via text messages and will store the messages in the case of reconnection. The message service is implemented with both mongoDB and SocketIO.

Upon opening up the Chat UI, a socket connection is established with the server Socket and the user join a room as per their session Id given.

When the user types something and send a message, the UI calls `createMessage` to the Message Controller which creates a message is the *MessageModels* Collection in our MongoDB Database. Upon successful creation of message in *MessageModels* Collection. The frontend will send the message through its Socket into the Room it is in, server socket in the Communication service receives this message and forwards it to all the users' sockets who are in the room. Users will not be able to send message into the chat room if message is not successfully created in the MessageModel Collection to ensure message accuracy and message persistence.

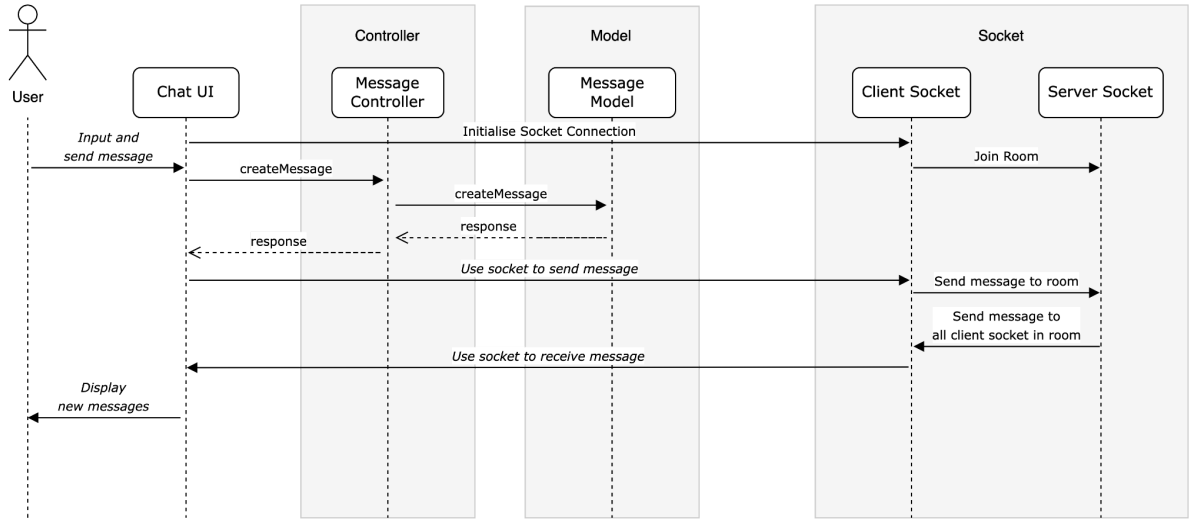


Fig: Sequence Diagram to submit and receive new messages

HTTP Request Method	Endpoint	Body	Response	Purpose
GET	/api/communication/messsage/:sessionId	-	**200** { **message:string** messages:[] }	Retrieves all messages belonging to this session
POST	/api/communication/message	{**sessionId**: String, **senderName**: string, **senderId**: string, message: string}	**201** { **message**: String data:resp } **400** Could not create a new message or Invalid or missing sessionId/senderName/senderId/message { **message**: String }	Creates a message to send to the session

Matching System

The Matching System is a feature that makes use of the Matching Service from the UI to allow users to be matched with another user that has indicated the same difficulty level for their practice session. After selecting the difficulty level, and selecting the option to find a match. The Matching System will help find a match for the user for 30 seconds until the user cancels the match request or a match is found. The UI calls sendMatchRequest to the Matching Controller in Matching Service. The Matching Controller checks for the corresponding match request by calling findMatchRequest

If there is another request with the same difficulty level in the current MatchingModels Collection of the matchServiceDB database, a match for the user is found and Matching Service will update this entry to true for ‘isMatched’ with both user’s information before creating an entry in the MatchSessionModels Collection. If not Matching Service will create a new match request in the MatchingModel Collection. After creating a new match request, the Matching Controller will continue to call checkMatchRequestIsMatched within 30 seconds. If a match request is not matched after 30 seconds, deleteMatchRequest will be called to delete the match request by the user in the MatchingModel Collection.
Once a match is found, the response will be returned to the frontend Match UI, and the user will be redirected.

