CS3219 Software Engineering Principles and Patterns
AY23/24 S1
Group 40 Report

Group Members:
Chia Yu Hong, Chinthakayala Jyothika Siva Sai, Justin Widodo, Tan Yuan Zheng, Xu Richard Chengji

Website:
https://peerprep.yuanzheng.pro

Code Repository:
https://github.com/CS3219-AY2324S1/ay2324s1-course-assessment-g40

Table of Contents

1. Introduction

1.1 Background and Purpose of the Project

This project involves the design and implementation of PeerPrep, a platform on which users are able to prepare for technical interviews by matching with peers and practicing interview questions in a collaborative space.

Users are able to match based on a selected programming language and question difficulty. A successful match directs them to a collaborative space in which a question of the selected difficulty is provided, along with a shared code editor, and a chat interface for communication. Users are able to save their attempt at a solution and view a history of their past attempts at the displayed question while in the collaborative space, and are also able to search for and select a new question to display to both users in the room. Outside of the collaborative space, users are able to view the available questions on PeerPrep, as well as a history of their past attempts at questions.

2. Contributions

2.1 Individual Contributions

Chia Yu Hong [*Sub-group α*]	**Implementation**
	- Implement base Question Service.
	- Set up containerization for frontend, API gateway, microservices, and MongoDB databases with Docker
	- Set up Docker Compose for local deployment of PeerPrep as a multi-container application with container networking
	- Implement Collaboration Service to handle concurrent real-time updates for a frontend consisting of a shared code editor, syntax highlighting and a text-based chat interface
	- Implement an API Gateway to handle routing of requests from the frontend to the respective backend microservice
	- Configure the API Gateway to authenticate requests with a sub-request to User Service
	- `README.md`
	Report
	- **[1.1]**, **[4.1]**, **[5.1.2]**, **[5.1.4]**, **[5.1.5]**, **[5.1.6]**, **[5.1.7]**, **[5.2]**, **[5.4]**, **[5.5]**, **[5.5.5]**, **[5.8]**, **[5.8.1]**, **[6.1]**, **[6.2]**, **[6.3]**, **[8.1,1]**
	- **Architecture Diagrams:**
	  - *Local Deployment of PeerPrep*
	  - *General Microservice Structure*
	  - *User Service*
	  - *Collaboration Service*
	  - *Cloud Deployment of API Gateway and Microservices*

Chinthakayala Jyothika Siva Sai [*Sub-group ß*]	**Implementation** - Implement Question Service using local storage. - Implement Frontend User Interface(UI) for all pages using the Angular Framework in Typescript. - Implement History Service Backend using Node.js and extend existing Question Service to accommodate a new Tags field. - Implement a searching feature by using text indexes in MongoDB for Question Documents. - Test and fix bugs in the application including matching timer random reset bug **Report** - **[5.1.1]**, **[5.3]**, **[5.3.1]**, **[5.3.2]**, **[5.3.3]**, **[5.3.4]**, **[5.3.5]**, **[5.5.2]**, **[5.5.2.1]**
Justin Widodo	**Implementation**

[*Sub-group α*]	- Implement User Service CRUD on the backend - Implement User Service on the frontend (login, register, update, delete) - Implement user Authentication using JWTokens and Cookie Sessions - Implement matching service in the backend using a local data structure - Integrated API Gateway with existing authentication scheme to query the respective service - Develop unit tests for backend matching, question and user, history services **Report** - **[3.4]**, **[4.2]**, **[4.3]**, **[5.1.4]** ,**[5.5.1]**, **[5.5.1.1]**, **[5.5.1.2]**, **[5.5.1.3]**, **[5.5.1.4]**, **[5.5.4]**, **[5.7.2]**, **[7.1]**, **[7.3]**
Tan Yuan Zheng **[*Sub-group ß*]**	**Implementation** - Implement App Database - Design Database Schema - Implement CD pipeline, automating build and deployment procedures - Set up DNS and proxy for frontend deployment - Set up TLS certification for frontend server and backend API - Set up Kubernetes deployment for Containerised applications **Report** - **[5.1.3]**, **[5.1.9]**, **[5.5.3]**, **[5.5.3.1]**, **[5.6]**, **[5.8.2]**, **[5.8.2.1]**, **[5.8.2.1.1]**, **[5.8.2.2]**, **[5.8.2.2.1]**, **[8.1.2]** - **Architecture Diagrams:** - *Cloud Deployment of Frontend*
Xu Richard Chengji **[*Sub-group ß*]**	**Implementation** - Implement CI pipeline, integrating build and testing procedures - Helped developed unit tests for frontend components - Implement serverless function to fetch questions from third party source - Debugged frontend components that were failing CI pipeline **Report** - **[5.1.8]**, **[5.7]**, **[5.7.1]**, **[5.9]**, **[7.2]** - **Architecture Diagrams:** - *Serverless Function*

3. Requirements Specification

3.1 Must-have [Mx] and Nice-to-have [Nx] features

[Mx]	Must-have	[Nx]	Nice-to-have	[*sub-group*]
[M1]	User Service	**[N1]**	Communication mechanism	***α***
[M2]	Matching Service	**[N2]**	History Service	***ß***
[M3]	Question Service	**[N4]**	Enhance Question Service	***ß***
[M4]	Collaboration Service	**[N5]**	Enhance Collaboration Service	***α***
[M5]	Basic UI	**[N9]**	GCP Deployment	***ß***
[M6]	Local Deployment	**[N11]**	API Gateway	***α***

3.2 Functional Requirements [FRs]

		Priority	Sprint
FR1. User Service - *user profile management, authentication*			
FR1.1	PeerPrep should allow users to register an account with a username and password	High	1
FR1.2	PeerPrep should store the authentication information of a user.	High	1
FR1.3	PeerPrep should authenticate users and allow login based on credentials.	High	1
FR1.4	PeerPrep should allow users to logout of their account.	High	1
FR1.5	PeerPrep should allow users to change their password.	Medium	4
FR1.6	PeerPrep should allow users to delete their accounts.	Medium	4
FR2. Matching Service - *criteria-based user matching*			
FR2.1	PeerPrep should match users based on a set of criteria, i.e., preferred programming language and complexity.	High	3
FR2.2	PeerPrep should match users within a period of time if a valid match exists.	High	3
FR2.3	PeerPrep should timeout after a period of time and	High	3

	indicate that no match was found if no valid match exists.		
FR2.4	PeerPrep should allow users to cancel their current matching while waiting for a valid match.	High	3
FR3. Question Service - *complexity-indexed question repository management*			
FR3.1	PeerPrep should allow only users with the admin role to store question data, indexed by complexity.	High	1
FR3.2	PeerPrep should allow only users with the admin role to update question data.	High	1
FR3.3	PeerPrep should allow only users with the admin role to delete question data.	High	1
FR3.4	PeerPrep should allow only authenticated users to read question data.	High	1
FR3.5	PeerPrep should allow only users with the admin role to tag questions.	Medium	6
FR3.6	PeerPrep should allow only authenticated users to filter questions by their title or tag.	Medium	6
FR4. Collaboration Service - *real-time collaborative space for matched users*			
FR4.1	PeerPrep should provide a shared code editor in the collaborative space in which matched users may simultaneously edit a solution to the provided question.	High	4
FR4.2	PeerPrep should provide a random question of the selected difficulty to the matched users in the collaborative space.	High	4
FR4.3	PeerPrep should provide code formatting and syntax highlighting for the selected programming language in the shared code editor.	High	5
FR4.4	PeerPrep should allow users to communicate in the collaborative space through the use of a text-based chat interface.	High	5
FR4.5	PeerPrep should allow users to retrieve a new question to display in the collaborative space.	Medium	4
FR4.6	PeerPrep should allow users to save their attempts at a question in the collaborative space.	Medium	5
FR4.7	PeerPrep should allow users to view the past	Medium	5

	attempts at the question displayed in the collaborative space.		
FR5. User Interface			
FR5.1	PeerPrep should provide an interface with which users can register an account with a username and password.	High	1
FR5.2	PeerPrep should provide an interface with which users can login to their account with valid credentials.	High	1
FR5.3	PeerPrep should provide an interface with which users can change their password with valid credentials.	High	4
FR5.4	PeerPrep should provide an interface with which users can view the available questions.	High	1
FR5.5	PeerPrep should provide an interface with which users can initiate matching with a preferred programming language and difficulty.	High	3
FR5.6	PeerPrep should provide an interface with which two users can view and edit a solution simultaneously.	High	4
FR5.7	PeerPrep should provide an interface with which users can view a history of their question attempts.	High	5
FR6. History Service - *complexity-indexed question repository management*			
FR6.1	PeerPrep should allow authenticated users to view a history of their own question attempts after submission.	Medium	5
FR6.2	PeerPrep should allow authenticated users to view the user they worked with for each question attempt.	Medium	5

