CS3219 Group 26

Project Report

PeerPrep

Team Members

Tan Chin Kiat A0218137H

Felix Ong Wei Cong A0217387W

Lee Qi An A0217608A

Ng Choon Siong A0196600H

Kum Wing Ho A0217689L

Content Page

1. Project Introduction

1.1 Background

Students will encounter technical interviews when applying for jobs or internships. This may seem daunting to students, especially those who are attending online assessments for the first time. These assessments test students on their coding knowledge and the ability to handle time pressure.

Practice is essential in order to ace such assessments. Current platforms such as LeetCode allow students to practice commonly tested coding questions.

However, current platforms do not allow for collaborative practice and solo practicing of these questions can become tedious. To tackle these issues, we have created an online assessment preparation platform PeerPrep, where users collaborate with other users during their preparation.

1.2 User Stories

As a student practising for my technical interviews, I want to …
1.2.1. login to the website with my Google Account.
1.2.2. select the difficulty level and programming language to practise.
1.2.3. get matched with peers with the same criteria (level of difficulty and programming language).
1.2.5. know if I am being matched and be able to rematch if the matching failed.
1.2.6. be given an appropriate question based on the criteria I selected.
1.2.7. be able to enter my solutions in a code editor with proper syntax highlighting and formatting.
1.2.8. make changes to the solution and view changes made by my matched peer in near real time.
1.2.9. communicate with my matched peer over text chat in real time.
1.2.10. be notified if my matched peer has left the collaboration session.
1.2.11. communicate with a generative AI chatbot to seek help.
1.2.12. be able to view all the questions available.
1.2.13. be able to filter questions by difficulty level, title or category.
1.2.14. be able to edit my name in my profile.
1.2.15. view my history of question attempts.

1.3 User Flow

1. A student who wants to prepare for their upcoming technical interview decides to visit the site.
2. He signs in with his Google Account.
3. Upon logging in, the student will be able to view all of the questions available for practice.
4. He filters the questions based on the complexity, title and categories of the questions to view a list of questions of a certain complexity, title and category.
5. He clicks on the question title to view the details of the questions.
6. When he is ready to start coding, he selects the question difficulty level and the programming language he wants to attempt.
7. The student then waits to be matched with a user who wants to attempt a question with similar programming language and question difficulty.
8.1. The matching will timeout after 30 seconds if there are no suitable users to match with and the student will be given the choice to try again.
8.2. Upon successful matching, the student and his peer are redirected to a page with the question detail and an online integrated development environment space to enter their code.
9. The coding space allows both the student and his peer to view each other's inputs in real time and collaborate on solving the question.
10. There is also a chat box provided to allow both users to communicate with each other.
11. The chat box also serves as an artificial intelligence chatbot, where both users can toggle into a chatbot mode and ask a generative AI chatbot questions.

2. Software Requirements

2.1 Functional Requirements

Requirements	Priority	Sprint
1.1 User Service - responsible for user profile management (M1)		
1.1.1 The system should allow users to register and create new profiles.	H	9
1.1.2. The system should allow users to login with a google account	H	8
1.1.3. The system should allow users to edit their profile name.	M	9
1.1.4. The system should allow users to delete their account.	H	8
1.1.5. Some users can have the maintainer role which grants them special access to the question service	M	11
1.2 Matching Service (M2)		
1.2.1. The system should be able to match users by their choice of question difficulty.	H	7
1.2.2. The system should be able to match users by language preference.	M	11
1.2.3. The system should be able to match users by both choice of question difficulty and topic preference	H	11
1.3 Question Service (M3)		
1.3.1 The system should allow users to search for questions by question title.	M	11
1.3.2 The system should allow users to filter questions by difficulty.	M	11
1.3.3 The system should allow users to filter questions by topic.	M	11
1.3.4 The system should allow users to view a selected question.	H	10
1.3.5 The system should allow for adding a new question by a user with the maintainer role.	H	9

Requirements	Priority	Sprint
1.1 User Service - responsible for user profile management (M1)		
1.1.1 The system should allow users to register and create new profiles.	H	9
1.1.2. The system should allow users to login with a google account	H	8
1.1.3. The system should allow users to edit their profile name.	M	9
1.1.4. The system should allow users to delete their account.	H	8
1.3.6 The system should allow for updating questions by a user with the maintainer role.	H	9
1.3.7 The system should allow for deleting questions by a user with the maintainer role.	H	9
1.4 Collaboration Service (M4)		
1.4.1 The system should enable authenticated and matched users to edit the same code at the same time.	H	9
1.4.2 The system should be able to show the changes in real-time made by both matched users to both users	H	9
1.5 User Interaction (M5)		
1.5.1 The system should allow users to select the question difficulty level (easy, medium, or hard).	H	8
1.5.2 The system should allow users to select the question topic to attempt.	M	9
1.5.3 The system should allow users to select the programming language.	H	8
1.5.4 The system should provide a workspace for two users to write their code.	H	9
1.5.5 The system should allow users to end the collaborative session.	H	9
1.5.6 The system should prompt users to confirm if they want to end the collaborative session.	H	9
1.5.7 The system should allow users to sign in with their Google account.	H	7

Requirements	Priority	Sprint
1.1 User Service - responsible for user profile management (M1)		
1.1.1 The system should allow users to register and create new profiles.	H	9
1.1.2. The system should allow users to login with a google account	H	8
1.1.3. The system should allow users to edit their profile name.	M	9
1.1.4. The system should allow users to delete their account.	H	8
1.5.9 The system should provide a profile page where users can view their question attempt history.	M	10
1.5.10 The system should provide a chat for communication with generative AI	L	12
1.6 Deployment (M6, N9, N10)		
1.6.1 The application and services should be containerized.	H	11
1.6.2 The system should be hosted on a cloud platform.	H	11
2.1 Communication (N1)		
2.1.1 The system should allow users to communicate with each other through a text-based chat service	M	10
2.2 History (N2)		
2.2.1 The system should allow users to view their question history, with information such as date-time of attempt.	M	11
2.2.2 The system should allow users to view their attempt.	M	11
2.2.3 The system should allow users to view suggested solutions for the question.	M	11
2.3 Enhancing question service (N4)		
2.3.1 The questions should be sorted in increasing order of complexity.	L	12
2.3.2 The system should allow users to filter questions by complexity.	L	12
2.3.3 The system should allow users to click on a category to	L	12

Requirements	Priority	Sprint
1.1 User Service - responsible for user profile management (M1)		
1.1.1 The system should allow users to register and create new profiles.	H	9
1.1.2. The system should allow users to login with a google account	H	8
1.1.3. The system should allow users to edit their profile name.	M	9
1.1.4. The system should allow users to delete their account.	H	8
view all questions belonging to that category.		
2.3.4 The system should allow users to search for questions by title	L	12
2.3.5 The system should allow users to search for questions by category	L	12
2.4 Enhancing collaboration service (N5)		
2.4.1 The code editor should have auto formatting/indentation.	M	11
2.4.2 The code editor should have syntax highlighting for one or more languages based on users' matched language.	M	11
2.5 Generative AI (N7)		
2.5.1 The system should allow users to chat with a generative AI when in the collaboration workspace	M	12

Table 1. Functional Requirements

2.2 Non-Functional Requirements

Non-functional requirements	Priority	Sprint
NF1 Performance		
NF1.1 The editor for the collaboration service should update for all users within 0.5s of any change made by a user.	M	11
NF1.2 The system should allow at least 10 concurrent users without significant lag	H	11
NF1.3 Users should be able to log in within 2s of submitting a log-in request 95% of the time.	H	11
NF1.4 When there's an available match for a user under the collaboration service, the system should match both users within 5s.	M	11
NF2 User Experience		
NF2.1 The system should be intuitive and easy to use	H	11
NF2.2 The system should be available to use across multiple OS/browsers	H	11
NF3 Security		
NF3.1 The system should use a reliable authentication method (e.g. Google OAuth provider)	H	9

Table 2. Non-Functional Requirements

3. Developer Documentation

Architecture Overview

The overall architecture of the application is shown in Diagram 1. The backend follows a microservices architecture where each service is deployed as a separate Kubernetes deployment. Incoming requests to the backend from the frontend are routed to the appropriate services by an ingress controller.

