Peer Prep

Practice Leetcode with others!

trypeerprep.com

Team

Lee Shao Wee github.com/shaowi	A0240790H
Kartikeya github.com/kxrt	A0235401W
Kelvin github.com/chuakid	A0235179X
Peigeng github.com/Bacon-Strips	A0230391N

Background

In recent years, there has been a significant surge in the demand for individuals with technical skills, especially in fields like software engineering and computer science. This demand has been further fueled by a shift in interview methods, as companies increasingly favour practical, whiteboard-style assessments, emphasizing problem-solving and coding abilities. Existing resources often need more suitable platforms for students to connect with practice partners and engage in collaborative preparation. Recognizing the value of peer learning, "PeerPrep" was designed to empower students to connect, practice, and learn from one another effectively. Moreover, the app addresses the time constraints often faced by job seekers, efficiently matching users and facilitating focused and productive interview practice.

This project outlines the creation of a technical interview preparation platform and peer matching system named "PeerPrep." This platform is designed to cater to students preparing for technical interviews.

Purpose

To provide a platform where students can connect with peers to practice whiteboard-style interview questions.

This platform aims to facilitate the following actions for students:

1. Creating an account and logging in.
2. Selecting the difficulty level of interview questions (easy, medium, or hard) and choosing a specific topic.
3. Match with another online student who has chosen the same difficulty level and topic.
4. Collaborating on a real-time solution to the provided interview question with their matched peer.
5. Allowing users to terminate collaborative sessions gracefully when needed.

This project aims to help students prepare effectively for technical interviews by providing a user-friendly platform for practising and collaborating with peers.

Application Requirements (M- Must-have, N - Nice-to-have)

Type	Functional Requirement	Priority	Sprint	Allocated to
User Service M1				
M	Users can create an account and log in securely	H	1	Shao
	Users can change their user profile name	M	1	Shao
	Users can delete their accounts	M	1	Shao
	Users should not be able to sign up multiple times with the same email address	H	1	Shao
N	Users can change their password	M	1	Shao
	Users can reset their password	M	1	Shao
	Users can see other users they have collaborated with	M	2	Shao
	Users can see their submission history	M	2	Shao
	Users can see their attempted questions	M	2	Shao
Matching Service M2				
M	Implement a matching algorithm that pairs users with similar difficulty levels.	H	1	Kart, Shao
	Allow users to terminate collaborative sessions when they finish their practice gracefully.	H	1	Kelvin
	If a user is not matched within 5 minutes, they should be timed out and notified.	L	3	Peigeng
	Users can view a countdown of the time they have left until the matching period expires	L	3	Peigeng
	Users who are matched together should receive the same question	H	1	Shao

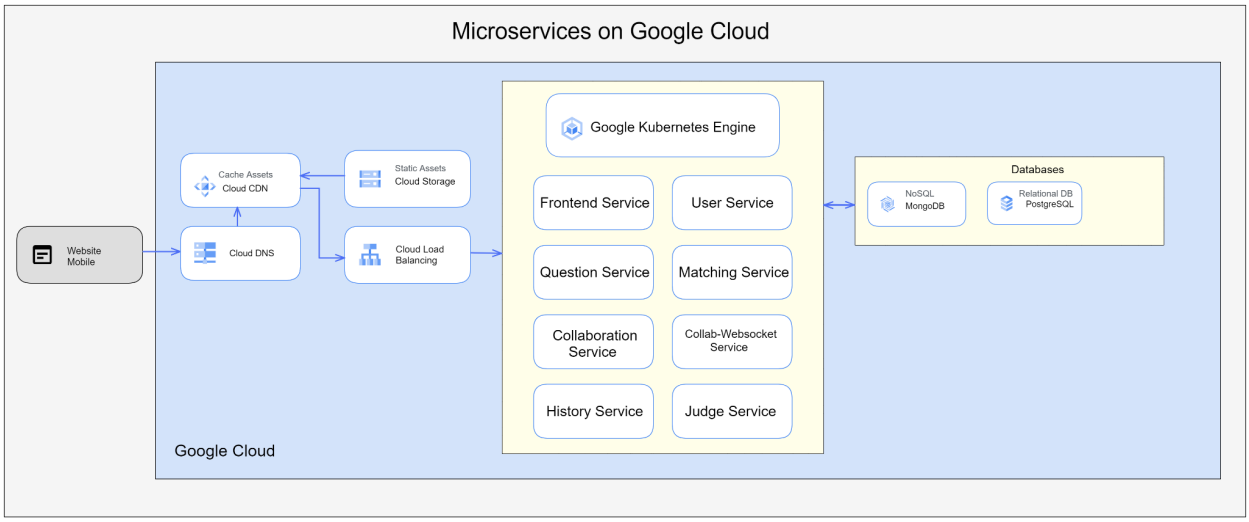
N	Users can be matched up with other users by difficulty levels	M	2	Shao, Peigeng
	Users are informed when their partner disconnects, with an option to terminate the session	L	2	Peigeng
Question Service M3				
M	Users can view and filter questions by difficulty level (easy, medium, or hard)	H	1	Kart
	Admins can update questions and test cases	H	2	Kart
Collaboration Service M4				
M	Users can work on a question with another user on the same editor interface	H	2	Kelvin
N	Users can compile and execute the code they have written	M	2	Peigeng
	Users can provide their test cases and view the result	M	2	Peigeng
	Users can communicate with each other via text chat	M	3	Kelvin
Frontend Service M5				
M	Users can register on the frontend	H	1	Kelvin
	Users can log in on the frontend	H	1	Kelvin
	Users can view questions on the frontend	H	1	Kelvin
	Users can write code for the questions	H	2	Kelvin
N	Users can see run-time statistics	L	3	Kelvin
	Users can see other people’s profiles	L	3	Kelvin
	The user should be able to use an improved code editor with formatting and syntax highlighting	L	3	Kelvin
<Blank>				
	Non Functional Requirements			

