CS3219 Software Engineering Principles and Patterns

Final Report

Team G20

Name	Matriculation Number
Aiken Wong Xiheng	A0217784U
Bryan Kwok Yan Fai	A0217657W
Bryann Yeap Kok Keong	A0217687N
Chow En Rong	A0222834J
Toh Wei Jie	A0223589U

Table of Contents

1. Introduction

1.1 Background

As a person preparing for technical interviews in a tech-related role, it can be a difficult and daunting task when doing it alone. To better prepare for such interviews, it is beneficial for someone to be able to partner with fellow programmers and practise problem sets together. This would allow them to possibly learn a different method of solving the current question and gain new perspectives on how to solve similar problems in the future. Furthermore, pair programming can help one solve more challenging problems, helping them to build up confidence for an actual interview.

1.2 Purpose

PeerPrep aims to be a platform for those who are looking to prepare for technical interviews with peers, allowing them to practise whiteboard-style interview questions together.

1.3 Project Scope

PeerPrep provides a diverse bank of technical interview questions to try from, and a peer matching system to enable a pair of users to work on a given question together in a collaborative workspace.

The platform will also allow people to create an account as a normal user, while users granted the maintainer role will be able to perform maintenance duties. Apart from being able to access and test the features of the website as a regular user, maintainers are also able to add more questions, or edit the current bank of questions to improve the website.

Finally, PeerPrep keeps track of the questions that you have attempted in your lifetime, and allows you to continue working upon that attempt again anytime that you wish.

1.4 Target Audience

PeerPrep is aimed at people who wish to practise their technical interviews on a whiteboard style interview platform, while collaborating with another person, be it to get help or to learn different approaches for various questions from the matched peer on the platform.

2. Functional Requirements

High priority functional requirements are features that are deemed to be most essential for users to accomplish the primary tasks of the application. Functional requirements that are of medium or low priority are features that help fulfil secondary user tasks (user tasks that are less common), or act as supplements for a higher priority feature.

Functional requirements that have many other functional requirements dependent on them are done in earlier sprints. This is to allow smoother development and integration of newer features.

2.1 Question Service

The question service is responsible for storing, retrieving and updating question information.

ID	Description	Priority	Sprint
1	Users should be able to view the question title, description, category, difficulty of the questions.	High	Sprint 1
2	Users with the normal role should be able to read questions, but not create, update or delete them.	High	Sprint 1
3	Users with the maintainer role should be able to create, read, update, delete questions.	High	Sprint 2
4	Unauthenticated users should not be able to create, read, update or delete questions.	High	Sprint 2

2.2 User Service

The user service is responsible for storing, retrieving and updating user information, as well as authentication and authorization.

ID	Description	Priority	Sprint
1	Users should be able to register for a new account.	High	Sprint 1
2	Users should be able to login and logout of their account.	High	Sprint 1
3	Users should be able to view their own profile.	High	Sprint 1
4	Users should be able to update their own profile.	High	Sprint 1
5	Users should be able to deregister their profile.	High	Sprint 1

2.3 Matching Service

The matching service is responsible for letting users find another user to do a question with, based on certain matching parameters (e.g. question difficulty).

ID	Description	Priority	Sprint
1	Two users should be matched by the difficulty level of the question they want to work on.	High	Sprint 2
2	Users that join the matching queue will timeout (i.e. removed from the matching queue) if no match is found within 30 seconds, and will be allowed to retry or cancel the matching.	High	Sprint 2
3	Upon successful matching with another user, the user will be directed to the collaboration workspace to start the collaborative session.	High	Sprint 2
4	Users should be shown a 'waiting timer' animation while waiting to be matched.	High	Sprint 2
5	Users should be able to exit from the matching queue while they are looking for a match but cannot find one	Medium	Sprint 2

2.4 Collaboration Service

The collaboration service is responsible for managing collaboration sessions between users.

ID	Description	Priority	Sprint
1	Users should be able to see the question they are working on in the collaborative workspace.	High	Sprint 2
2	Users should be able to write their programs concurrently with the matched peer in the collaborative workspace.	High	Sprint 2
3	The collaboration workspace editor should support the use of JavaScript and TypeScript.	High	Sprint 2
4	When a user changes languages in the editor, the matched peer's editor language should also reflect this change.	High	Sprint 2
5	When collaborating in different languages, the editor state should be stored independently for each language and reinstated when the corresponding language is toggled to.	High	Sprint 2
6	**N3:** Users should be able to execute their programs in a sandboxed environment and present the results inside the collaboration workspace.	High	Sprint 2
7	**N5:** Users should have access to code formatting and syntax highlighting for multiple languages in the collaboration workspace.	High	Sprint 2
8	**N2:** Users should be able to view the questions they have attempted, along with other information like date attempted, and the attempt code itself from the user's profile page.	Medium	Sprint 3
9	Users should be able to check the status of all of their collaborative attempts.	Medium	Sprint 3
10	Users should be able to rejoin the collaborative session if it is still ongoing.	Medium	Sprint 3

11	Users should be able to review their own attempt if the collaborative session is closed, and work on it in a non-collaborative code editor.	Medium	Sprint 3
12	Users should be able to check their attempts for the same collaborative session in different languages.	Medium	Sprint 3
13	Users' edits to past attempts' code should not be persisted in the database	Medium	Sprint 3
14	The state of users' edits to their past attempts' code should not be persisted when toggling language changes in the non-collaborative code editor	Medium	Sprint 3

2.5 Chatbot Service

The chatbot service is responsible for managing user queries to and responses from generative AI.

ID	Description	Priority	Sprint
1	**N7:** Users should be able to query LLMs to seek an explanation of code written by the matched peer in a collaborative session.	High	Sprint 3
2	Users should be able to view their chat history for a collaborative session	Medium	Sprint 3
3	The chatbot should be aware of the user's chat history and question context, and reply with reference to such context	Medium	Sprint 3

3. Non-Functional Requirements

Below are non-functional requirements that have been fulfilled by our team.

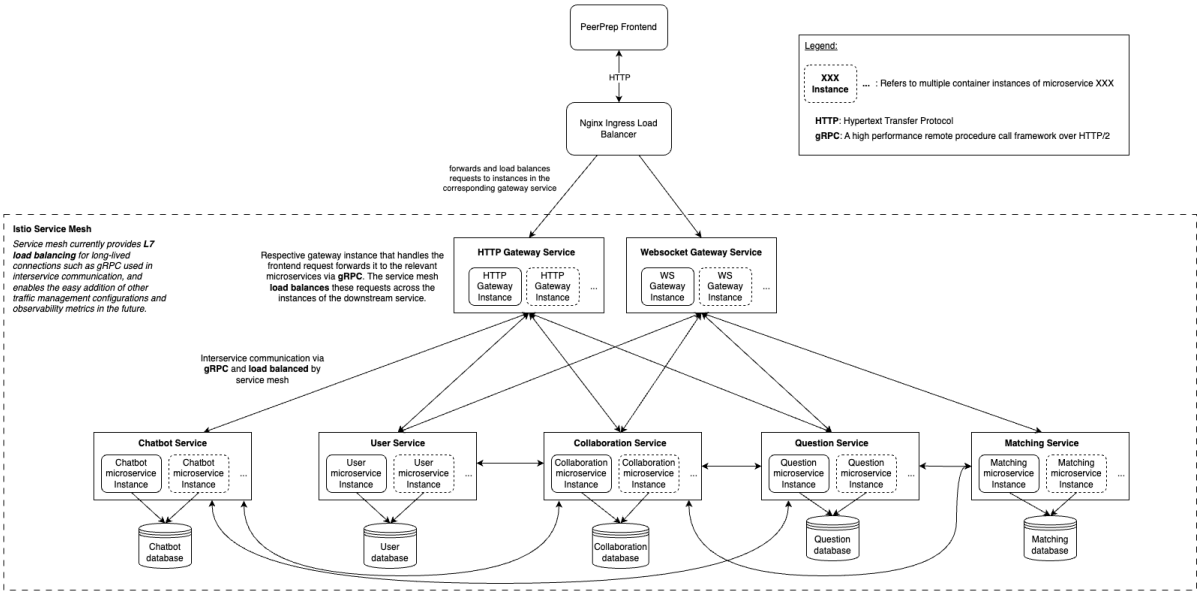
Non-functional requirements are ranked in priority in terms of their level of impact on their respective stakeholder's satisfaction or security.

ID	Description	Priority	Attribute
1	New developers should have access to documentation to set up the project locally.	Medium	Documentation
2	The web application should work on all common Operating Systems (i.e. Windows, macOS, Linux).	High	Portability
3	The web application should be compatible with major web browsers.	High	Portability
4	The web application should be able to support up to 500 different questions.	High	Performance (Data capacity)
5	Starting a collaboration session should start the collaborative 'workspace' in less than 5 seconds.	Medium	Performance (Response time)
6	The application frontend should be a Single Page Application.	Medium	Performance (Response time)
7	Users’ ‘logged in’ status should be persisted through browser page refreshes, and during navigation to other pages within the application frontend.	High	Usability
8	The application should be responsive (provide a mobile interface and a desktop interface).	Medium	Usability
9	The application should present an intuitive	Medium	Usability

	interface for users to match and collaborate with each other.		
10	The application should use an authentication token (JWT token) when accessing protected API endpoints.	Medium	Security
11	The application should use HTTPS and WSS protocols for all communication between the deployed application frontend and backend.	High	Security
12	The application should use OAuth2.0 protocol for authenticating users.	High	Security
13	Websocket connections to the application backend should be authenticated with a valid single-use websocket ticket generated by the backend. Connections that are unable to be authenticated will be terminated and closed from the backend.	Medium	Security
14	The application should be able to automatically scale horizontally when application load is high.	Medium	Scalability
15	API requests and interservice communication should be load balanced between all available instances of the corresponding microservice.	Medium	Scalability

4. Architecture

4.1 Architecture Diagram



Our architecture for Peerprep consists of the following main components:

- Frontend
- Backend
    - HTTP Gateway
    - Websocket Gateway
    - Question Microservice
    - User Microservice
    - Matching Microservice
    - Collaboration Microservice
    - Chatbot Microservice

Each backend microservice that requires persistent storage contains its own separate database as per the *Database per service* pattern, storing only data relevant to its responsibilities.

The frontend communicates through REST APIs and websockets with the backend. Requests first go through a load balancer which distributes the type of request to the corresponding instances of the HTTP Gateway or Websocket gateway. The gateway instances then facilitate the transport of requests to the appropriate microservices through interservice gRPC communication.

4.2 Architecture Considerations

4.2.1 Microservice VS Monolithic

In monolithic architectures, the application is one single indivisible unit that is often under one codebase. In contrast, a microservice architecture splits the application into different components, defined according to the bounded contexts of the problem at hand.

The following table provides a brief comparison between the 2 architecture paradigms.

	Monolithic	**Microservice**
Debugging and testing	Easier to debug and test, requiring less coordination	Harder to debug and test, since each service can be developed independently and requires coordination. Requests may call multiple downstream microservices, making things harder to debug.
Level of coupling	More coupled components; changing one might lead to unintentional effects on other parts of the app	Less coupled components; each service can be deployed, developed, operated, scaled without affecting the functioning of other services since there is separation of concerns
Deployment complexity	Simpler deployment process	More difficult to deploy, with multiple services to manage and configure
Scalability	Only the whole application can be scaled	Different parts of the application that are responsible for different bounded contexts can be

		scaled independently
Choice of software	More limited; entire application is built on a set of tools and frameworks	More flexible and customizable; each service can choose which technology stack it wants to use as services communicate through an API interface
Development process	Due to tighter coupling, development of one component of the application could depend on another, leading to a more restrictive development process	Services can be developed independently in parallel with less friction and merge conflicts

Our team decided to move forward with a microservice architecture for Peerprep since there are many bounded contexts that the problem can be divided into, thus allowing us to specify different microservices to address each sub-problem while developing them independently and in tandem. This is essential, considering the short timeframe of one semester and a small development team. It is thus imperative that everyone is able to contribute productively to the final application, and being able to develop services independently is a critical enabler which allows each subteam to separate responsibilities and develop in parallel in a loosely coupled manner.