3.3 Non-Functional Requirements [NFRs]

		Priority
NFR1. Performance		
NFR1.1	PeerPrep should be able to handle at least 6 concurrent users without a noticeable drop in performance.	High
NFR1.2	PeerPrep should be able to retrieve question data within 2 seconds or display an error message upon failure.	High
NFR1.3	PeerPrep should respond to user input in <10 ms	High
NFR2. Storage		
NFR2.1	PeerPrep should be able to store user data for up to 300 users.	High
NFR2.2	PeerPrep should be able to store question data for up to 300 questions.	High
NRR2.3	PeerPrep should be able to store solution data for each user attempt at a question.	Medium
NFR3. Compatibility		
NFR3.1	PeerPrep should be able to work on modern browsers, i.e., Chrome, Safari, Internet Explorer, FireFox.	High
NFR4. Usability		
NFR4.1	PeerPrep should be intuitive enough for new users to navigate and use.	High

3.4 User Stories

1. As a user, I want to log in to my account to access the site's functionality. **[FR 1.2]**, **[FR1.3]**, **[FR5.2]**
2. As a user, I want to be able to view the technical interview question list to see all available questions. **[FR 3.4]**, **[FR5.4]**
3. As a user, I want to select a question difficulty level (easy, medium, or hard) to customize my technical interview preparation experience. **[FR 2.1]**, **[FR 5.5]**
4. As a user, I want to be matched with another online user who has selected the same difficulty level and language, to collaborate on technical interview questions. **[FR 2]**
5. As a user, I want to receive a technical interview question when successfully matched with a peer. **[FR 4.2]**
6. As a user, I want access to a collaborative space where my matched peer and I can work together on solving the provided technical interview question. **[FR 4.1]**, **[FR 5.6]**
7. As a user, I want to be able to see syntax highlighting for my chosen programming language when working in the collaborative space. **[FR 4.3]**
8. As a user, I want to be able to chat with the peer I am currently working with to better coordinate with them. **[FR 4.4]**
9. As a user, I want to be able to review my past attempts for attempted questions. **[FR 5.7]**, **[FR 6.1]**
10. As a user, I want the option to save the solutions from collaborative sessions for future reference. **[FR 4.6]**
11. As an admin, I want to be able to create new questions for the question repository. **[FR 3.1]**
12. As an admin, I want to be able to edit and delete any existing questions in the repository. **[FR 3.2]**, **[FR 3.3]**

4. Software Development Process

4.1 Methodology

The software development process sought to emulate the Agile development process. To this end, weekly sprints were planned, and sprint meetings held at the start of each sprint to plan the tasks prioritized for each sprint as well as to keep the development team updated on the status of each task and any issues faced. There was also a focus on iterative and incremental development, with the goal of ensuring a working product increment at the end of each sprint. A set of sprint logs **[8.2]** were kept for tracking and accountability purposes.

4.2 Review Process

The review process for the project was such that any pull requests to the master branch required the review and approval of another team member before merging. This was enforced by branch protection rules on the `master` branch, requiring any merging to the branch to require a pull request, and that pull requests have the approval of the most recent reviewable push by a member other than the member who pushed it. Additionally, the rules also require status checks to pass and branches to be up to date prior to merging. These checks ensure code quality is maintained in the `master` branch, and helps to catch any oversight. Where applicable, team members with the relevant expertise are also requested to review a pull request prior to merging, and the pull request is merged only with their approval.

4.3 Development Stages

PeerPrep's development began with gathering the requirements of the project through the development of the Functional **[3.2]** and Non-Functional **[3.3]** requirements. Any ambiguities faced during the requirements gathering were resolved by consulting the project mentor. This process allowed the team to build a comprehensive set of software requirements to provide a basis for the development of the project. Additionally, at this stage there was a brief discussion about which **Nice-To-Have (Nx)** features were feasible, with the aim of ensuring their consideration while working on the **Must-Have (Mx)** features.

Once a solid set of software requirements were developed, work on the **Must-Have (Mx)** features began, with tasks being distributed evenly amongst the team. On completion of these features, a Minimum Viable Product had been developed that encompassed all the Must-Haves. Based on the Agile development process, the team dedicated time to testing and refining features for the MVP before moving on to the stage.

Following the testing and refinement of MVP features, the team made concrete decisions on which **Nx** features were to be developed. These decisions relied on the feasibility of the **Nx** features in relation to the current implementation and tech stack.

After the **Nx** features had been completed the team once again dedicated time to testing and refining features of the overall project, ensuring each **Mx** feature works seamlessly with the **Nx** features.

5. Design

5.1 Tech Stack

Frontend	Angular
Backend	Node.js, Express.js
Database	MongoDB
Authentication	JSON Web Token (JWT)
Code Editor	Monaco Editor
Event-based Communication	Socket.IO
API Gateway	NGINX
Containerization	Docker
CI	GitHub Actions
Cloud Deployment	Google Kubernetes Engine (GKE)

5.1.1 Frontend - Angular

Angular is a popular and powerful front-end framework that is often chosen for Single Page Applications (SPA) CRUD (Create, Read, Update, Delete) applications due to its Modularity and Reusability. Angular encourages the use of components, services, and modules, making it easier to create modular and maintainable code. Components can be reused across different parts of the application, improving code organization and reducing redundancy.

There are 2 main features of Angular that contributed to the choice of the Angular framework.

Angular offers two-way data binding, which means changes in the UI are automatically reflected in the underlying data model and vice versa. This simplifies the development of CRUD operations where data synchronization between the view and model is crucial. The logic to update data from backend services in the front end used by PeerPrep is abstracted and reused so that development can focus on higher-level logic like organization and layout of data.

Angular's dependency injection system allows for the easy management and injection of services into components. This promotes the use of decoupled and

testable code, making it easier to replace or upgrade components without affecting the entire application. This enables PeerPrep's front end to adhere to the Separation of Concerns Principle since different pages viewed by the user are not dependent on previously navigated pages and logic handling backend requests and responses can be separated for each microservice.

5.1.2 Backend - Node.js and Express.js

Node.js was chosen for its throughput and scalability in developing server-side applications, as well as access to packages with the node package manager (npm). Express.js, a minimalist Node.js web framework, was chosen as it provides features suitable for web development, such as HTTP request handlers, while also providing access to useful features with middleware packages.

5.1.3 Database - MongoDB

A NoSQL database (MongoDB) was chosen due to its scalability and performance over an SQL database.

In terms of scalability, NoSQL databases have the advantage of being more horizontally scalable, allowing them to handle a large amount of traffic and data by adding more servers to the database. This scalability is often achieved through sharding, or partitioning data horizontally across multiple servers, which is supported by MongoDB. This ability to natively support sharding is especially advantageous in the Peerprep app, as many of the collections in the schema benefit from sharding and its advantages. One such example is the `history` collection, which can be sharded on the `userId` shard key. This provides performance benefits such as only having to query a specific shard that contains the relevant data, instead of performing the query on the whole collection.

NoSQL databases are optimized for specific types of queries and access patterns, making them performant in certain use cases. For CRUD operations, especially in scenarios where data retrieval is more straightforward, NoSQL databases can offer faster read and write speeds compared to SQL databases. MongoDB in particular is optimized for document-oriented data that may have varying/undetermined columns/fields and other data types like images, like the `questions` collection.

Detailed justification of the advantage of this design choice will be discussed further in section 5.6 Database Schema.