N	Application to be deployed on GCP	M	3	Peigeng
	The application should be scalable using Kubernetes	L	3	Kart
	The application should have automated CI/CD	M	2	Peigeng
	Question service is secure from code submissions from malicious users	M	2	Peigeng
	Passwords should not be stored in plaintext	H	1	Shao
	JSON Web Tokens (JWT) should expire within an hour for security purposes	M	2	Shao
	Ensure backend services can only be accessed by authenticated users	H	1	Shao, Kart
	Users can access PeerPrep from any modern web browser on at least a 1440px screen width	L	3	Peigeng
	Users should be able to run test cases for a question within 10 seconds	H	3	Peigeng

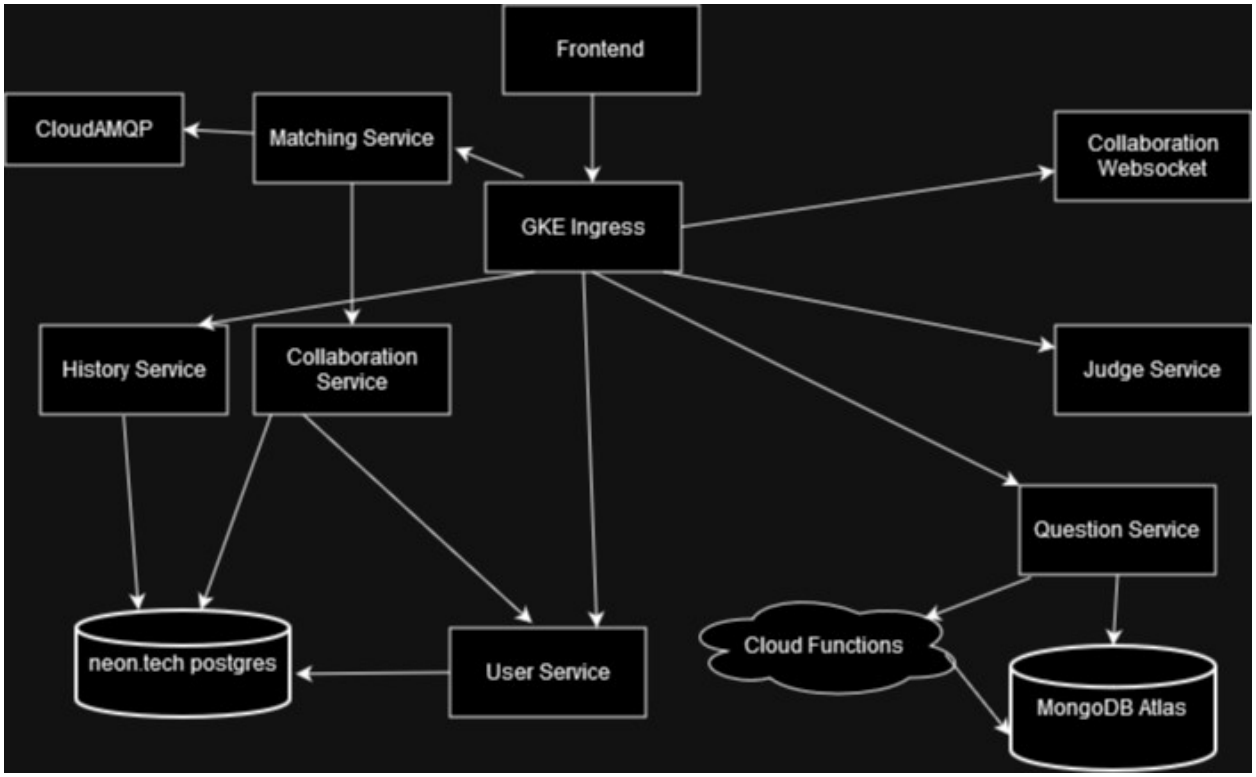
Tech Stack

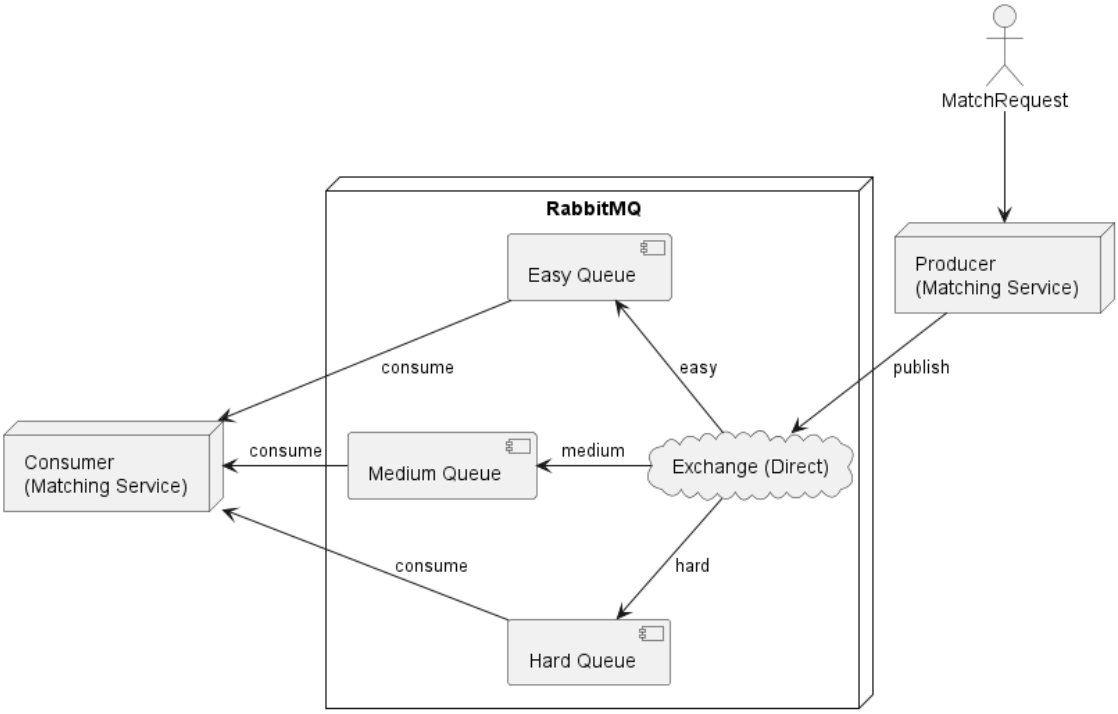
Languages: TypeScript, Go, Python, SQL
Frameworks: React.js, FastAPI, Gorm
Tools: RabbitMQ, MongoDB, PostgreSQL, WebSockets, GCP

System Architecture



Deployed on GCP





Frontend

Description

The frontend is built in Typescript React with Mantine UI as the component framework of choice. React-router is used to manage frontend routing.

Vite is used as the bundler and development server.

We use eslint and prettier to ensure coding standards are met.

We use Tanstack Query to manage our query states. The frontend sends all calls to the reverse proxy and lets it decide which service to redirect the call.