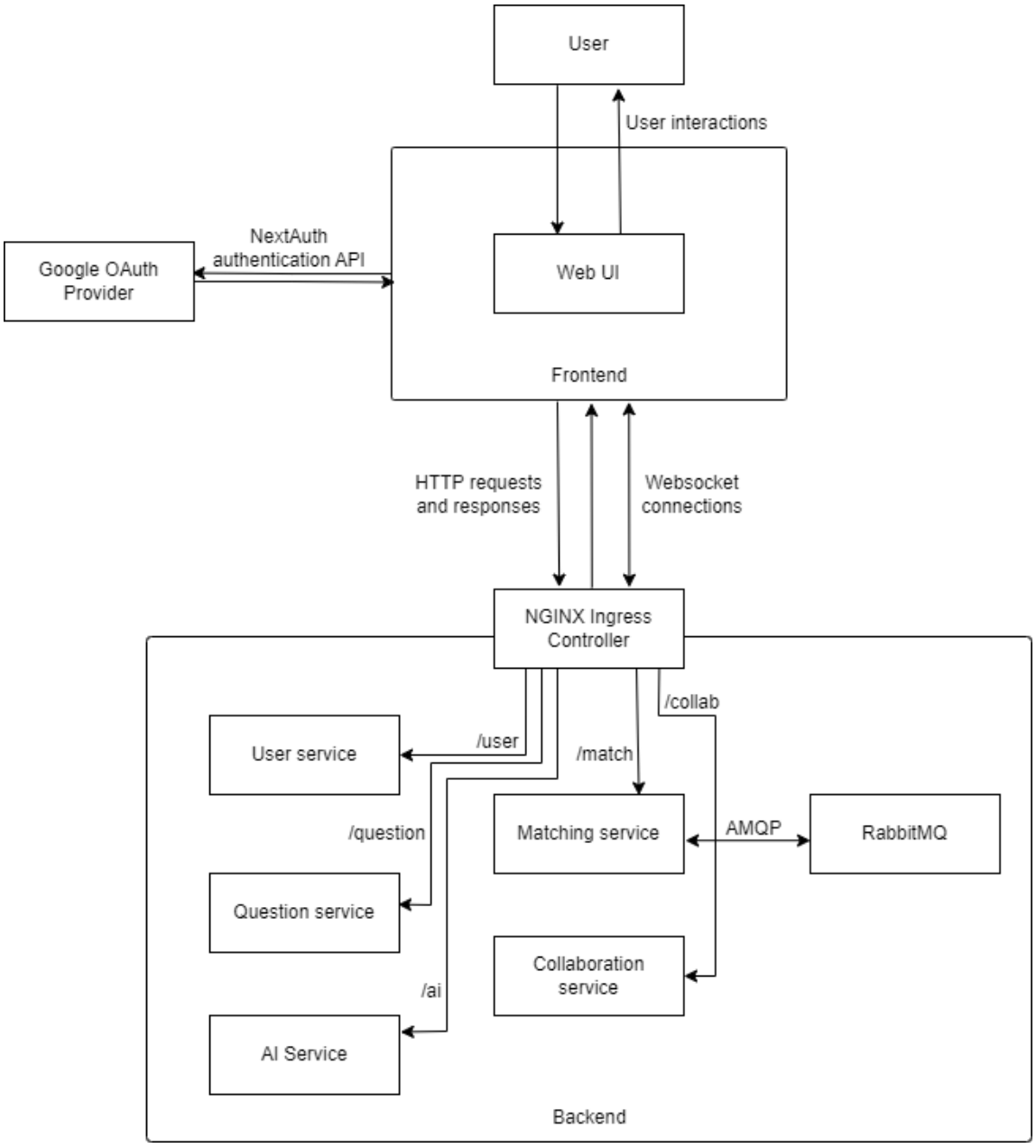


Diagram 1. Architecture Overview

3.3 Tech Stack

Component	Technologies used
Frontend	Next.js Tailwind CSS Axios remark with remark-html and remark-gfm plugins
Backend servers	Axios Express.js Node.js
Authentication	NextAuth.js
Authorisation	JSON Web Token (JWT)
User service and Attempts service	PostgreSQL express-validators
Question service	Mongoose MongoDB
Collaboration service	yjs y-websocket
Matching service	RabbitMQ AMQP socket.io
Containerization	Docker
Deployment	Google Kubernetes Engine
Project management	GitHub issues, milestones and project
Tooling	Prettier

Table 3. Tech stack

3.4 Implementation details

3.4.1 Frontend

Next.js is used as the framework for the frontend and Tailwind CSS is used for styling. In general, we try to adopt a minimalist design that is intuitive and easy to use.

3.4.1.1 Authentication and Session Management

Authentication is implemented using NextAuth.js with Google as the provider to allow users to login with their Google account using OAuth 2.0 to access the Google credentials API. Upon successful login, user information including name, email and profile picture are retrieved from the user service if it is an existing user or retrieved from the Google account if it is a new user, and stored in the session. The user's role type is stored in a JWT to be used in backend authorisation for the question service. The Provider component, which is the parent of all components, ensures that the same session information is available to all children components.

3.4.1.2 Authorisation

For frontend authorisation, two higher-order components (HOC) were created, PrivateRoute, which pushes a user to the homepage if they try to access any page that requires authentication, and MaintainerRoute, which shows a message that the user is unauthorised to access the page if they do not have the maintainer role. This is done by checking the status of the session.

```
const page = ({ params }) => {
  return (
    <div className="flex justify-center">
      <QuestionCard questionId={params.id} />
    </div>
  );
};

export default PrivateRoute(page);
```

Diagram 2. Example of using PrivateRoute HOC to protect the individual question page

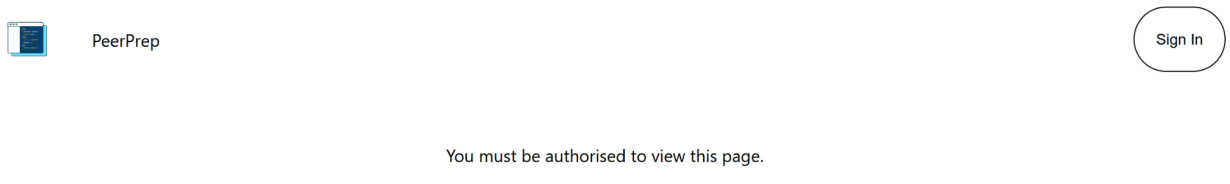


Diagram 3. Example of message shown if user is not a maintainer.

3.4.1.3 User profile management

Each user has a profile page, showing their user information stored with our application.