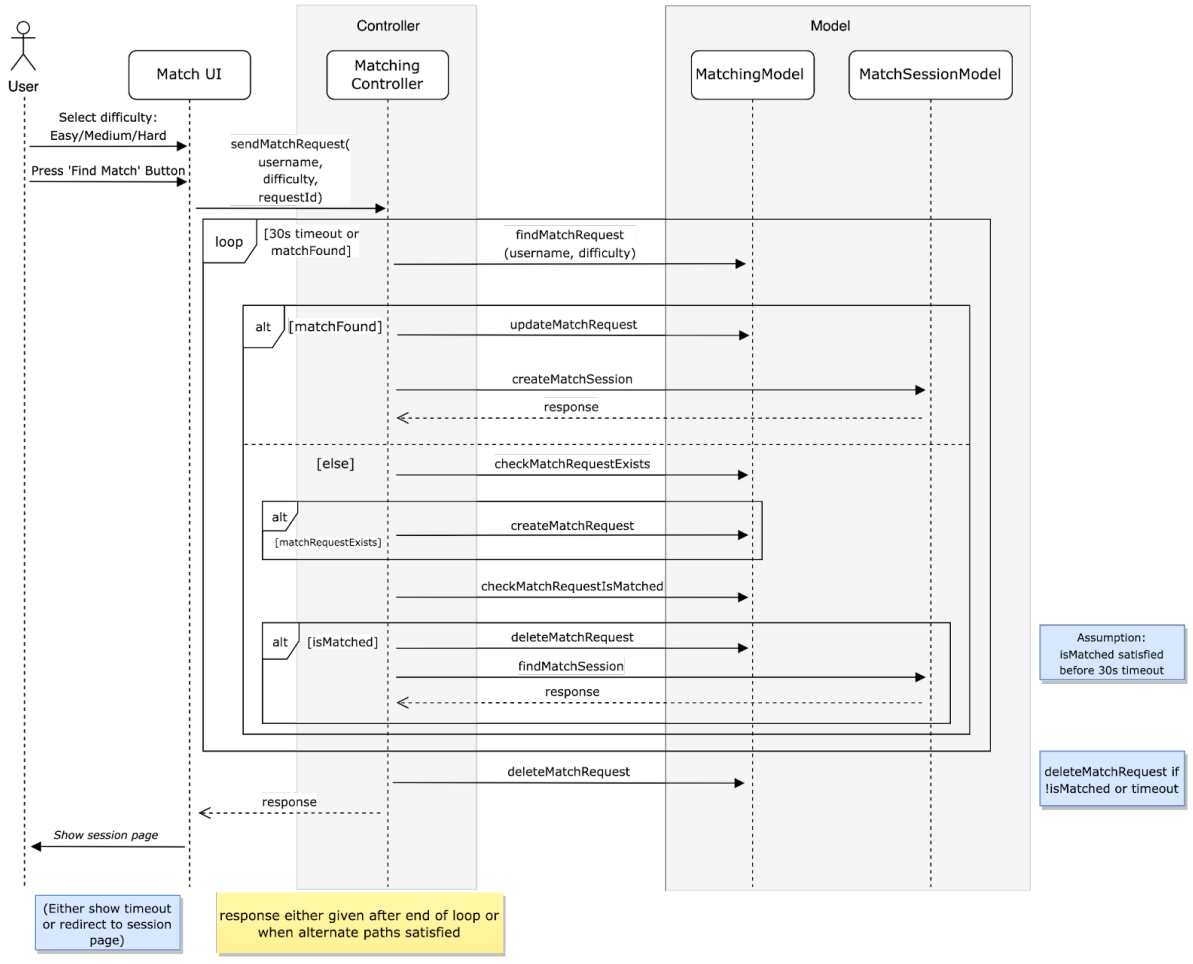


Fig: Sequence Diagram demonstrating the process of matching users in match page

HTTP Request Method	Endpoint	Body	Response	Purpose
GET	/api/match/sessions/:sessionId	-	**200** { data: { difficulty: String, username1: String, user1RequestId: String username2: String, user2RequestId: String, question: String, createdAt: Date } } **400** No sessionId or No such session with sessionId Found { **message**: String}	Retrieves session information including users and question
POST	/api/match/request	{**username:** String, **difficulty:** String, **requestId**: String}	**200** { **message**: String } **400** Invalid or missing username/difficulty/requestId or Match request cancelled Or Could not find a match after 30s { **message**: String }	Match users according to the difficulty that they selected
POST	/api/match/cancel	{**username:** String, **difficulty:** String}	**201** { **message**: String } **400** Invalid or missing username/difficulty or No such match request { **message**: String }	Updates match request that it is cancelled

DELETE	/api/match/request	{**username:** String, **difficulty:** String}	**201** { **message**: String } **400** Invalid or missing username/difficulty or No such match request { **message**: String }	Deletes match request

History Service

History Service is an integral part of the Learning Pathway system. It aims to record and maintain the session histories of all its users, and also to give qualitative feedback for users.

Whenever a session is initiated between 2 users, a new history record is logged into the matching database through a post-save hook from the matching database. This introduces some coupling between the 2 microservices, but it is only minimal, since this hook is a trigger in MongoDB. The completion of the create session does not rely on the result of the hook (creating a history record). The history service relies on the matching service to work, but only for it to trigger a createRecord call.

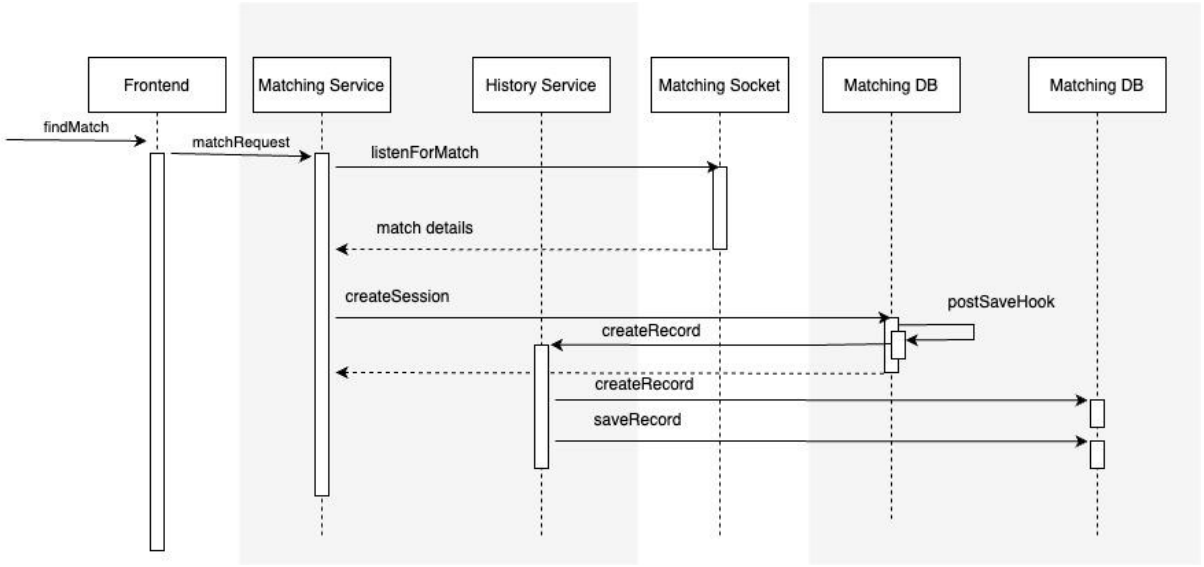


Fig: Sequence Diagram showing how records are saved from new sessions

HTTP Request Method	Endpoint	Body	Response	Purpose
GET	/api/history /records /:username (?limit=[number] &offset=[number])		**200** { **records**: RecordModel[]; }	Gets all historical records of user
POST	/api/history /records /:username	{ **questionName**: String; **firstUsername**: String; **secondUsername**: String; **startedAt**: Date; **questionDifficulty**: String; **duration**: Number; }	**200** { **message**: String; } **400** Invalid or missing RecordModel attributes	Adds historical record of user
GET	/api/history		**200**	Gets list of

	/completed /:username		{ **questions**:String[]; }	question titles done by user
GET	/api/history /completed /difficultyCount /:username		**200** { **Easy**: number; **Medium**: number; **Hard**: number; }	Gets number of questions completed by user for each difficulty
GET	/api/history /completed /monthCount /:username		**200** { **counts**: { **month**: number; **year**: number; **count**: number; }[] }	Gets number of questions completed by user per month
GET	/api/history /experience /:username		{ **experiencePoints**: number; **experienceLevel**: string; }	Gets the experience points and experience level obtained by user

Editor System

The editor system is an integral feature that allows users to collaborate on an editor in real time. After being allocated a partner to work on the problem together, they will both be able to make use of an editor that works similarly to Google Docs. The information keyed inside the editor is immediately synced between two people and is also saved at an interval of every 2 seconds. This is so that the content does not disappear on refresh or reconnect. Since the editor system is handled entirely by sockets where all the information needed between the frontend and the backend goes through the socket instead of REST APIs. Thus there will be no API table for this section.