4.2.2 Interservice Communication

For interservice communication, our team conducted a thorough examination of the following 3 communication transport protocols: TCP, RabbitMQ (RMQ) and gRPC. We kicked off the project using TCP, before shifting to RMQ midway and ending with a complete migration to gRPC. While maximising our learning of the various protocols taught in lecture was one of our key considerations for trying out these many protocols in the project, the rest of this section will discuss the various considerations that our team had, and the rationale for switching protocols twice.

We initially used TCP as our main transport protocol, as it was the default standard protocol for NestJS microservice communication. It is a low-level, reliable, and secure protocol for

transmitting data between applications under a *request-response* paradigm. It provides a straightforward method for sending and receiving data, and is best for low-volume and simple interservice communication between microservices.

However, our team quickly ran into scalability issues while using TCP for interservice communication. As TCP connections are persistent, when an instance of a microservice ‘A’ connects to an instance of another microservice ‘B’, all subsequent requests between these two microservices will be made through this persistent TCP connection, effectively not distributing the load to other instances of microservice ‘B’, rendering them idling.

To address the above, we decided to make load balancing inherent in our architecture, and swapped out the interservice transport protocol to RMQ. RMQ sets up a message broker where all instances of the various microservices subscribe to a particular queue corresponding to each microservice type. Since multiple instances of a particular microservice can consume from the same queue, this would enable us to seamlessly scale our microservices while load balancing requests to each microservice instance (i.e. consumer) in a round-robin fashion as configured in RMQ. In fact, by making all our microservices subscribe to and post messages on the relevant queues in RMQ, we effectively removed the need for service discovery. The usage of a message broker for interservice communication also decreased coupling further, while ensuring reliable asynchronous message delivery and improving the fault tolerance of our application by storing messages in the messages for eventual delivery should a downstream microservice go down.

While the benefits above are numerous, our team eventually decided that using RMQ for asynchronous interservice communication may be somewhat inappropriate for our application where the majority, if not all, our interservice communications are of the request-response paradigm with simple fetch calls and short response times. This was an atypical use case for RMQ, which is usually used for Event-driven messaging, and was a code smell.

Upon further research into scaling long-lived interservice connections that use the synchronous *request-response* paradigm, we came to learn that a widely-accepted way to scale such persistent interservice communication setups was to use a service mesh to perform intelligent load balancing. This will be discussed in detail in the next section on Load Balancing. A service mesh could be used in tandem with gRPC, which establishes a long-lived connection between the relevant microservices, and is a common protocol used in

interservice communication today. The gRPC protocol provides significantly better upgrades from TCP. It utilizes strongly typed contracts in the form of protocol buffers to serialize and deserialize data during transmission, enforcing data types sent and ensuring that the upstream and downstream services agree on the same interface. As gRPC protocol buffers are language agnostic, gRPC protocol buffers to TypeScript interface generation tools greatly simplified the writing of interfaces within the application backend itself. Compared to the standard HTTP REST + JSON communication that most are used to, there is higher performance with significantly smaller message sizes. Since we were trying to choose an internal communication protocol, speed was more of a concern than data being in a commonly used and readable format like JSON. The performance and extensibility of the gRPC protocol was worth the additional development effort, and opens up the application to the possibility of supporting data streaming should the need arise.

4.2.3 Load Balancing

This subsection will now briefly expound on the load balancing techniques used in the application.

When using the TCP and gRPC protocols for interservice communication, our team initially experienced a ‘quirk’ in our Kubernetes cluster where all our interservice API calls from a particular instance of microservice ‘A’ are served by the same instance of the downstream microservice ‘B’, even though there are multiple instances of microservice ‘B’ available. Upon further debugging, we found out that this was because TCP and gRPC connections open a long-lived connection between the upstream and downstream service, resulting in all subsequent interservice service calls from this instance of upstream microservice ‘A’ to be made to the same instance of microservice ‘B’.

Further investigation revealed that the Kubernetes *Service*, which abstracts over all the pods in a Deployment, only provides L3 load balancing. In other words, it provides ‘load balancing’ at the IP table level. When our instance of upstream microservice ‘A’ first initiates a connection to microservice ‘B’, the request is routed to an instance of microservice ‘B’ with equal probability. The long-lived connection is then established and all subsequent communication goes through this persistent connection, which was not the load balancing we expected nor wanted, where requests are distributed evenly across all instances of the downstream microservice.

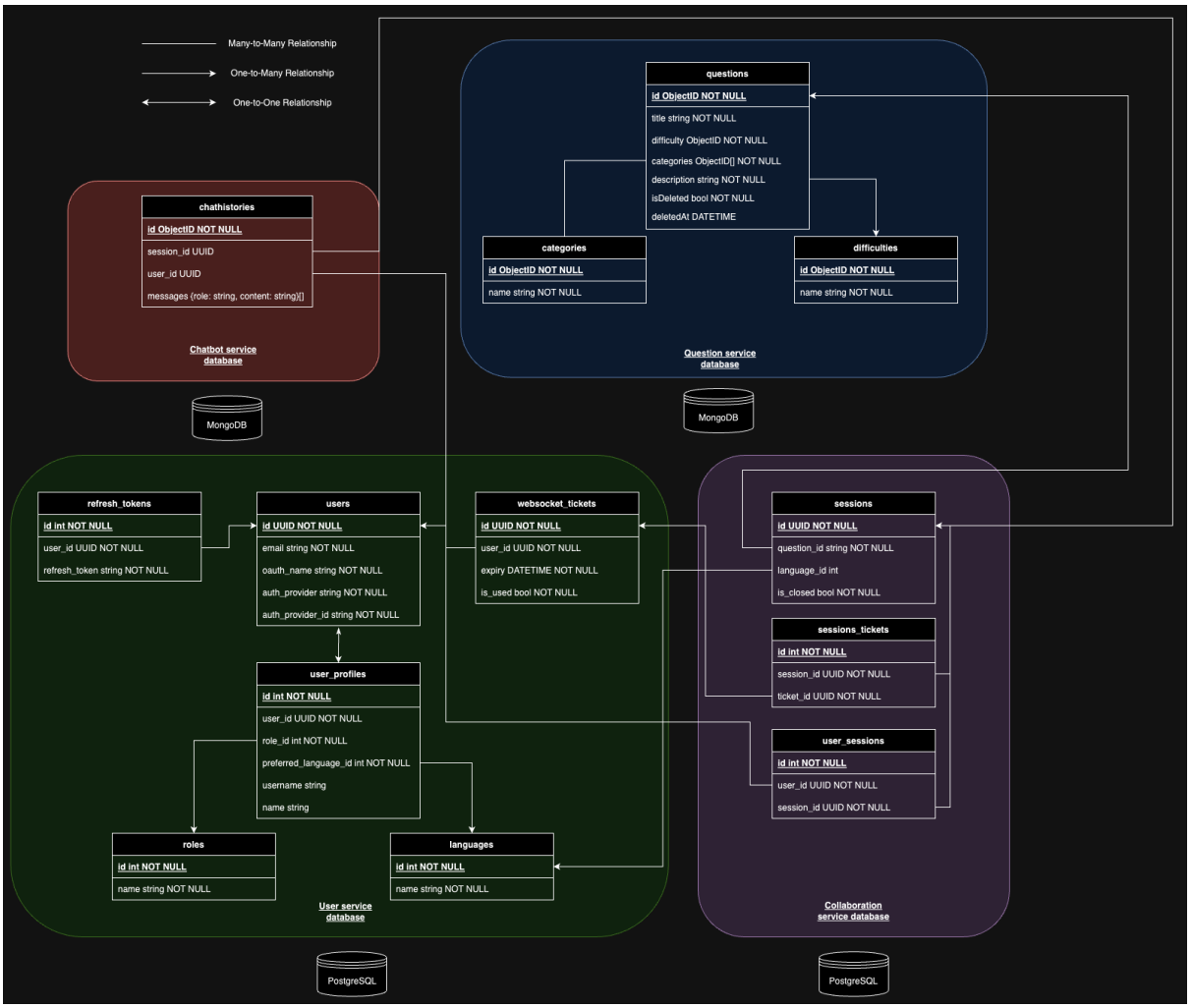
To properly scale our application horizontally, we needed to perform L7 load balancing at the application layer. This is typically implemented in the form of service meshes, which adds another layer on top of the interservice communication with the introduction of a sidecar container as per the Kubernetes sidecar pattern. Service meshes work in tandem with service registries to do service discovery, and perform intelligent load balancing by inspecting and distributing requests across all instances of the downstream microservice. Additionally, service meshes offer other important functionalities such as added observability (through the three pillars of observability: metrics, traces and access logs, by inspecting all interservice communication), and performing security related tasks such as implementing mutual TLS (mTLS) to establish secure connections between our microservices. However, this comes with the downside of adding significant resource overhead by needing to attach a sidecar to every pod in the deployment.

Our team eventually decided to use the Istio service mesh with a basic configuration to get L7 load balancing in our distributed production environment to distribute requests evenly to all instances of downstream services when performing interservice communication. While the current setup is simple, the addition of the service mesh paves the way for us to eventually introduce better observability and transport layer security between our services should business needs evolve to warrant so.

4.2.4 HTTP Gateway and Websocket Gateway

To achieve better separation of concerns in our API gateway, our team decided to split up the API gateway into a HTTP gateway and Websocket gateway, where the HTTP gateway exposes our REST API and Websocket gateway handles websocket related connections. This architecture also enables us to scale the gateways independently when the need arises, while achieving better SRP.

4.3 Entity-Relation Diagram



**Note that every table row in PostgreSQL has timestamps for creation and last updated times*

5. Design

5.1 Design Patterns

5.1.1 Pub-sub Messaging Pattern

We decided to make use of Redis together with the native WebSocket API as a pub-sub mechanism to allow for bi-directional, low latency communication between the frontend and the backend.

WebSockets provide a convenient way for bi-directional communication between the frontend and the backend, which is required for the matching and collaboration services. For example, in the matching service, when users want to find a match, a message is sent from the frontend to inform the backend. The backend can then respond to this message to find a match. Should the backend find a match, it then fires a message back to the frontend stating that a match has been found, along with the details of this match. The frontend can then respond to this message accordingly (i.e. start a collaboration session).

```
const handleMatchFound = (sessionId: string) => {
  handleCancelSearch();
  setSearchStatus(matchFound);
  enqueueSnackbar(matchFound, { variant: 'success' });
  navigate(`/session/${sessionId}`);
};

const handleWebsocketOpen = () => {
  sendWsMessage(ws, { questionDifficulty: difficulty._id }, 'get_match');
  ws.onmessage = (message) => {
    const data = JSON.parse(message.data);
    if (data.event === 'match') {
      handleMatchFound(data.data.sessionId);
    }
  };
};

// Functions to handle the searching and cancelling of searching for a partner
const handleSearch = async () => {
  ws = await getMatchingWebSocket();

  ws.onopen = handleWebsocketOpen;

  setIsSearching(true);
  setSearchStatus(searching);
};
```

Frontend code requesting a match and responding to a match

```ts
@SubscribeMessage('get_match')
async getMatch(
  @MessageBody() data: MatchingDto,
  @ConnectedSocket() connection: AuthenticatedWebsocket,
) {
  // wait for ticket to be set, or connection to close
  while (
    connection.ticket === undefined &&
    connection.readyState !== connection.CLOSED
  ) {
    await new Promise((resolve) => setTimeout(resolve, 100));
  }

  if (connection.readyState === connection.CLOSED) {
    return;
  }

  const { userId } = connection.ticket;
  this.websocketMemoryService.addConnection(userId, connection);
  await lastValueFrom(
    this.matchingService.requestMatch({
      userId,
      questionDifficulty: data.questionDifficulty,
    }),
  );
}
```

```ts
[matchingEntry, possibleMatch].forEach((entry) => {
  this.webSocketClient.emit(
    WebsocketServiceApi.EMIT_TO_USER_AND_DELETE_WEBSOCKET,
    {
      userId: entry.userId,
      event: 'match',
      payload: { sessionId: collabSessionId },
    },
  );
});
```

Backend code responding to frontend request and sending a message if a match is found

Within the backend, Redis is used as a pub-sub mechanism to quickly respond to changes in the collaboration session among multiple instances of API gateways should the application scale up in that way. For instance, when a user first starts a collaboration session, an instance of the API gateway will hold the websocket connection for the user. When there is a language change, this message goes to a random instance of the gateway, which will publish a language change event. Whichever gateway instance holds the user's

current websocket in their current session will thus respond appropriately and send a message back to the user.