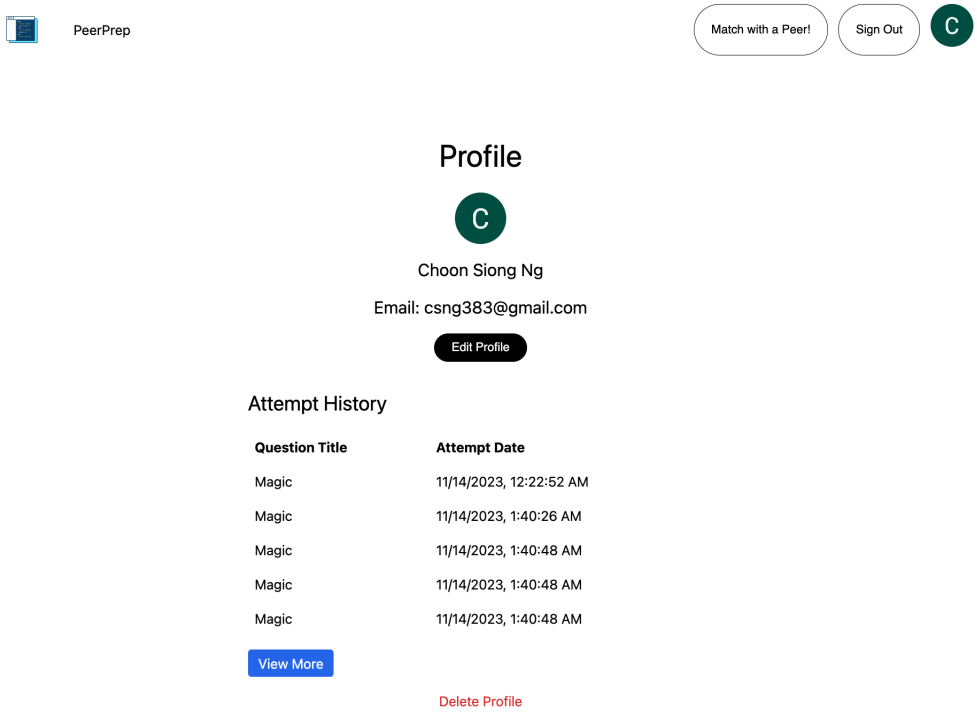
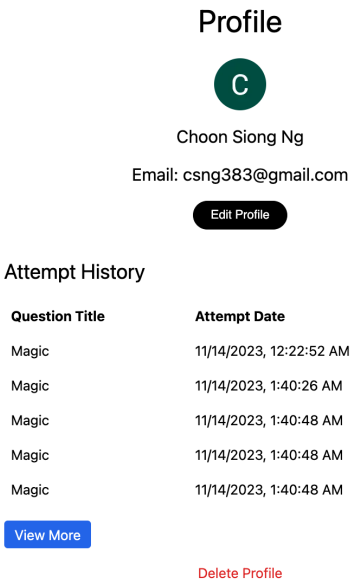


Diagram 4. User interface for User profile

The user profile information is retrieved from the JWT token and loaded into the session at load time, and populated on the screen.

Editing and Deleting a User

This page allows users to edit and delete their profile, which updates the user's jwt token and session on the frontend, and sends an asynchronous request to the backend to save the transaction.



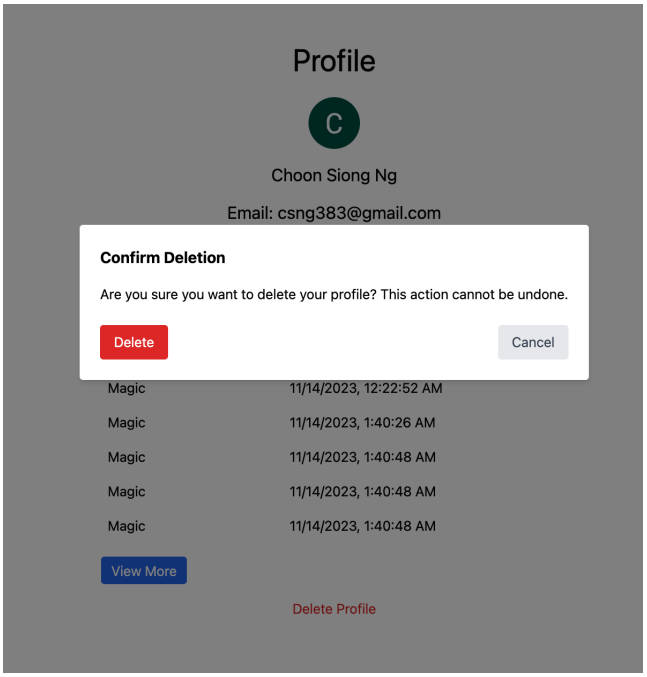


Diagram 5. User interface for editing and deleting user profile

3.4.1.4 Question Forms

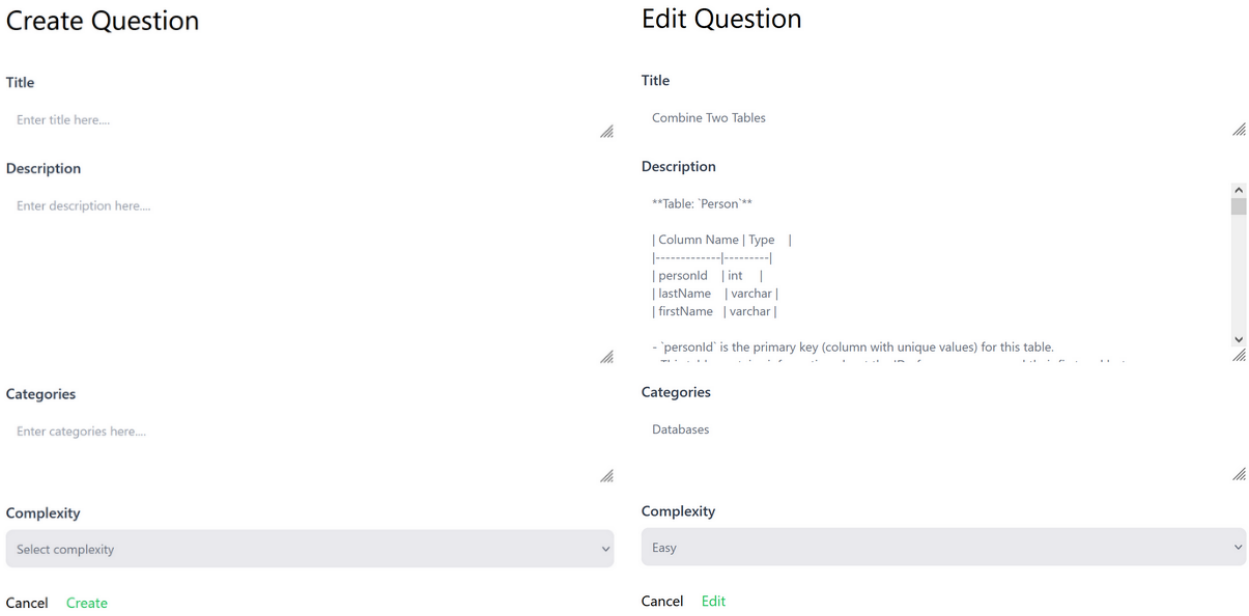


Diagram 6. Reusable form for create and edit question

A reusable question form was created and used for the create and edit question pages. This allows for a consistent design language.

3.4.1.5 Question Rendering

Each question may differ from other questions in terms of its structure, for example, whether it has tables or images, making it difficult to come up with a general HTML template for all questions.To allow for greater flexibility and dynamic rendering according to the question structure, the remark tool is used to convert Markdown into HTML. Additional plugins such as remark-html and remark-gfm were also used to allow for creation of tables. Github-style styling

was also used to provide a sense of familiarity.