The synchronization aspect of the code editor is implemented using yjs, which allows us to synchronize data structures like text and maps. We use this to synchronize user names and chat as well.

For deployment, the frontend is built with Vite and deployed to an nginx container.

Dependencies

- **MantineUI v6**
- **React**
- **React-router**
- **Vite**
- **Tanstack Query**

- **yjs**

Backend - User Service

Description
The user service handles operations related to users such as user creation, retrieval, deletion and sign-in. These users are stored in the user table in a relational database.

The service is interacted with by RESTful API and built with Gorm (ORM library for Golang). Gorm is a popular Object Relational Mapping (ORM) library for the Go programming language. It simplifies database interactions by offering a clean and dynamic syntax, supporting multiple database backends, providing struct mapping and automatic migration for table creation. Its ease of use, compatibility, and features make it a preferred choice for us to work with a relational database.

Dependencies
- PostgreSQL: Database to store the users
- Gorm: ORM library for Golang

API

Endpoint	Description

GET
/api/v1/user
	Retrieves user by user id from the database.
	
	Status Codes
	200 - Success
	401 - Unauthorised
	
	Response Schema
	{
	"user_id": uint,
	"email": "string",
	"name": "string",
	"access_type": uint // 0: user, 1: admin
	}

POST
/api/v1/user
	Retrieves user by JWT from the database.
	