Alternatives considered

We have considered Kafka as an alternative for publishing and subscribing to messages. Below are some of the considerations between the 2:

	Redis (Pub-Sub)	**Kafka**
Size of data	Able to handle relatively smaller amounts of data before latency is affected noticeably	Able to handle relatively larger amounts of data (even up to 1GB) before latency is affected noticeably
Latency	Lower due to it being in-memory	Higher
Reliability of message delivery	Unreliable	Has automatic failover, but can be configured to provide unreliable message delivery
Durability	Not durable (in-memory)	Durable (disk based)
Scalability	Increase number of nodes in a cluster to increase broker concurrency	Enabled by consumer groups and topic partitions, which enable very high consumer concurrency backed by broker concurrency (multiple nodes and partitions).
Throughput	Relatively lower	Relatively higher
Dependencies / Complexity of deployment	Simpler deployment as we already have Redis service running for in-memory storage for Matching Service	Relatively more complex, as we also need to set up and configure Zookeeper service, on top of Kafka

In summary, Redis offers lower latency due to it being in-memory, and is designed to be used in cases where there are low numbers of subscribers.

We decided to go with Redis for our use case, since it offered a simpler and faster solution for performing Pub-sub interservice communication. Since we are only dealing with small messages to represent events like "language change" and "match found" events, we also do not need the higher bandwidth that Kafka offers. Furthermore our Pub-sub architecture was relatively simple, and Redis required fewer configurations out of the box to fulfil it. Kafka however, was designed for more complicated Pub-sub architectures, with concepts like Topics and Partitions that we did not need to deal with.

5.1.2 Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Our team decided to employ factory methods when creating some of our modules, because we wanted to standardise some common details across our microservices. For instance, our microservices using SQL databases would use a PostgreSQL database with similar configuration options like hostname, username, password, etc.. Thus, we created a factory method for creating an SQL database module that would help us connect to the database using the options we provide.

This allows us to standardise configuration options and not have to repeatedly pass in the same arguments options repeatedly and do them from a selected point within the code, enforcing DRY principles.

The code below is an example of how this is done. The factory method creates an instance of SqlDatabaseModule with the appropriate options passed through via a configuration service.

```
export class SqlDatabaseModule {
  static factory(
    models: (typeof BaseModel | typeof BaseModelUUID)[],
  ): DynamicModule {
    const modelProviders = models.map((model) => ({
      provide: model.name,
      useValue: model,
    }));

    return {
      module: SqlDatabaseModule,
      providers: [
        ...modelProviders,
        {
          provide: 'KnexConnection',
          inject: [ConfigService],
          useFactory: async (configService: ConfigService) => {
            const databaseOptions = configService.get(
              'databaseConfigurationOptions',
            );
            if (databaseOptions === undefined) {
              throw Error(
                'Database configuration not specified in ConfigModule!',
              );
            }
```

Example factory created so that microservices using the same configuration for accessing SQL databases do not repeat configuration code

5.1.3 Dependency Inversion

In order to decouple the high-level modules from the low-level modules, we made use of dependency inversion. Interfaces were created to define methods that were implemented by controllers in each microservice, so that the gateway knows what methods it can call by accessing these interfaces. Resultantly, the low-level implementation details of the service classes can be changed without affecting the high-level gateway, since the methods called are still the same.

Below is an example of the chatbot microservice controller implementing an interface, so that the gateway can depend on the abstraction instead.

```
Aiken Wong, 3 weeks ago | 1 author (Aiken Wong)
@Controller()
@ChatbotServiceControllerMethods()
export class ChatbotController implements ChatbotServiceController {
  constructor(private readonly chatbotService: ChatbotService) {}
```

ChatbotController implementing ChatbotServiceController interface

```
Aiken Wong, 3 weeks ago | 1 author (Aiken Wong)
export interface ChatbotServiceController {
  createChatQuery(
    request: ChatbotRequest,
  ):
    | Promise<ChatMessagesResponse>
    | Observable<ChatMessagesResponse>
    | ChatMessagesResponse;

  getChatHistory(
    request: ChatHistoryRequest,
  ):
    | Promise<ChatMessagesResponse>
    | Observable<ChatMessagesResponse>
    | ChatMessagesResponse;
}
```

ChatbotServiceController interface

```
@Post('query/:sessionId')
getSessionAndWsTicket(@Req() req, @Param('sessionId') sessionId) {
  const body = req.body;
  const { query, language, userSolution } = body;
  const userId = req.user.id;

  if (query !== null && language !== null) {
    return this.chatbotService.createChatQuery({
      sessionId,
      query,
      userId,
      language,
      userSolution,
    });
  }
}

@Get('history/:sessionId')
getChatHistory(@Req() req, @Param('sessionId') sessionId) {
  const userId = req.user.id;

  if (sessionId !== null) {
    return this.chatbotService.getChatHistory({ sessionId, userId });
  }
}
```

Gateway dependending on the interface

5.1.4 Layered Architecture

In our application, we have implemented the Layered Architecture pattern to handle requests. At the forefront of our backend architecture, we made use of a Data Transfer Object (DTO) layer to validate incoming request JSON object fields, ensuring data integrity and adherence to defined specifications. The controller layer then acts as the interface between the presentation layer and the underlying services, strictly limiting its role to calling the service layer and managing the flow of information. The service layer, in turn, plays a pivotal role by orchestrating business logic, calling upon other services or DAOs (Data Access Objects) as necessary. Notably, the DAO layer abstracts over the database layer, making our application agnostic to the choice of database and ORM (Object Relational Mapping). This modular structure enhances code maintainability and contributes to overall modularity of our application.

```ts
export default class RefreshDto {
  @IsString()
  @IsNotEmpty()
  refreshToken: string;
}
```

Example DTO for a refresh token

```ts
@Public()
@Post('refresh')
async refreshTokenFlow(@Body() body: RefreshDto) {
  return this.userAuthService.generateJwtsFromRefreshToken({
    refreshToken: body.refreshToken,
  });
}
```

POST endpoint that checks that data received conforms to the RefreshDTO

```ts
@Controller()
@UserAuthServiceControllerMethods()
export class AuthController implements UserAuthServiceController {
  constructor(private readonly authService: AuthService) {}

  generateJwts(user: User) {
    return this.authService.generateJwts(user);
  }

  generateJwtsFromRefreshToken({ refreshToken }: RefreshTokenRequest) {
    return this.authService.generateJwtsFromRefreshToken(refreshToken);
  }
```

Controller layer that calls the service layer

```
@Injectable()
export class AuthService {
  private tokenConfig: JwtTokenConfig;

  constructor(
    private readonly jwtService: NestJwtService,
    private readonly configService: ConfigService,
    private readonly refreshTokensDaoService: RefreshTokensDaoService,
    private readonly userDaoService: UserDaoService,
    private readonly websocketTicketDaoService: WebsocketTicketDaoService,
  ) {
    this.tokenConfig = configService.get('jwtTokenConfig');
  }

  private generateAccessToken(payload: JwtPayload) {…
  }

  private generateRefreshToken(payload: JwtPayload) {…
  }

  private verifyRefreshToken(refreshToken: string): JwtPayload {…
  }

  /**…
  async generateJwts(user, oldToken?: string) {
    const payload: JwtPayload = { id: user.id };
    const accessToken: string = this.generateAccessToken(payload);
    const refreshToken: string = this.generateRefreshToken(payload);

    if (oldToken) {
      await this.refreshTokensDaoService.patchByRefreshToken(
        oldToken,
        refreshToken,
      );
    } else {
      await this.refreshTokensDaoService.create(user.id, refreshToken);
    }
```

Service layer that manages flow of information

```
@Injectable()
export class RefreshTokensDaoService {
  constructor(
    @Inject('RefreshTokenModel')
    private refreshTokenModel: ModelClass<RefreshTokenModel>,
  ) {}

  create(userId: string, refreshToken: string) {
    return this.refreshTokenModel.query().insert({ userId, refreshToken });
  }

  patchByRefreshToken(oldRefreshToken: string, newRefreshToken: string) {
    return this.refreshTokenModel
      .query()
      .patch({ refreshToken: newRefreshToken })
      .where({ refreshToken: oldRefreshToken });
  }

  findByUserId(userId: string) {
    return this.refreshTokenModel.query().select().where({ userId });
  }
}
```

Example DAO for modifying refresh tokens in the database

5.2 Design Considerations

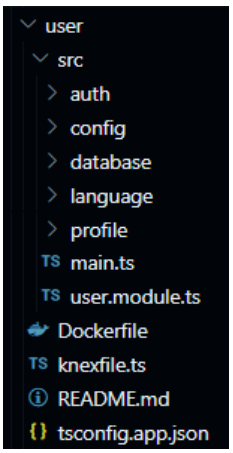
5.2.1 NestJS

In order to create a scalable distributed application that is maintainable, we made use of the NestJS, which is an enterprise-ready server-side NodeJS framework to build our backend. NestJS also enables us to fulfil many of our design considerations as follows.

5.2.1a Modularization

A NestJS application is usually divided into modules, with each module containing its own controllers, services, configurations, schemas, and other elements, serving its own functionality. This modularity makes the application more maintainable and extensible by splitting the system up into differing levels of dependence, and concealing the complexities of each part behind abstractions. Moreover, with modularization, that means that there is a strict boundary for each service and to access the services of a module, it must be done via

interaction with a well-defined interface. As a result, we are improving cohesion and we also manage the coupling of the modules.



Example of modularity in user microservice

5.2.1b Dependency Injection

Dependency injection is used to make a class independent from its dependencies, thus encouraging a loosely coupled program. It relieves code modules from the responsibility of instantiating their own dependencies, and instead allows such dependencies to be swapped easily in the constructor. NestJS has built-in decorators that makes it easy to inject dependencies into controllers, services, and other modules, thus promoting the use of dependency injection.

```
You, yesterday | 3 authors (En Rong and others)
@Injectable()
export class CollaborationService implements OnModuleInit {
  private userAuthService: UserAuthServiceClient;
  private questionService: QuestionServiceClient;
  private userProfileService: UserProfileServiceClient;
  private languageService: UserLanguageServiceClient;
  private mdb: MongodbPersistence;

  constructor(
    @Inject(Service.USER_SERVICE)
    private readonly userServiceClient: ClientGrpc,
    @Inject(Service.QUESTION_SERVICE)
    private readonly questionServiceClient: ClientGrpc,
```

Use of Dependency Injection using NestJS common decorators

5.2.2 Object-Relational Mapping (ORM) Integration

ORMs bridge the gap between object-oriented programs and databases. It is an additional layer of abstraction over our databases that allows our microservices to interact with the database through their provided interfaces. This enables us to query our databases directly using the programming language that the backend is written with.

ORMs also allow us to create models and specify relations between them based on the actual persisted model in the database as per the Entity-Relation Diagram above, greatly simplifying development time and ensuring that our application models match those found in the database at every point of development. The creation of such ORM models also enables us to easily perform a ‘graph fetch’ of related entities without writing the database ‘join’ queries ourselves.