Combine Two Tables

Easy

Table: `Person`

Column Name	Type
personId	int
lastName	varchar
firstName	varchar

- `personId` is the primary key (column with unique values) for this table.
- This table contains information about the ID of some persons and their first and last names.

Diagram 7. Example of rendered question

3.4.1.6 Viewing Attempts

From the user's profile (3.4.1), the user can also view saved attempts made on a question. The list of attempts is fetched client-side during run time, and populated on the profile's page. Clicking on an attempt brings up a separate page that sends an API request to the Attempts Service (3.4.7) to fetch the code.

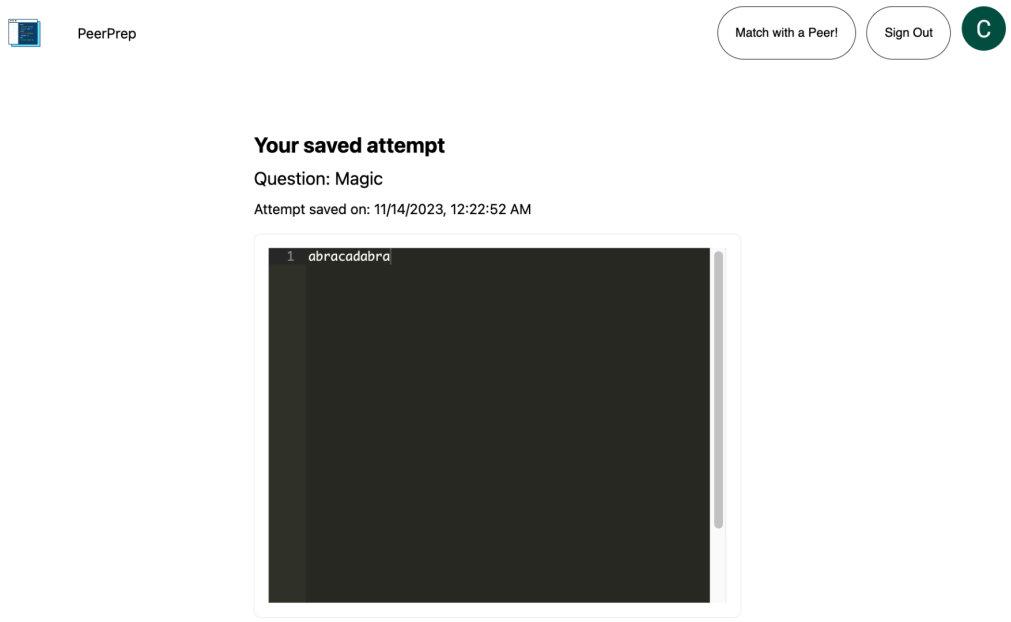


Diagram 8. User interface for viewing attempts

3.4.1.7 Collaborative Page Rendering

The collaborative page consists of a few components integrated together.
There is the question card component (3.4.1.4), a code editor with syntax highlighting depending on matched users' language choice, a toggle-able chatbox with an option to talk to AI, and a button to save current code as saved attempt. Upon clicking the save current code button, the button greys out and becomes unclickable, notifying the user that the code has been saved.

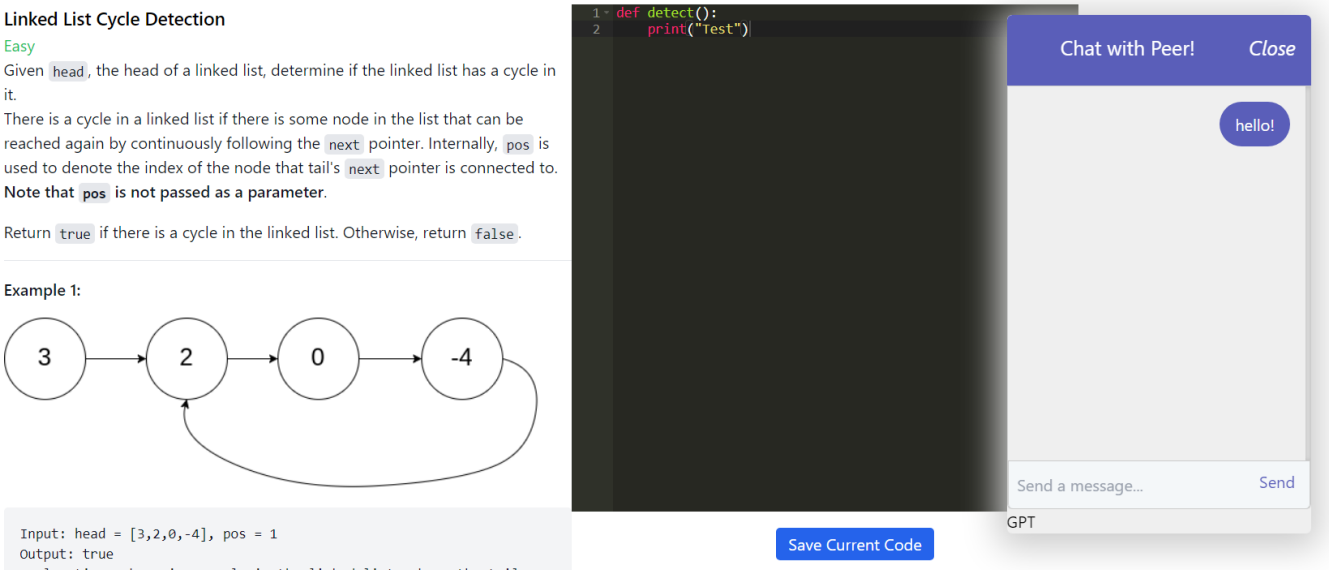


Diagram 9. Example of collaboration session

3.4.2 User Service

The User Service manages user accounts, authentication, profiles and other user-related data and functions. It is implemented as an independent microservice with a PostgreSQL database.

Architecture
The User Service follows a layered architecture with the following components

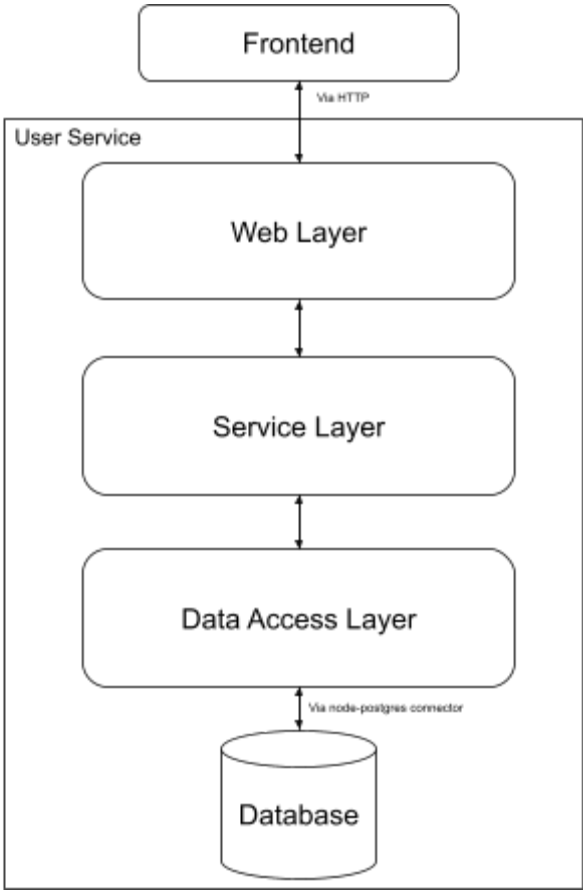


Diagram 10. User Service Architecture

Web Layer	The web layer handles the public APIs for the User Service. It consists of a Node.js Express application that exposes REST endpoints for client and other services to integrate with.
Service Layer	The service layer contains the business logic for user management and coordinates operations across components. It handles tasks like authentication and request validation and consists of the middlewares and routes.
Data Access Layer	The data layer abstracts the underlying PostgreSQL database. It consists of functions that define database operations in the controller.
Database	The PostgreSQL database provides a relational database with ACID transactions for persistent storage. It stores user accounts, profiles, roles, permissions and other related data.