5.1.4 Authentication - JSON Web Token (JWT)

JWT is an open standard that defines a compact and self-contained way to securely transmit information between parties as a JSON object. In PeerPrep it is primarily used to authenticate a user as well as store key information about the user such as information about their roles. The JWT issued upon login uses the "HS256" algorithm which is a symmetric keyed hashing algorithm that uses one secret key which allows

us to validate JWT by providing the secret key again. JWT was thus chosen given its robust standardization and widespread support across various tech stacks.

5.1.5 Code Editor - Monaco Editor

Monaco Editor was used to implement the code editor interface as it provides options to configure the features of the provided code editor as required, and its model based content management lends itself well to implementing concurrent updates to the editor interface in the collaborative space with event-based communication. Integration into the frontend was done with an Angular module, `ngx-monaco-editor-v2`, which allows utilization of Monaco Editor as an Angular Component, providing better compatibility and management with the rest of the frontend.

5.1.6 Event-based Communication - Socket.IO

Socket.IO was chosen as it provides bidirectional, low-latency event-based communication, which makes it suitable for implementing concurrent updates to different components between two users in the collaborative space. This serves to fulfill **NFR1. Performance**, in terms of the responsiveness of the collaborative space. Furthermore, given that a Socket.IO connection is established with different low-level transports based on the capabilities of the browser, this choice also helps fulfill **NFR3. Compatibility**.

Additionally, in the case that future extensions are made with respect to scalability, Socket.IO was chosen as it provides options for horizontal scaling.

5.1.7 API Gateway - NGINX

NGINX was chosen to implement the API gateway as it provides options to configure policies to enforce on APIs, which provides finer access control. In particular, PeerPrep specifies the `auth_request` directive to enforce request authentication.

NGINX also provides configuration for load balancing, which leaves room for future extensions to be made to the API gateway in the future to support horizontal scaling.

5.1.8 Containerization - Docker

Docker was chosen as it is well suited to the deployment of PeerPrep as a multi-container application. For local deployment, Docker Compose simplifies the management and configuration of the containers with a `.yml` file, which streamlines service configuration and orchestration during development and testing. Cloud deployment is also streamlined given how Docker is compatible with the Google Kubernetes Engine (GKE), where Docker containers are leveraged for deployment on GKE to capitalize on the scalability and flexibility of Kubernetes.

5.1.9 CI - GitHub Actions

Continuous Integration (CI) is a software development practice that enables developers to integrate code changes into a shared repository regularly. We implemented automated tests in the GitHub repository through GitHub Actions, leveraging a `.yaml` file to define the workflow. GitHub Actions was chosen because GitHub Actions is tightly integrated with GitHub repositories. This means the CI configuration lives in the same repository as the code, allowing for easier tracking of code changes, and easier code maintenance. Additionally, GitHub Actions allows automation of software development workflows. This automation helps catch issues early in the development process and ensures that code changes don't introduce regressions. GitHub Actions provides visibility into the CI/CD pipeline directly within the GitHub repository. The status of workflows and view logs can be observed, which provides helpful insights into the success or failure of each step. This visibility helps in troubleshooting and understanding the health of the codebase.

A `.yaml` file was used to define our workflow for GitHub Actions. YAML is a human-readable, easy-to-write, and easy-to-understand configuration language. The declarative nature of YAML allows us to define the CI workflow steps clearly. The workflow was split into builds and tests. Initially, a build for PeerPrep is executed to verify that all the services could build and run without failure. Next is the testing phase, which was split between backend tests and frontend tests. This way, if any test case fails, it can be easily pinpointed where the problem is coming from.

5.2.0 Cloud Deployment - Google Kubernetes Engine (GKE)

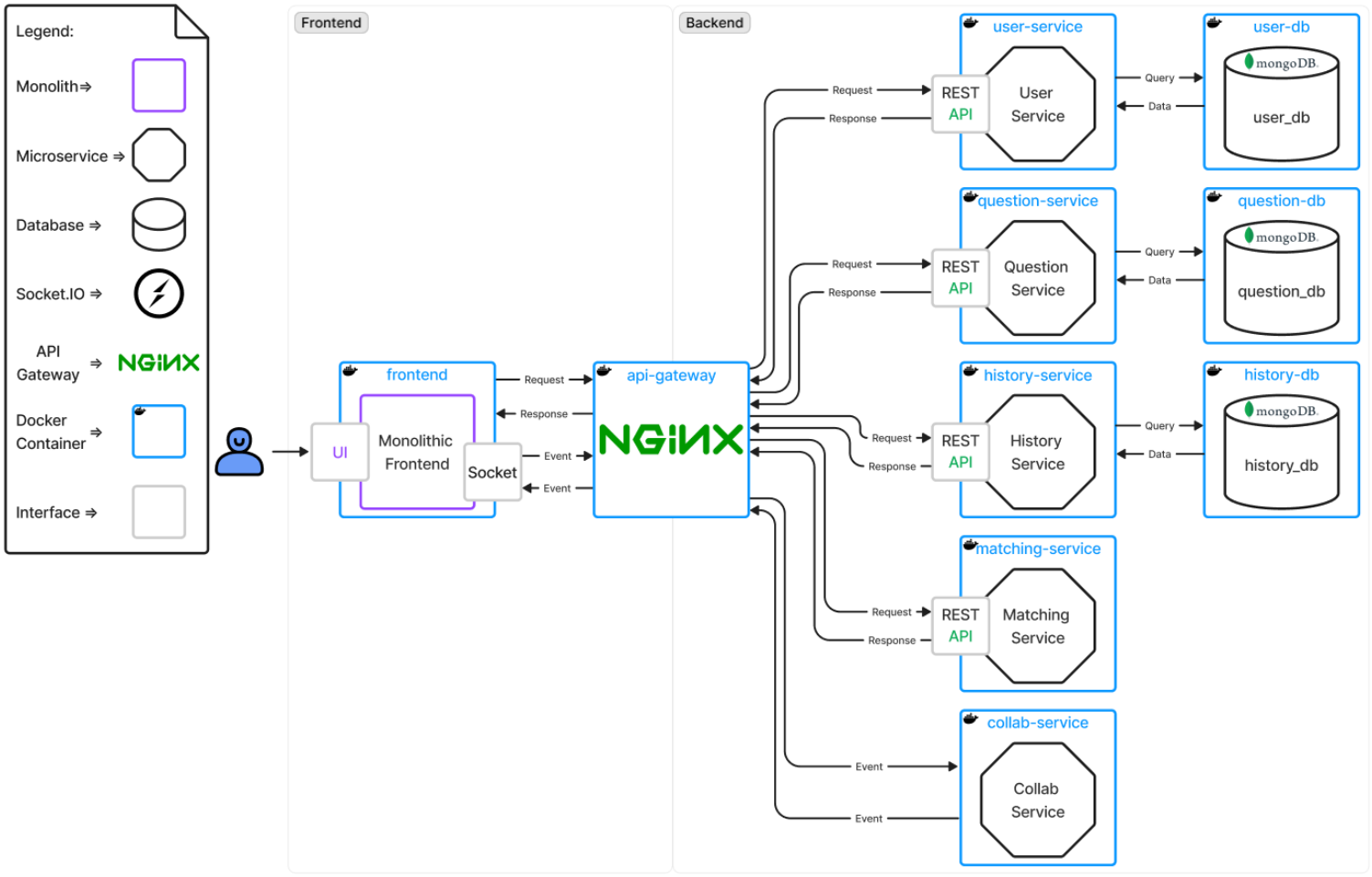
Google Kubernetes was chosen as the container orchestration tool of choice for cloud deployment due to the fact that our backend microservices use the containerized deployment mechanism. Kubernetes as a framework provides us the ability to run our containerized microservices resiliently. It automates important non-functional tasks, such as scaling and failover.

GKE also allows for autoscaling of the application backend by automatically resizing the number of nodes in a given node pool based on the workload demands. When demand is high, the cluster autoscaler adds nodes to the node pool, and conversely, when demand is low, the cluster autoscaler scales back down to a preconfigured minimum size automatically without developer intervention.

