CS3219 Project

PeerPrep Final Report

Group G37

GOH YONG JING (A0230206Y)

KONG HEYI (A0238046E)

LEE WEI CHONG STEFAN (A0229939J)

RANDALL NG HONG RONG (A0235128J)

TAN WEI ZHE (A0233610W)

Table of Contents

1. Background and Purpose

As the tech industry evolves, the value of technical interviews are increasingly emphasised as a means to evaluate applicants. These interviews require candidates to demonstrate both their problem solving skills as well as extensive understanding about computer data structures and algorithms.

Our product PeerPrep is a platform specifically designed to help Computer Science students prepare for such technical interviews. With PeerPrep, you can collaborate with your peers in real-time and tackle coding questions together. Whether you are just starting out or are diving deep into more complex challenges, we want to help ensure that your preparation is comprehensive.

2. Individual / Sub-group Contributions

2.1 Individual Contributions

Goh Yong Jing	Technical: ● Questions Service ○ Set up the questions server API endpoints (create, read, update, delete questions) ○ Set up MongoDB database to store questions ● Matching Service ○ Built Frontend to allow the user to initiate finding a match, cancel match requests ○ Provide feedback to user for time elapsed and match not found after timeout ○ Set up web socket connections between the Frontend and the Auth Service to send match requests and responses ○ Set up AMQP connections between RabbitMQ, Auth Service and the Matching Service to send match requests and responses ○ Handle server side match queue and cancel match logic on the Matching Service ○ Containerize Matching Service using Docker ● Sync Questions Service (Assignment 6) ○ Build Frontend buttons to allow users to fetch the question of the day and the 20 sample questions from Leetcode ○ Set up API endpoints for serverless functions to fetch the question of the day and the 20 sample questions from Leetcode, as well as send requests to update the Questions Service Non-technical: ● Added README (Assignment 5 and 6) ● Wrote the Assignment 5 PDF document for the Matching

	Service ● Recorded the demo video (Assignment 6)
Kong Heyi	Technical: ● Single Page Application (Assignment 1) ○ Set up a local server to host the questions ○ Wrote the Frontend UI components to create, read, update and delete questions ● User Service and Questions Service (Assignment 2) ○ Integrate the Frontend with both microservices using HTTP API endpoints ○ Show error messages to the user when duplicates (e.g., duplicated usernames, duplicated question titles) are detected ○ Add a sign-up form to create new users ○ Show the current user's username and display name ○ Allow users to update their profile (e.g., they can change their display names) ○ Allow users to delete their account ● Authentication and authorisation (Assignment 3) ○ Integrate with the Auth Service to persist user sessions (e.g., when the user refreshes the page, they are still logged in) ○ If the current user is unauthenticated, redirect them to the login page ○ If the current user is already authenticated, redirect them away from the login / signup page and onto the questions page ○ Only "admin" users can add / update / delete questions (i.e., "basic" users can only view the questions) ● Collaboration Service ○ Show a list of Easy / Medium / Hard questions on the collaboration page for both users to work on together Non-technical: ● Recorded demo video for Assignment 1 ● Drew architecture diagrams for final report ● Reorganised structure of final report
Lee Wei Chong Stefan	Technical: ● Authentication and authorisation (Assignment 3) ○ Persistent user sessions using cookies (i.e., the user remains logged in even after refreshing the page) ○ Check that the user is authenticated and authorised before forwarding the requests to the appropriate microservice (e.g., User Service, Questions Service) ○ Check that the user is authorised to add / update / delete questions before forwarding the requests to the Questions Service ● Sync Question Service (Assignment 6) ○ Containerize the Sync Question Service using Docker Non-technical: ● Review pull requests from other team members

	• Recorded demo video for Assignment 3 • Edited all the demo videos to ensure that they are less than 3 minutes long • Standardise the READMEs for all the assignments

Randall Ng Hong Rong	Technical:
	• Containerization (Assignment 4)
	  ○ Created Docker files for each microservice – Frontend, Auth Service, User Service, Questions Service
	  ○ Created Docker compose files to build each Docker image for each microservice more efficiently
	  ○ Incorporated Docker secrets to help protect sensitive information in the containers
	• Collaboration Service
	  ○ Added a code editor to the collaboration page
	  ○ Connect matched users to the Collaboration server using web sockets to enable concurrent code editing
	
	Non-technical:
	• Added README for Assignment 4 on how to run docker compose with secrets
	• Recorded demo video (Assignment 4)

Tan Wei Zhe	Technical:
	• User Service (Assignment 2)
	  ○ Set up PostgreSQL database to store user information
	  ○ Set up the users server API endpoints (Add, retrieve, edit and delete users)
	• Collaboration Service
	  ○ Handle server logic for synchronising two clients' collaboration page when user types code
	  ○ Handle edge cases when two users are collaborating (e.g., when one user refreshes the page / disconnects and reconnects back, restore the code that was in the previously in code editor)
	  ○ Handle chat functionality between two users
	  ○ Containerize the Collaboration Service using Docker
	
	Non-technical:
	• Recorded demo video (Assignment 2)
	• Organised weekly team meetings and delegated work to each team member for that particular week

2.2 Sub-group Contributions

Sub-group 1 :: Lee Wei Chong Stefan :: Randall Ng Hong Rong	N2: History Service • Set up another PostgreSQL database to store the attempts of each user • Set up the history server API endpoints • Users can save their code to the History Service at any time during the collaboration session

	● User can view their past attempts in the History page in the front-end ○ Each history entry includes the question title, the user's code with syntax highlighting and the timestamp of the attempt N3: Code execution ● Integrate with Judge0 API to execute code in a secure sandboxed environment ● Code execution is supported for multiple programming languages including JavaScript, Python, C/C++ and Java ● Display code output beneath the code editor for faster development N5: Code formatting and syntax highlighting ● Integrated an opinionated code formatter Prettier to indent and format the user's code to ease readability ● Added syntax highlighting for popular programming languages in the collaboration code editor N11: API Gateway ● The Auth server check that the user is appropriately authenticated and authorised before routing the requests to the correct microservice
Sub-group 2 :: Goh Yong Jing :: Kong Heyi :: Tan Wei Zhe	N1: Communication ● Added text-based chat functionality to the collaboration page ● Chat bubbles are positioned based on the roles of sender / receiver (e.g., if Alice sends a message to Bob, Alice will see her message on the right hand side of the chat window whilst Bob will see her message on the left hand side and they are coloured differently) ● Display the timestamp and sender's username beside each message ● Typing status indicator to signify when the other user is typing a message ● Able to send code blocks in chat messages N4: Enhanced Question Service ● Search for questions by title ● Sort questions by title, complexity level ● Filter questions by tags (i.e., question category), complexity level, attempted status N9: Deployment ● Deployed the PeerPrep application using an Amazon Elastic Compute Cloud (EC2) instance ● Transferred environment variables and API keys securely to the production environment using `scp` ● Prepared scripts and config files to ease DevOps of deployment N10: Scalability

	• Deploy each microservice on separate Auto Scaling groups to increase number of launched EC2 instances based on CPU utilisation rate of current instances • Deploy each microservice with Elastic Load Balancer (ELB) to distribute the app traffic between EC2 instances

3. Project Requirements

3.1 Must-have features

M1: User Service	**Priority**
Functional	
M1.F1: Allow users to create a new account with username, display name and password	high
M1.F2: Allow users to view their account details	high
M1.F3: Allow users to update their own account details (e.g., change display name and password)	high
M1.F4: Allow users to delete their own account	high
M1.F5: Ensure that usernames are unique	high
M1.F6: Allow users to log in and out of their account	medium
M1.F7: User sessions are persistent (e.g., when the user refreshes the page / closes and reopens the tab, they should still remain logged in)	medium
M1.F8: Each user is assigned either the “basic” role or the “admin” role (new user accounts will automatically be assigned the “basic” role)	medium
Non-Functional	
M1.N1: Handle up to 10 concurrent users	medium
M1.N2: Hash and salt user passwords before storing them in the database (i.e., do not store plaintext passwords)	medium