Without ORMs, one would have to write SQL statements (for relational databases) or NoSQL queries from scratch, which can be much more complex than using ORMs. Additionally, by limiting the use of raw SQL statements, ORMs protect the application against vulnerabilities such as SQL injections. There are undoubtedly instances where writing raw SQL statements might be better, such as cases when complex ORM join queries are generated and written inefficiently. However, optimization is not currently a high priority in our application, and thus we deemed it acceptable to make that tradeoff. The two ORMs used in our application are Objection.js for PostgreSQL and Mongoose for MongoDB.

Example of Objection.js queries

Example of Mongoose queries

5.2.4 Sandboxing of Code Execution in Frontend

The decision to sandbox code execution in the frontend rather than the backend was a strategic choice driven by a few reasons. Firstly, relying on external APIs like Judge0 posed limitations due to their constrained daily rates or high costs, making it an impractical solution for our application. Even the prospect of a self-hosted Judge0 instance raised concerns about managing scaling and deployment for our backend infrastructure. Given that the code execution feature was deemed a low-priority element in the context of our MVP, we opted for a pragmatic approach. Running code execution in the frontend within a separate worker thread allowed us to support two languages, TypeScript and JavaScript, aligning with our initial goal of providing basic multi-language support without the need to handle a multitude

of languages from the outset. Moreover, by isolating code execution in a separate worker thread, we achieved enhanced control and security. The ability to set a timeout for code execution mitigates the risk of prolonged execution times, ensuring a responsive user experience. Additionally, isolating the code execution environment in the frontend enables us to secure it further by restricting access to certain browser functions and variables to perform dangerous operations. This approach strikes a balance between functionality and resource efficiency, aligning with the priorities of an MVP development phase.

5.2.5 Websocket Authentication

Websockets do not come out-of-the-box with authentication protocols that check who can access them. As such, we decided to implement our own authentication flow for websockets. For a user to connect to a websocket in our application, they must first obtain a ticket. This ticket is a one-time-use ticket, and only authenticated users can access the endpoint needed to obtain a ticket. Users will then use this ticket to connect to a websocket, in which the ticket will be consumed. Without a valid ticket, users will not be able to connect to a websocket.

```ts
/**
 * Authenticates incoming websocket connection with accompanying connection ticket.
 *
 * `handleConnection` returns true if connection is successful, else closes the connection
 * and returns false.
 */
async handleConnection(
  connection: AuthenticatedWebsocket,
  request: Request,
): Promise<boolean> {
  const ticketId = BaseWebsocketGateway.getTicketIdFromUrl(request);

  if (!ticketId) {
    return BaseWebsocketGateway.closeConnection(
      connection,
      BaseWebsocketGateway.UNAUTHORIZED_MESSAGE,
    );
  }

  const ticket = await firstValueFrom(
    this.userAuthService.consumeWebsocketTicket({
      id: ticketId,
    }),
  ).catch(() => null);

  if (!ticket) {
    return BaseWebsocketGateway.closeConnection(
      connection,
      BaseWebsocketGateway.UNAUTHORIZED_MESSAGE,
    );
  }

  connection.ticket = ticket;
  connection.onerror = (event) => {
    console.error(event);
    throw new WsException(event);
  };
  return true;
}
```

Code showing how the websocket gateway verifies a connection using a ticket

Alternatives considered

Socket.io is a library that supports authenticated websockets through authentication headers. However, we are limited by our choice of Y.js to manage collaborative documents, which does not support Socket.io. Using Socket.io for the matching service and implementing our own authentication for the collaboration service will lead to 2 different kinds of websockets that the team will have to manage, making development more complex in general. Developing our own custom authentication protocol to be used across all

websockets in our application allows us to not repeat code for authentication in websockets, and creates a reusable component for future use should the team need it.

6. Technology Stack

Purpose	**Technologies**
Frontend	React, Material-UI, Monaco Editor, Vite
Backend	NestJS
E2E Testing	Jest
Database	PostgreSQL, MongoDB
Database ORMs	Knex (for PostgreSQL) and Mongoose (for MongoDB)
Project Management	Github Projects
CI/CD	Github Actions
Collaborative Document Editing	Yjs
Pub-Sub Messaging + In-memory database	Redis
Deployment	Cloudflare Pages, Google Cloud Platform
Containerization	Docker
Container Orchestration	Kubernetes
Service Mesh for Service Discovery and L7 load balancing	Istio
Load balancer/Ingress Controller	NGINX
Docker Image Repository	DockerHub

7. DevOps

7.1 CI/CD

This section will give an overview of the various aspects of our CI/CD pipeline.

7.1.1 Pre-commit Hooks

Pre-commit hooks are scripts that are run when the developer tries to commit a file locally. For NodeJS, we used the Husky package to write these pre-commit hooks. Within these hooks, we run our linters and formatters to ensure that every developer on the team only commits code that follows a standardized style, keeping the codebase neat and tidy while removing unnecessary imports/unused variables. We also build and run our tests locally via these hooks, adding a layer of sanity check before developers commit code to their branch.

```
#!/usr/bin/env sh
. "$(dirname -- "$0")/_/husky.sh"

cd backend; yarn lint; yarn format; yarn test;
```

In our main repository on Github, we used Github Actions to format, lint, build, and run tests on our code.

7.1.2 GitHub Actions

For the portion of the pipeline on our code repository, we configured GitHub Actions to run the same linters, formatters, before building and testing our code in the remote environment whenever a new pull request is made on the repository.

On merge to the master branch, the same checks are applied, and the Docker images are built for the linux/amd64 platform before being pushed to DockerHub. This facilitates the eventual deployment of microservice containers on our Kubernetes cluster on Google Cloud by pulling the images directly from DockerHub.

Our GitHub Action workflows can be found in the repository under the '.github/workflows' directory.

```yaml
name: backend-ci

on:
  pull_request:
  push:
    branches:
      - master

jobs:
  test:
    name: Backend Tests
    runs-on: ubuntu-latest

    strategy:
      matrix:
        commands: ['format:ci', 'lint:ci', 'build:ci', 'test']

    steps:
      - name: Check out repository code
        uses: actions/checkout@v3
      - name: Use Node.js
        uses: actions/setup-node@v2-beta
        with:
          node-version: '18'
      - name: Install backend dependencies
        run: cd backend && yarn install --frozen-lockfile;
      - name: Run backend tests
        run: cd backend && yarn run ${{ matrix.commands }}
      - name: Stop containers
        if: always()
        run: cd backend && docker compose -f ./docker-compose.test.yml --env-file=.env.test down
```

Screenshot of an example yaml file used to configure Github Actions for our backend CI

7.1.3 Testing

For testing, we used Jest to write end-to-end (e2e) tests. These tests simulate the HTTP and Websocket gateways and ping them at their respective endpoints to access resources from our backend. E2E tests run within a containerized environment, trying to replicate the app's true environment in production as much as possible.

```
it(`(POST) should return 401 unauthorized when user has non-maintainer role`, () => {
  const data = {
    question: MOCK_QUESTION_1,
  };

  request(app.getHttpServer())
    .post(endpoint)
    .send(data)
    .set('Authorization', `Bearer ${MOCK_USER_1_TOKEN}`)
    .expect(401);
});

it(`(POST) should return 201 created when user has maintainer role`, async () => {
  const data = {
    question: MOCK_QUESTION_1,
  };

  const { body } = await request(app.getHttpServer())
    .post(endpoint)
    .send(data)
    .set('Authorization', `Bearer ${MOCK_ADMIN_TOKEN}`)
    .expect(201);

  mockQuestionId = body._id;

  expect(body).toEqual(
    expect.objectContaining({
      _id: body._id,
      ...data.question,
    }),
  );
});
```

Example tests showing the expected responses when accessing an endpoint

7.2 Dockerization

Our backend microservices are containerized via Docker. Locally, we can orchestrate the building and running of these Docker containers using *docker compose*. Instructions on running the backend using Docker can be found here. An example of our docker-compose.yaml file can be found here.

To facilitate faster build times for our Docker containers, we added some yarn caching when building the Docker containers. This allows us to not reinstall our node_modules every time we rebuild the Docker containers.

```
17  # Cache yarn dependencies
18  RUN --mount=type=cache,target=/root/.yarn YARN_CACHE_FOLDER=/root/.yarn yarn install --frozen-lockfile
19  RUN yarn build user
20
```

Line that adds yarn caching when building our Docker containers. An example of the full file can be found here.

7.3 Infrastructure as Code

To facilitate easy (re)deployment and configuration, our team decided to write down our deployment configurations as part of Infrastructure as Code (IaC). This allows us to track configuration changes in version control, and ensures that every single change in deployment configuration is noted down. In fact, the versioning and commit history would be self-explanatory and helps a new DevOps developer to onboard quickly and easily, while being able to understand the different rationale for past configuration changes. Additionally, he would be able to redeploy a similar environment from scratch through this IaC.

For this project, we mainly focused on using the IaC approach for deployment of our Kubernetes cluster. Our own literature review revealed that there are two widely used tools, Helm and kustomize, for writing Kubernetes manifests for IaC deployment. We proceeded to try out these tools in our own deployment configurations.

Helm is like the package manager for Kubernetes manifests. Developers are able to write and publish Helm charts, which other developers are able to pull and run to deploy a certain service on the Kubernetes cluster. We used Helm extensively to install additional services we needed from the command line in our cluster, such as Istio, ingress-nginx and CertManager for TLS certificate management. Additionally, we defined our own custom Helm charts to specify the base manifests for our own microservices and Nginx ingress, where we then inject custom values for each microservice into. An example usage of our custom Helm charts can be found here. Helm then combines all of these manifests into a single YAML file when running the *helm template* command.

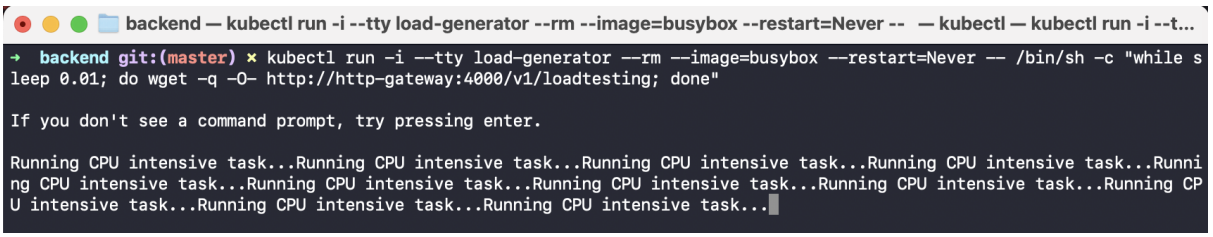
Subsequently, we use kustomize to inject deployment environment specific variables into the Helm generated manifests. Some examples include injecting the respective dev and prod environment variables, and overriding (aka patching) certain portions of the manifests for different environments. An example of the production environment patches that kustomize makes to the Helm generated manifest can be found here. To build the complete manifest, we can then run the *kustomize build* command before applying the complete manifest to the Kubernetes cluster by piping the output to *kubectl apply*. The combined instructions for the usage of Helm and kustomize to deploy our project on a Kubernetes cluster can be found here.

Do note that our microservices are all deployed in the ‘default’ Kubernetes namespace for easy verification by the teaching team if the need arises. However other services installed

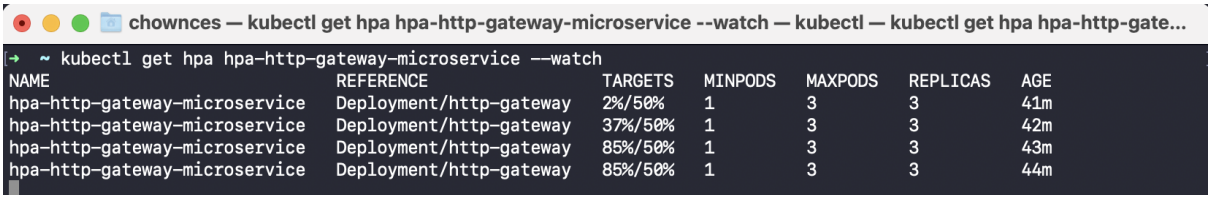
via Helm charts such as Istio, CertManager and ingress-nginx are deployed in their own separate namespaces. The exact namespaces for each of these services can be found in the commands when running Helm install. In a real life production scenario, a good practice is to separate the project backend deployment into its own namespace instead of using the 'default' one. This would achieve better separation of concerns on the Kubernetes cluster itself.