GKE also ensures service health by automatically restarting containers that fail, replacing containers, killing containers that don't respond to health checks, and refraining from advertising containers until they are ready to serve.

Aside from these two main benefits, GKE also offers many other quality of life benefits like load balancing and secret management.

5.2 Architecture



Architecture Diagram for Local Deployment of PeerPrep

PeerPrep uses a microservice architecture for its backend, with a monolithic frontend that interacts with microservices through an API gateway.

The use of a microservice architecture for the frontend was avoided as while such an architecture would provide scalability in larger projects, it is not as suitable for the design of PeerPrep as a small project, and could instead hinder its development due to increased development and deployment complexity. As such, a monolithic frontend architecture was chosen as it was deemed more suitable for the size and complexity of PeerPrep, and would also allow the design and development of more consistent features in the frontend.

A microservice architecture for the backend was chosen as it separates the functionality of the backend into a collection of independent services with well-defined capabilities, creating a modularized architecture in which each service can be developed, deployed and scaled independently. This reduces the complexity of the development process and is thus more suitable for the small development team as it allows each member team to independently develop, test and deploy

individual loosely coupled microservices based on the requirements of each service, which leads to faster development.

Instead of having the frontend send requests directly to each microservice, an API gateway was used to route requests from the frontend to the microservices. This design was chosen as it allows the API gateway to handle authentication for certain requests through a sub-request to the authentication service, instead of having each microservice authenticate with a request to the authentication service, which would increase coupling. Additionally, this prevents the frontend from requiring knowledge of the address of each microservice. Instead, the address of each microservice is known by the API gateway, and the frontend simply sends requests to the API gateway, which in turn routes the requests to the microservices.

5.3 Frontend

The front end is split into components like the Matching Component (the matching page viewed by the user), services like the Matching Service (the interface used to manage and implement backend communication with the Matching Service) and the Router which enables navigation between pages by replacing components on the page viewed by the user. PeerPrep is a Single-Page Application (SPA) that uses the Angular Framework and several Design Patterns stated below.

5.3.1 MVC Pattern

The MVC design pattern corresponds to the architecture of a project consisting of three distinct parts. Models are used to represent transit data used in the application (interface or class without logic) which are the Decorators in this case. The views correspond to the user interface (the HTML template). Finally, controllers allow linking of the two previous layers which are the typescript files of each component.

5.3.2 Dependency Injection Pattern

Dependency Injection (DI) is an important design pattern for developing large-scale applications. Angular has its own DI system, which is used in the design of Angular applications to increase efficiency and scalability. Indeed in the constructor of the classes (components, directives, services), one asks for dependencies (services or objects). It is an external system (the injector) which is responsible for creating the instance according to the configuration. This facilitates the development but also the tests. Directives offered by Angular such as `NgIf` and `NgFor` are used heavily in the frontend to abstract for-loops and if-statements for HTML component creation like table entries that follow the same structure in HTML but have different corresponding data.

5.3.3 Model-Adapter Pattern

The design pattern model-adapter transforms the format of data retrieved from an external source into a data format suitable for consumption by the Angular client. For example, if you retrieve data from an API the dates will be formatted as String. It is possible to perform the transformation upon receipt of the response from the API and instantiate a date in javascript format. This pattern is used heavily in the application by transforming HTTP responses to Observable Objects that facilitate the transfer of data from services to respective components and the transformation of HTTP response body data to String or Object data types to be used/displayed by components.

5.3.4 Lazy Pattern

The lazy design pattern makes it possible to develop scalable and high-performance applications because the principle of this pattern is to provide an application divided into several independent parts. This makes it possible to reduce the size of the main application and then have different modules which will be downloaded by the user only if he wishes to access this specific part of the application.

5.3.5 Decorator pattern

The decorator design pattern is an alternative to subclasses for extending an object using composition instead of inheritance. In fact, the decorator attaches additional responsibilities to an object. The main concept is to have an object that wraps another object. The one wrapping the object is the decorator and it is the same type of the original object but it has also an object of the same type of the object. Decorators used in PeerPrep's front end include `@Component`, `@Service` and `@Router` which offer a structure for different parts of the front end to communicate and differentiate their uses as well as reuse standard structures for these components.

5.4 API Gateway

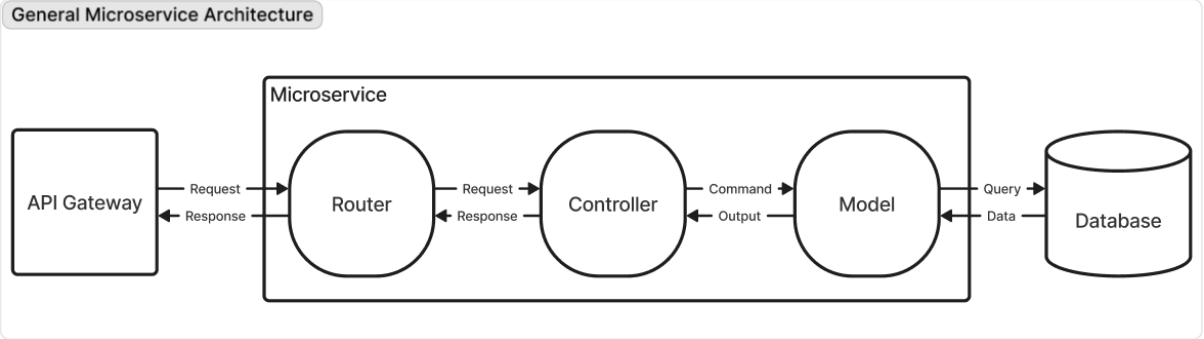
PeerPrep implements an API gateway with NGINX as a single entry point for requests from the frontend. Any requests from the frontend are handled by the API gateway by being routed to the appropriate microservice. This hides how microservices are partitioned, and removes the need for the frontend to determine the exact location of each microservice. Instead, the API gateway exposes a set of APIs, which the frontend may utilize through API requests which are routed to the appropriate service in the backend.

Route configuration is handled in the `default.conf` file, where routes are defined with the names of the respective service, which mirror the container names in the case of local deployment, and service names in the case of cloud deployment,

leveraging the DNS provider of the Docker network and GKE respectively to resolve the names to addresses.

The API gateway also implements security by verifying that clients are authorized or have sufficient permissions to perform certain requests. This was chosen in favor of each service sending requests to the user microservice for authentication in order to reduce coupling between microservices, and prevent each microservice from needing to determine the location of the user microservice. This authentication is implemented with the `auth_request` directive, which is defined on certain routes that require authorization and/or certain permissions. When configured on a route, the directive sends a sub-request through a configured `internal` route for authentication. This `internal` route sends a request with the credentials of the requesting client to the authentication sub-service of user-service, which returns a `200 OK` status code in the case that the requesting client is authorized for the route, in which case the original request is allowed through, else the request is blocked.