	Status Codes
	200 - Success
	401 - Unauthorised
	
	Request Schema
	{
	}

	Authorization: Bearer <JWT>
	
	Response Schema
	{
	"user_id": uint,
	"email": "string",
	"name": "string",
	"access_type": uint // 0: user, 1: admin
	}

GET
/api/v1/user/all
	Randomly all the users from the database.
	
	Status Codes
	200 - Success
	500 - Internal Server Error
	
	Response Schema
	[
	{
	"user_id": uint,
	"email": "string",
	"name": "string",
	"access_type": uint // 0: user, 1: admin
	}
	]

POST
/api/v1/user/register
	Add a user into the database.
	
	Status Codes
	201 - Created
	500 - Internal Server Error
	
	Request Schema
	{
	"email": "string",
	"password": "string", // must be at least 6 characters long
	"name": “string" // must be at least 3 characters long
	}
	Authorization: Bearer <SecretKey>
	
	Response Schema
	{
	"user_id": uint,
	"email": "string",
	"name": "string",
	"access_type": uint // 0: user, 1: admin
	}
	* Registered user is set as admin only if a correct SecretKey is provided.

POST
/api/v1/user/deregister
	Remove a user from the database.
	
	Status Codes
	202 - Accepted
	500 - Internal Server Error
	
	Request Schema
	{
	"user_id": uint
	}
	
	Response Schema
	{
	"user_id": uint,
	"email": "string",
	"name": "string",
	"access_type": uint // 0: user, 1: admin
	}

POST
/api/v1/user/login
	Logs in a user from the database.
	
	Status Codes
	200 - Success
	500 - Internal Server Error
	
	Request Schema
	{
	"email": "string",
	"password": "string", // must be at least 6 characters long
	}
	
	Response Schema
	{
	"jwt": "string"
	}

POST
/api/v1/user/changepassword
	Change the password of a user from the database.
	
	Status Codes
	200 - Success
	500 - Internal Server Error
	
	Request Schema
	{
	"email": "string",
	"oldPassword": "string", // must be at least 6 characters long
	"password": "string", // must be at least 6 characters long
	}
	
	Response Schema

	{
	"message": "string"
	}

POST
/api/v1/user/changename
	Change the name of a user from the database.
	
	Status Codes
	200 - Success
	500 - Internal Server Error
	
	Request Schema
	{
	"email": "string",
	"name": "string", // new name (must be at least 3 characters long)
	}
	
	Response Schema
	{
	"message": "string"
	}

Backend - Question Service

Description

The question service handles operations related to programming questions that users can attempt on the PeerPrep website. These questions are stored as documents in a MongoDB instance.

It exposes a CRUD API that allows the reading, creating and deleting question documents in MongoDB. It further calls on a cloud function every 24 hours to keep the list of questions updated and in sync with what one may find on leetcode.com. We offloaded the fetching and processing of questions to the cloud function to follow the Single Responsibility Principle, as the question service is not responsible for populating the questions itself.

This service was written in Python's FastAPI framework. We selected this framework because of its connection with Pydantic, which allowed us to implicitly validate data models for questions to protect the integrity of our NoSQL database structure. FastAPI also depends on the OpenAPI 3.0 specification, allowing us to present Swagger docs for quick interfacing with the API during testing.

Dependencies

- MongoDB: Storing question documents

- Cloud Function: Fetch questions from leetcode.com

Environment

- MONGODB_URL: URL for a specific database to store questions
- QUESTIONS_URL: URL for cloud function to fetch questions from leetcode.com

API

Endpoint	Description

GET
/api/v1/questions/all
	Retrieves all questions from the database.
	
	Status Codes
	200 - Success
	500 - Internal Server Error
	
	Response Schema
	[
	{
	"_id": "string",
	"title": "string",
	"description": "string",
	"categories": [
	"string"
	],
	"complexity": "string"
	}
	]

GET
/api/v1/questions/easy
	Randomly retrieve an easy question from the database.
	
	Status Codes
	200 - Success
	500 - Internal Server Error
	
	Response Schema
	{
	"_id": "string",
	"title": "string",
	"description": "string",
	"categories": [
	"string"
	],
	"complexity": "Easy"
	}

GET /api/v1/questions/medium	Randomly retrieve a medium question from the database. Status Codes 200 - Success

	500 - Internal Server Error
	
	Response Schema
	{
	"_id": "string",
	"title": "string",
	"description": "string",
	"categories": [
	"string"
	],
	"complexity": "Medium"
	}

GET
/api/v1/questions/hard
	Randomly retrieve a hard question from the database.
	