7.4 Horizontal Pod Autoscaling

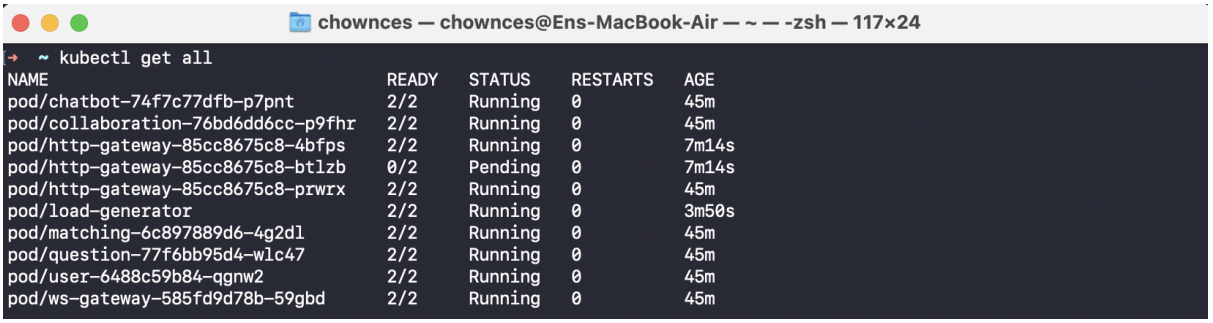
Horizontal Pod Autoscaling (HPA) is also set up on our Kubernetes cluster. The Kubernetes manifests related to HPA can be found in our base-microservice custom Helm chart here. Detailed instructions to test the horizontal pod autoscaling on a Kubernetes cluster can be found here. We have included a few screenshots demonstrating the HPA here.



Generating load to the http-gateway service at a dev environment only endpoint for CPU intensive task



Watching for the CPU load to increase beyond target threshold



http-gateway scaling up to 3 pods as specified in values.yaml

8. Deployment

8.1 Backend

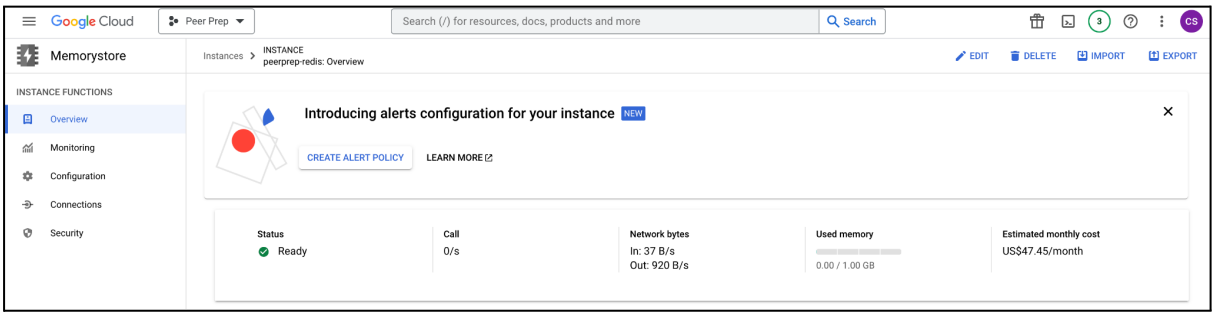
For our backend deployment, we chose to do it using Google Cloud Platform.

8.1.1 Databases

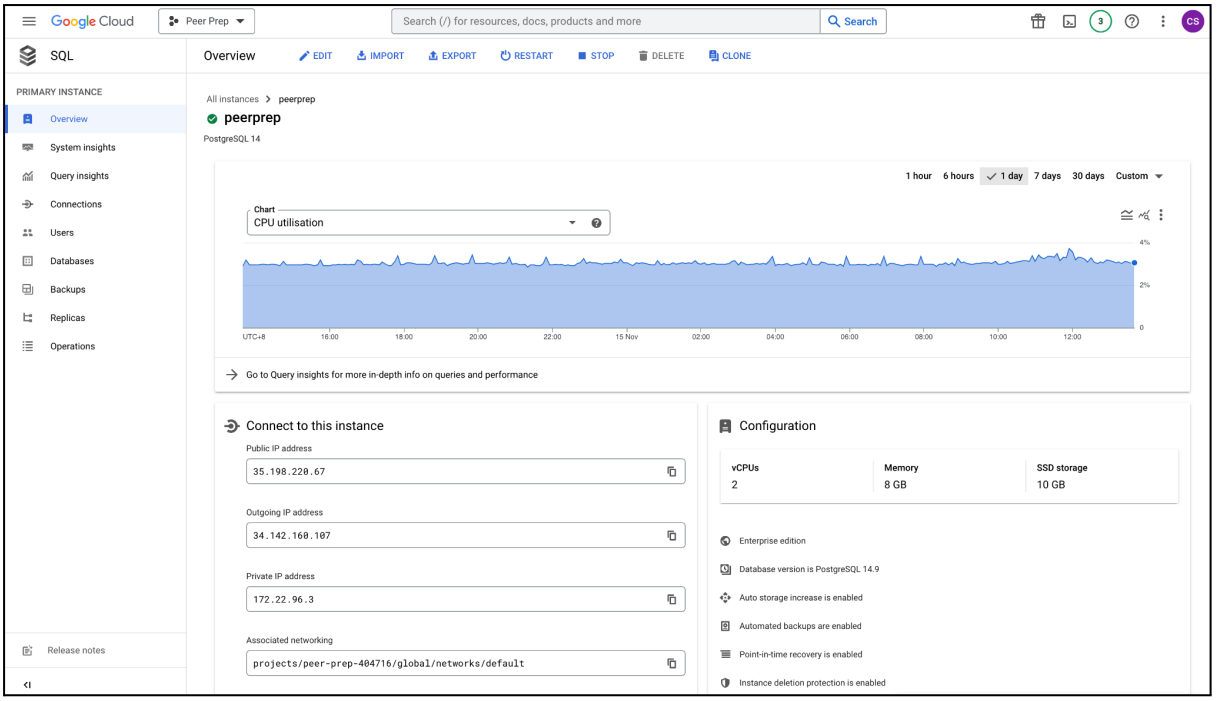
For our PostgreSQL, MongoDB and Redis, we deployed them on Cloud SQL, MongoDB Atlas on GCP, and GCP Memorystore respectively. These services are exposed within the VPC with private IP addresses. Although a public IP address is exposed, we configured it to only allow whitelisted IP addresses (i.e. from the developers) to connect to these services. Non whitelisted IP addresses would be prevented from connecting.

The team notes that in a real life production scenario, the good practice is to disable public access to these databases to improve security and reduce attack vectors. However, since this project is not going to production yet, we allow whitelisted IP addresses to connect for ease of demonstration to the teaching team during the project presentation. The public IP address can easily be removed after the presentation.

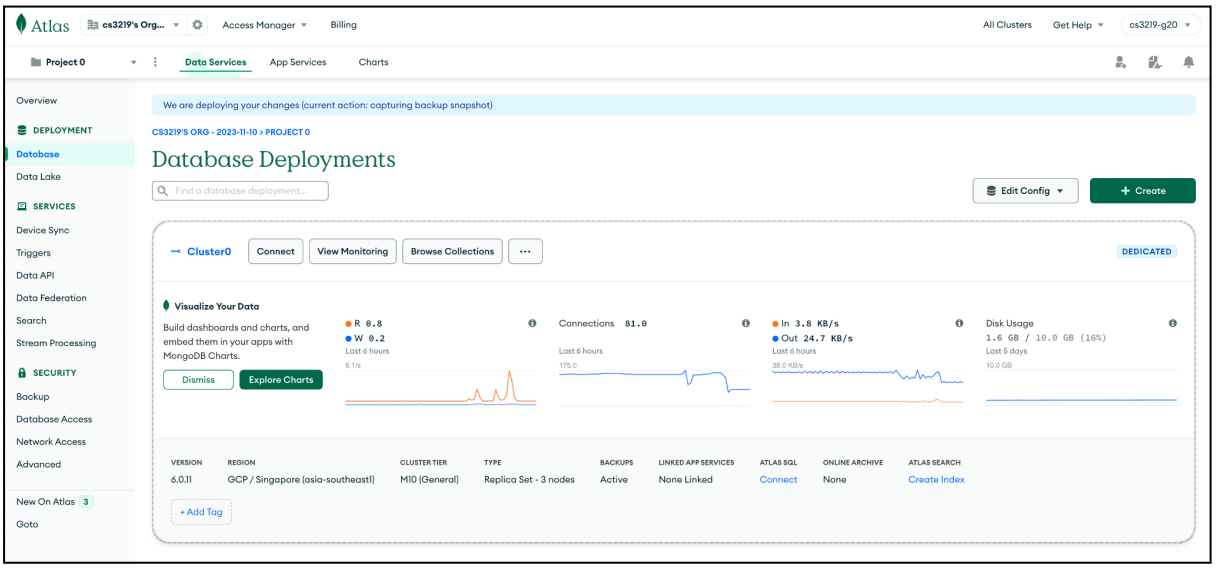
Here are some screenshots of the deployments as follows.



Redis instance deployed on GCP Memorystore



PostgreSQL deployed on GCP Cloud SQL



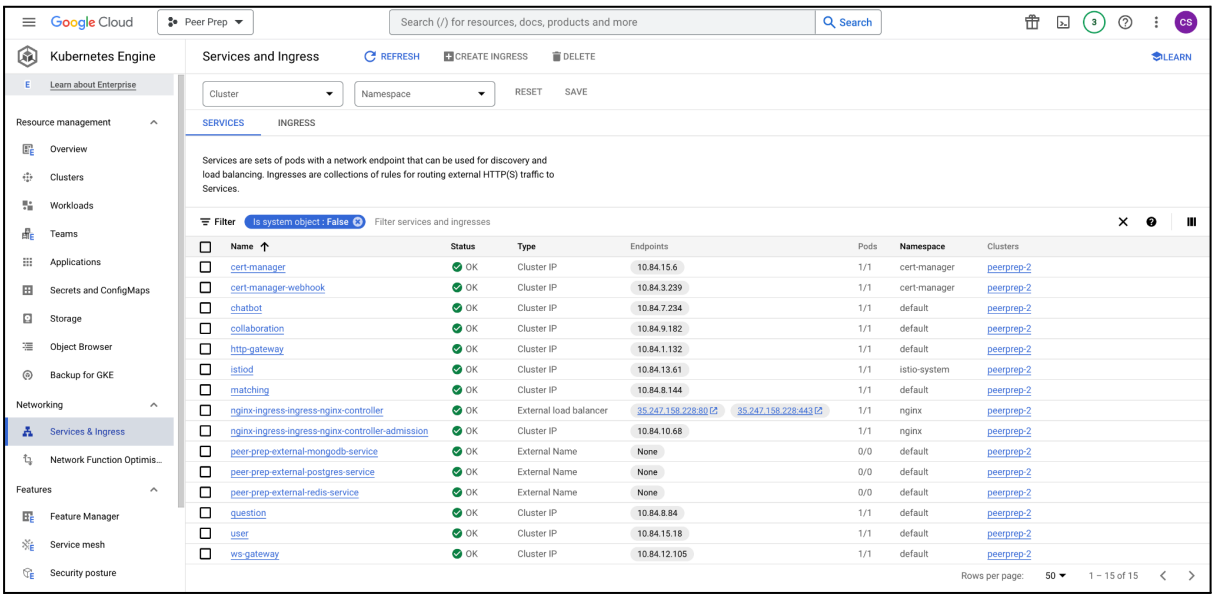
MongoDB deployment on MongoDB Atlas on GCP

8.1.2 Microservices

For the actual backend microservices, we decided to move forward with Kubernetes as our container orchestration tool, as it is widely used and an industry standard. This allowed us to gain working knowledge on Kubernetes in a small to medium scale school project. Additionally, Kubernetes enables us to easily implement service discovery (inbuilt) and manage horizontal pod autoscaling for our application.