5.5 Microservices



Architecture Diagram for General Microservice Structure

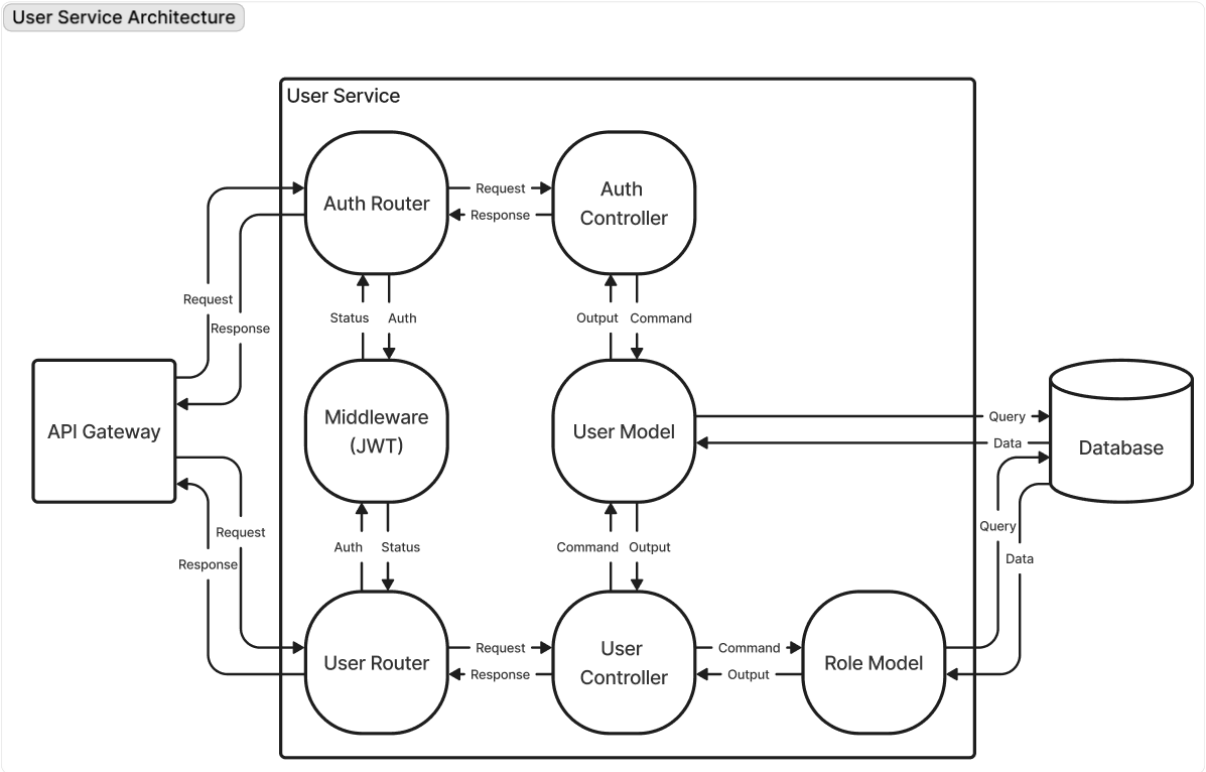
The design of each microservice uses the service instance-per-container pattern, and where applicable, the database-per-service pattern.

The use of the service instance-per-container pattern ensures that each container image consists of the minimal applications and libraries required to run the respective service instance, allowing the content and complexity of each container to be managed independently, and for containers to be as lightweight as possible.

The database-per-service pattern was used so that each service has its own database schema, should a database be necessary for the service. Compared to the shared database pattern, where a single database may not be able to satisfy the storage and access requirements of each service, the database-per-service pattern was chosen as it allows more flexibility in defining a different database schema for each service, allowing each service to use a database best suited to its requirements, which better suits the microservice architecture. This also ensures that services are loosely coupled, allowing changes to be made to the database schema of one service without impacting other services, while also reducing the data

coupling between each service and making only relevant data known to each service.

5.5.1 User Service



Architecture Diagram for User Service

The User Service is logically separated into two distinct components: the User Sub-service, which manages User Objects, and the Authentication Sub-service, which oversees authentication checks and session management. The service uses a synchronous request and response architecture, where clients send HTTP requests through the frontend, and the service processes these requests and provides responses.

5.5.1.1 User Sub-service

The User Sub-service is responsible for managing User Objects within the User Database. User Objects store essential user account information, including the user's username, email, and roles. The current implementation of the User Sub-service supports creating, reading, updating, and deleting User Objects. Furthermore, a user's username and email will be unique as there are checks in place to verify their uniqueness before creation.

Notably, the User Sub-service restricts user profile updates to password changes only. This design choice reflects our emphasis on prioritizing the collaborative aspects of the application over extensive user profile management. Moreover, to enhance security, user passwords are hashed using the “bcryptjs” library before

being stored in the database. This ensures that even in the event of a data breach, user passwords will still remain confidential and inaccessible.

Another thing to note is that the “roles” of a User Object is stored as an array, this decision was made to facilitate a role-based permission system, where permissions are assigned based on the user’s roles. Each role defines its own set of permissions that can aggregate to allow a user to perform a certain subset of tasks. This implementation grants PeerPrep a more flexible account management system, allowing for the seamless addition of new roles with their own unique permissions in the future. As it stands, the current implementation only has two roles based on the project requirements, a `user` role which grants users the ability to view questions and participate in collaborative sessions, and an `admin` role which encompasses the full range of permissions, enabling the creation, update and deletion of questions, on top of all other user capabilities.

5.5.1.2 Authentication Sub-service

The Authentication Sub-service is responsible for session management and authentication of JWT. During a login attempt, the service first retrieves the user’s username, and then it compares the `bcryptjs` hash of the entered password with the pre-existing hashed password in the database. Subsequently, it generates a JWT using the secret key found in the `auth.config.js` file, the user’s id and a boolean value that is set to true if the user is an admin. The JWT, username and an array of roles is then encrypted with the cookie-session library and stored on the Set-Cookie header value in the response to the user.

In the front end, an `HttpInterceptor` is configured to automatically set the `withCredentials` flag to true for any outbound HTTP request originating from itself. This configuration ensures that the cookie header persists in all interactions between the frontend and the various microservices in the backend. Without this `HttpInterceptor` in place, it would be necessary to include the '`withCredentials: true`' flag on every route, resulting in a more complex and repetitive setup.

Requests requiring some form of authentication authenticate with a sub-request sent to this sub-service by the API Gateway before being routed to their intended services. JWT validation is performed by the `authJwt` middleware which uses the secret key to verify any given JWT. Two primary authentication functions are in place: `verifyToken` and `isAdmin`. The `verifyToken` function checks the validity of the token, while `isAdmin` not only verifies the token but also examines the boolean value within the JWT to determine if the user has an admin role. This separation of concerns was implemented to ensure that each function is dedicated to a specific task. Responses from this middleware works hand in hand with the API Gateway’s `auth_requests`. For the corresponding responses based on these conditions, please refer to the table below:

Response	Condition
`200 OK`	All checks passed
`401 Forbidden`	No token/session found
	Invalid Token Provided
`403 Unauthorized`	Not an Admin (for `isAdmin`)

Secret keys, such as the ones used for JWT and `cookie-session`, were managed within the repository. The current system checks if there is an environment variable currently set in the `process.env.JWT_SECRET`. If not, it uses a default value for the secret which is defined in the same file.

5.5.1.3 Session Management - Cookie-session

Cookie-session is a lightweight middleware used to maintain a session through browser cookies. It stores session data on the client's web browser, which enables users to stay logged in across multiple instances. The decision to use this is influenced by the fact that a cookie persists in the user's browser as opposed to just their active tab. Consequently, this approach allows a user to open up a new tab and still remain logged into the same session. Alternatively, they could also close the browser and the session would still persist until the next time they log out. This greatly improves user experience as they would not need to re-login each time. The `cookie-session` encrypts the session variables using a single secret key so as to not reveal the variables in plaintext. To further increase security, the `httpOnly` flag is also set to true to prevent client side scripts from retrieving the session information.

5.5.1.4 User Database Schema

There are two tables within the user database schema. The User table which follows the definition of a User Object, hence the schema includes username, email, password, and roles. The Role table which maintains the different roles currently supported by PeerPrep, its schema is simply just the name of the role. The roles field in the User table stores an array of MongoDB's `ObjectId` type, these are references to Roles objects that are stored in the Role table.

5.5.2 Question Service

The Question Service uses a simple Model-View-Controller(MVC) pattern and the Express library to implement this. The View component consists of all the routes employed to make different requests by the front end. The Model consists of the database schema, modeled in javascript Objects using Mongoose ODM, and the MongoDB connection which is queried by the methods of the Controller class. The

Controller's methods are called upon by the routes and they return HTTP responses to the respective routes.

Since MongoDB is a non-relational Database, an additional Counter collection is used to maintain the count and subsequent identification number (ID) of newly added documents to the collection. There is one counter object created when the question service is run for the first time, which is updated by subsequent create requests to the database. If this counter document's count variable exceeds the limit of the integer variable maximum space limit (which is 16MB), a new counter object can be created to extend the existing count. This scenario is unlikely given that the maximum number of questions required to be held by the application is likely not more than 1 million. This imposes some additional computation for each request but it does not have a significant effect on the speed of execution of question creation requests.