M2: Matching Service	**Priority**
Functional	
M2.F1: Allow users to find a match based on their chosen question complexity	high
M2.F2: Allow users to cancel the matching process before they get a match	high

M2.F3: If a user fails to get a match within 20 seconds, prompt them if they would like to retry	medium
M2.F4: Display a timer to the users, indicating how long they have been waiting for a match	medium
Non-Functional	
M2.N1: Handle up to 10 concurrent users matching amongst themselves	medium

M3: Questions Service	**Priority**
Functional	
M3.F1: Display a list of questions to the user, along with the question title, category, and complexity.	high
M3.F2: When the user selects a question, show the full question description	high
M3.F3: Allow only maintainers (i.e., users with the “admin” role) to add new questions	high
M3.F4: Allow only maintainers to update the question details	high
M3.F5: Allow only maintainers to delete questions	high
M3.F6: Ensure that all question titles are unique	medium
Non-Functional	
M3.N1: Handle storing at least 50 questions	medium

M4: Collaboration Service	**Priority**
Functional	
M4.F1: Provide a code editor for users to write code	high
M4.F2: Synchronise the updates in the code editor by either user	high
M4.F3: Allow either user to end the collaboration session at any time	high
M4.F4: Display a list of questions that fulfil the matching criterion (see M2.F1)	high
M4.F5: Pause the collaboration session if one party disconnects from the room (e.g., they accidentally refreshed the page)	medium
M4.F6: Resume the collaboration session when both users are re-connected	medium
Non-Functional	
M4.N1: Synchronise the updates in the code editor within 3 seconds of any	medium

change from either user	

M5: Basic UI for user interaction	**Priority**
Non-Functional	
M5.N1: Display text in English language	high
M5.N2: Design the UI to be available in light mode	low

M6: Deployment on local machine	**Priority**
Non-Functional	
M6.N1: Support deployment on a local machine such that it is accessible by other devices in the same Local Area Network (LAN)	medium

3.2 Nice-to-have features

N1: Communication	**Priority**
Functional	
N1.F1: Allow users to chat via text during collaboration	medium
N1.F2: Show the display name of the other user for incoming chat messages	low
N1.F3: Show the timestamp of incoming chat messages	low
N1.F4: Differentiate incoming and outgoing messages in the chat window	low
N1.F5: Indicate that the other user is typing	low
Non-Functional	
N1.N1: Chat messages should be delivered to the other party within 3 seconds	low

N2: History Service	**Priority**
Functional	
N2.F1: Allow users to save their code and the question attempted	medium
N2.F2: Allow users to view their saved code alongside the question and timestamp of the attempt	medium
N2.F3: View the history entries sorted by most recent attempt first	low

N2.F4: Add language-specific syntax highlighting for each attempt (see N5.F2)	low
N2.F5: If the attempted question has been deleted, the user's attempt for that question should still remain	low

N3: Code execution	**Priority**
Functional	
N3.F1: Support code execution for popular programming languages like JavaScript, TypeScript, Python, C/C++ and Java	medium
N3.F2: Display the results of the code execution to the user	medium
N3.F3: Limit the maximum amount of time and memory the user's code is allowed to consume	low

N4: Enhanced Question Service	**Priority**
Functional	
N4.F1: Allow users to sort questions by title and complexity	medium
N4.F2: Allow users to filter questions by category, complexity level and attempted status	medium
N4.F3: Allow users to search for questions based on their titles	medium

N5: Enhanced code editor	**Priority**
Functional	
N5.F1: Provide code formatting for JavaScript and TypeScript	medium
N5.F2: Provide syntax highlighting for popular programming languages like Python, C/C++ and Java	medium

N9: App deployment	**Priority**
Functional	
N9.F1: Deploy the application to the cloud so that it is publicly accessible on the internet.	medium
Non-Functional	
N9.N1: Be able to run on both Chrome and Firefox on a computer	low

N10: Scalability	Priority
Non-Functional	
N10.N1: Ensure that each microservice is able to scale independently of each other	low

N11: API Gateway	Priority
Functional	
N11.F1: Provide an API gateway that routes requests to the correct microservice	medium

4. Developer Documentation

4.1 Architecture

4.1.1 Components

We used Domain Driven Design to decide on the different microservices. The main differentiating factor of PeerPrep is the smooth integration of multiple common services together to provide a coherent service to the user to solve coding questions with a friend.

- Core sub-domain
    - User Interface (Frontend): Serves as the primary interface for user interactions.
    - Collaboration Service
        - Shared code editor with syntax highlighting and code formatting
        - Chat feature
    - Questions Service
    - Matching Service
- Supporting sub-domain
    - User Service
    - Auth Service
    - Collaboration Service
        - Code Execution
    - History Service
- Generic sub-domain
    - Deployment, load balancing, scalability

Based on our chosen sub-domains, we can split up the domains into different microservices and also decide if we want to develop these features in-house or outsource them. Our final product consists of 6 microservices, each of which is containerized separately.

- Auth Service: Handles user authentication and authorisation. It also acts as an API gateway for all backend requests which will check that the user is appropriately authenticated and authorised before routing the requests to the appropriate microservice.
- User Service: Manages user profiles and identity-related functionalities.
- Questions Service: Handles the operations related to questions, including creating, reading, updating and deleting.
- Matching Service: Matches two users based on some criteria to collaborate together.
- Collaboration Service: Facilitates real-time collaboration between users with chat functionality and an auto syncing code editor.
- History Service: Manages users' historical data, such as past questions attempted and the corresponding code written.

The Architecture Diagram below provides a high-level design overview of PeerPrep:

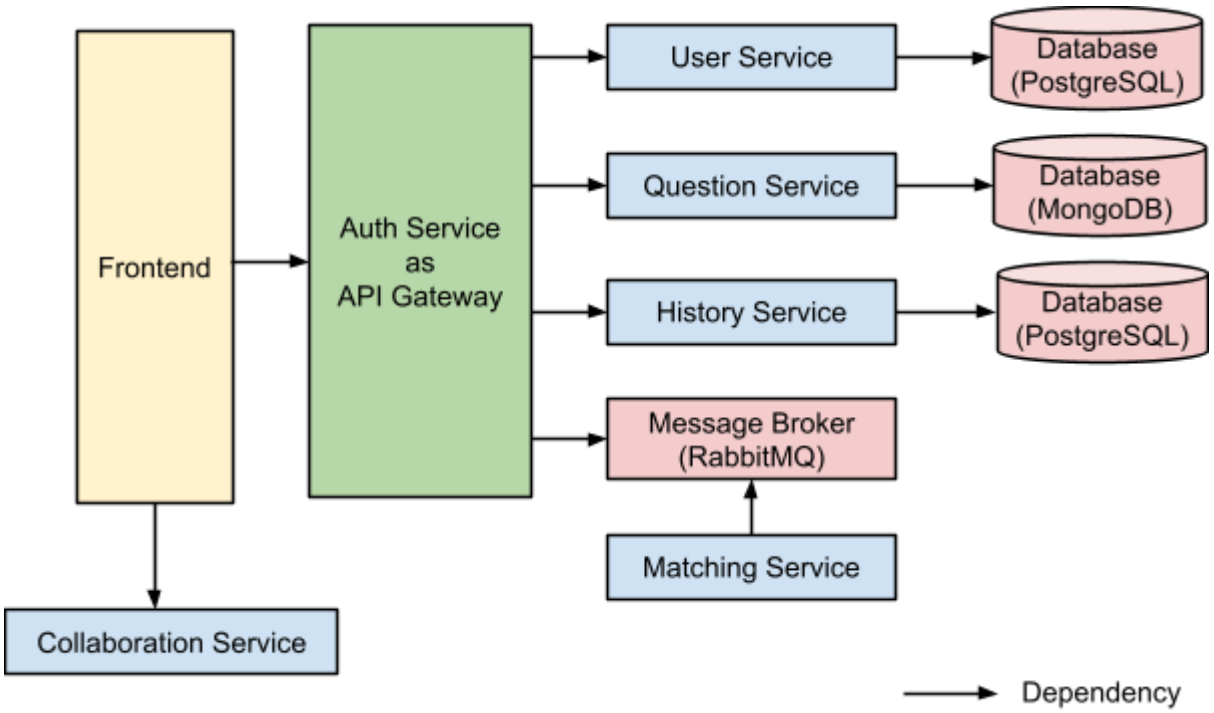

Figure 4.1.1a

4.1.2 General Design Decisions

We made a series of design decisions and used multiple best-practice design patterns to ensure the app's scalability and robustness under diverse usage circumstances:

- Microservices Architecture (see 4.3)

    - Each service is a self-contained module, allowing developers to work on one part of the application without clashing with others.
    - Each microservice can be scaled up or down independently based on the demand.
- Separate Databases
    - Each microservice (i.e., User, Auth, Questions and History) has its own separate database. This gives each microservice greater autonomy over how it structures and manages its own data – different microservices can choose to adopt different database systems (e.g., PostgreSQL, MongoDB) based on their needs.
- API Gateway (see 4.3.1)
    - The Auth Service acts as a single entry point that redirects requests to appropriate microservices. This improves security as the Auth server can verify that the user is indeed authenticated and authorised to access this internal service.
- REST API
    - We standardised the naming conventions of our API endpoints for consistency and ease of concurrent development:
        - Auth Service: `/api/auth`
        - Questions Service: `/api/questions`
        - User Service: `/api/users`
        - History Service: `/api/history`

4.1.3 Deployment

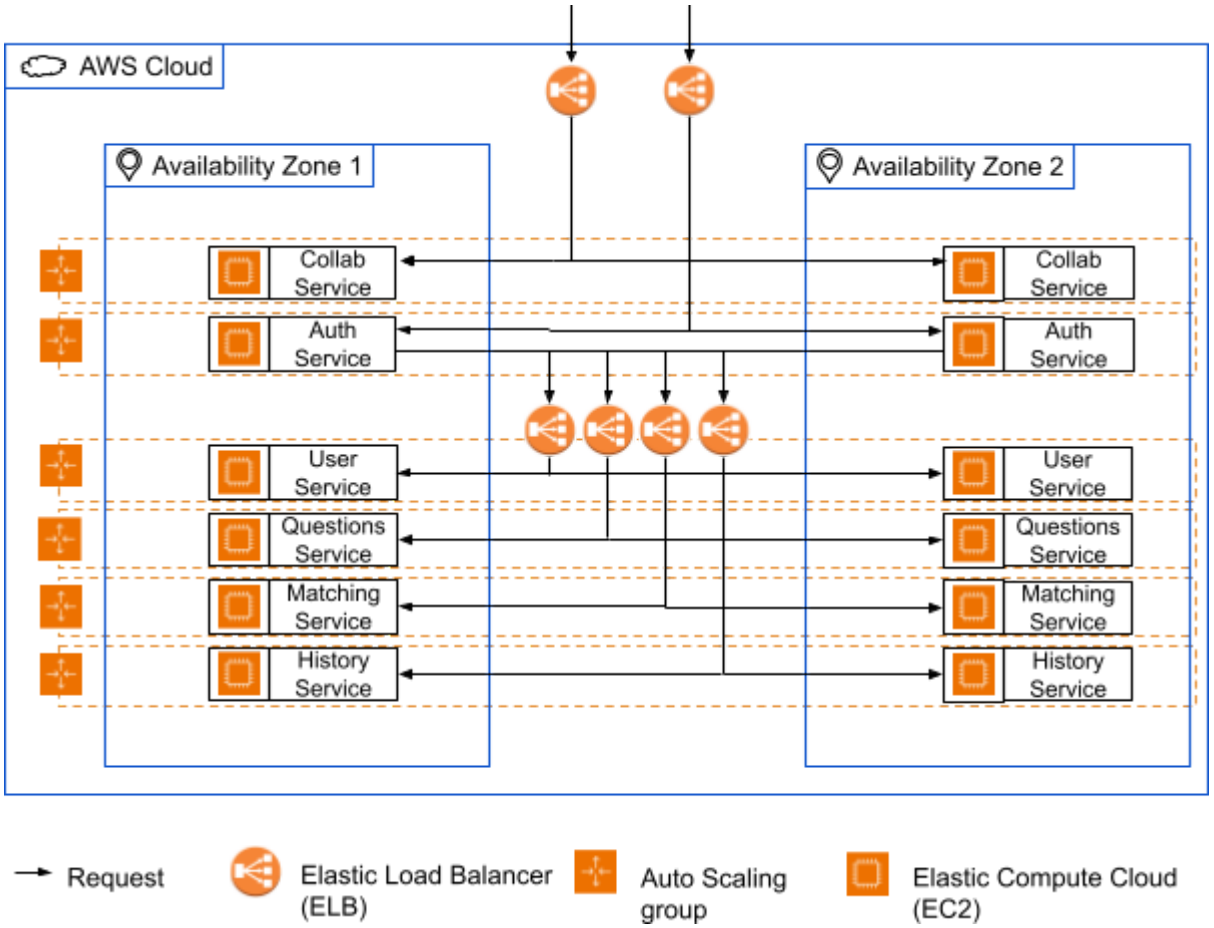

Figure 4.1.3a

AWS Elastic Compute Cloud (EC2)

- Each microservice is on a separate instance and can be scaled independently from each other

Auto Scaling group

- Automatically launches new EC2 instances if the CPU utilisation rate of current instances exceed 70%
- EC2 instances are launched in different isolated availability zones (AZ) to minimise chances of outage of services due to catastrophic events affecting a single AZ

AWS Elastic Load Balancer (ELB)

- Serves as the service's API gateway. If an EC2 instance goes down and is replaced with another instance with a different IP address, other microservices continue to send requests to the same API gateway with no additional configuration needed.
- Evenly spreads the app traffic to different servers handling the same microservice.

4.2 Frontend

4.2.1 Architecture

The Frontend component of PeerPrep is the central interface through which users interact with the platform's diverse functionalities. Its primary objective is to offer an intuitive, responsive, and seamless user experience when interacting with various underlying microservices.

The Frontend component is built using React, TypeScript and Material UI, and it encompasses several key pages, each tailored for the user interactions with a specific microservice. The architectural diagram below gives an overview of the components in the Frontend:

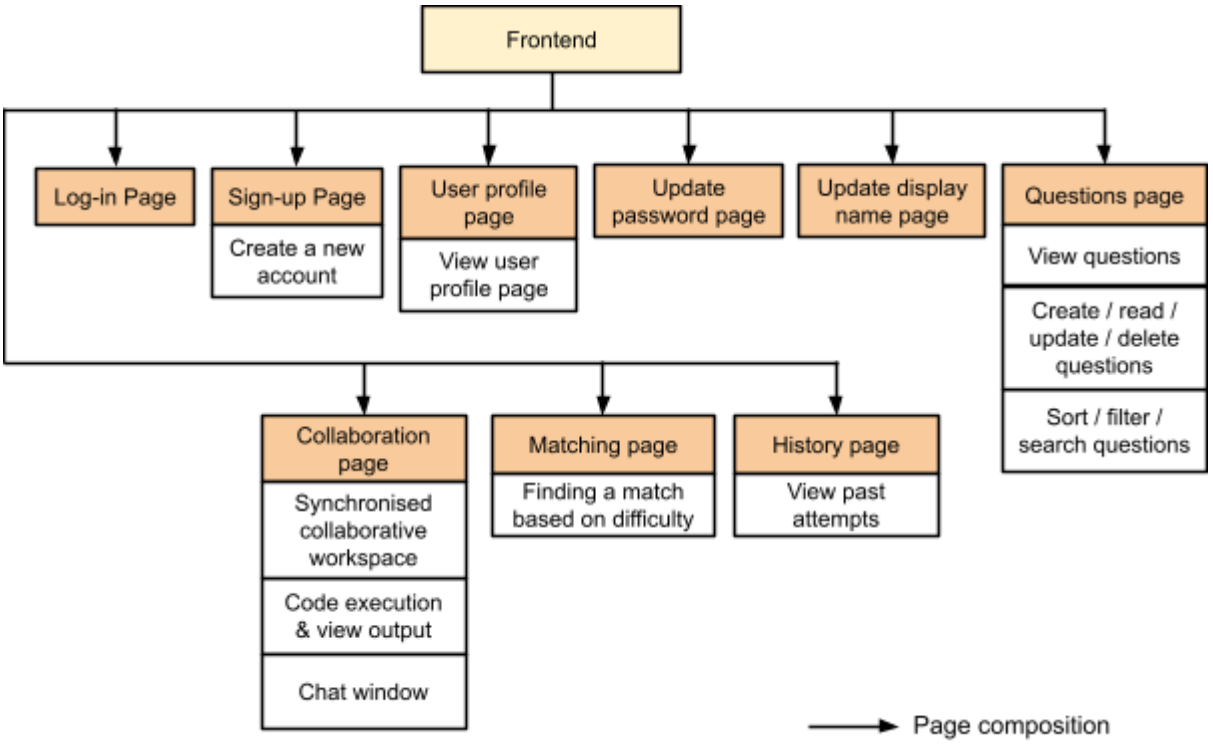

Figure 4.2.1a

4.2.2 Design Principles

- Ease of Navigability: We implemented a toolbar that is present on every page to give users easy access to all the main features of PeerPrep with minimal clicks.
- Separation of Concerns: Adhering to the principle of separation of concerns, each microservice has its own dedicated Frontend page. This ensures a more organised website and codebase, enhancing both the user experience and maintainability of the project.

4.2.3 Components

User Component

Features

- View user profile: Users can view their own profiles which includes their username (unique and non-changeable), display name and role (either “basic” / “admin”).
- Updating user profile: Users can update their display name and password.
    - Users must provide the correct old password before they can change to a new password
- Deleting their own account: Users can also choose to delete their PeerPrep account once logged in.

Auth Component

Features

- Access Control: The Auth Component automatically redirects all unauthenticated users to the log-in page. This ensures that only authenticated users gain access to the platform’s features.
- Session Management: Upon successful authentication, the Auth Component maintains user sessions across page reloads and user navigations.
- Logging out: When the user logs out from PeerPrep, the auth information in the Frontend is cleared for security purposes.

Implementation

- Upon successful log-in, instead of manually passing down the user’s auth information (e.g., username, display name, role) from component to component (which is prone to human error), the Auth Component populates the user context object with the user’s profile information to make it available in any Frontend component. This reduces potential bugs and clutter due to ‘prop drilling’, and speeds up the development process. The following diagram illustrates a sample use case of React Context:

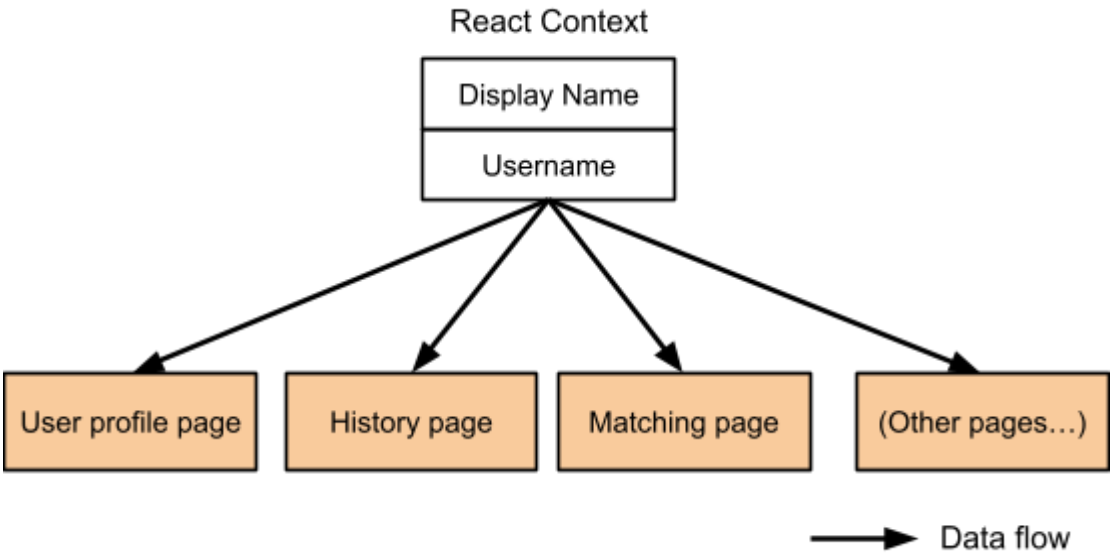

Figure 4.2.3a

Question Component

Features

- Question exploration: Users can view all the questions alongside their title, categories and complexity level. Clicking on the question title shows the full description of the question.
- Searching for questions: Users can search for questions based on the question's title.
- Filtering questions: Users can filter the questions based on category, attempted status and complexity level
- Sorting questions: Users can sort the questions based on the question title or complexity level
- Question management ("admin" users only): "admin" users can add, update and delete questions

Implementation

- Redux Integration: The Question Component uses Redux for state management which allows for more efficient handling of user interactions and the state changes that result from it. The diagram below gives an overview of data flow using Redux:

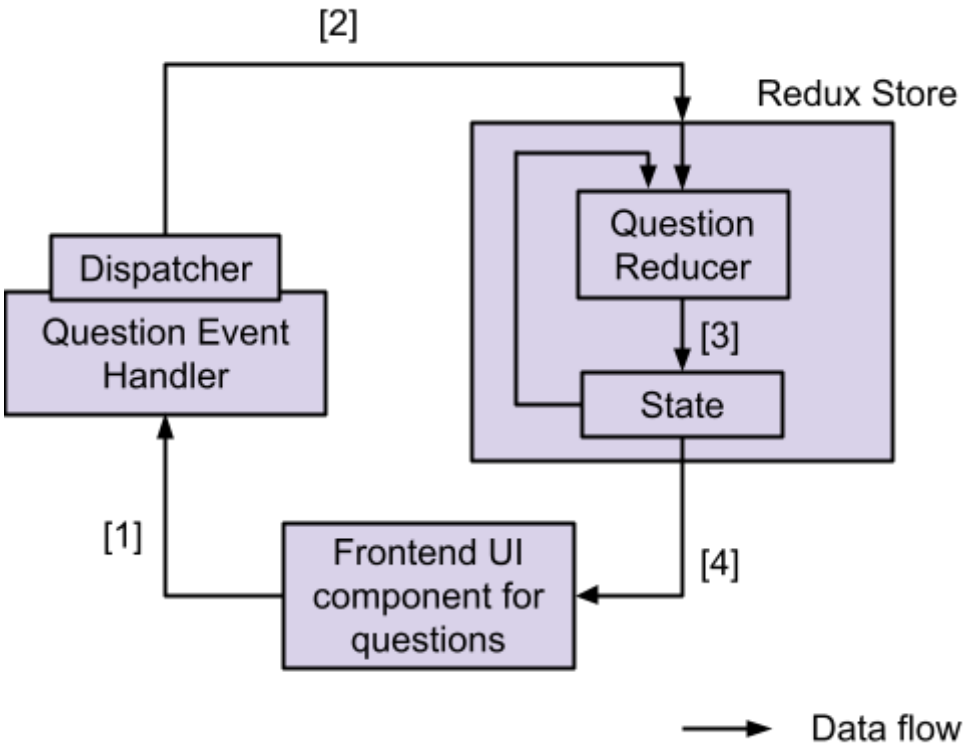

Figure 4.2.3b

1. An event related to the questions component occurs (e.g., the user clicks a button)
2. The question event handler dispatches an action (e.g., delete question) with the appropriate payload (e.g., the question identifier) to the Redux store
3. API requests are made to the Questions Service to update the questions database. On success, we update the Redux store as appropriate (e.g., the new Redux store now has one less question)
4. The Frontend subscribes to the Redux store. Whenever the Redux store changes, the Frontend automatically updates the UI as needed (e.g., the Frontend displays one less question to the user)