	Status Codes
	200 - Success
	500 - Internal Server Error
	
	Response Schema
	{
	"_id": "string",
	"title": "string",
	"description": "string",
	"categories": [
	"string"
	],
	"complexity": "Hard"
	}

GET
/api/v1/questions/:id
	Retrieve a question from the database with a specified ID. This is in line with the question numbers on leetcode.com, and not the internal _id field used by MongoDB. The _id field in the response reflects this. The question ID to retrieve is taken in as a path parameter.
	
	Status Codes
	200 - Success
	404 - Question not found
	
	Response Schema
	{
	"_id": "string",
	"title": "string",
	"description": "string",
	"categories": [
	"string"
	],
	"complexity": "string"
	}

PUT /api/v1/questions/update (Admin-Only)	Update a question in the database with a specified ID. This is in line with the question numbers on leetcode.com, and not the internal _id field used by MongoDB. The _id field in the response reflects this. Status Codes 202 - Updated/Accepted 500 - Internal Server Error
POST /api/v1/questions/add (Admin-Only)	Create a question in the database. To set the ID for this question, it takes the current highest question ID and increments it by 1. Status Codes 201 - Created 500 - Internal Server Error
DELETE /api/v1/questions/:id (Admin-Only)	Delete a question in the database with a specified ID. This is in line with the question numbers on leetcode.com, and not the internal _id field used by MongoDB. The _id field in the response reflects this. The question ID to retrieve is taken in as a path parameter. Status Codes 200 - Success 400 - Question not found

Backend - Cloud Function

Description
A Google Cloud Function was configured to fetch question data from Leetcode's GraphQL API, parse it, and then return it as a list of question objects. The API requires specifying the number of questions to retrieve. Since the number of questions available on Leetcode is variable, we used 2 requests to achieve this - one to fetch the total number of questions available, and the second to fetch this specified number of questions.

At the time of writing this, there are over 2,300 questions retrieved by the Cloud Function. We then need to filter out premium-only questions from this list, as their data is not provided by the API. This process obviously takes a while to run (12-15 seconds), and we felt it wasn't appropriate triggering this every time a user loads up PeerPrep - it would degrade the user experience with large wait times. Hence, we configured the Question Service to run this cloud function every 24 hours, and synchronize the MongoDB instance with the most updated list of questions. This allows us to keep PeerPrep updated on a daily basis, while ensuring a reliable user experience.

The Cloud Function is written in TypeScript, which is transpiled to JavaScript before being deployed through GCP. We have a Makefile that simplifies this process, allowing one to test the function locally before deploying the latest version.

The question objects are returned in the following format:

```
{
  "id": string, // synchronous with question numbers shown on Leetcode
  "title": string,
  "description": string, // question content
  "categories": string[],
  "complexity": string (stored as an Enum: "Easy", "Medium", "Hard")
}
```

Backend - Matching Service

Description

The matching service facilitates match requests between users for collaboration. Message queues are created for matching users based on question difficulty and the cancellation of match requests. We used RabbitMQ message broker as it is relatively easy to use and supports multiple programming languages with Golang included. It also ensures message durability and can be scaled by adding more nodes to the cluster, making it a preferred option over Apache Kafka for our app.

Dependencies

- RabbitMQ
- WebSocket

API

Endpoint	Description

/ws	User requests for a match.
	
	Request Schema
	{
	"user_id": uint,
	"action": “Start”,
	"difficulty": “string” // Easy/Medium/Hard
	}
	
	Response Schema

	{
	"user_id": uint,
	"room_id": uint,
	"action": “string”,
	"difficulty": “string” // Easy/Medium/Hard
	}

	User cancels match request or match request times out.
	
	Request Schema
	{
	"user_id": uint,
	"action": “Cancel”,
	"difficulty": “string” // Easy/Medium/Hard
	}

	User stops collaboration with the matched user.
	
	Request Schema
	{
	"room_id": uint,
	"user_id": uint,
	"action": “Stop”
	}
	
	Response Schema
	{
	"error": "string" // Empty if there is no error
	}

Backend - Collaboration Service

Description

The collaboration service handles the collaboration of users after being matched successfully with room creation and closing. These rooms are stored in the room table in a relational database. The service is interacted with by RESTful API and built with Gorm (ORM library for Golang).

Dependencies

- PostgreSQL: Database to store the users
- Gorm: ORM library for Golang

API

Endpoint	Description

GET
/api/v1/room?id=1
	Retrieves a room with id of 1 from the database.
	
	Status Codes
	200 - Success
	401 - Unauthorised
	
	Response Schema
	{
	"room_id": uint,
	"question_id": "string",
	"user_a_id": uint,
	"user_b_id": uint,
	"is_open": bool,
	"created_on": "string" // format: "YYYY-MM-DD HH:MM:SS"
	}

POST
/api/v1/room/create
	Add a room to the database.
	