We provisioned our Kubernetes cluster on GCP Kubernetes Engine (GKE), and followed the deployment steps as outlined here. In total, we installed the Istio Service Mesh, ingress-nginx, CertManager for managing TLS certificates and our backend microservices on the cluster. An overview of the services deployed on the cluster can be seen as follows. The backend domain for TLS cert registration was obtained from NoIP.

We connect to our databases via private IP addresses within the VPC. As our MongoDB Atlas cluster was installed on a different VPC, our team had to read up on additional configuration to enable VPC peer networking so that the MongoDB service can be accessible via a private IP address.



Kubernetes cluster deployed on GKE, and the corresponding services running

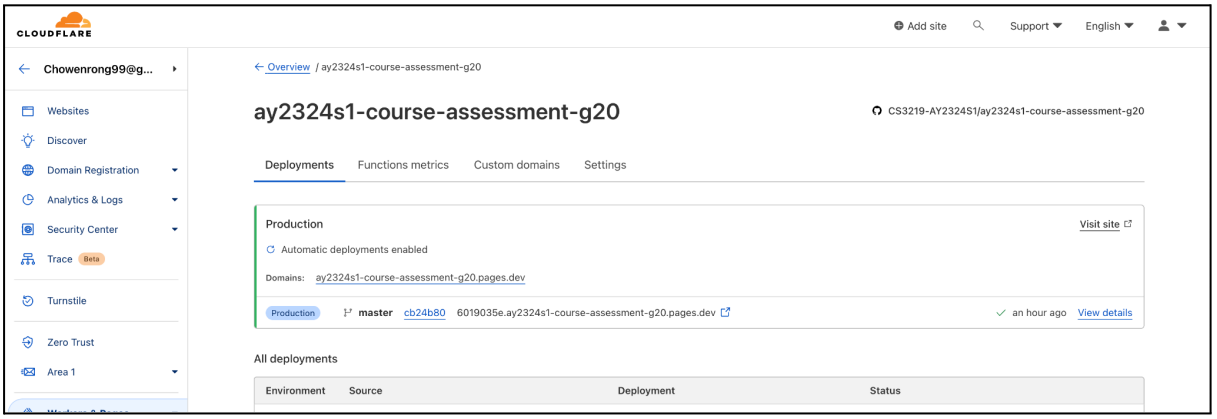
8.1.3 External Services

Our team also had an external service being used, namely the gpt3.5 turbo LLM API by OpenAI. We configured an API key with a set cost limit for the production deployment. This would prevent the team from being overcharged when the API cost limit has been fully utilised.

8.2 Frontend

Our frontend was deployed using Cloudflare Pages. Cloudflare Pages has an easy to use interface and is integrated with GitHub CI/CD to deploy the application frontend to production on merge to master branch.

Our application frontend can be found at this link: https://ay2324s1-course-assessment-g20.pages.dev. However, do note that the backend would have been taken down immediately after the project presentation due to the costs involved, and the application frontend would not work properly. The following screenshot demonstrates the frontend deployment on Cloudflare Pages.



9. Project Management

9.1 Sprints

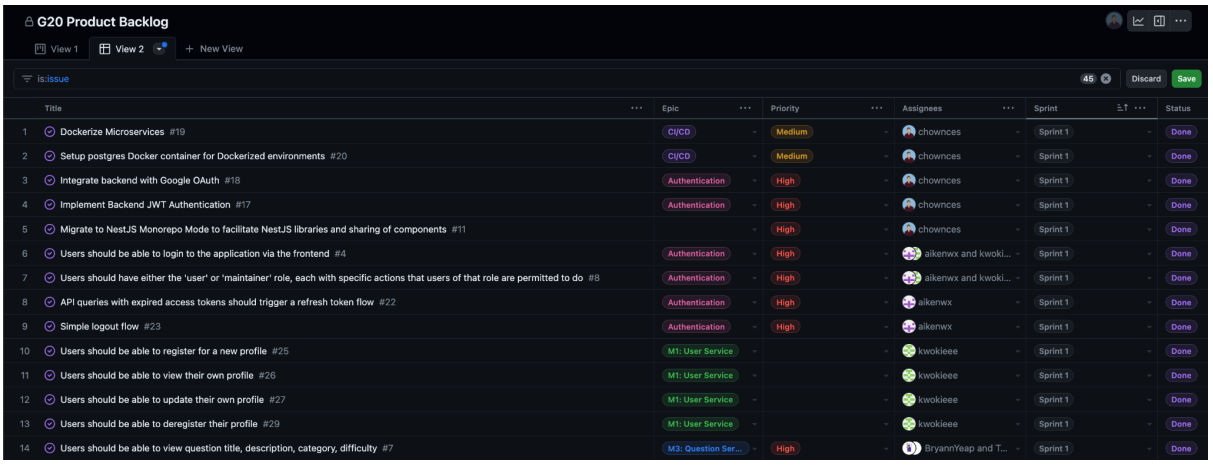
Sprints during this project last between 2 weeks to a month. During a sprint, the team will design, implement and test new features.

The team meets up weekly and syncs up regarding the progress they have made, what outstanding tasks there are remaining, and any problems they are currently facing.

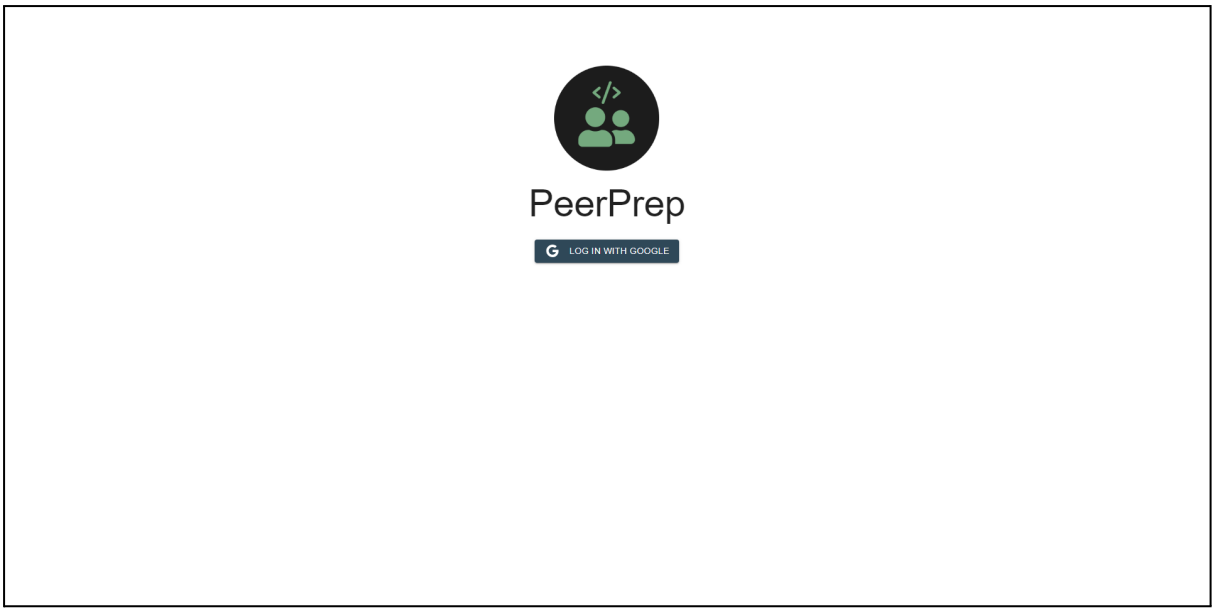
When planning for sprints, the team will discuss and assign priorities to the features to be implemented (High, Medium, or Low).

9.2 Product Backlog

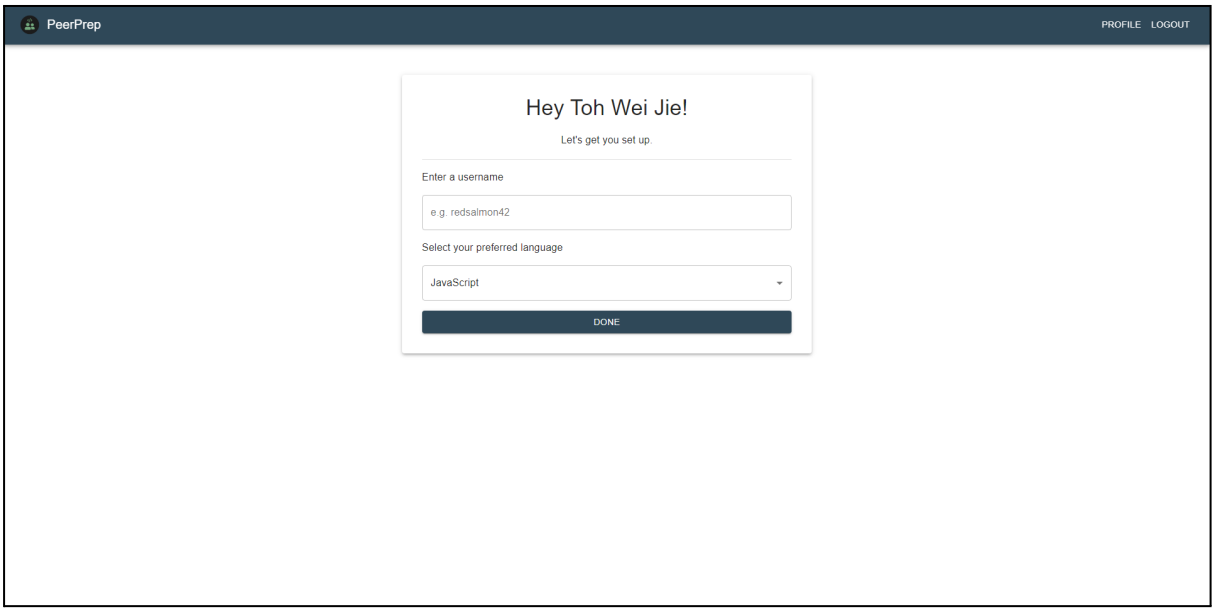
Using Github Projects allows us to seamlessly assign issues for developers to work on, link issues to milestones and sprints to know what needs to be done by when, and link pull requests to issues so that we can track the code associated with each issue. Below is a screenshot of our product backlog for sprint 1.