History Component

Features

- Viewing past attempts: Users can view their history of past attempts for all collaboration sessions. Each entry includes the question title, the code written during the collaboration session (with syntax highlighting), and the timestamp of the attempt
    - The entries are sorted by the most recent attempt first. Users can view the full timeline of their past attempts to gain insights into their learning journey.
- Persistence of attempts: In cases where the attempted question has been deleted, the History Service will retain the attempt and display to the user that the particular question has been deleted.

Implementation

- Syntax highlighting: The History Service stores the programing language of the attempt to allow the Frontend History Component to apply the correct syntax highlighting for each attempt using highlight.js
- Redux Integration: Given that both the Question Component and the History Component have a dependency on each other (the Question Component needs the list of history attempts to correctly filter by attempted status and the History Component needs the list of questions to show the question title next to each attempt), we decided to bring in Redux to manage the state of the History Component too. For brevity, the Redux data flow diagram for the History Component is omitted as it is similar to that of Question Component.

Matching Service Component

Features

- Selecting complexity level: Users can select their preferred complexity level before initiating the “Find Match” process. This allows the system to tailor match users that have similar skill levels with each other.
- Timer and timeout: The Frontend displays a timer to the user indicating the amount of time that has elapsed since the “Find Match” process began. If a match is not found within 20 seconds, the Frontend will automatically cancel the match request and prompt the user if they would like to try again.
- User cancellation of match requests: Before the 20 seconds is up, the user can also manually cancel the matching request.
- Integration with collaborative space: Upon finding a match, users will be automatically redirected to the collaborative space that is unique to each matched pair to allow them to work on the coding questions together.

Implementation

- Simplifying state: Instead of managing multiple pieces of state (e.g., one for when the user sends a match request, one for if the user cancels the match request, another for when the timer exceeds 20 seconds), we reduced the complexity by condensing them all into just two states – either the match request timer is running or not.
    - When the button is clicked
        - If the match request timer is running, the Frontend knows that there is an ongoing match request and the user wants to cancel it.
        - If the match request timer is not running, the Frontend knows that the user wants to send a new match request.
    - When the timer reaches 20 seconds, the Frontend knows to cancel the match request and reset the timer

Features

- Synchronized Code Editor: Both users can write code in the shared code editor at the same time; changes made by either user are automatically synchronised and displayed on screen in real-time.
- Rich Text Editor: Our code editor incorporates QuillJS, a rich text editor. The integration of this rich text editor aims to enable users to easily format their code with various layout options such as bold, italic, underline, highlight, code-block etc.
- Question Exploration: Users can view the full list of questions that match their chosen complexity level in the collaborative space.
- Chat Functionality: Users can communicate with each other in real-time using the integrated chat function. This allows both parties to discuss various strategies, brainstorm different solutions and clarify their doubts they have with each other which helps to enhance the collaborative problem-solving process. The chat comes with timestamps, typing status indicators, and users can even send syntax highlighted code blocks.
- Programming Language Selection and Syntax Highlighting: Users have the flexibility to select the programming language they wish to work in and to apply syntax highlighting. This helps to improve the ease of coding and also enhances the coding experience. Users can also specify “plaintext” if they prefer no syntax highlighting.
- Code Formatting: For certain programming languages like JavaScript and TypeScript the coding space includes a “Format Code” functionality whereby users can format their written code using specified conventions and rules. This improves code readability which translates to better understanding and collaboration between users.
- History Records: The code written during these collaborative sessions can also be recorded in the History Service. This feature provides users with a useful log of their collaborative efforts and coding progress.
- Ending of collaboration session: Users can end their collaboration session at any time by clicking the “End session”. If the user has unsaved changes in their code editor, they will be alerted to save their attempt to the history service.
- Leaving and rejoining the collaboration session: When one user leaves the collaboration session, the other user will have a pop up to inform them that they are waiting for the other user and give them the option to navigate back to the Find Match page. If the other user rejoins the collaboration session, their collaboration page will be synchronised and updated with the remaining user in the session.

Implementation

- Real-time chat using Socket.io: There is a Socket.io client instance which emits chat events to the collaboration server and listens to incoming chat events coming from the collaboration server. The events being emitted are the “send message”, “user typing” and “user stopped typing” events. The events being listened to are the “incoming message”, “other user typing” and “other user stopped typing” events.
    - The “user typing” event is emitted when there is a change to the text in the chat input with the help of the onChange handler. A “user stopped typing"

event will be emitted when the user sends the message or when the user clicks outside of the chat input.

- Real-time code editor using Socket.io: We chose Quill.js as our code editor as Quill.js allows us to track the content and changes in the editor using deltas. The code editor shares the same Socket.io client instance with chat and emits the delta whenever the user changes the contents in their editor. The other user in the same collaboration room will then update their code editor using the received delta, ensuring synchronisation of both editors.
- Syntax highlighting using highlight.js: We integrated highlight.js into our code editor to provide extensive support for syntax highlighting for popular programming languages like Python, C/C++ and Java. The user can also select their preferred code editor themes from a drop down list.
- Code formatting with Prettier: We integrated Prettier into our code editor to provide automatic formatting and indentation for JavaScript and TypeScript.
- User rejoining collaboration session: When a user rejoins the collaboration session, the other user that remained in the session will emit their selected programming language and code editor content to the Collaboration Service using their Socket.io client, which will then emit the content to the incoming user. The incoming user's Frontend will then be updated with the received content, synchronising all users in the collaboration session.

Code Execution Component

Features

- Users can run the code that they have written in the code editor and see the results displayed below the code editor. This helps to speed up the feedback loop for quicker development.
- The output window also displays error messages to the user (if any).

Implementation

We decided to incorporate Judge0 into our application to handle the execution of code in a secure and sandboxed environment.

We submit code to the Judge0 execution service through their online API library found at RapidAPI. The Judge0 API has been configured such that each code execution is limited to 5 seconds of runtime and 128MB of memory to minimise the load on the Judge0 API. (see N3.F3)

How the Code Execution component works:

- User clicks on the "Run Code" button.
- The Frontend sends a request to the Judge0 API along with the code text.
- The Judge0 execution service runs the code and responds with the results / error messages.
- Finally, the Frontend displays the output / error messages to the user as appropriate.

Design decisions

Following their GitHub page, we initially incorporated the Judge0 Docker container into our application as it gives us better control on the CPU and memory limits. In addition, there is no limitation on the number of executions per day, unlike the online Judge0's API which imposes a restriction of 50 submissions per day. This allows our app to scale to more users.

The above setup worked on our local testing environments. However, when we tried to deploy our application on AWS, there were errors related to the AWS Linux operating system not being compatible with the Judge0 container.

Since we consider the code execution service as a Supporting Sub-domain, we decided to use the online Judge0 API hosted on RapidAPI for our deployment.

4.3 Microservices

4.3.1 Auth Service

The Auth Service handles authentication and authorisation, and it encapsulates most of the backend servers of PeerPrep (except for Collaboration Service).