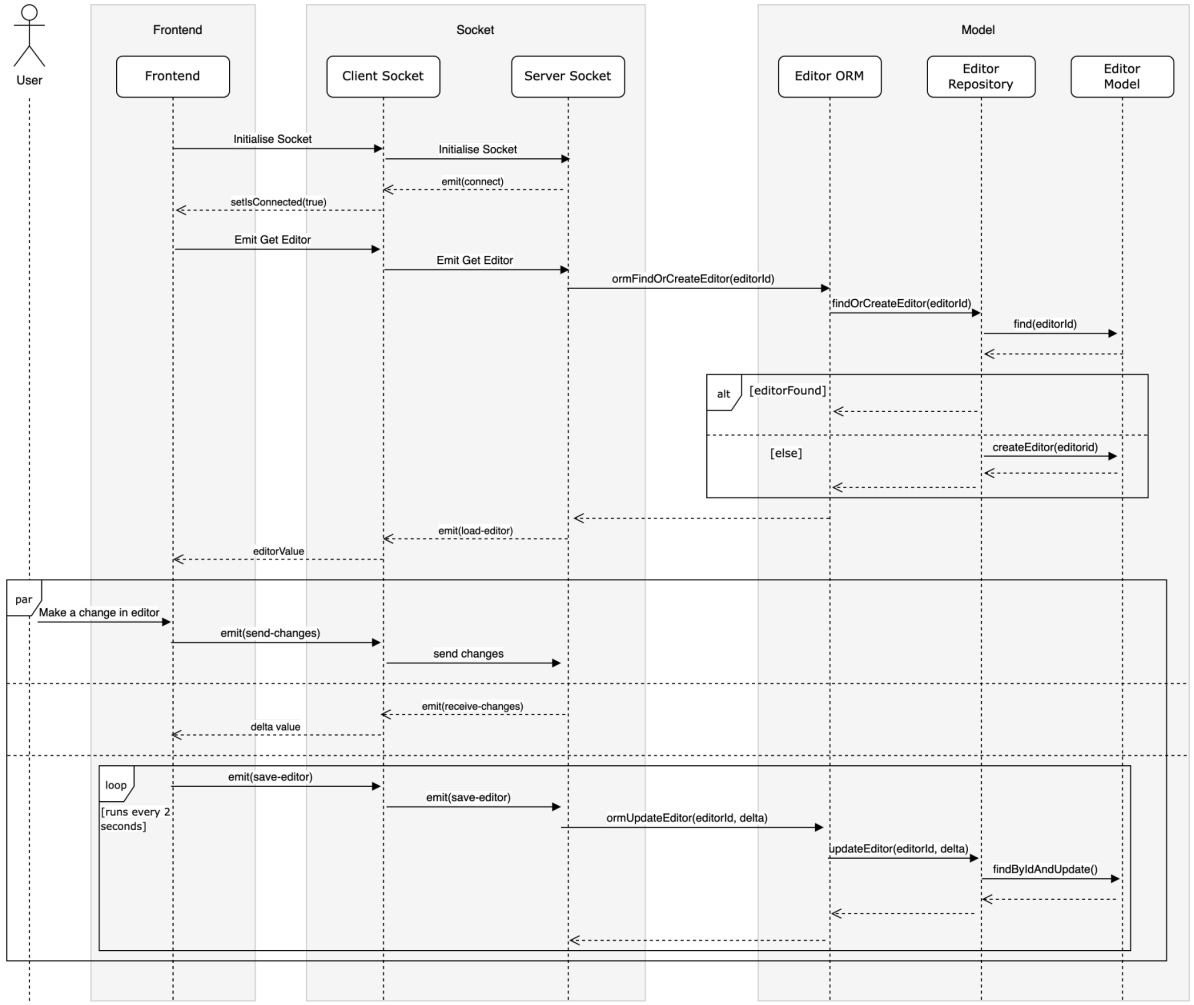


Fig: Sequence Diagram demonstrating the process of loading editor and updating editor contents

Functional Requirements

The table shown below are the functional requirements defined for the different services throughout our application. The different requirements have been assigned different priorities, either **High (H)**, **Medium (M)**, or **Low (L)**.

Most of the functional requirements shown in the table have been implemented thus far, and have been listed as either **Yes (Y)** or **No (N)** in the table below.

Functional Requirements				
Service	**ID**	**Functional Requirement**	**Priority**	**Implemented**
User	1.1	The system should allow users to create an account with username and password.	**H**	**Y**
	1.2	The system should ensure that every account created has a unique username.	**H**	**Y**
	1.3	The system should allow users to log into their accounts by entering their username and password.	**H**	**Y**
	1.4	The system should allow users to log out of their account.	**H**	**Y**
	1.5	The system should allow users to delete their account.	**H**	**Y**
	1.6	The system should allow users to change their password.	**M**	**Y**
	1.7	The system should allow users to customize their profile information. (name, email …)	**L**	**N**
Matching	2.1	The system should allow users to select the difficulty level of the questions they wish to attempt.	**H**	**Y**
	2.2	The system should be able to match two waiting users with similar difficulty levels and put them in the same room.	**H**	**Y**

	2.3	If there is a valid match, the system should match the users within 30s.	**H**	**Y**
	2.4	The system should inform the users that no match is available if a match cannot be found within 30 seconds.	**H**	**Y**
	2.5	The system should provide separate/unique rooms to different pairs of users who matched.	**H**	**Y**
	2.6	The system should allow users to cancel their match request.	**H**	**Y**
	2.7	The system should provide a means for the user to leave a room once matched.	**M**	**N**
	2.8	The system should allow users to join specific rooms.	**L**	**N**
	2.9	The system should allow users to select their preferred topics/categories before matching or before starting session	**L**	**N**
	2.10	The system should be able to match two waiting users with similar preferred topics/categories and put them in the same room.	**L**	**N**
Question	3.1	The system should retrieve a question of appropriate difficulty level to the shared workspace once the participants are matched.	**H**	**Y**
	3.2	The system should provide the user with a suggested solution.	**M**	**N**
	3.3	The system should have a comment/forum page where users can discuss about the questions and their different solution	**M**	**Y**
	3.4	The system should be able to give users a set of test cases.	**L**	**N**
	3.5	The system should be able to take in some test cases and output the result of that test case for any given solution.	**L**	**N**

	3.6	The system should be able to verify the correctness of the written program by running it through our test cases or against our suggested solution	**L**	**N**
	3.7	The system should provide hints on request.	**L**	**N**
	3.8	The system should match and retrieve a question based on the user's selected topics during the matching, if specified.	**L**	**N**
Collaboration	4.1	The system should provide a collaborative code/text editor in the room that allows users to collaborate with each other.	**H**	**Y**
	4.2	The system should provide syntax highlighting and code IntelliSense in the editor	**L**	**N**
Communication	5.1	The system should provide chat support for participants to send messages in the room.	**H**	**Y**
	5.2	The system should provide voice calling support for participants to communicate in the room.	**L**	**N**
	5.3	The system should provide video calling support for participants in the room.	**L**	**N**
History	6.1	The system should save all records of a user's sessions.	**L**	**Y**
	6.2	The system should provide the records of the questions that the user has attempted.	**L**	**Y**
	6.3	The system should provide the records of other user collaboration instances.	**L**	**Y**
	6.4	The system should calculate and output the experience points that a user has accumulated.	**L**	**Y**
	6.5	The system should provide aggregated data of a user's history such as the count of completed questions by month and question difficulty.	**L**	**Y**

Table x: Functional Requirements of all services

Non-functional Requirements

Similar to the functional requirements section, this section displays a table containing the non-functional requirements gathered for all services in this application, including the metric that it falls under.