Table 4. User Service Architecture Layers

Design Decisions

Decoupled architecture

A key advantage of the implementation is the decoupled architecture:

- Web Layer acts as interface adapters to the external client
- Routes in the Service Layer encapsulate core business logic
- Controllers in the Data Access Layer acts as interface to the database
- Database abstracts data persistence

This separation of concerns provides loose coupling and maintainability.

Technology stack

We chose PostgreSQL as the database as it has ACID-compliant transactional support for user management, which we felt was critical for handling important information about our users. We also chose the express-validators library to perform request validation and sanitisation, which was important to prevent malicious attacks like SQL injection or Cross-Site scripting attacks.

Session Management

We chose to store the user information and role in an embedded JWT token. This acts as the session state for the logged in user. This was done to minimise complexity as no separate session store is required.

User Schema

```
CREATE TABLE IF NOT EXISTS clientuser (
    id    SERIAL PRIMARY KEY,
    name  VARCHAR(50)        NOT NULL,
    email VARCHAR(50) UNIQUE NOT NULL,
    image VARCHAR(255)       NOT NULL
);
```

Diagram 11: Schema for Users

To avoid excessive collection of personal information, we decided to only store the minimum information necessary for our application, so we decided to store the name, email and the URL to the user's image on Google. The schema was named `clientuser` as `user` is a reserved keyword in SQL.

Control Flow

User Authentication Flow

[Insert sequence diagram showing control flow – prob don't need to include for all flows]

1. User clicks Google login button on frontend
2. Frontend calls NextAuth Google provider OAuth handler
3. Google provider prompts user consent and extracts authorization code
4. Authorization code is exchanged for access token via Google SDK
5. Google profile information (name, email, avatar URL) is extracted

6. Frontend calls backend `GET /auth/signin-new` route with user info in the request body
7. Request is forwarded by Web Layer to the auth route in the Service Layer, which calls middlewares to ensure that:
    a. Email parameter must be a valid email
    b. Name and image field must be present
    c. All fields are sanitized by replacing special characters with their HTML entries
8. The Service Layer then interacts with the Data Access Layer to check whether the user exists in the database
    a. If not, creates new user profile with 'user' role
    b. If user exists, retrieve existing user
9. The user information consisting of the saved name, image (as a link), and the user role is passed upstream and eventually to the frontend via the Web Layer, which generates the JWT token and stores it as a user session.

User Profile management flows – Update and Delete
The control flow for updating and deleting a user profile are similar, with differences only at a few steps to either update or delete a user.

The control flow for updating a user's profile is as follows:

1. Client makes PUT request to `/user/:email` with name in request body (for deleting user profile)
2. Request is forwarded by the Web Layer to `userRoute` in the Service Layer which runs the middlewares.
3. Request middleware in `userMiddleware` validates input:
    a. Email parameter must be a valid email
    b. Name field must be present (for updating user)
    c. All fields are sanitized by replacing special characters with their HTML entries
4. If validation fails, 400 error is returned with failure messages
5. If validation succeeds, code extracts email and name from request
6. Another middleware `checkUserExists()` queries the database via `userController` in the Data Access Layer to verify user exists for the email
7. If user not found, 404 error is returned
8. If user found, the database is updated with new name value (when updating a user) or database has user information removed (when deleting a user) via `userController`
9. 200 success response is returned to client upstream via the Web Layer

3.4.3 Question Service and Backend Authorisation

The question service communicates with a MongoDB database hosted on MongoDB Atlas, where all the questions are stored. Express.js is used to build RESTFUL APIs with Node.js, and Mongoose is used to create the schema for questions and connect to the hosted MongoDB

database. To allow for greater flexibility of the structure of the question description, it is stored as a Markdown, which is converted into HTML on the frontend using the remark tool. To populate the database with questions, a seed function was written to create and insert 20 sample questions if the database is empty.

```
const QuestionSchema = new Schema({
  title: {
    type: String,
    required: [true, 'Title is required'],
    trim: true,
    unique: true,
  },
  description: {
    type: String,
    required: [true, 'Description is required'],
    trim: true,
  },
  categories: {
    type: [String],
    required: [true, '1 or more categories are required'],
    trim: true,
  },
  complexity: {
    type: String,
    enum: ['Easy', 'Medium', 'Hard'],
    required: [true, 'Complexity is required'],
    trim: true,
  },
});
```

Diagram 12. Question Schema

While frontend authorisation is in place to prevent rendering protected content to unauthenticated and unauthorised users, backend authorisation is also implemented to prevent access to the questions even if they can access the backend. This is implemented via the use of a verifyRole middleware, which prevents unauthenticated users and unauthorised users that do not have the maintainer role from accessing the questions, and creating, editing or deleting the questions respectively.

```
router.post('/new', verifyRole('maintainer'), createQuestion);
router.get('/', verifyRole(['user', 'maintainer']), getAllQuestions);
router.get('/:id', verifyRole(['user', 'maintainer']), getQuestionById);
router.patch('/:id', verifyRole('maintainer'), updateQuestion);
router.delete('/:id', verifyRole('maintainer'), deleteQuestion);
```

Diagram 13. Use of verifyRole middleware to protect endpoints

From the frontend, API calls are made to the question service and JWT are signed using a secret key and sent in the HTTP Authorization request header as a bearer token, containing the user's role. On the backend, the same secret key is used to verify the JWT and check whether the user has an appropriate role type to access the endpoints. If requests are not made by a

logged in user from the frontend, there will not be a valid authorization request header and a HTTP 401 status code is also returned.

3.4.4 Matching Service

The matching service makes use of a websocket server and a message queue service.

Architecture

Diagram 7 shows the architecture of the matching service. The message queue service maintains multiple message queues to store matching requests for each combination of matching criteria (programming language and question difficulty level), and a notifications queue to queue notifications of successful matches.

The matching server is an Express server which communicates with the message queue service via an AMQP connection and also maintains a websocket connection with the client.