	Status Codes
	201 - Created
	401 - Unauthorised
	
	Request Schema
	{
	"question_id": "string",
	"user_a_id": uint,
	"user_b_id": uint
	}
	
	Response Schema
	{
	"room_id": uint,
	"question_id": "string",
	"user_a_id": uint,
	"user_b_id": uint,
	"is_open": bool,
	"created_on": "string" // format: "YYYY-MM-DD HH:MM:SS"
	}

POST
/api/v1/room/close
	Close a room from the database.
	
	Status Codes
	202 - Accepted
	401 - Unauthorised
	
	Request Schema
	{
	"room_id": uint
	}

	Response Schema
	{
	"room_id": uint,
	"question_id": "string",
	"user_a_id": uint,
	"user_b_id": uint,
	"is_open": bool,
	"created_on": "string" // format: "YYYY-MM-DD HH:MM:SS"
	}

Backend - Collaboration WebSocket Service

Description

This service is used to synchronize the CRDTs between users when collaborating. It sends synchronization packets to the other user in the same room, synchronizing aspects like chat, code editor and names.

Backend - History Service

Description

The collaboration service maintains the records of the questions attempted by the user(s). These attempts are stored in the attempt table in a relational database. The service is interacted with by RESTful API and built with Gorm (ORM library for Golang).

Dependencies

- PostgreSQL: Database to store the users
- Gorm: ORM library for Golang

API

Endpoint	Description

GET /api/v1/history/attempt?userId=1	Retrieves the attempts of a user with the user id of 1 from the database.
	
	Status Codes
	200 - Success
	401 - Unauthorised
	
	Response Schema
	[
	{
	"attempt": {
	"attempt_id": uint,
	"question_id": "string",

	"user_id": uint, "code": "string", "language": "string", "passed": bool, "attempted_on": "string" }, "question": Question }]

POST
/api/v1/history/attempt
	Adds an attempt from an user to the database.
	
	Status Codes
	201 - Created
	401 - Unauthorised
	
	Request Schema
	{
	"question_id": "string",
	"user_id": uint,
	"code": "string",
	"language": "string",
	"passed": bool
	}
	
	Response Schema
	{
	"question_id": "string",
	"user_id": uint,
	"code": "string",
	"language": "string",
	"passed": bool
	"attempted_on": "string" // format: "YYYY-MM-DD HH:MM:SS"
	}

GET
/api/v1/history/collaboration
?userId=1
	Retrieves the collaborations of a user with user id of 1 from the database.
	
	Status Codes
	200 - Success
	401 - Unauthorised
	
	Response Schema
	[
	{
	"collaboration": {
	"room_id": uint,
	"question_id": "string",
	"user_a_id": uint,

	"user_b_id": uint, "created_on": "string" }, "question": Question }]

Backend - Judge Service

Description

The Judge service executes the code attempts made by the user(s). These attempts contain the code typed by the user alongside the input and expected output of the program. They are run by the workers of the service and the result of the attempt is returned, such as “Accepted”, “Wrong Answer”, “Compile Error” etc. These results are stored in a Postgres database and the service fetches the attempt results from the database.

Initially, we intended to host the judge service alongside our other backend services onto Google Cloud. While it works when deploying on local docker containers, there are problems deploying the Judge0 image onto Kubernetes. Hence we decided to subscribe to Judge0’s public API and use that for the code execution within the deployment.

Dependencies

- Judge0: Open-source code executor system
- PostgreSQL: Database to store the attempt results
- Redis

API

- https://ce.judge0.com/

Deployment

We have 4 types of deployment available depending on the environment to work in:

- Local Development (LD) through Docker Compose
- Local Production (LP) through Docker Compose
- Bare Metal Production (BMP) through Kubernetes with Nginx Ingress
- Cloud Production (CLP) through Kubernetes with GKE Ingress

The first three will work out of the box with our repository (after replacing environment variables), whereas the fourth will require some extended manual configuration.

LD/LP

We used LD for development and LP for local testing. LD uses bind mounts to enable hot reloading of our files while working in a containerised environment, while running the contained services in development mode. LP is similar but excludes bind mounts, and runs the services in production mode.

LD and LP both contain the following services:

- Frontend
- User Service
- Matching Service
- Question Service
- History Service
- Collaboration Service
- Collaboration WebSocket Server
- PostgreSQL (locally hosted)
- RabbitMQ (locally hosted)
- Judge0 (locally hosted)
- MongoDB (locally hosted)
- Mongo Express (to explore MongoDB through GUI, locally hosted)

The environment variables for these are covered in their respective Docker Compose files (*docker-compose.dev.yaml* and *docker-compose.yaml*). Since we are running these instances locally for development and testing, these environment variables don't need to be replaced and work out of the box. The Judge0 API key environment variable, however, needs to be replaced in *frontend/src/services/JudgeAPI.ts*.