Non-Functional Requirements					
Service	**ID**	**Non-Functional Requirement**	**Metric**	**Priority**	**Implemented**
User	1.1	A user must verify the deletion of their account	**Robustness**	**H**	**Y**
	1.2	A strong password (Longer than 8 characters, including at least 1 alphabet, number, and special character) must be used when registering for an account.	**Security**	**H**	**Y**
	1.3	A user must be authenticated and authorised before they can utilise the application's features and APIs.	**Security**	**H**	**Y**
	1.4	The login and signup operations should not take more than 2 seconds to complete.	**Performance**	**M**	**Y**
	1.5	Users' passwords should be hashed and salted before storing in the Database.	**Security**	**M**	**Y**
	1.6	The system must be able to persist/store at least 100,000 user profiles without performance degradation	**Scalability**	**L**	**Y**
	1.7	The user login session should persist until the user logs out or clears the browser cache.	**Usability**	**L**	**Y**
Matching	2.1	The matching process should/should exit after cancelling the match request.	**Robustness**	**H**	**Y**
	2.2	Users should not be waiting for a match if there are other suitable users seeking for a match.	**Usability**	**H**	**Y**
	2.3	The system should be able to handle more than 200 concurrent users requesting for a match without huge performance degradation.	**Scalability**	**M**	**N**
	2.4	The collaborative platform should have a latency of fewer	**Performance**	**M**	**N**

		than 3 seconds			
Question	3.1	The question bank should be able to save at least 1000 questions without any significant performance degradation.	**Performance**	**H**	**Y**
Collaboration	4.1	The collaborative platform should have a latency of fewer than 3 seconds	**Performance**	**M**	**Y**
	4.2	The collaborative platform should not allow users who do not belong to the room to access	**Security**	**L**	**N**
Communication	5.1	Messages should not take more than 2 seconds to send.	**Performance**	**H**	**Y**
	5.2	The communication service should not allow users who do not belong to the room to access	**Security**	**L**	**N**
History	6.1	Should be able to maintain 1000 last records for each user.	**Performance**	**L**	**Y**

Table x: Non-functional Requirements of all services

NFR Tests

Hare a few examples of tests we run to check our NFRs

Security

NFR 1.3 A user must be authenticated and authorised before they can utilise the application’s features and APIs.
We implemented a middleware in our microservices which checks for authorization to access the services' specific endpoints. Below is an example of matching-service.

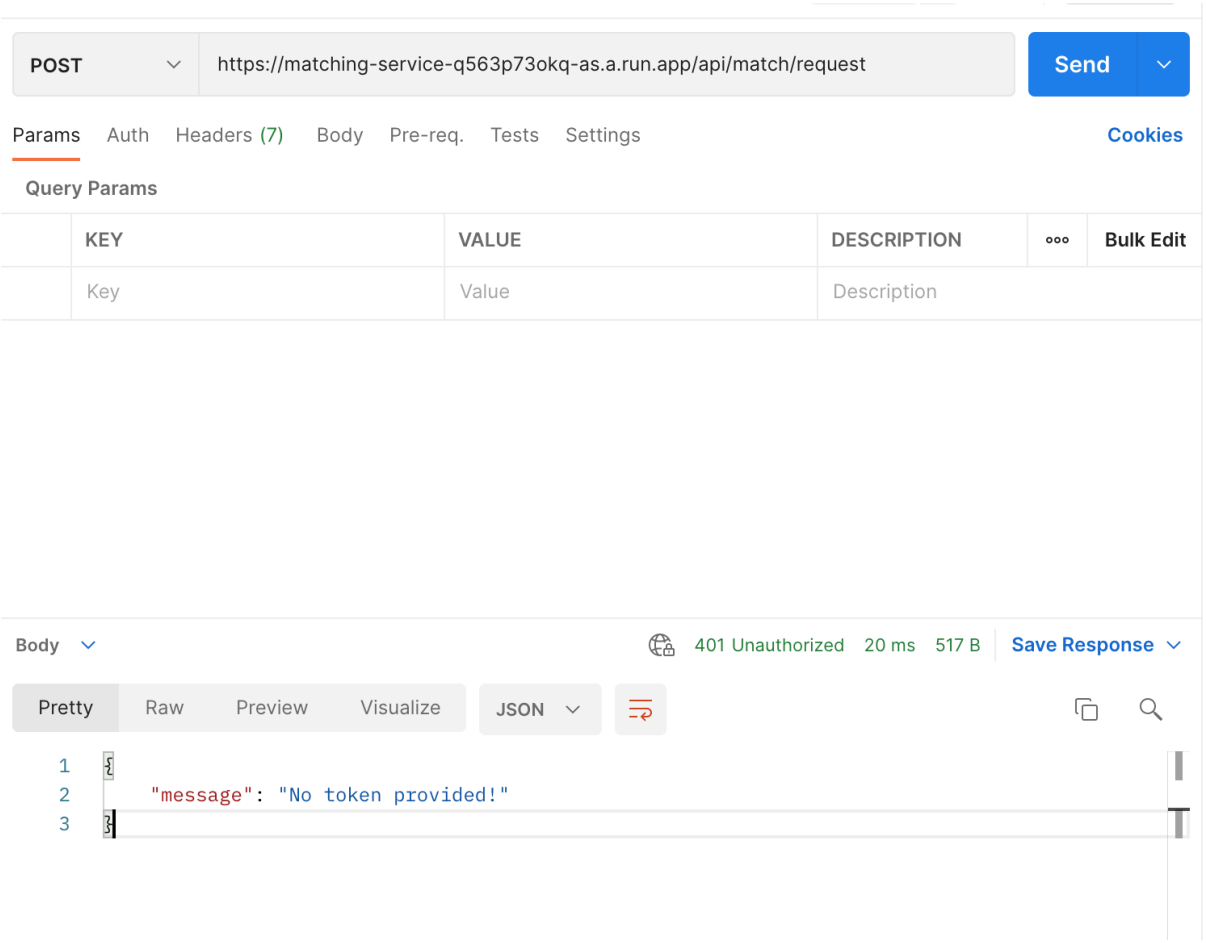


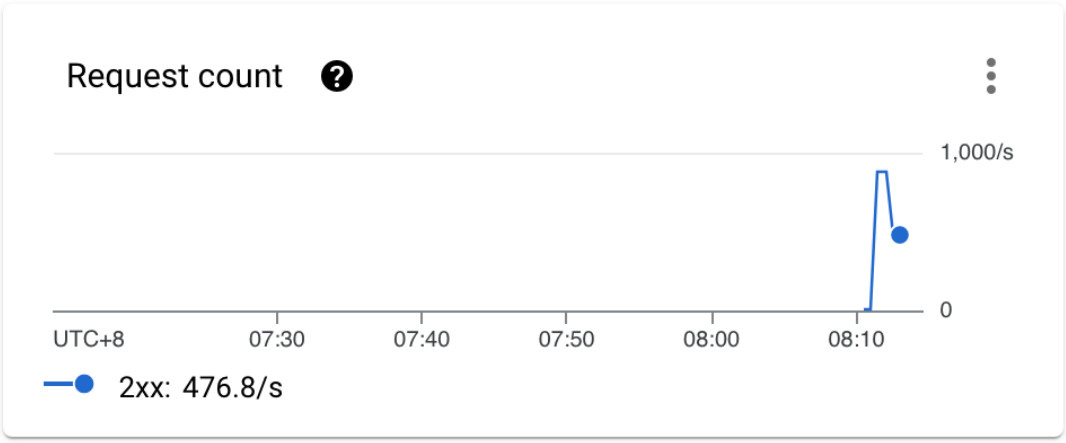
Fig: Postman testing whether endpoint is protected