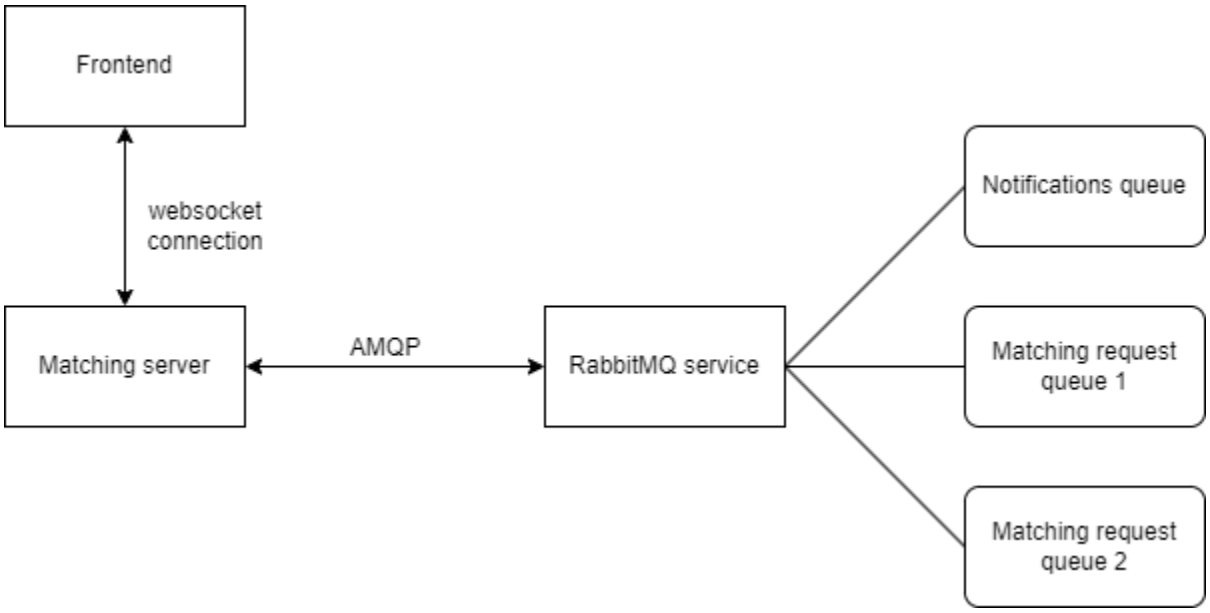


Diagram 14: Architecture of matching service

Flow

In the “matching” page, the user can select his/her desired matching criteria and click on the match button to initiate a match request. A websocket connection is set up between the matching server and the client. Then, the user’s data is queued on the appropriate matching request queue.

The matching server continually checks the matching requests queues. If there are 2 or more user requests queued on any of the queues, the matching server consumes those requests

from the message queue, creates a new websocket room and adds a notification message for each user to the notifications queue.

Simultaneously, the matching server continually checks the notifications queue for successful match notification messages for each user. If a successful match notification is found, the server consumes it and adds the user to the websocket room. The 2 matched users will be added to the same websocket room where they will be able to communicate with each other via the Collaboration Service and the text-based chat.

If no successful match notification message is found after 30 seconds, the server disconnects the websocket connection with the client and the client is given the option to try requesting to match again. The match requests by the user will also be removed from the appropriate message queues.

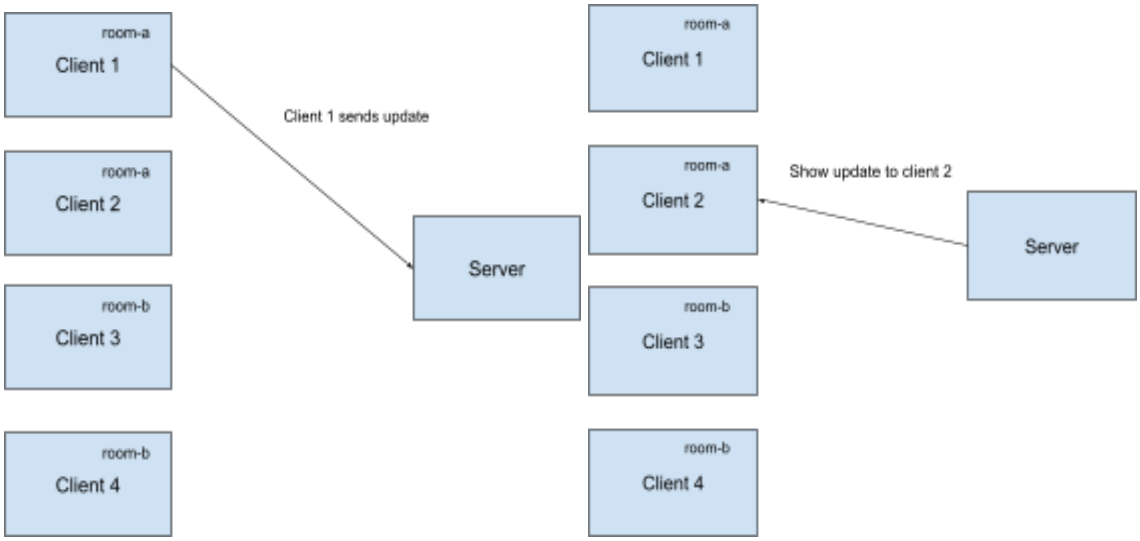
Note: There were issues with deploying a RabbitMQ cluster on Google Kubernetes Service where we are hosting the microservices, so we had to “substitute” the message queues with javascript data structures like arrays.

3.4.5 Collaboration Service

The Collaboration microservice is implemented using [Yjs’s y-websocket], and is integrated with the Matching Service and the Attempts Service to create the collaborative workspace as shown in 3.4.1.5.

Architecture

The code primarily utilises several components such as Collab-Page, WorkSpace and CodeEditor to render the user interfaced. This is backed by y-websocket for shared editing.



Overview: Client 1 types something, Client 2 gets notified of Client 1’s new changes

Control flow

User Matching to Session Initiation

The collaboration session begins when the Matching Service signals a successful match through the match_found event. This triggers the onMatch function in the CollabPage component, initializing the collaboration session. It sets up the required states such as roomId, questionId, and programming language, marking the transition into the active coding environment.

Session Lifecycle Management

The lifecycle of a session encompasses everything from its creation to its termination, including user disconnections. The CollabPage component manages the session state, ensuring a stable and persistent collaboration environment throughout the user interaction.

Real-Time Editing and Synchronization

At the heart of the Collaboration Service is the CodeEditor component, which interfaces with Yjs to enable real-time code editing. Changes made by one user are then reflected on the other user's editor. Users are then allowed to save their attempts using the Attempts Service by clicking on a button that will save whatever is in the editor into an attempt object.

3.4.6 AI Service

The AI microservice allows the users to chat with an AI chatbot via the chat box in the frontend. This service utilizes OpenAI API to access OpenAI models GPT-4. It listens on port 4444 and accepts text inputs via POST request. Upon receiving a POST request, it will asynchronously make an API call to OpenAI and return the query results.

The reason for using the chat box for the frontend was to reduce the need to have many different UI components and to merge similar chatting functions into the same UI element.

3.4.7 Attempts Service

The Attempts service handles tracking and managing coding attempts by users. It allows users to review their past attempts at a question.

Architecture

Similar to the User Service (section 3.4.2), the Attempts Service follows a layered architecture with the same layers. For simplicity and performance sake, the code is stored in the same folders (but in separate files) as the User Service to ensure efficient use of resources.