To run LD, run `make dev` in the project root. The frontend will then be accessible on *localhost:8080*. To run LP, run `make prod` in the project root. The frontend will then be accessible on *localhost:80 (port can be omitted)*. Since the databases are hosted locally, these images will take a while to build and start running. The services are also coupled with the databases as dependencies, and hence will only start running after the databases are up. This may require rerunning the services if one of the database deployments is slow to start up.

For security, all requests are checked against the authentication service for a valid JWT token.

<u>BMP</u>

When it came to deploying for users, we felt we could do better than running all the services dependent on building the databases, as that wasn't too different from a monolithic setup. We created Kubernetes configuration files for each of the services,

and shifted to the following cloud providers for our databases, message queue and for Judge0:

- MongoDB Atlas
- Neon.Tech (PostgreSQL)
- CloudAMQP (RabbitMQ)
- Judge0 Shared Cloud

This allowed us to reduce the number of services we manually ran to the first 7 in the list above. The environment variables for these are stored in their respective yaml files in the *k8s* folder. The Judge0 environment is placed within the frontend folder, under *src/services/JudgeAPI.ts.* Since we are using hosted versions of databases, these environment variables need to be replaced to use the desired hosted services. In the future, we will switch these to use Kubernetes secrets for greater security in deployment.

To run updated versions of services, their images need to be pushed to a Docker repository. These images can then be indicated in their respective yaml files in the *k8s* folder. The default configurations use our pushed images. Our Kubernetes configurations have an always pull policy to ensure the latest images are always used.

To run BMP, run `make pipeline` in the project root. This deploys the services, sets up a bare metal Nginx Ingress, and directs requests from port 8080 to the respective services. There is a preconfigured sleep delay between downloading the Nginx Ingress and deploying it, because of a slow startup time. This may be needed to return the Ingress setup if it fails, which can be done by running `make ingress` in the project root. The frontend will then be accessible on *localhost:8080*.

CLP

For deploying on GCP, we chose to use the native GKE Ingress to benefit from Google's Cloud Load Balancing, which allows us to run each service as a separate deployment and distributes these deployments across different geographic locations, running as a distributed service. The website accessible on trypeerprep.com (HTTP-only) uses this architecture.

Manual configuration is required to deploy and run CLP. In the future, we will use deployment scripts to automate this. After creating a cluster on Google Kubernetes Engine, cloud the repository and run `make cloud` in the project root. Once the services are live, create a GKE Ingress that directs to the frontend service. The yaml for it in Cloud Shell will contain this:

```
spec:
  defaultBackend:
    service:
      name: front-end-service
      port:
        number: 80
```

After setting up Cloud DNS for a Cloud Domain that directs to this GKE Ingress, add the following rules to the config (replacing the 2 items in square brackets):

```
spec:
  defaultBackend:
    service:
      name: front-end-service
      port:
        number: 80
  rules:
  - host: [Replace with your domain]
    http:
      paths:
      - backend:
          service:
            name: user-service
            port:
              number: 3000
        path: /api/v1/user/*
        pathType: ImplementationSpecific
      - backend:
          service:
            name: history-service
            port:
              number: 3008
        path: /api/v1/history/*
        pathType: ImplementationSpecific
      - backend:
          service:
            name: matching-service
            port:
              number: 8082
        path: /ws
```

```
        pathType: ImplementationSpecific
      - backend:
          service:
            name: collaboration-service
            port:
              number: 3005
        path: /api/v1/room/*
        pathType: ImplementationSpecific
      - backend:
          service:
            name: collab-ws-server-service
            port:
              number: 4444
        path: /collab/ws/*
        pathType: ImplementationSpecific
      - backend:
          service:
            name: question-service
            port:
              number: 8080
        path: /api/v1/questions/*
        pathType: ImplementationSpecific
status:
  loadBalancer:
    ingress:
    - ip: [Replace with your external IP]
```

About 10-15 minutes later when GKE updates the ingress, the platform will be running. For latency testing purposes, we deployed this in asia-australia1 instead of asia-southeast1. We are happy to say there is no noticeable delay from this, ensuring the platform can now serve a global audience.