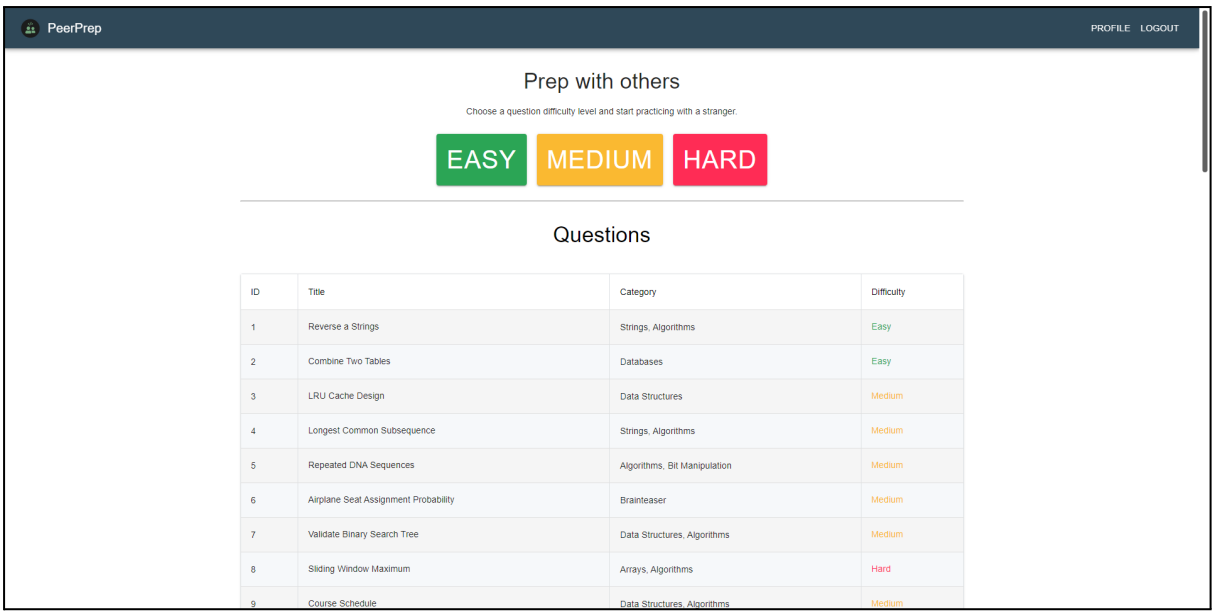
10. Application Screenshots



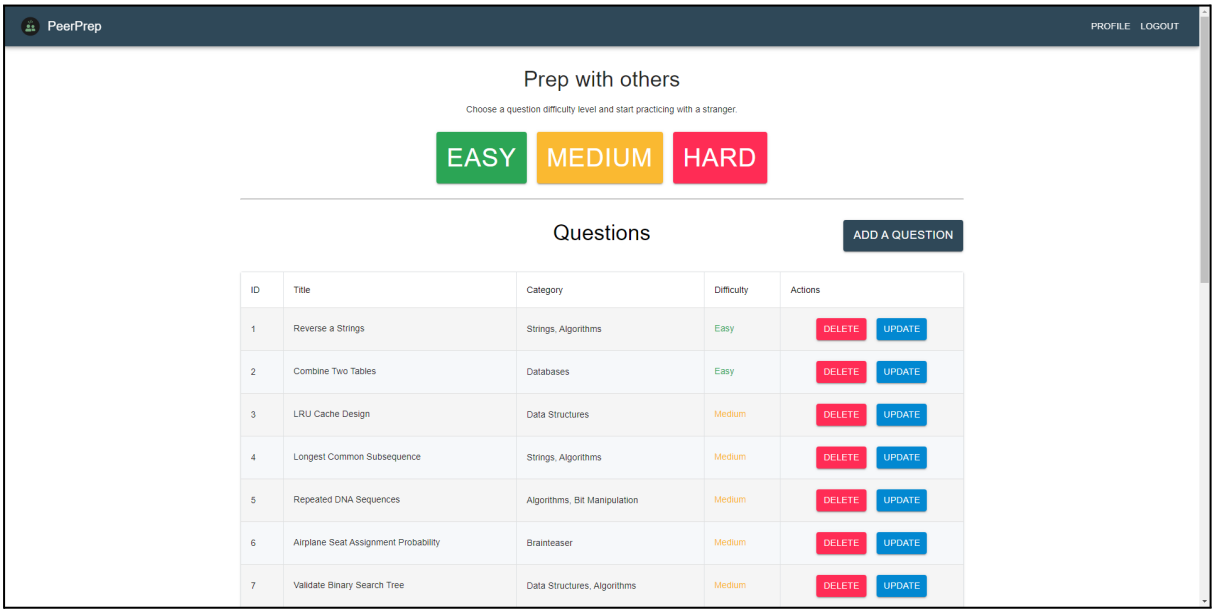
Login/Signup Page



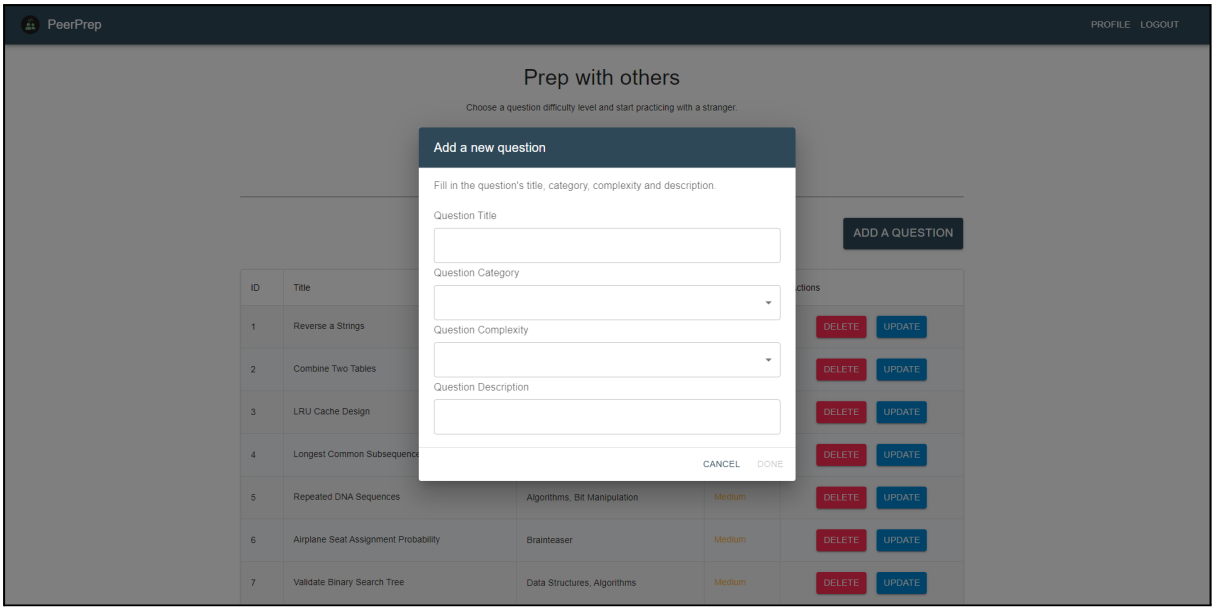
Onboarding Page



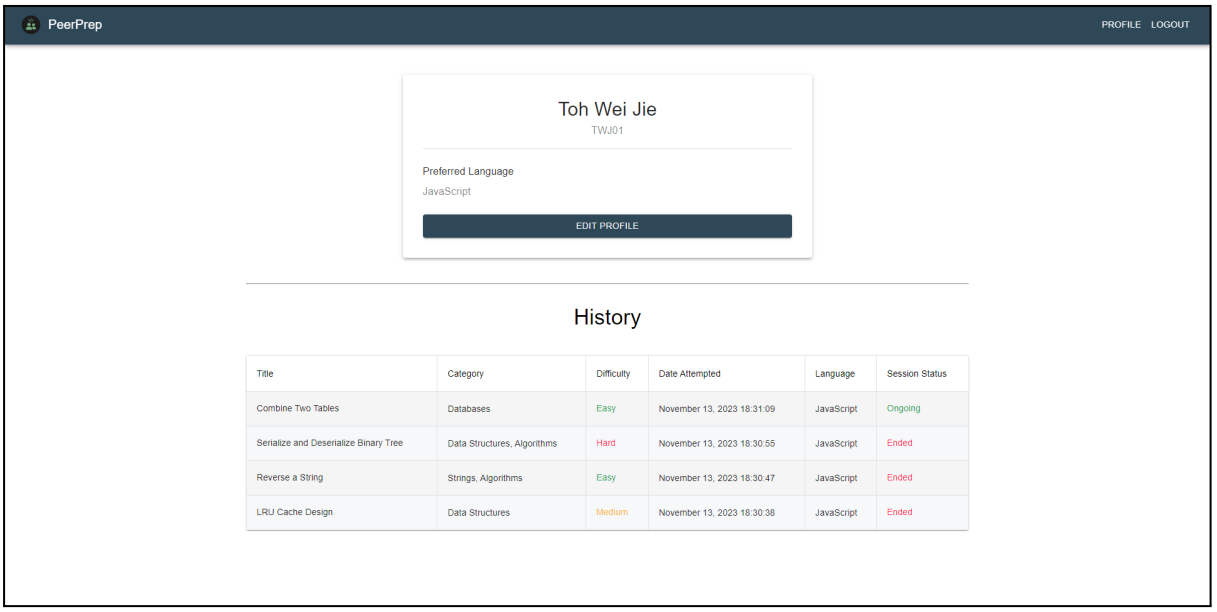
Normal User's Main Menu



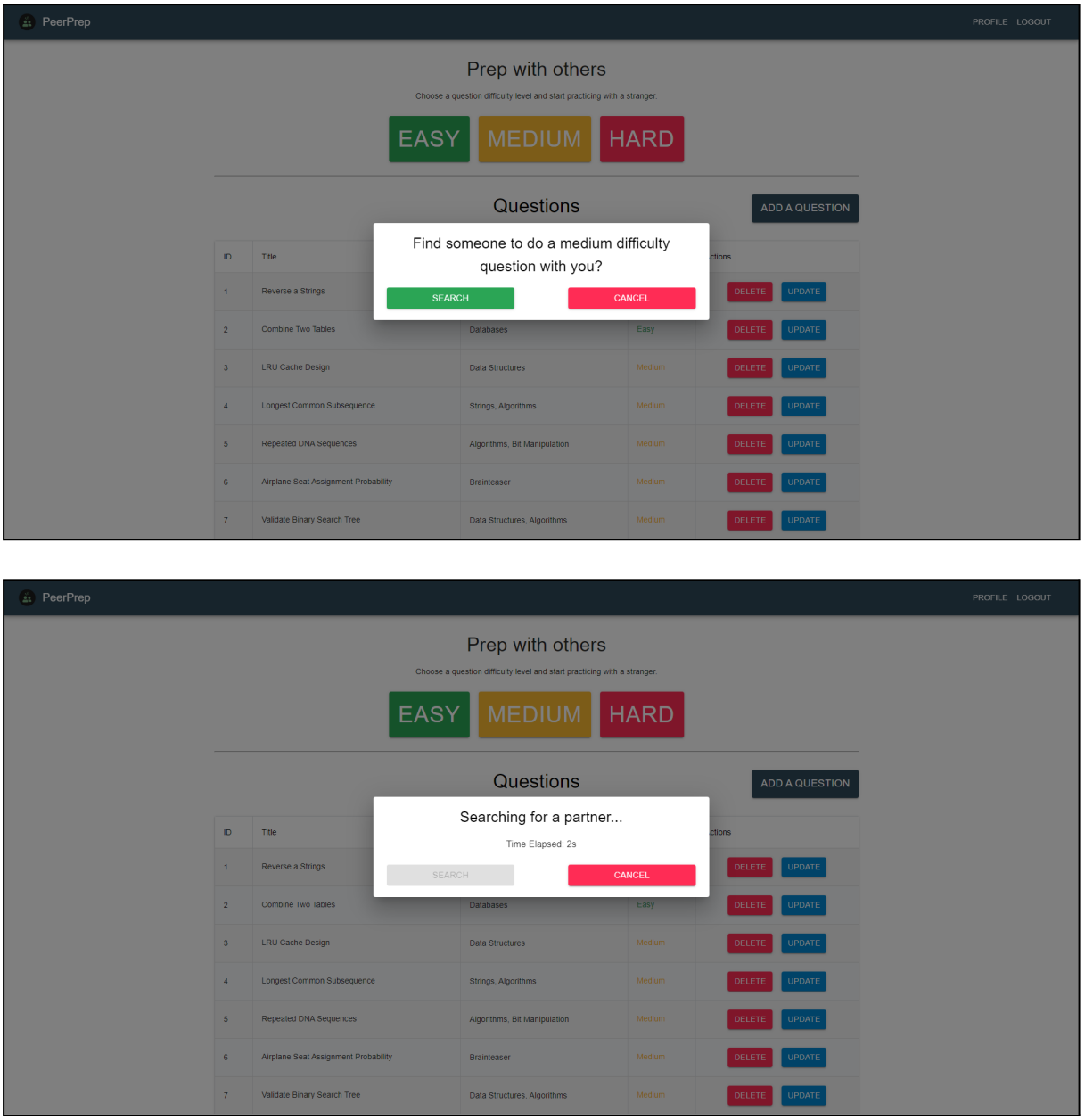
Maintainer's Main Menu



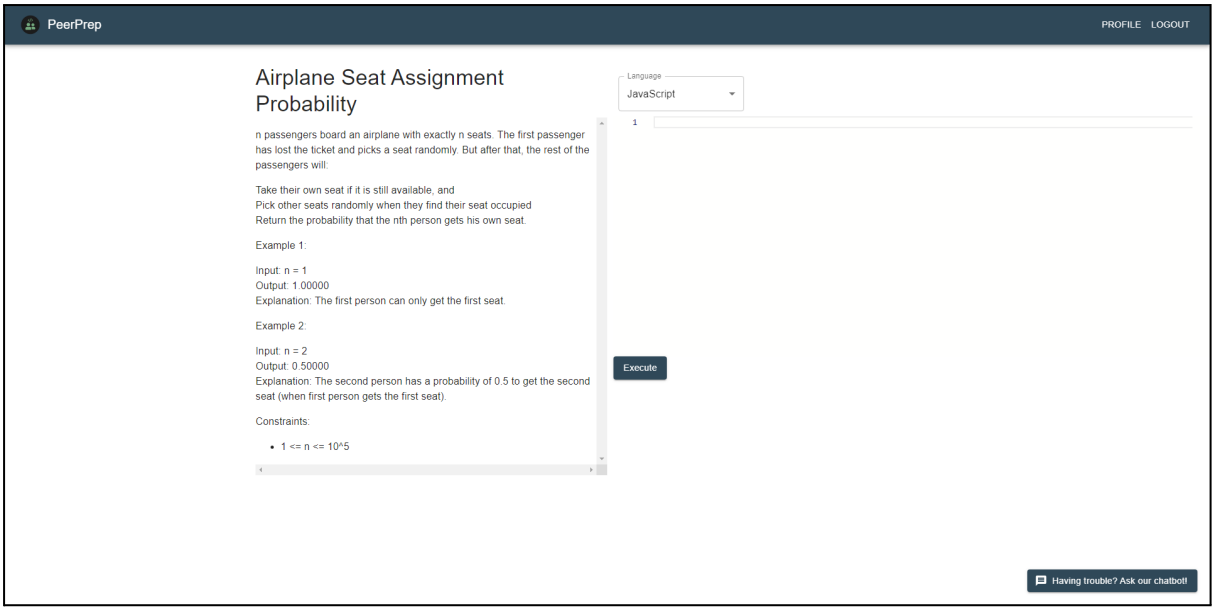
Question Form Popup



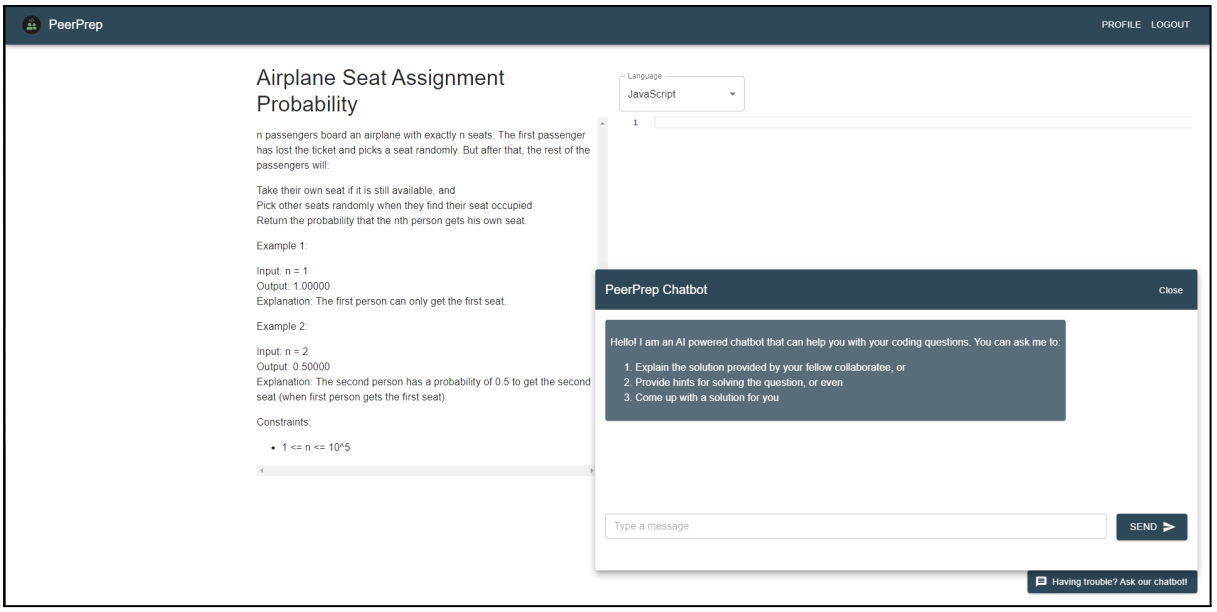
User Profile Page



Matching Service Popup



Collaborative Space



PeerPrep Chatbot

11. Potential Improvements and Enhancements

11.1 Pagination for questions

If the application has to handle displaying many questions available to the user, it would be good if pagination was implemented on the frontend to prevent loading large amounts of data in one go. It would also provide for a more user-friendly experience where the user can view a limited number of questions on each page instead of having to scroll through many questions on one page.

11.2 Table sorting / filters / search

In the same vein of creating a user-friendly table, users currently do not have the ability to sort the order of or filter and search for questions. A good enhancement would be to enable the sorting and filtering of questions based on the difficulty and categories. Similarly, a search bar could be implemented where users can search for a specific question to look at instead of having to manually scroll through the table for their desired question.

11.3 Support more languages

For the application to expand in the future to handle more languages (e.g. C++, Java, etc.), it would be more scalable if we did compilation and execution of code in the backend. Due to the limited timeframe of the project within this module, as well as the absence of free code compilers and executors, we decided to use the React Monaco Editor, which supports compiling JavaScript and TypeScript to make use of the browser's JavaScript execution thread, since it comes out of the box with syntax highlighting and autocompletion. The downside to this would be that we had to check the user's input for any potentially dangerous code that might affect the browser. For example, users should not have access to the process object, which would give them control over the current Node.js process.

11.4 Selection of criteria for matching

Currently, our matching of users who wish to enter a collaboration session is done solely based on the difficulty of the questions they chose. A further enhancement we could make would be to leverage on the categories that are assigned to each question. It would be possible to provide the user with a checklist consisting of the different difficulties and categories available and allow them to select all the options they desire (could be any combination of difficulty and category) and two users that have the same subset of choices would be matched into the same collaborative session.

11.5 Video call / live chat functionality

Currently, the only way for users to communicate is via the code editor. A further enhancement we could make is to allow users to make a video call with their collaborator. This would allow users to more freely exchange their ideas via the video call. This would also allow users to more closely experience what an actual interview may feel like. Alternatively, we could have a live chat box instead to allow for easier and separate method of communication between the users in the same session.

11.6 Viewing the user profiles of other users

Currently, on the frontend we only support the viewing of the user profile that belongs only to the user themselves. We do not provide any method on the frontend side to check out or navigate to other users' profile, although, users can manually view it themselves by changing the URL of the user profile name to other users' username. A possible feature would be to include a search bar where users can search for other existing users' accounts and click on them to view their profile.

12. Concluding Reflections

As for some, this was our first exposure to utilising a microservices architecture, there was a significant amount of research required to be done in order to learn how such microservices could be implemented and the possible tools that would enable us to program the microservices. As a result, this project provided us a platform to utilise a lot of different tools that we normally otherwise would not.