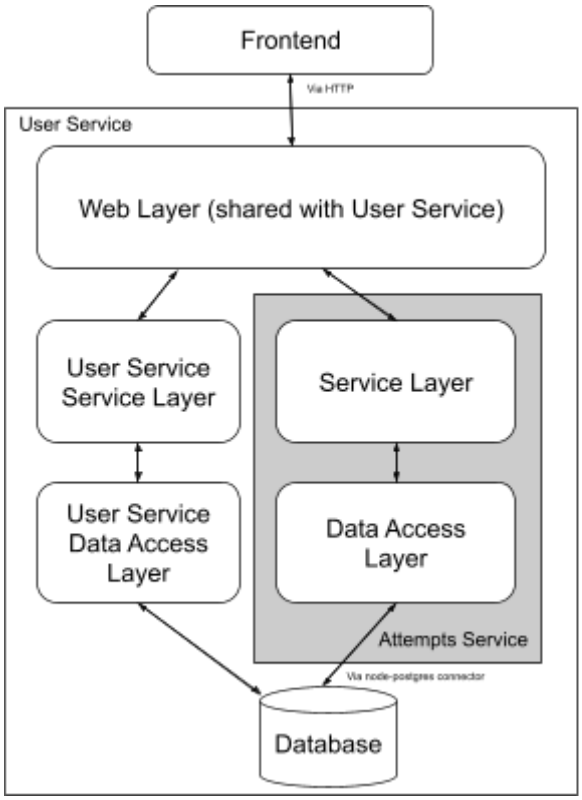


Diagram 15: Architecture diagram of Attempts Service

Design Decisions

Simplified Implementation with User Service

For the initial app, the Attempts Service was implemented as a pseudo-service within the same container in the backend as the User service and utilizes a shared database. This was done to benefit early stage development as it prevents the need for joins across separate databases and reduces overhead of communication across separate services. This thus increased the speed of development for our early application, with some slight technical debt. However, code for the Attempts Service downstream of the Service Layer comprising logic and querying of databases are stored in separate files from the User Service to minimize coupling. This code doesn't interact in any way with code from the User Service. Code for calling the Attempts Service in the frontend is also stored in separate files.

As PeerPrep scales, we intend to separate the Attempts Service into a separate container and service with an API layer for managing communication between services. The separation of files was hence intended as a transitory step towards this.

Decoupled architecture
As the Attempts Service was modeled after the User Service and stored in the same container, it shares the same benefits of a decoupled architecture within the internal service components.

- The Web Layer acts as interface adapters to the external client
- The Routes in the Service Layer encapsulate core business logic
- The Controllers in the Data Access Layer acts as interface to the database
- The Database abstracts data persistence

This separation of concerns provides loose coupling and maintainability within the service itself.

Attempts Schema

```
CREATE TABLE IF NOT EXISTS attempts (
    attempt_id INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    email VARCHAR(50),
    question_id VARCHAR(30),
    question_title VARCHAR(30),
    attempt_datetime TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    code VARCHAR(500)
);
```

Diagram 16: Schema for attempts saved

Control Flow

Attempt Submission Flow

1. User attempts the question and clicks on the save button in the frontend
2. Frontend calls `POST /attempts/:email/:id` with question details `question_id` and `question_title` and the code in the request body, which is forwarded by the Web Layer to the attempts route in the Service Layer.
3. The route handler validates input by calling middlewares to:
    a. Ensure that email is valid
    b. Check that required fields are present
4. If validation fails, a 400 error is returned with failure messages
5. If validation succeeds, it further calls `attemptsController::addAttempt` in the Data Access Layer which inserts the attempt into the database.
6. A status code of 200 is returned if no error is thrown.

View Attempt Flow

1. When a specific attempt is clicked from the list of attempts on the profile page, the frontend retrieves the email from the JWT token and calls `GET`

`/attempts/:email/:id`, which is forwarded by the Web Layer to the attempts route in the Service Layer.
2. The Service Layer queries the Database using `attemptsController::getAttemptById` inside the Data Access Layer to retrieve the attempt information.
3. This information is used to verify that the provided email matches the email of the attempt in the Database.
    a. If it doesn't match, a 401 Unauthorized error is returned
    b. If it matches, the attempt is returned upstream to the frontend via the Web Layer with a 200 status code.

3.5 Code Organisation

Repository Structure
Our repository has two main directories - a frontend directory, which contains a NextJS application, and a backend directory. Within the backend directory, we have a directory for each microservice, which contains the respective server applications and Dockerfiles needed to containerize each service.

For ease of testing and development, we can run each microservice independently in its own directory. There is also a docker-compose configuration file in the backend directory to help run the backend services with docker-compose.

3.6 Deployment process

3.6.1 Containerization
Each microservice and the frontend was containerized using Docker. A docker-compose configuration file has also been added in the backend directory to facilitate local testing.

We decided to containerize our applications to increase portability, making the process of deploying it easier to manage.

3.6.2 Cloud deployment
Our application is deployed on Google Kubernetes Engine (GKE), which is a managed Kubernetes service for containers and container clusters running on Google Cloud infrastructure.

We decided to deploy it using Kubernetes because it is a powerful and flexible container orchestration platform, helping to automate the deployment, scaling, and management of our containerized applications. It is also cloud-agnostic, meaning it can run on various cloud

providers (such as AWS, Azure, Google Cloud) as well as on on-premises infrastructure. This portability allows organizations to avoid vendor lock-in and choose the best environment for their needs. We ended up hosting it on GKE due to its gentler learning curve as compared to AWS and Azure.

3.6.3 API Gateway Implementation

We also implemented an API gateway using an NGINX ingress controller, which routes requests from the frontend to the appropriate microservices.

The implementation of an API gateway is beneficial due to the centralization and simplification of external access to services. This makes it easier to implement scalability and security improvements like load balancing and network security protocols.

4. Suggestions for future improvements

4.1 General Features across Services

- Gamification features like points, badges and leaderboards can motivate participation further in our application.
- Tagging questions by subject, and highlighting popular questions by top companies could increase relevance to certain subsets of users and allow efficient use of our application

4.2 User Service

- User review feature to allow users to rate and review other users could encourage users to improve collaboration experience. It can also be used to identify experienced users that could become mentors.

4.3 Code Execution Service: Code compilation and execution

Currently, users cannot compile and run the code that they have collaboratively worked on during a session.

To make this possible, an extra service such as an Execution Service can be set up, and integrated into the collaborative session.

This microservice would receive the code from the frontend of the session, compile/interpret it accordingly with a suitable compiler/interpreter.

Once the code execution is completed, the results output and logs will be sent back to the frontend. This can either be successful code outputs or any compilation or runtime errors.

5. Project Management

Design Process
The project followed an iterative waterfall approach for design and development.

Requirements Gathering
Initial requirements were gathered and categorized into must-have and nice-to-have features based on priority.

Must Have Features
The must-have features are:
User Service - For user profile management and authentication
Matching Service - To match users for practicing on questions
Question Service - To maintain a repository of questions
Collaboration Service - For real-time code collaboration

Nice to Have Features
The nice-to-have features are:
N1: Communication - Text-based chat service in the collaboration page
N2: Attempt History - To track user's past question attempts
N4: Enhanced Question Management - For organizing and searching for questions
N5: Improved Collaboration - With code editing and highlighting
N7: AI Assistance - Using ChatGPT to provide assistance when practising
N9: Cloud Deployment - Deployment of application to Google Kubernetes Service (GKE)
N11: Implementation of API gateway - with NGINX ingress controller

Version Control
GitHub was used for source control with feature branch workflow:

- Main branch - Production deployment
- Feature branches - For development of features

After each feature is developed, integration testing is performed by the developer and a reviewer before merging to the main master branch.