Scalability

We make use of k6 and javascript scripts to run load testing on our services. Below is an example of testing history-service.
We create 2000 Virtual Users(VUs) which are 2000 concurrent sessions querying the ’GET’ endpoint request for 5 seconds at the same time to stress test the scalability and performance of the system. Every Virtual User will perform the GET request, in a continuous loop, as fast as it can.

```
     /\      |‾‾| /‾‾/   /‾‾/
    /\  /  \     |  |/  /   /  /
   /  \/    \    |     (   /   ‾‾\
  /          \   |  |\  \ |  (‾)  |
 / __________ \  |__| \__\ \_____/ .io

  execution: local
     script: test/getScripts.js
     output: -

  scenarios: (100.00%) 1 scenario, 2000 max VUs, 35s max duration (incl. graceful stop):

running (31.5s), 0000/2000 VUs, 2288 complete and 0 interrupted iterations
default ✓ [======================================] 2000 VUs  5s

     ✓ status is 200

     checks.........................: 100.00% ✓ 2288      ✗ 0
     data_received..................: 9.7 MB  308 kB/s
     data_sent......................: 1.6 MB  51 kB/s
     http_req_blocked...............: avg=2.52s    min=0s       med=2.38s    max=5.92s    p(90)=5.19s    p(95)=5.55s
     http_req_connecting............: avg=250.67ms min=0s       med=191.42ms max=1.16s    p(90)=661.14ms p(95)=1.01s
     http_req_duration..............: avg=13.71s   min=275.16ms med=14.38s   max=29.45s   p(90)=23.96s   p(95)=25.53s
       { expected_response:true }...: avg=13.71s   min=275.16ms med=14.38s   max=29.45s   p(90)=23.96s   p(95)=25.53s
     http_req_failed................: 0.00%   ✓ 0         ✗ 2288
     http_req_receiving.............: avg=2.34ms   min=16µs     med=81µs     max=697.41ms p(90)=1.1ms    p(95)=2.09ms
     http_req_sending...............: avg=86.57µs  min=14µs     med=74µs     max=665µs    p(90)=150µs    p(95)=183.65µs
     http_req_tls_handshaking.......: avg=2.26s    min=0s       med=1.97s    max=5.73s    p(90)=4.95s    p(95)=5.28s
     http_req_waiting...............: avg=13.7s    min=275.07ms med=14.36s   max=29.45s   p(90)=23.96s   p(95)=25.53s
     http_reqs......................: 2288    72.743146/s
     iteration_duration.............: avg=16.23s   min=711.06ms med=16.74s   max=31.44s   p(90)=28.39s   p(95)=29.35s
     iterations.....................: 2288    72.743146/s
     vus............................: 54      min=54      max=2000
     vus_max........................: 2000    min=2000    max=2000
```

Fig: Metrics shown for http requests



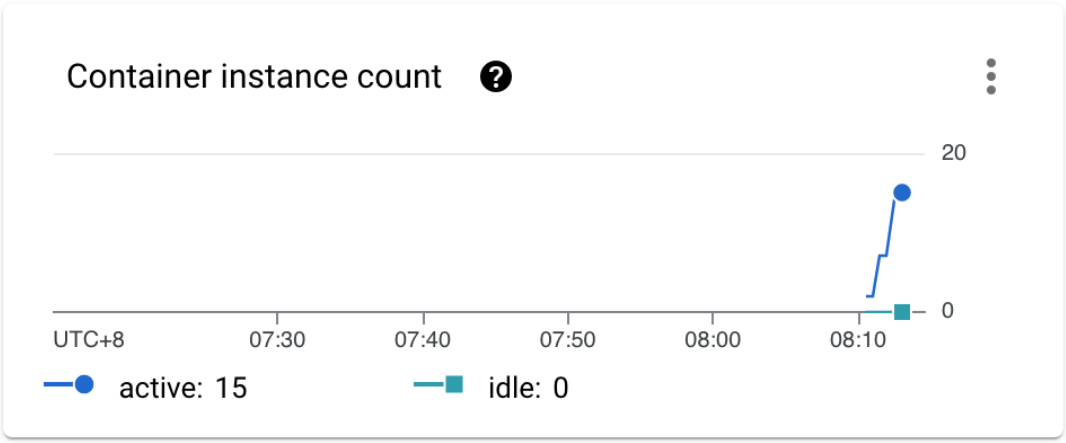


Fig: Container instances increase as requests increase

Reflections

Challenges Faced

Implementation of WebSocket

For some of our services, we make use of Socket.IO which is a framework for WebSocket in order to implement. As none of our team members had exposure to WebSocket development before. This was a steep learning curve. The configuration of payloads through Socket.IO is very flexible and is up to our own design and standardisation. There were a lot of different opinions and styles online, hence we had to pick and implement our own standards. Also, we spent ample time implementing matching-service with Socket.IO, only to refactor it to use mongoDB solely instead, this undoubtedly affected the amount of time left for other features as we spent over 2 weeks implementing and debugging Socket.IO for matching-service.

Unit Testing

For each of our services, unit testing was done using chai and Mocha. The priority is to ensure that the REST APIs that each service provides is functional. However during the testing of our REST APIs, we encountered many issues with time out and environment issues where when we added these unit test files onto the workflow for continuous integration, it suddenly broke and was not working as intended. There were also times where the test files will accidentally delete some of the data that we have pre-populated into the database. Hence causing some of our features to not be behaving as intended. However, we were able to narrow down the problem and remove it as a team and get the whole unit testing and our features to work well in deployment.

Deploying Microservices

We weighed the pros and cons of each cloud service from multiple different providers and decided on either Google App Engine or Google Cloud Run. Ultimately both gave us what we needed, and a big plus of Google App Engine is the single-entrypoint feature that it provided. However, we faced multiple issues trying to get App Engine to work properly with Socket.IO. We researched long about how to solve this issue, but there were too many configurations and settings to configure. Google-Github-Actions plugin doesn't support environment variables for App Engine out of the box either, which further complicated the deployment process.

Learning Points

Software Design

From trying to design and implement our own microservice architecture from scratch, we faced the need to make many design decisions for our different services according to our requirements. We had to set priorities for our different requirements in order to have a better idea of what design decisions we will make. The team has grown to appreciate and understand the importance of different Software Architectural Patterns as well as Design Patterns.

Importance of Scrum

Scrum involves proper pre-planning, where we divide our tasks into smaller sub tasks in each sprint, to better manage our product backlog. This will allow us to focus better on these smaller tasks whilst increasing efficiency at the same time. With scrum, our team members are not subjected to an organizational hierarchy, giving each other equal importance and respect. This has allowed us to effectively communicate our issues and problems with each other, ruling out some minor conflicts that our team had.

Effective Communication

Our team has learnt a lot about the importance of effective communication. We started off being very reserved and didn't give each other the right critiques and opinions. This led to a lot of inefficiency when we realised that something has to be changed at the later date. Luckily our team was quick to warm up with each other and by being professional, everyone gave constructive feedback without holding back and we were able to convey thoughts and ideas among our team clearly. With effective communication the team found itself to be efficient during the weekly sprints and the project was completed smoothly.