Moreover, as this project was not as strict on the requirements, many decisions were needed to be made by the team, for instance, the architecture design and pattern, making a decision on what qualifies as a duplicate question, what criteria would we allow the users to match with each other for a collaborative session. There was a need for the team to justify our design choices over other alternatives (some of which are brought up in the potential improvements and enhancements section above). This to us was both a bane and a boon as with more open-ended requirements, we were free to choose the way we wish to implement something, but in the same vein, we also encountered scenarios where we were unsure if our choice would be accepted by the teaching team, as such we needed to seek for clarification from our mentor.

Although we planned out our progress into sprints and set soft deadlines on completion of certain services, our time management for this project could be better. Due to the need to balance our workload between modules, we were cutting it close with the sprint deadlines for the completion of programming of certain tasks. Moreover, although we did our own local testing throughout the project, we only managed to come together to perform a more thorough integration testing and stress test the application nearer to the final deadline. In hindsight, we should have set aside a proper week dedicated fully on testing to find any leftover bugs, and tried to set a harder deadline for completion of certain features to prevent overflow into the next sprint.

Appendix

Contributions

Individual

Name	Technical contributions	Non-technical contributions
Aiken Wong Xiheng	- Matching service - In-app code editor and execution - Syntax highlighting - Chatbot Service	- Code Review - Report Writing - Documentation and READMEs - Project management - Requirements specification - QA Testing
Bryan Kwok Yan Fai	- Frontend design - User service - E2E tests	
Bryann Yeap Kok Keong	- Question service - History service - Exceptions Filter + Error-Handling	
Chow En Rong	- Architecture - Authentication - Collaboration service - Deployment + Kubernetes Manifests - Container orchestration	
Toh Wei Jie	- Question service - Matching Frontend - Horizontal scaling	

Subgroup

Subgroup 1: Aiken, Bryan, Bryann

ID	Feature
N2	History: Maintain a record of the questions attempted by the user. With progressive levels of difficulty: maintain a list of questions attempted along with the date-time of attempt, maintain a list of questions along with the attempt, maintain a list of questions along with the attempt and retrieve correct answers.
N3	Code execution: Implement a mechanism to execute code in a sandboxed environment, and retrieve+present the results in the collaborative workspace.
N5	Enhance collaboration service by providing an improved code editor. With progressive levels of difficulty: code formatting, syntax highlighting for one language, syntax highlighting for multiple languages.
N7	Incorporate generative AI to assist during the preparation. For example, users can seek help from ChatGPT, Copilot (or such other services) to seek explanation of code written by the other participant. Users can seek help from gen AI to solve the puzzle by providing prompts from within the interface.
N8	Extensive (and automated) unit, integration, and system testing using CI. Teams can also use various test frameworks or demonstrate effective usage of CI/CD in the project.

Subgroup 2: En Rong, Wei Jie

ID	Feature	How to verify
N9	Deployment of the app on the production system (AWS/GCP cloud platform).	Details under the ‘Deployment’ section of this report.
N10	Scalability – the deployed application should demonstrate easy scalability of some form. An example would be using a Kubernetes horizontal pod auto-scaler to scale up the number of application pods when there is a high load.	Details, screenshots and instructions to reproduce under the ‘Horizontal Pod Autoscaling’ section of this report.
N11	The application should have an API gateway of some kind that redirects requests to the	Our application uses ingress-nginx in the Kubernetes

	relevant microservices. An example would be using an ingress controller such as NGINX ingress controller if using Kubernetes (https://kubernetes.GitHub.io/ingress-nginx/).	cluster deployment. Additionally, we implemented two separate API gateways (http-gateway and ws-gateway) as outlined in the 'HTTP Gateway and Websocket Gateway' section of this report.
N12	The application should demonstrate service discovery or implement a service registry of some kind. For example, Kubernetes has built-in DNS features and service networking to pods (https://kubernetes.io/docs/concepts/services-networking/dns-podservice/).	Kubernetes has built-in DNS features and service networking already.