Features

- Manages sign-up, log-in and log-out
- Generates and validates signed user cookies to maintain user sessions
- Acts as a API gateway for all other microservices, forwarding user requests to the various API endpoints of other microservices only if the user is appropriately authenticated and authorised

Implementation

The Auth Service follows the REST architecture and authenticates users using Passport.js. The Auth Service also acts as an API gateway for most of the other microservices – upon receiving a request, it uses express-session to ensure that users are authenticated before forwarding the request to other services. When attempting to add, update or delete questions in the Questions Service, the Auth Service will also check that the user is authorised to do so (i.e., the role must be "admin"). The Auth Service uses a MongoDB database to keep track of users' session cookies.

- For security purposes, we salt and hash the user's passwords using bcrypt before storing them in our database (i.e., we do not store their passwords in plaintext)
- When a user log-ins successfully, the server will generate a session cookie to be sent to the Frontend. This cookie will be used to authenticate users on subsequent requests.

The diagram below shows an overview of the data flow in the Auth Service.

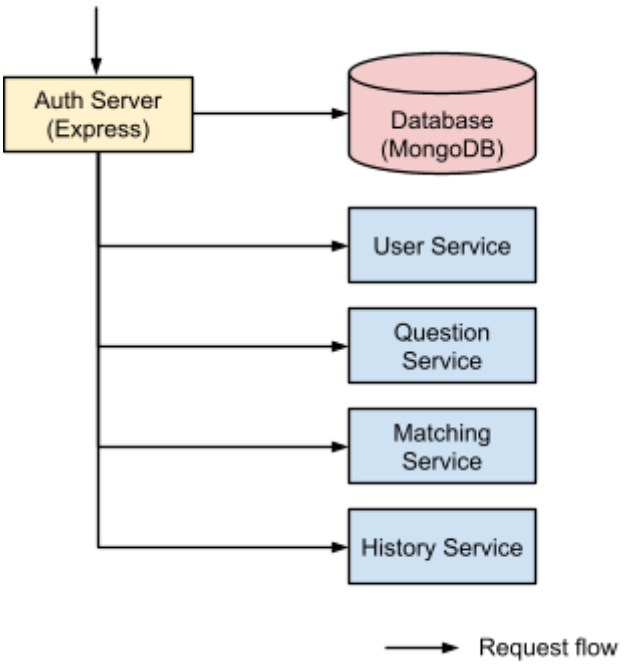

Figure 4.3.1a

Design Decisions

- Focused Scope: Auth Service only manages user authentication (e.g., sign-up, log-in, log-out). Other features like user profile management are delegated to a dedicated microservice like User Service (see 4.3.3). This ensures a clear separation of concerns.
- API Gateway: The Auth Service plays a critical role in providing an additional layer of abstraction for the various microservices and also enhances security by centralising authentication and authorisation checks.

4.3.2 Questions Service

The Questions Service manages questions and their details in a persistent MongoDB database.

Features

- Create, read, update and delete questions

Implementation

- The schema of question consists of one table which has the following columns: title, categories, complexity and description.
- The router receives GET, POST, PUT and DELETE HTTP requests and calls the appropriate controller.
- The controller validates the data before sending the request to the Model

    - The Auth Service (and not the Questions Service) will handle the checking of authorisation for the users when attempting to add, update and delete questions.
- The Model sends the request to our MongoDB database in the cloud and returns the response to the controller.
    - If there are errors received from the database, the controller will also return a response to the Frontend with the appropriate status codes.

The diagram below gives an overview of data flow in the Questions Service.

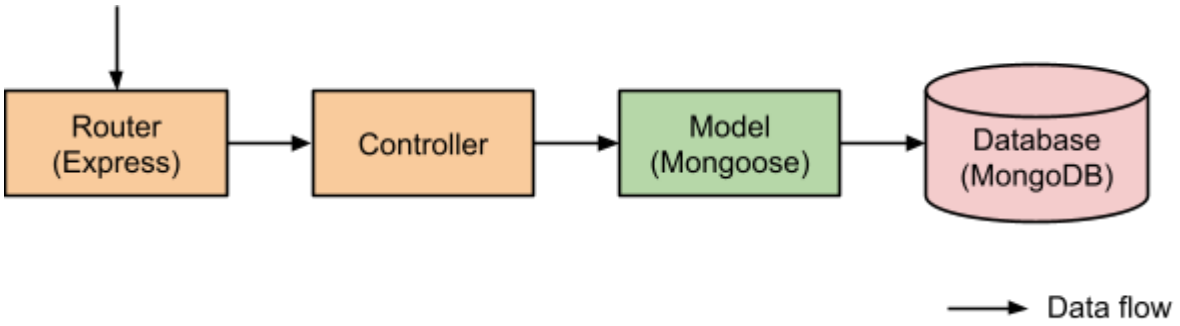

Figure 4.3.2a

Design Decisions

The Questions Service follows the MVC architecture. It handles:

- Validating new questions
- Adding of valid questions to the database
- Retrieves questions based on title, complexity or category
- Validates and updates questions by id
- Deletes questions by id
- Prevention of questions with duplicate titles

The Questions Service does not handle:

- Authentication of requests (see 4.3.1)

4.3.3 User Service

The User Service follows the REST API standard and handles the details of users in a persistent PostgreSQL database. It is hidden from the client as it only interacts with the Auth Service.

Features

- Create, read, update and delete users

Implementation

This service uses the Express.js framework which runs on Node.js. The database is a PostgreSQL database which is hosted in the cloud. The schema consists of one table which has the following columns: username (Primary key), display name, (hashed) password and role (“basic” / “admin”).

- The router receives GET, POST, PUT and DELETE HTTP requests and calls the appropriate controller.
- The controller checks for the validity of data and sends the request to the service handler
    - The controller also catches errors received from the database and returns the responses with appropriate status codes.
- The service handler builds the relevant SQL queries to create, read, update or delete data and sends the query to the Database Instance which is created from the pg library (a PostgreSQL client for Node.js).
- The Database Instance sends the request to our external PostgreSQL database in the cloud and returns the response to the service handler.

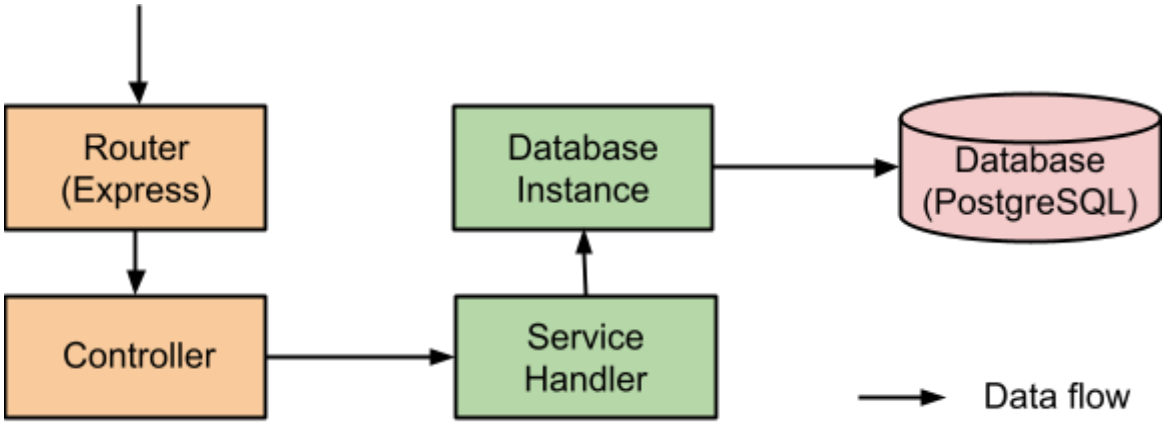

Figure 4.3.3a

Design Decisions

The request handlers are broken down into controllers and services. Controllers are responsible for handling the request from their respective endpoints. They make use of services which encapsulates the business logic and data manipulation. This allows for better separation of concerns. This follows the MVC architecture.