Github Issues and Projects

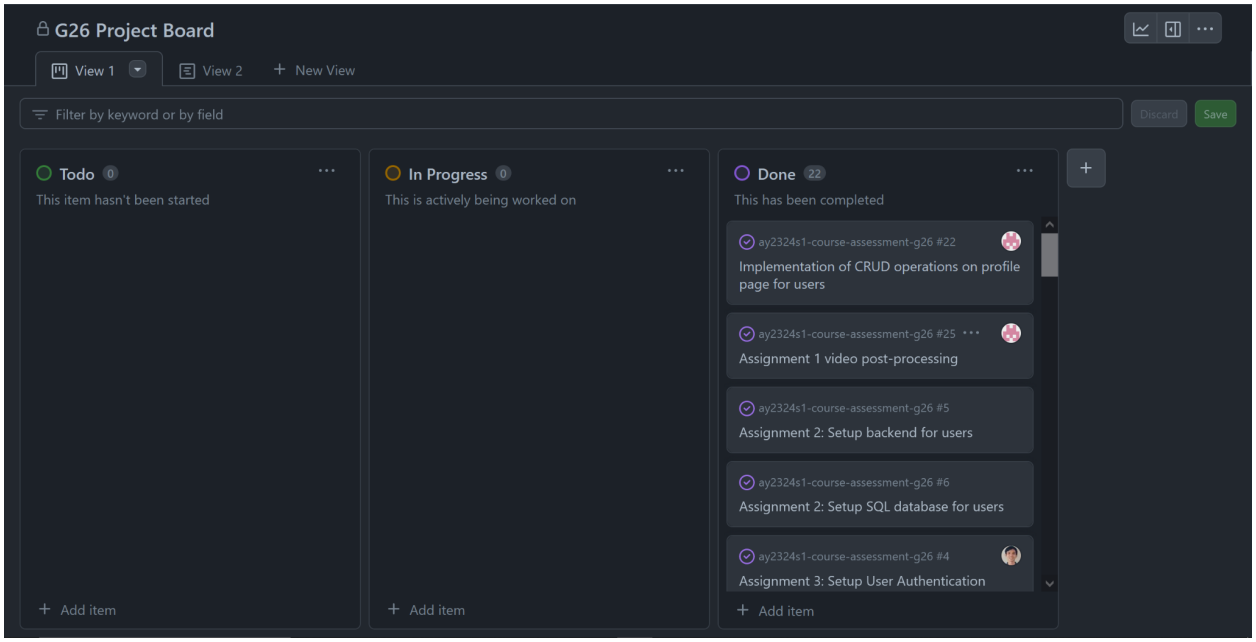


Diagram 17. Use of Github Issues and Projects

Github Issues are created according to the assignment or feature being developed. To further improve the tracking of our progress, Github Projects was used to easily change the status of an issue between the different states.

6. Reflections

6.1 Time management

During the course of our project, we faced some challenges in managing our timelines effectively. A key issue we encountered was the consistent need to push back our deadlines, primarily due to unforeseen technical difficulties. As a team new to web development, resolving issues was a time-consuming learning process.

Key Challenges:

1. **Unexpected Technical Issues:** We often stumbled upon technical roadblocks that were not anticipated in the initial planning phase. These included issues related to integration, compatibility, and performance, which were complex and required extensive research and troubleshooting.
2. **Learning Curve:** Given our lack of experience in web development, each technical hurdle presented a steep learning curve. The time required to understand and resolve these issues significantly exceeded our initial estimates.
3. **Underestimation of Task Complexity:** Many tasks turned out to be more complex than initially anticipated. This underestimation led to a cascading effect on the project timeline.

Adopted Strategies and Their Effectiveness:

1. **Incremental Learning and Application:** We adopted an approach of learning and immediately applying new knowledge to tackle technical issues. This strategy, while effective in resolving issues, was time-intensive.
2. **Regular Team Meetings and Status Checks:** Frequent team meetings were held to assess progress and re-align our goals according to the current status. These meetings helped in identifying delays early on, but they also highlighted the need for more realistic timeline planning.

Lessons Learned:

1. **Early Start and Buffer Time Allocation:** Reflecting on our experiences, a key takeaway is the importance of starting early. An earlier start would have allowed us more time to deal with unexpected challenges without impacting the overall project deadline.
2. **Realistic Time Estimation:** Allocate buffer times that account for potential technical difficulties and learning requirements.

6.2 Importance of effective communication

One of the pivotal factors that influenced the success and efficiency of our project was the level of communication within the team. Initially, we encountered several instances where ineffective communication led to unproductive work and a lack of clarity on task ownership and progress.

Lessons Learned:

1. **Improved Regular Updates:** In future projects, we plan to establish a more rigorous routine for progress updates. This could involve daily stand-ups or regular check-ins, ensuring that everyone is on the same page.
2. **Effective Use of Communication Tools:** Leveraging communication tools more effectively, such as shared calendars, ensuring notifications are turned on in messaging apps, and project management software, can help maintain a steady flow of information and keep everyone informed.

7. Contributions

7.1 Individual Contributions

Name	Contributions
Choon Siong	User Service ● Implemented and integrated user registration/login and user profile management flows on the backend and frontend with Google OAuth integration ● Designed and developed REST APIs for user profile management ● Implemented input validation and sanitization in requests ● Designed and implemented user profile page and windows for confirmation when user is deleted ● Implemented authorisation routes on the profile page ● Setup PostgreSQL database and deployed to Google Cloud SQL Attempts History/Attempts Service ● Implemented backend service within User Service to handle API requests for adding and retrieving attempts, covering request validation, logic and database storage ● Implemented methods to store, fetch and display attempts data on the frontend by integrating with backend ● Designed and implemented table of attempts component on profile page, and styled the attempts viewer
Felix	Frontend ● Setup Next.js application and created basic user interface with homepage and navigation bar ● Created components related to questions Authentication and Authorisation ● Setup NextAuth.js with Google Provider ● Create higher-order component for frontend authorisation of questions ● Used JWT for backend authorisation

	Question Service ● Setup MongoDB database on MongoDB atlas ● Designed and developed REST APIs for question management ● Generated markdown representation for sample questions
Qi An	Frontend ● Set up coding/collab page with various components ● Styled and improved chat box UI Collaboration Service ● Setup y-websocket server for collaborative editing ● Set up client connections to collaboration server ● Create Code Editor component ● Added syntax highlighting based on matched language Frontend Integration for Attempts Service ● Integrated Attempts Service into collaborative workspace ● Added code viewing in an editor for an attempt
Chin Kiat	Frontend ● Set up matching page Matching service ● Set up matching service ● Helped with integration with collaboration service Deployment ● Containerized microservices and frontend ● Deployed application on GKE ● Implemented NGINX ingress controller
Wing Ho	Frontend ● Edited the components needed for the AI service ChatBot AI Service ● Set up the AI ChatBot service Deployment ● Created the DockerFile for some microservice

7.2 Sub-group Contributions

Sub-group	**Contributions**
1 **(**Chin Kiat, Qi An**)**	**Communication:** ● Implemented text-based chat service for participants in

	collaborative session ● Added chatbox-like styling **Collaboration Service Enhancement** ● Enhance code editor by including code formatting and syntax highlighting for languages that users are matched for **Cloud Deployment** ● Containerized microservices and frontend with Docker ● Deployed microservices on Google Kubernetes Service (GKE) ● Deployed frontend on GKE as well **Implementation of API Gateway** ● Added an NGINX ingress controller which routes requests to the appropriate services
2 **(**Felix, Wing Ho, Choon Siong**)**	**Question Service Enhancement** ● Added question management features such as filtering by complexity, searching by title and category to make it easier to find questions **Generative AI Service** ● Added a tab in the chat window to allow the user to communicate with ChatGPT for help **Attempts Service/Attempt History** ● Attempt History is a feature that allows users to save their attempts while coding and retrieve it for review purposes later ● It consists of backend logic and APIs for recording and retrieving question attempts, with request validation for API calls and integration with User Service for further validation ● There is a save button during a coding session that allows users' to save their code, a table on profile page to view the list of saved attempts, and an attempt page after a saved attempt is clicked.