Suggestions

Video and voice communication

As of now, LeetWarriors is able to allow users to collaborate on a real-time in-sync editor while also allowing them to communicate with each other by sending texts on the same page. Since technical interviews are usually done physically or over a zoom call, it would be beneficial if we can get them to get used to using video calls while solving problems and using it to communicate their thoughts and ideas to their problem-solving skills.

It would also enhance communication between peers giving rise to a better collaboration experience. To implement this feature, we could have an additional microservice that, like the editor and chat, uses a socket to pass the information.

Interviewer/Interviewee option

Leetwarriors is currently designed to allow two users to collaborate on their work currently as equals. In the future, we want to implement an option for users to choose whether to collaborate on coding problems or to take turns and assume the role of interviewer and interviewee. Where the interviewer is given a set of questions and prompts for them to read out and test the interviewee. The interviewee will then stimulate it as a real interview and attempt to perform live coding and explain his thought process while he is coding. Once the time is up they will then be able to switch roles.

This would allow users to not only get familiar with problem solving with another person but also get comfortable with the setting of interview live coding hence preparing them for an actual interview. If we do want to add on this feature, it would have to be implemented in the matching service where they will be allowed to choose.

Preview Builds

Cloud Run gives us a huge flexibility in terms of creating multiple different versions of our services and deploying them to different instances (each instance a different container). It is possible and feasible to design a preview build deployment into our CI/CD pipeline, where pull requests targeting the main branch should open a new preview build on some url. This url is calculated based on the commit name and author. This determined url should also be provided to the frontend preview through a preview config file, allowing the frontend preview to be integrated with the backend preview, setting up a full app preview for PRs.

Deployment

Microservices

Microservices deployed on GCP Cloud Run

- Scales automatically based on CPU resource consumption
- Can delegate traffic across multiple instances

Frontend

Frontend deployed on Vercel

- CI/CD pipeline out-of-the-box
- Preview builds for testing/development
- NextJS + Vercel can handle routing (can handle redirecting/rewriting requests to microservices from the server config)