4.3.4 Matching Service

The Matching Service handles requests to match other users with a specific complexity level, and responds to both requests when a match is found.

Features

- Finds a match for a user requesting to work with a chosen question complexity
- Cancel match requests

Implementation

- User selects a question complexity level and clicks the "Find Match" button,
- The client (browser) sends the request to the Auth Service using WebSockets.
- Upon receiving a matching request, the Auth Service server sends this request to RabbitMQ, the message broker, using the Advanced Message Queue Protocol (AMQP).
- The Matching Service is subscribed to RabbitMQ for new match requests. When receiving match requests, it queues them up. When it receives two requests with the same chosen complexity level, it sends 2 messages to rabbitMQ responding to each of the clients.
- The Auth Service server receives the response from rabbitMQ and relays the response to the respective clients through the same WebSocket. Both clients are then redirected to the shared collaboration space where they can begin their joint coding session.

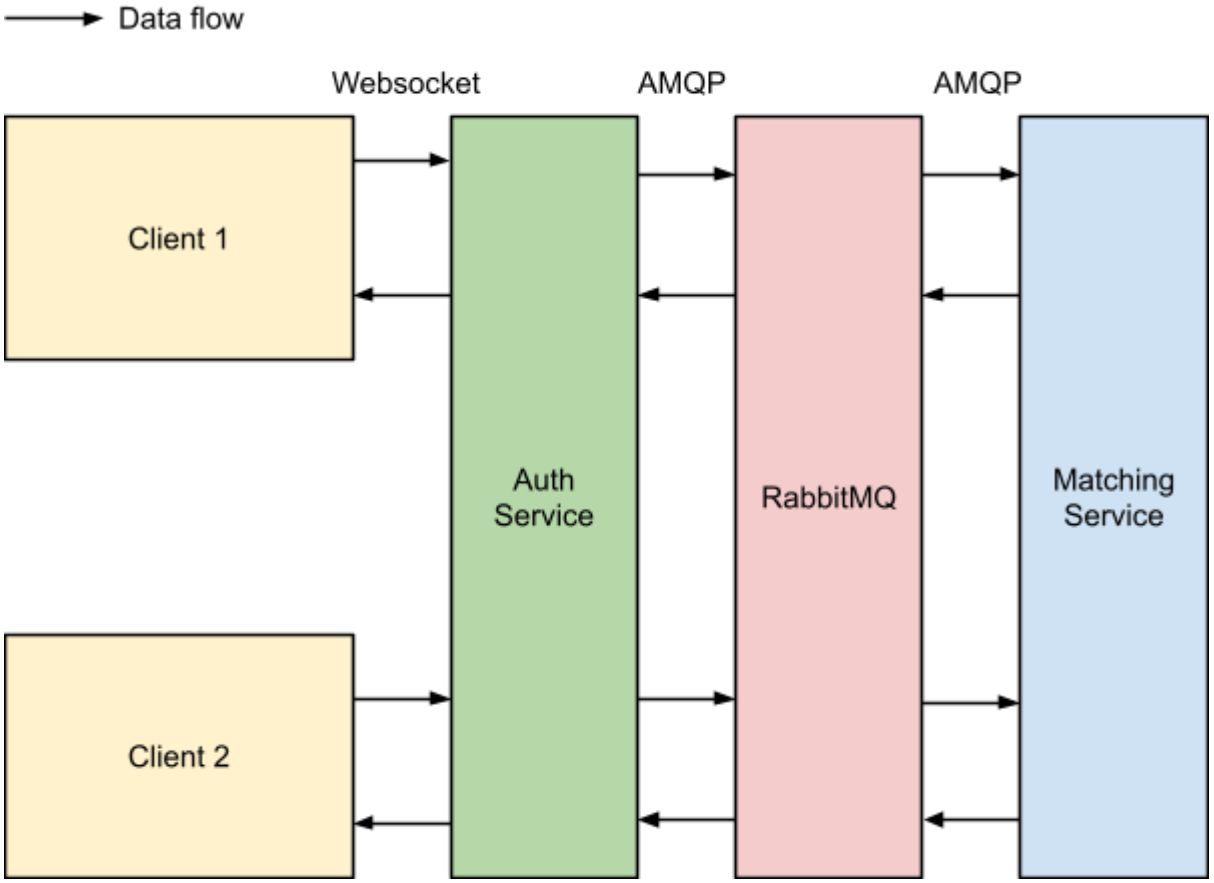

Figure 4.3.4a

Design Decisions

Message Patterns

The Point-to-Point Message Channel Pattern is used between the Auth Service and Matching Service. Each client sends a request to the Matching Service, and its reply (the corresponding match found) is only useful to that client.

- *Return Address*
  Since the Auth Service receives match requests from multiple clients, it uses the return address field to inform the Matching Service of the correct channel queue to send its reply to.
- *Correlation ID*
  Since each client is able to send multiple requests in quick succession (Change preferred question complexity, cancel previous request), Correlation ID is used for the Auth Service to match requests and replies to the same client.

Communication Patterns

Transient Asynchronous Communication Pattern is used between the client and server.

- *Transient*
  Both the client and server are expected to be running while the request and reply are being sent and processed. It is not useful to store the requests persistently if the server does down for any reason; The user would have already stopped waiting for a match by the time the server comes back online to receive the message. Similarly, it is not meaningful to store replies persistently for the client if the client has already disconnected and is not available to join a session with the other user.
- *Asynchronous*
  Finding a match takes up to 20 seconds. The process of match finding should be asynchronous so that the browser can continue to handle rendering of loading animations and handling clicking of the cancel match button while waiting for a match.

Due to the above reasons, we used RabbitMQ as the message broker between the 2 services. By having a message broker, we can independently scale the Auth Service by running more instances of servers, without needing to update the Matching Service which can continue listening on the same channel on RabbitMQ.

4.3.5 Collaboration Service

The Collaboration Service allows two users to share a collaborative workspace together. It handles the sending and receiving of chat messages, and also automatically syncs both code editors and the selected programming language.

Features

- Both users that have been matched with each other will be automatically placed in the same collaboration room.
- Automatically sync changes to the code editor and the selected programming language to both parties
- Inform users of when the other party leaves / rejoins the collaborative space.
- Handles chat events like sending / receiving messages and showing the typing status indicators of each party.

Implementation

The Collaboration Service operates on an Event Driven Architecture (EDA) where the events of each client will be broadcasted to other clients. Each client represents a user engaging in the collaboration room using the Socket.io client. The Collaboration Service represents the Socket.io server that listens and emits events to clients.

- When two users are paired, they connect to the Collaboration server using the Socket.io client, specifying the room ID given by the Matching Service in the query.
- The Collaboration server puts the clients in the same Socket.io room as specified.
- Any action or code modification made by one user is immediately transmitted via the connected Socket.io client to the server, which then emits the changes to the other user in the same Socket.io room.
- The other user in the same room receives the emitted changes and performs updates to its own Collaboration page to ensure synchronisation between the two users.
- The chat events for messages and typing status indicators are also emitted and listened to by the server between the two users in the same room.

The following are the actions that would be emitted to other users in the same room:

- When a user joins or exits the room.
- Changes to the code editor.
- Changes to the language selected.
- Chat messages
- When the user is typing / stopped typing

Design Decisions

For other parts of the app, Auth Service, acting as an API gateway, handles incoming requests for various services. However, routing Collaboration Service requests through this gateway would inevitably introduce additional latency.