Individual/Sub-group contributions

Subgroup	Name	Technical	Non-technical
1	Lee Shao Wee	User Service - Created user model - Built user controller and API	- Draw the system architecture diagrams

		routes for the following features: user register, deregister, login, logout, reset/change password, change name and user(s) retrieval - Setup JWT for user sessions <u>Matching Service</u> - Built upon the base template and added logic of matching and unmatching users based on difficulty level - Handle cancel match by users - Call the question service to retrieve a random question - Call the collaboration service to create room when matched <u>Collaboration Service</u> - Created room model - Built room controller and API routes for the following features: room creation, closing and retrieval <u>History Service</u> - Created attempts model - Built attempt, collaboration controller and API routes for the following features: attempt creation, retrieval and user's collaboration	- Created report template
1	Kelvin Chua	<u>Frontend</u> Developed most of the frontend - User authentication and authorisation flow - Registration - display of questions on the landing page - Searching and sorting of questions on the landing page - individual question display page - Question creation UI	Architecture diagrams

		- Users profile page (both current user and other users) - Statistics on user profile page - Previous submission history (N2) and collaboration history on user page (N2) - Password change UI - Collaboration page, including implementation of CRDT to synchronize chat (N1), code editor and user status - Implement code editor with syntax highlighting and formatting (N5) <u>Set up CRDT websocket server for synchronization between clients</u> <u>Helped with docker setup (setting up docker compose for development and deployment)</u> <u>Architected front end design</u> - Separation into service layer, and query layers - Usage of react query to manage state <u>Adhoc bug fixing and feature implementation for the backend services</u> - <u>Usage of JWT to determine current user for profile edits</u> <u>Set up basic unit testing</u> <u>Implementation of testing for user service</u>	
2	Kartikeya	<u>Question Service</u>	- Updated report

		- Created service from scratch, with CRUD endpoints to manage questions stored in the database - Created separate admin and user routers to ensure role-based access to administrative routes (creating, updating, deleting questions) - Created question model for consistency in the MongoDB instance - Configured updating of questions with Cloud Function every 24 hours - Documented API endpoints following the OpenAPI 3.0 specification, viewable through Swagger Docs <u>Matching Service</u> - Created service from scratch, with endpoints to check service health and upgrade to a WebSocket connection - Created message models to ensure consistency in WebSocket communication - Created WebSocket message parsing functions, to ensure only the right requests reach RabbitMQ - Connected with RabbitMQ to stream matching requests from users into Easy, Medium, Hard queues - Created matching algorithm using goroutines with a custom	template - Created template for documenting APIs

		time-based map data structure to ensure connections are upheld only for a short duration, and prevent unnecessary memory hogging <u>Cloud Function</u> - Created functions to fetch all question data from Leetcode's GraphQL API - Deployed Cloud Function on GCP <u>Deployment</u> - Created Kubernetes configuration files for elastic deployment - Set up external hosted services for a distributed environment - Created deployment pipelines in a Makefile to enable different deployments based on enclosing environments - Implemented GKE Ingress to route external requests to PeerPrep - Set up a Cloud Domain with Cloud DNS for a custom URL to redirect to PeerPrep - Manual testing to ensure deployment worked <u>Miscellaneous</u> - Created Makefile to simplify complex commands - Read through multiple PRs to check for possible bugs	
2	Peigeng	<u>Frontend</u> Developed parts of the frontend	- Some developer testing to check if

		- Implementation of frontend's matching service - Implementation of frontend's judge service - Implementation of frontend's history service to save code submissions - Update individual question display page to include code editor - Update individual question pages and collab page to include code execution - Update code editor to use a smaller set of commonly used languages - Implementation of code submission results display <u>Judge Service</u> - Implementation of code execution service using Judge0 image <u>Docker</u> - Setting up docker compose for development and production	services are working as intended

<u>Improvement Suggestions/Enhancement</u>

- Implementing separate databases for distinct services is crucial to prevent potential race conditions during database updates. The current configuration, where our user, collaboration, and history services share a common database, poses a risk of data inconsistencies. To enhance system reliability, it is recommended to adopt individual databases for each service. This approach minimizes the likelihood of conflicts, ensuring a more resilient and consistent data management system.
- The optimization of the message queue system in the matching service can be achieved by reducing the number of queues required. The current implementation results in a maximum queue length of 2 users, as every pair of matched users is promptly removed from the queue. This renders the resources

allocated for creating the queue inefficient and wasteful. Streamlining the queue creation process to align with the specific needs of the service, where a maximum of two users can be accommodated, would lead to more efficient resource utilization.
- Implement refresh tokens to refresh the JWT as now they expire within an hour and there is no way to refresh it beyond logging in again.

Reflections and Learning Points

- Employing a microservice architecture approach streamlines team collaboration by enabling clear assignment of tasks to individual members.
- Docker is a powerful tool for deployment and rapid prototyping of services and frontend.
- There are a lot of open source projects that are available and deployable on Docker, which can be useful for prototyping or implementing services that are tricky to implement such as code executors.
- Diagrams are paramount as they serve as a vital tool for comprehending the architecture before embarking on the implementation process.
- Microservices allow for rapid scaling, but bring with them a lot of complexity that must be managed.
- Deployment on GCP was done towards the end, but we paid attention to decouple the services early on such that GCP deployment was seamlessly configured in a few days.