Application Screenshots

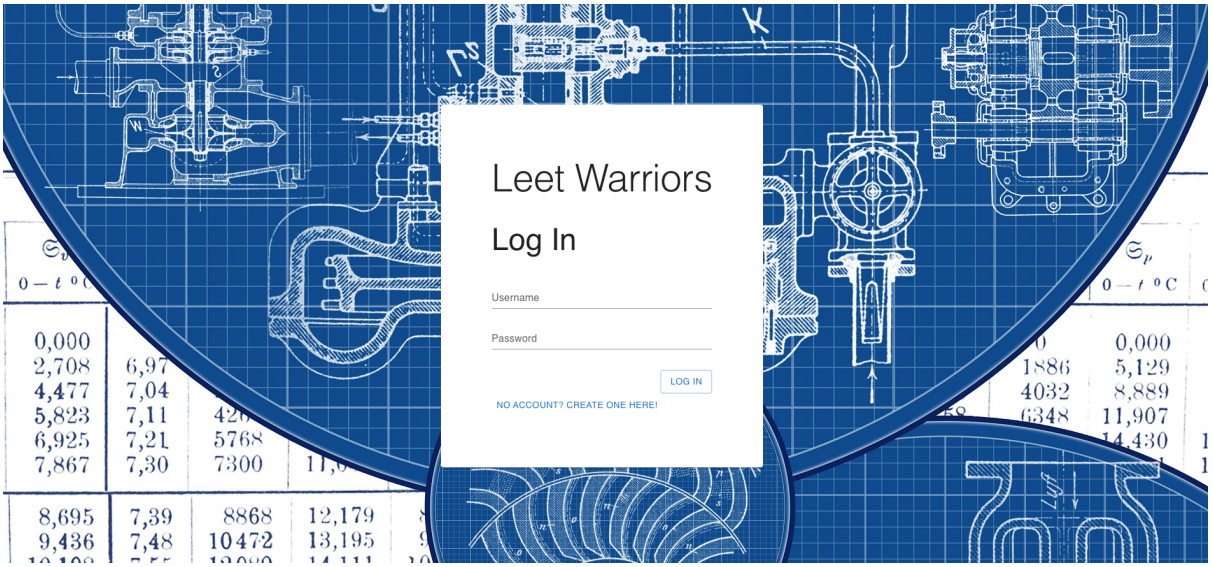


Fig: Login page

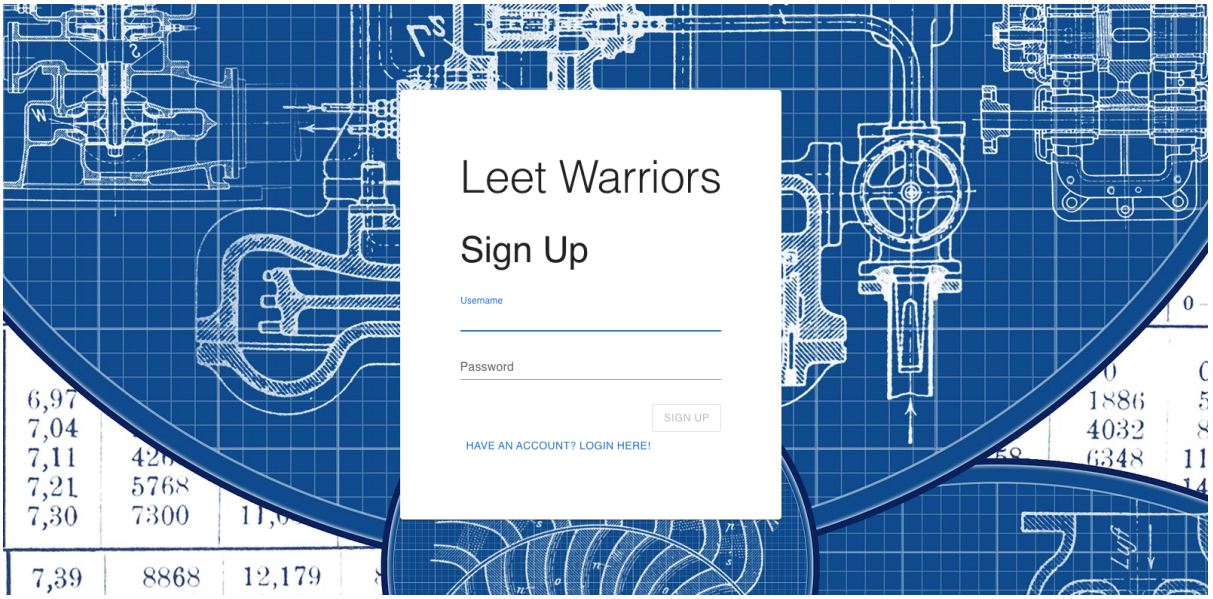


Fig: Sign Up page

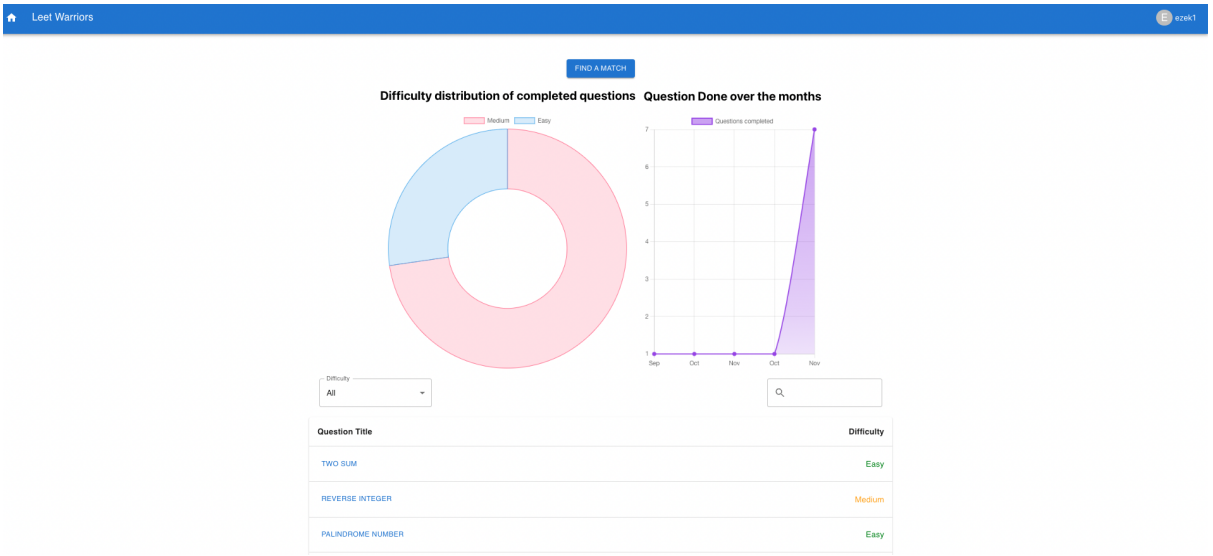


Fig: Dashboard page, where users can view a list of questions and their aggregated statistics

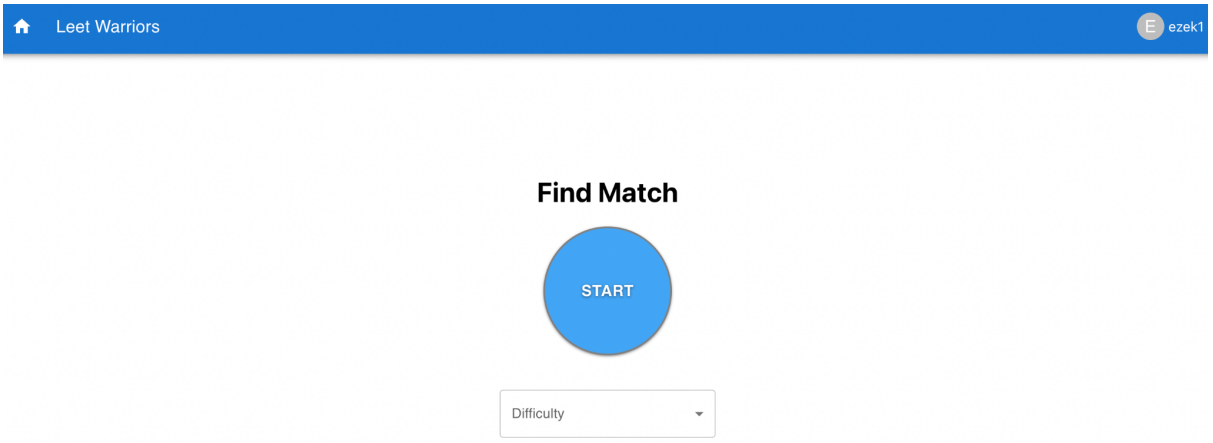


Fig: Matching page

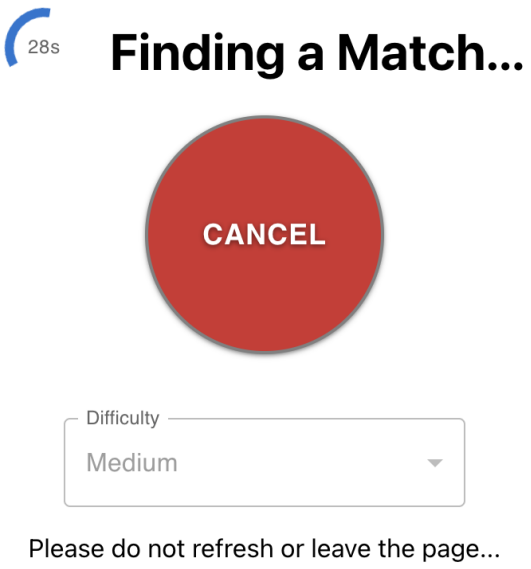


Fig: User is currently waiting for a match

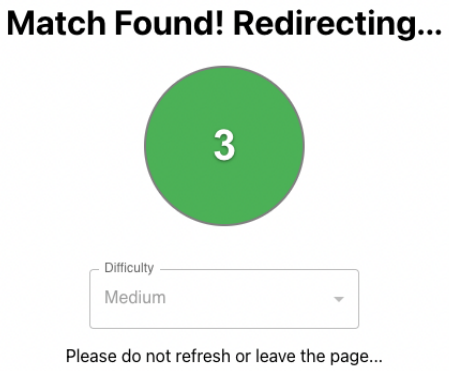


Fig: Match has been found, success message shown

Minimum Number Of Steps To Make Two Strings Anagram

Medium

Given two equal-size strings s and t. In one step you can choose **any character** of t and replace it with **another character**.

Return *the minimum number of steps* to make t an anagram of s.

An **Anagram** of a string is a string that contains the same characters with a different (or the same) ordering.

Example 1:

```
Input: s = 'bab', t = 'aba'
Output: 1
Explanation: Replace the first 'a' in t with b, t = 'bba' which is anagram of s.
```

Example 2:

```
Input: s = 'leetcode', t = 'practice'
Output: 5
Explanation: Replace 'p', 'r', 'a', 'i' and 'c' from t with proper characters to make t anagram of s.
```

Example 3:

```
Input: s = 'anagram', t = 'mangaar'
Output: 0
Explanation: 'anagram' and 'mangaar' are anagrams.
```