For collaborative applications, ensuring low latency is paramount for smooth user interactions. Hence, we have configured the Collaboration Service to bypass the Auth API gateway. Furthermore Socket.io makes use of Websockets which is faster than HTTP. This effectively reduces the processing time for each request, ensuring quick synchronisation and seamless user experiences in collaboration.

4.3.6 History Service

Oversees the adding of each user's question attempt.

Features

- Stores the username of the user, the shared code editor text, the programming language of the code editor text, the id of the attempted question and the timestamp of the attempt

Implementation

- Make use of PostgreSQL database to store the details of each history attempt.
- The primary key of the database is username, question id, and timestamp of the attempt

5. Development Process

Time	Weekly Progress
Week 3	● Decided on the tech stack and selected tools for development. ● Assigned specific assignment tasks and responsibilities to each team member. ● Set up a shared team repository for collaborative work. ● Defined the overall architecture of the project.
Week 4	● Drafted functional & non-functional requirements for the project. ● Set priority for each requirement. ● Developed a single-page application for the Questions Service. ● Implemented basic APIs for user, authentication, and question backend services.
Week 5	● Completed login, signup, and user session management features, integrating them with the existing question service. ● Established connections with backend databases for managing user & question data. ● Integrated question, authentication, and user functionalities into a cohesive system. ● Record assignment 1 video
Week 6	● Implemented error handling mechanisms for Questions service and User Service. ● Containerised services with Docker using Dockerfile. ● Integrated the Auth Service with the APIs of user and questions services, establishing it as the gateway API.
Week 7	● Added profile management functionality to the User Service. ● Record assignment 2 video ● Record assignment 3 video ● Explored serverless functions, queue mechanisms, and socket technology. ● Developed a basic framework for the collaboration page using socket.
Week 8	● Dockerized the existing microservices. ● Enabled code synchronisation during collaboration sessions via socket connections.

	• Managed matching requests using RabbitMQ, based on question complexity.
Week 9	• Enhanced Questions Service to include filtering, searching, and sorting functionalities. • Developed History Service to log timestamps and code for attempted questions. • Developed a basic structure for enabling chat between collaboration partners. • Created frontend pages for the Matching Service. • Recorded assignment 4 video.
Week 10	• Integrate matching service with collaboration service. • Implemented a frontend for the History Service, featuring syntax-highlighted code display. • Completed the chatting functionality within the Collaboration Service. • Set up serverless functions to fetch questions from the Leetcode API for assignment 5. • Recorded assignment 5 & 6 video.
Week 11	• Wrote the README for each assignment. • Enhanced the aesthetic appeal and user experience of the chat feature. • Implemented handling of disconnections and session termination in the Collaboration Service. • Adopted Redux for state management in frontend web pages. • Started the deployment process for the application.
Week 12	• Developed code execution functionality within the application. • Implemented code formatting and syntax highlighting in the collaboration code editor. • Integrated History Service to record work done during collaboration. • Completed the deployment of the frontend component of the application. • Synchronised user-selected options for programming language and theme between two users in the collaboration session. • Start writing the final report
Week 13	• Completed the Deployment of the application on AWS EC2 instances, incorporating load balancing and auto-scaling features. • Resolved issues related to credentials, cookies, and port configurations during deployment. • Implemented the use of innerHTML for rendering content in the chat box. • Finalised the project report and prepared the accompanying presentation slides.

Our team followed the Iterative Agile Development Process. We held weekly scheduled scrum meetings at the same time every week. During these scrum meetings, we engaged in group discussions and each member presented updates on their progress from the past week. Any unfinished task was placed in our product backlog and reconsidered for tasks and objectives for the upcoming week. This structured approach ensured that we remained aligned and focused on our top priority project goals.

During our development, we adopted the forking workflow. This helps to prevent unintended and unwanted modifications to the master branch as every code change must be approved by another member of the team through pull requests. Any pull request created to the master branch is also manually reviewed and tested by at least one reviewer before merging into the master branch.

The microservices architecture also allowed us to work in parallel as each member works on his/her own microservices and components simultaneously, without causing major git merge conflicts.

6. Future Improvements and Enhancements

- Integrated browser support for real-time video and audio sharing:
    - Current State: Users have to use the integrated chat feature to communicate with one another during the session.
    - Suggested Improvement: Allow users to enable their webcam and microphone for smoother collaboration. This allows users to have the ability to talk through their thinking process and solutions as they solve the questions.
    - Rationale: Technical interviews not only test users on their ability to code, it also tests the users ability to convey their thought process and work collaboratively with the interviewer. This is an essential soft skill that interviewers look out for during these technical interviews.

- Enhanced Question Selection in Matching
    - Current State: Presently, users match based on complexity levels alone.
    - Suggested Improvement: Introduce the capability for users to select questions based on specific categories or topics. This allows for a more tailored learning experience and can better align with users' current areas of focus or interest.

- User Profile Personalization
    - Current State: The user profile only displays the user's username, display name and password.
    - Suggested Enhancement: Allow users to upload and set their own profile images, adding a personal touch to their accounts.

- User Login Tracking
    - Current State: User login sessions are not visually tracked over time.
    - Suggested Improvement: Implement a login calendar or visual tracker that showcases the days a user has accessed the platform. This encourages consistent engagement and allows users to monitor their preparation patterns.

- User Attempted Questions Analytics
    - Current State: The History Service only records the code that the user saves..
    - Suggested Enhancement: Track all questions attempted and display a distribution chart of the different complexity levels of questions the user has

tackled. This provides users with insights into their progression and areas they may need to focus on.

- Popularity Metrics for Questions
    - Current State: There is no record of the popularity of each question.
    - Suggested Improvement: Implement a system that tracks the popularity or frequency of attempts for each question. Provide users with the option to sort or filter questions based on this popularity metric, guiding them to trending or highly-attempted questions.

- Questions management
    - Current state: Maintainers manually log in to add, update and delete questions. This is time consuming for maintainers.
    - Suggested Improvement: Implement serverless functions to automatically add in question of the day from Leetcode, as well as a serverless function to fetch and update the 20 sample questions in PeerPrep from Leetcode. This reduces the need for maintainers to add new questions periodically and eliminates the chance of errors when transferring questions from Leetcode manually.
    - Difficulty: A prototype has been implemented for Assignment 6. All it needs is deployment to a service for serverless functions, and addition of 2 buttons to sync the question of the day and the 20 sample questions.

- Deployment
    - Currently, the load balancer for the Auth Service serves as the API gateway. We can include AWS CloudFront and AWS Simple Storage Service (S3) to route front end and back end services to different servers and serve static content in a S3 bucket from multiple AWS Points of Presence to minimise latency.
    - Currently, we are using HTTP to send and receive requests. We should use AWS Route 53 to purchase our own domain name to set up our own TLS certificate so as to use https to secure our communication between the client and server

7. Reflections and Learning points

7.1 Development Process

- Technological Challenges
    - Utilising advanced technologies like Socket.io, AMQP, and Docker in our project presented a series of challenges, particularly when adapting to varying running environments and deployment contexts. These experiences underlined the necessity of flexibility and in-depth understanding when integrating new technologies.

- Code Architecture and Organization
    - Adopting structured code architecture and file organisation in our project proved invaluable. Such an approach not only facilitated smoother development, but also significantly eased the process for team members to review and understand each other's code.

- Attention to Detail
    - Focusing on the details, from page layouts to the delays in synchronisation, was essential. Addressing these details greatly enhanced the user experience. Additionally, preemptively considering potential issues, like a peer leaving a matching room abruptly, allowed us to implement more robust and user-friendly solutions.

7.2 Teamwork

- Timely Communication
    - We realised the significance of prompt and clear communication. This not only kept everyone aligned but also preempted potential misunderstandings that could derail our progress.
- Constructive Feedback
    - Providing each other with constructive feedback through GitHub PR reviews was a cornerstone of our teamwork. This not only fortified our working relationships but also facilitated collective growth as we learned from each other's ideas.