5.5.2.1 Question Database Schema

The database schema used for Question Service includes the fields Question Id, Question Title, Question Category, Question Complexity, Question Description and Question Tags where Question Tags is an optional field. This ensures that questions with empty fields are not inputted into the database as they are not useful to users of PeerPrep and use up Storage space. The Question Title field is also unique so that there is a field that can be used to uniquely identify a question by users and questions are not confused. Since these fields are all text fields, MongoDB can store these fields as indexes and perform faster search requests on all fields of question objects for users to find specific kinds of questions as well as specific questions.

5.5.3 History Service

The History Service is implemented similarly to the Question Service with an MVC pattern and MongoDB. The View component consists of all the routes employed to make different requests by the front end. The Model consists of the database schema, modeled in javascript Objects using Mongoose ODM, and the MongoDB connection which is queried by the methods of the Controller class. The Controller's methods are called upon by the routes and they return HTTP responses to the respective routes.

5.5.3.1 History Database Schema

The history service controls the CRUD operations of the history database, which models an `ATTEMPT` relationship between users and questions, or in other words, where an author (user) attempts a question. The ODM enforces the fields `questionId`, `language`, `solution`, `userId`, and `userId2` in the schema. A non-trivial design decision made is to store the fields `questionId`, `userId` and `userId2` as `strings`, rather than using a reference to the corresponding documents in `question` and `user` databases. The purpose of this choice is to stay in line with

NoSql database design philosophy and good practices, since embedded documents are preferred over references to minimize joins, which maximizes lookup efficiency. Furthermore, the finer details about users and questions are not needed for history service; hence, making this choice even more natural.

5.5.4 Matching Service

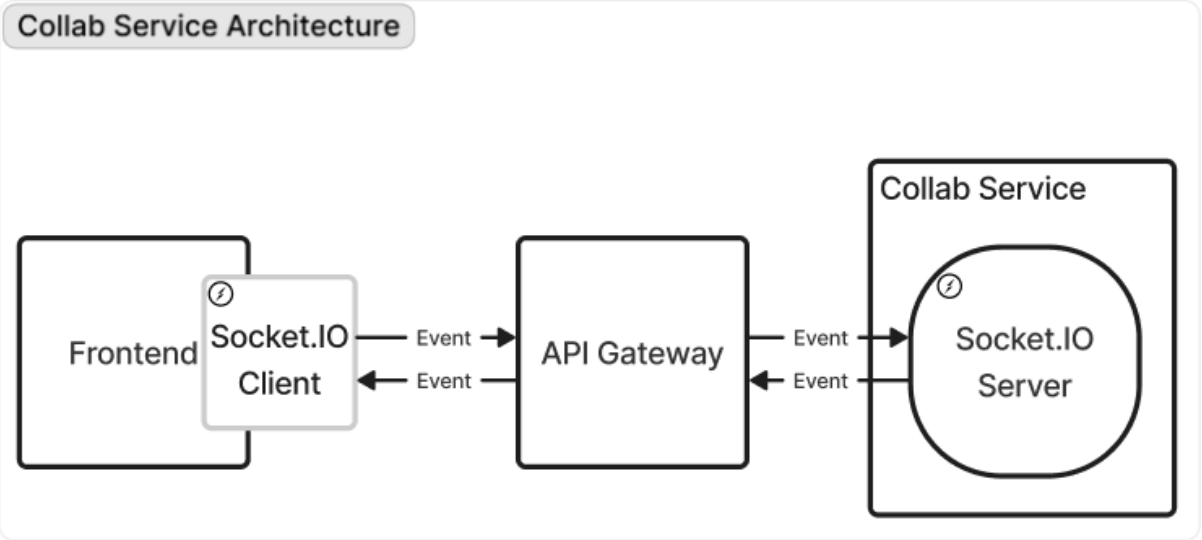
The Matching Service employs a custom queue data structure to manage Match Objects. These Match Objects represent an instance of a user looking for a match, storing information such as the user's username, chosen programming language, difficulty level and the delayed response object to the user. An asynchronous request-reply pattern is implemented, in which a user is not blocked upon initiating a match.

When a user initiates a match request, a request is sent to the Matching Service. It checks if there is another Match Object in the queue with matching programming language and difficulty level criteria. If a match is found, the corresponding Match Object is removed from the queue, and an HTTP response containing a success message, the `roomId`, difficulty level, programming language and the corresponding peer's `userId` is sent to both users. The `roomId` variable is generated with the usernames of the two users along with the value from `Date.now()`. Given how matching is done, this guarantees that the `roomId` value used to establish a session is unique given the uniqueness of each `userId` and how no two matches occur at the exact same time.

If no matching Match Object exists in the queue, the service adds the requesting user's Match Object to the queue. Subsequently, the user's frontend client subscribes to the Matching Service using `rxjs`, waiting for a response. A 30-second timer begins, and if the timer elapses without a match being found, the user's client stops listening to the service and sends a request to remove the Match Object from the queue.

A local queue was chosen for the implementation as a user would not enter the queue if a corresponding match is already present in the queue. Consequently, the queue is expected to contain a maximum of twelve Match Objects with unique criteria at any given time, based on the four programming languages and three levels of difficulty available. With this small, bounded number of objects, we decided that implementing a custom data structure was a suitable choice. This approach not only simplified the implementation but also maintained efficiency, while still meeting the project requirements. To enhance the functionality of the custom queue data structure, Lodash, a lightweight utility library was used. It allowed efficient operations on the queue, further optimizing the functionality of the Matching Service.

5.5.5 Collaboration Service



Architecture Diagram for Collaboration Service

The Collaboration Service implements a publisher-subscriber (pub-sub) pattern using a Socket.IO server to handle event-based communication in the collaborative space. The server listens for events emitted by sockets created in the frontend and publishes them to the relevant subscribers.

Event	Description
`"connection"`	Emitted by the Socket.IO server upon a new connection from a client socket.
`"join"`	Emitted by a client socket to join a room with the `roomId` generated by Matching Service. The client socket is subscribed to events emitted by other client sockets in the same room via the `.join()` method called by the Socket.IO server.
`"leave"`	Emitted by a client socket upon leaving their current room.
`"request"`	Emitted by a client socket to request a change in the currently displayed question in the collaborative space.
`"question"`	Emitted by a client socket to change the currently displayed question in the collaborative space.
`"change"`	Emitted by a client socket on any changes to the content of the code editor.
`"message"`	Emitted by a client socket on sending a message in the text-based chat interface.
`"disconnect"`	Emitted by a client socket upon disconnection from the Socket.IO server.

Upon successfully matching with another user, each user creates a client socket, which connects to the Socket.IO server running in Collaboration Service, and emits a "`join`" event with the `roomId` generated by Matching Service in order to subscribe to events in the room identified by the `roomId`. Updates to the collaborative space are handled by different events based on the component in question. These events are emitted to other client sockets in the room.

The code editor in the collaborative space was implemented with Monaco Editor, which supports code formatting and syntax highlighting through its options. Concurrent edits to the code editor are handled by the "`change`" event, which is emitted upon any change to the content of the code editor model originating from the user, and contains information on the change made to the model. The other user in the room receives these remote change events through their client socket and applies the changes to their own model, which keeps the contents of the code editors synchronized.

Updates to the text-based chat interface are handled by the "`message`" event, which is emitted by the client socket on sending a message, and contains information on the message sent by the user. Each user maintains their own copy of the messages history, and displays each message differently depending on if they were sent by the user or received from the other user.

The "`request`" and "`question`" events handle any changes to the question displayed in the collaborative space. The "`question`" event is emitted by a client socket to change the question currently being displayed in the collaborative space, and contains information on the question to change to. Upon receiving the "`question`" event, the displayed question for the user is changed to that of the question information contained in the event. The "`request`" event is emitted upon a request by a user to change the currently displayed question, and prompts a confirmation from the other user. Upon confirmation of the change request, a "`question`" event is emitted to change the displayed question for both users. If the change request is rejected, a "`message`" event is emitted to the requesting user with a message indicating that the change request has been rejected by the other user.

5.6 Database Schema

The database schema design is summarized in the table below. Fields highlighted in yellow are required fields as enforced by mongoose ODM in the backend.