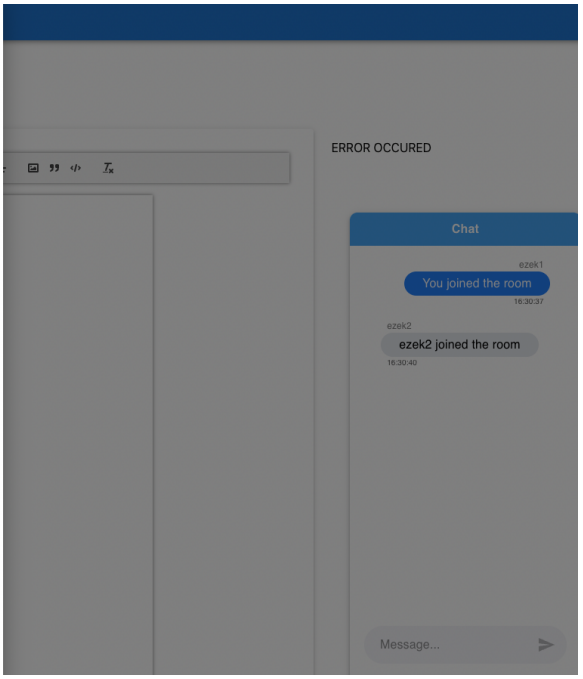


Fig: Question shown in an aside component

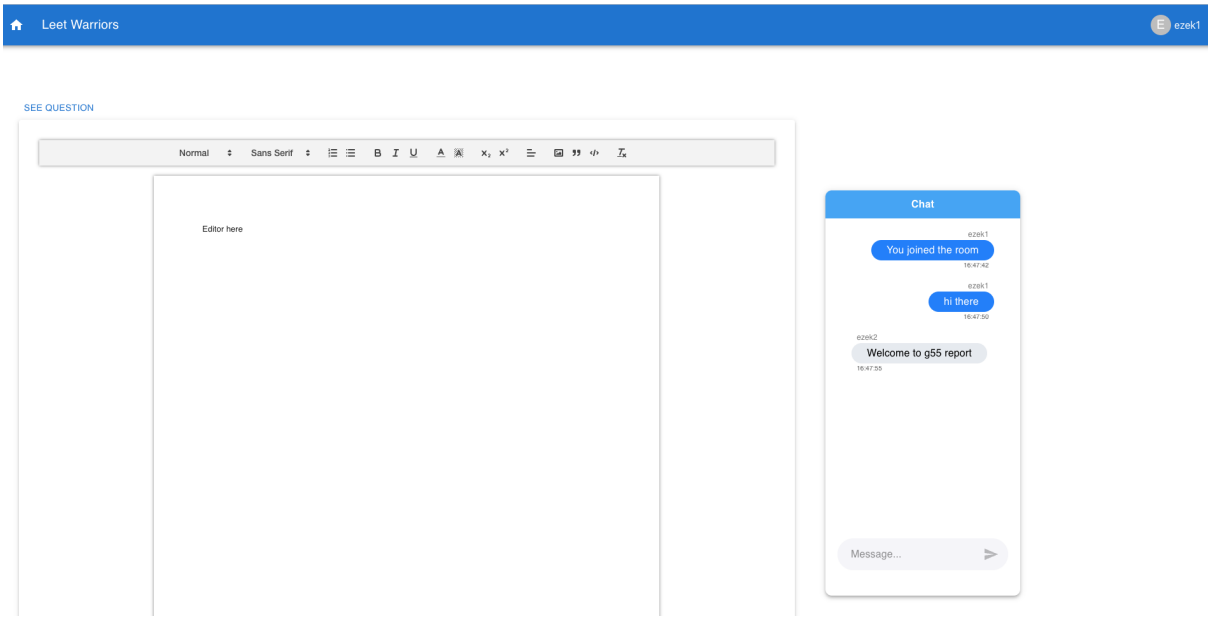


Fig: User's workspace in a session, with the editor and chat box.

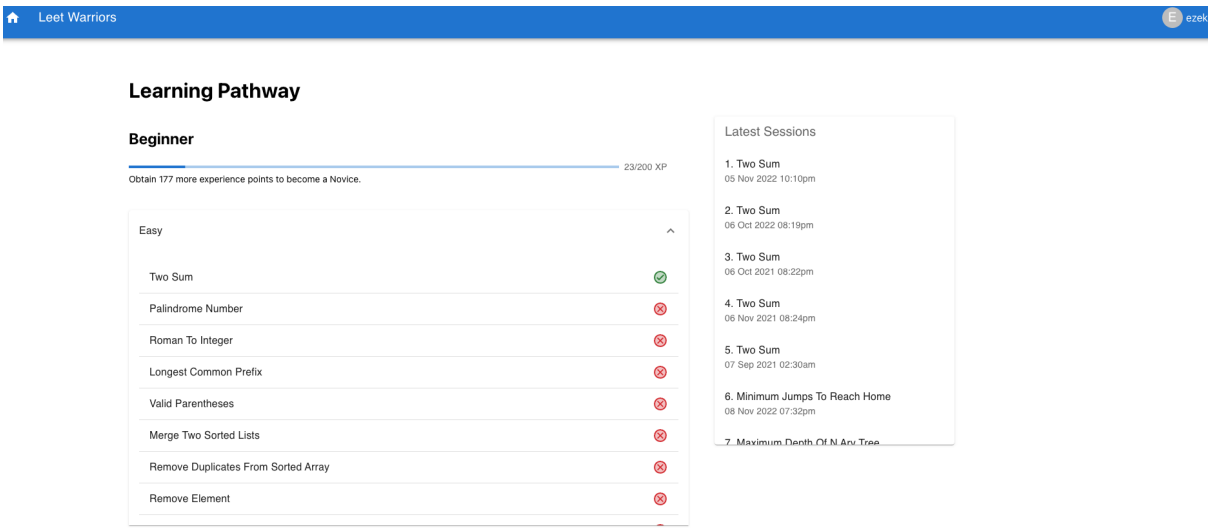


Fig: Learning pathway, shows a user's completed records and experience levels

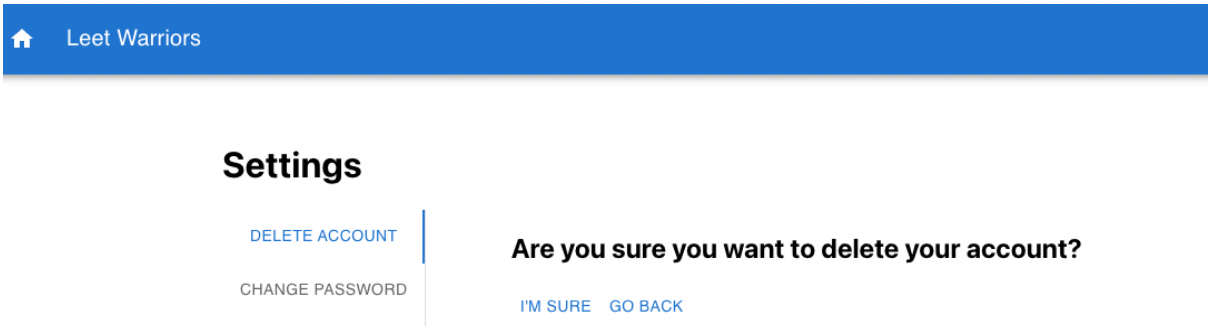


Fig: Settings page, users can delete their account or change password

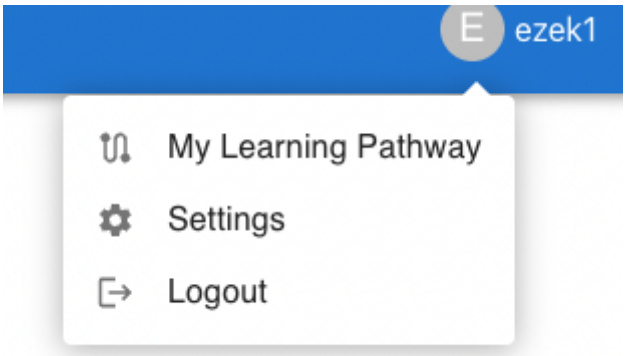


Fig: Avatar settings menu