Collection	Document Attributes	Remarks
users	username: String email:String Password: String roles: Array	roles is an array of ref: roles.ObjectId. username is unique.
roles	name: String	name is unique.
questions	questionId: Int32 questionTitle: String questionDescription: String questionCategory: Array questionComplexity: String questionTags: Array createdAt: Timestamp updatedAt: Timestamp	Both questionCategory and questionTags are arrays of String. questionComplexity is a potential shard key. questionId and questionTitle are unique.
histories	questionId: String Language: String Solution: String userId: String userId2: String	userId is a potential shard key. questionId is unique.
categories	name: String createdAt: Timestamp updatedAt: Timestamp	name is unique.
tags	name: String createdAt: Timestamp updatedAt: Timestamp	name is unique.
counter	id: String seq: Int32	

The database was designed in such a way to leverage the advantages of a NoSQL database such as MongoDB as much as possible.

The first advantage that MongoDB offers over traditional SQL databases is the ease of horizontal scalability using sharding. Hence, when designing the database, the attributes are created in a way to accommodate for potential shard keys, which are noted in the remarks column. This way, while the current database hosted on Atlas (the official MongoDB DBaaS) is not sharded due to the low load and traffic, the relevant collections can be sharded when needed. Sharding will allow for faster

query performance through the use of distributed computing, and in the case such as the `histories` collection, it may also be possible to zone shards to different locations based on the geolocation of the `userId` shard key for further performance improvements. Note that while sharding has not been implemented currently as the load does not necessitate it and the Atlas free tier does not support it, this schema is designed with future sharding in mind and can be upgraded easily if needed.

Another design consideration is to reduce the need for merges by using raw `String` values instead of mongo object refs to reduce joins. NoSQL frameworks like Mongo optimize performance and speedup by minimizing join operations, which require multi-collection lookups.

MongoDB in particular is optimized for document-oriented data that may have varying/undetermined columns/fields and other data types like images, like the `questions` collection. However, this freedom comes as a cost, which is the fact that data consistency is enforced much loosely as compared to SQL databases. This drawback is mitigated with developer enforced constraints in the backend code through the use of Mongoose ODM, which enforces constraints such as `NOT NULL` and `UNIQUE` which MongoDB does not natively enforce, reminiscent of SQL databases. Furthermore, developer defined checks are also in place in the database controller to enforce more complex constraints.

5.7 Testing

Through GitHub actions, whenever there are pushes to the master branch, the frontend and backend tests are automatically run in order to verify if the branch is okay to merge. The use of Github actions to automate the tests helps catch issues early in the development process and ensures that code changes do not introduce regressions.

5.7.1 Frontend Tests

Frontend testing was focused on unit tests for each Angular component using the Jasmine test framework. Angular defaults to creating and running tests using Jasmine and Karma. The choice to use Jasmine was that it works well with Angular and it is already the default framework to do so. For each Angular component, we had at least one test case, to test the instantiation and initialization of the component. Additional tests were added to the component depending on the properties of the component. The testing module for each component was done through the use of `beforeEach`. In `beforeEach`, there is a `TestBed`, which is an Angular testing utility that provides a testing module configuration for the unit tests. In the `beforeEach` function, `TestBed.configureTestingModule` is used to configure the testing module by declaring the necessary components to be tested, along with any imports needed. Next, the use of `Fixture` helps provide access to the component instance and allows interaction with the component's DOM.

5.7.2 Backend Tests

Backend testing also focused on unit tests for each microservice using the Jasmine test framework. The choice of this framework was based on its compatibility with Angular, and ensures a unified test framework for both the front and backend. This choice was aimed to minimize any integration issues between the two layers of the application. The `SuperTest` module simulates the functionality of each microservice by emulating an HTTP server environment throughout the duration of the tests. Requests can then be made to these servers to test if they work and respond correctly. Additionally, the `mongodb-memory-server` module serves as a mock database, providing an in memory database for services requiring database functionality. The unit tests themselves were straightforward and tested the functions of each controller in the microservices.

5.8 Deployment

PeerPrep is deployed as a multi-container application and can be deployed either locally with a Docker daemon, or on the cloud with Vercel and the Google Kubernetes Engine. The front end, API gateway, and each of the microservices have their own `Dockerfile`, which defines the instructions for building the image to set up the container for the component.

5.8.1 Local Deployment

Local deployment of PeerPrep is handled by the `docker-compose.yml` file, which defines the services in the project, the location of the `Dockerfile` in the repository with which to build the image for a service or the name of the image used by a service, the port mappings or exposed ports for a service, as well as the volumes used by a service, if applicable.

Port mappings are defined for the frontend and the API gateway with the `ports` configuration, mapping to a port on the host machine, which allows for external access from outside the Docker network. This allows access to the frontend through a web browser, as well as access to the API gateway through requests, which allows for local testing through software such as Postman.

In contrast, the ports for each microservice and each database container are defined by the `expose` configuration, which only exposes the respective port within the Docker network, preventing external access. This serves to isolate the backend from external access and prevent the circumvention of the API gateway, which uses sub-requests to the User Service to authenticate requests, ensuring that requests are protected from unauthorized access as intended.

Local deployment of the frontend was done in a container built with the NGINX image, serving the frontend as static content.

For local deployment, the API gateway resolves container names to addresses using the default DNS server for the Docker network, specified in the `default.conf` file with an address of `127.0.0.11`.

Microservices that require a database are provided with a database container built with the `mongodb/mongodb-community-server` image, mounted with a volume to persist the data in the database. The choice to deploy multiple database containers as opposed to a single database container was to allow for easier management of the volumes and testing of the local deployment. By providing a database container for each microservice that requires it, each volume contains only the data required by the microservice attached to the database container mounting the volume, and any changes to the schema which causes data for a microservice to be incompatible can be handled by simply deleting the respective volume. This allows the data for each service, contained in their respective volume, to be handled independently, and

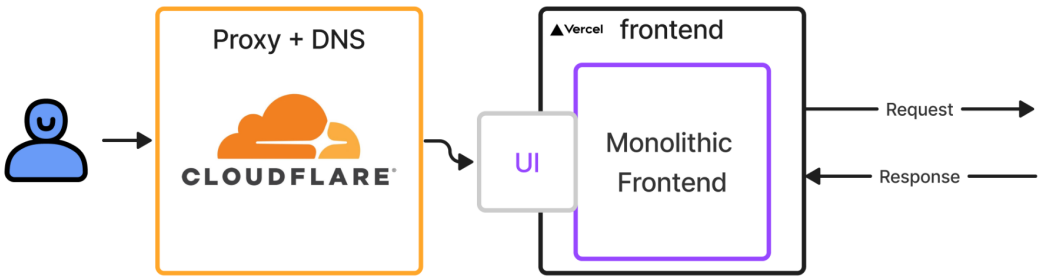
also restricts data management via command execution in a running container to only the relevant data.

5.8.2 Cloud Deployment

Cloud deployment of PeerPrep is handled separately for the frontend, API gateway, and the backend. Domain names are resolved by Cloudflare DNS. The frontend address, https://peerprep.yuanzheng.pro/ is proxied through Cloudflare to leverage it to optimize, cache, and protect all requests to the application, as well as to protect the server from DDoS attacks. The address of the API gateway for access to the backend is https://peerprepapi.yuanzheng.pro/.

The TLS certificates for the frontend and backend are provided by their cloud deployment services respectively. The Cloudflare SSL/TLS encryption mode is configured to Full, hence allowing it to provide end-to-end encryption for any traffic that is proxied through its network.

5.8.2.1 Frontend



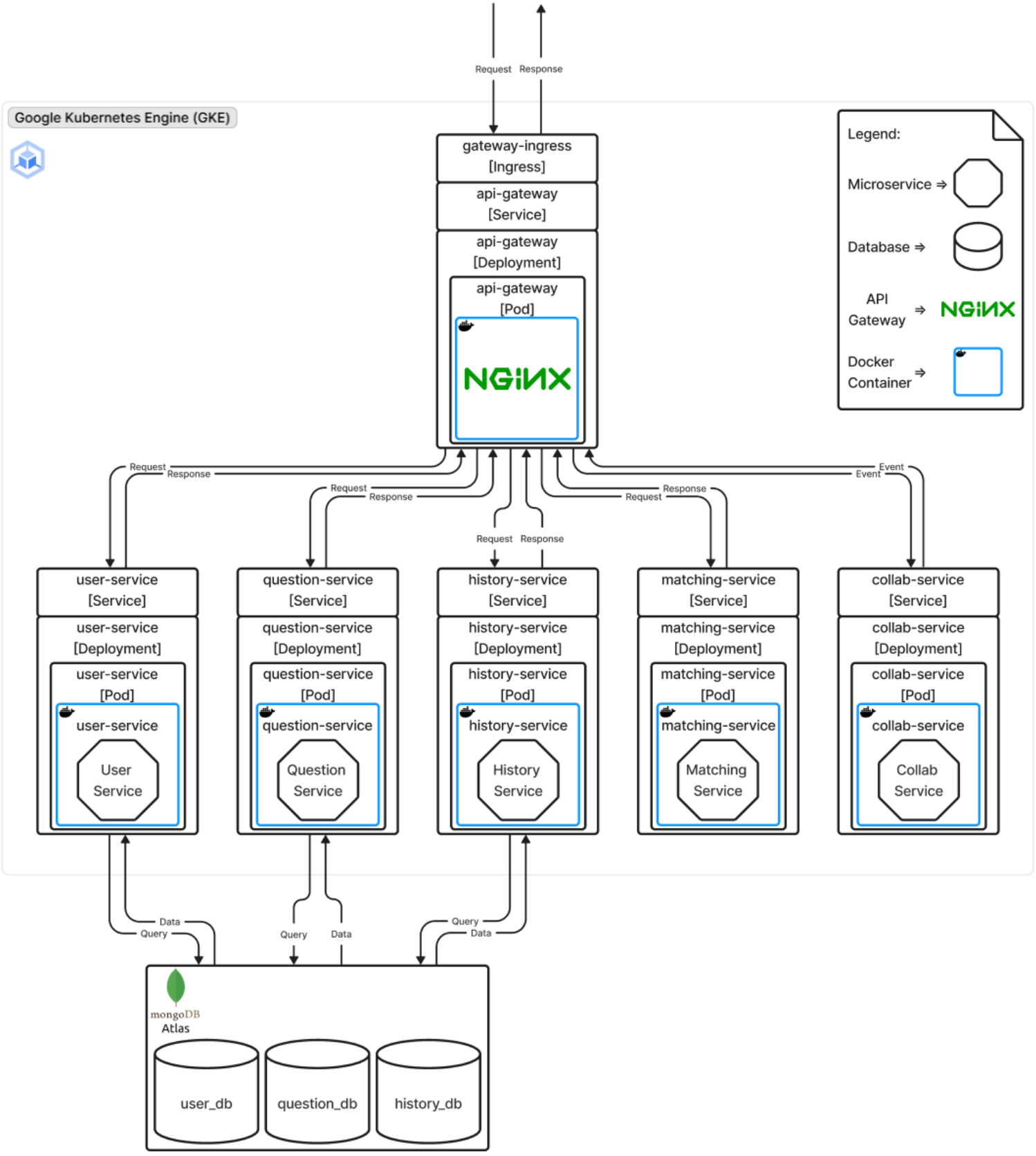
Architecture Diagram for Cloud Deployment of Frontend

The Angular frontend is hosted on Vercel, and deployment is configured to target the `./frontend` directory of the `master` branch.

5.8.2.1.1 Continuous Deployment

Vercel watches the target branch for new commits that are pushed or merged, and automatically builds and deploys whenever a change is made to the target branch.

5.8.2.2 API Gateway and Microservices



Architecture Diagram for Cloud Deployment of API Gateway and Microservices

The API gateway and microservices are deployed on the Google Kubernetes Engine (GKE). The deployment configuration of the gateway and each microservice making up the backend is defined by two files, a `<component>-deployment.yaml` file with the configuration for a deployment workload resource, which manages pods, and a `<component>-service.yaml` file with the configuration for a service, which groups pod endpoints into a single resource. The API gateway additionally has an `api-gateway-ingress.yaml` file, which configures an ingress to route external traffic to the API gateway deployed on GKE.

As is the case with local deployment, the cloud deployment avoids exposing any of the microservices to access from outside the cluster. Instead, external traffic is routed with the API gateway, which sets up an ingress to do so. This ensures that unauthorized access to certain routes is restricted as intended.

The configuration of the routing in the API gateway remains the same as in local deployment. The service names on GKE mirror the container names in local deployment, which allows the DNS provider for GKE to resolve the service names to the endpoint IP addresses within the cluster on GKE.

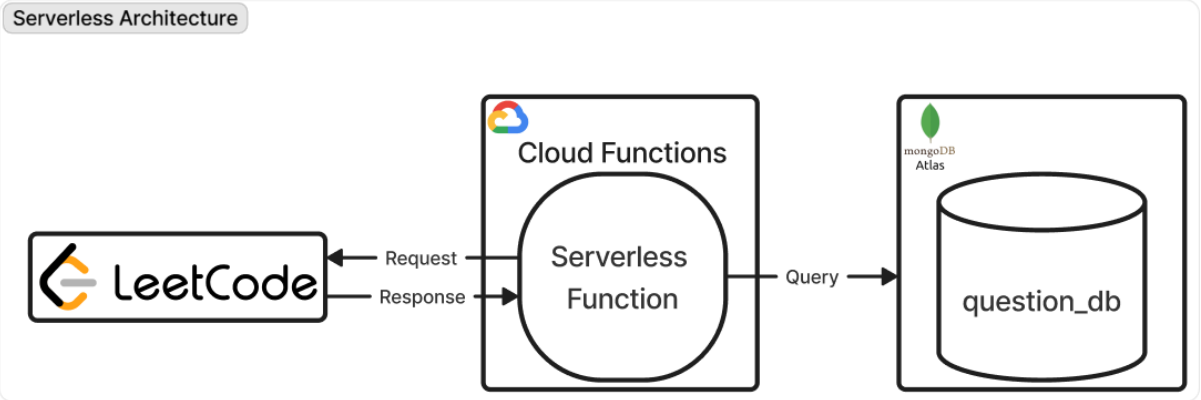
The MongoDB database is deployed on Atlas, the official MongoDB DBaaS platform. There are several advantages to choosing Atlas over a MongoDB volume in GKE, the first being the fact that Atlas offers peripheral DB utilities such as sharding, triggers and analytics on their console. The Atlas console offers useful insights into the database usage such as Opcounters, networking statics and other useful data analytics which will be otherwise difficult to synthesize when using a Docker volume. Furthermore, the Atlas console also allows developers to view and interact with the data, making for a better developer experience. While the current deployment is not sharded due to the low load and traffic and the fact that Atlas free tier does not support it, sharding can be added when the need arises.

5.8.2.2.1 Continuous Deployment

To deploy to GKE, a CD pipeline was set up, with a separate Github Actions workflow defined for each component in a `deploy-<component>.yml` file, which triggers on any push to the `master` branch affecting the sub-directory of the repository concerning the component. Each workflow was also configured to be able to be run manually with `workflow_dispatch`. Defining a separate workflow for each component allows finer control over how and when each microservice, or the API gateway, is deployed to GKE. This allows us to redeploy each component individually, instead of having to deploy every component on making any change to the repository. The steps of each workflow are shown below.

1.	Checkout the repository
2.	Authenticate to Google Cloud with a service account key for a service account with sufficient permissions, stored as a repository secret.
3.	Setup the Google Cloud CLI with the project ID, stored as a repository secret.
4.	Configure Docker to use the Google Cloud CLI as a credential helper for authentication.
5.	Retrieve GKE credentials for deployment to the cluster.
6.	Build the Docker image with the relevant `Dockerfile` and tag it with the commit SHA and the “latest” tag.
7.	Push the built Docker image to the Google Artifact Registry
8.	Deploy the Docker image to the GKE cluster with the relevant `.yaml` and files.

5.9 Serverless Function



Architecture Diagram for Serverless Function

The serverless function was implemented using JavaScript and deployed to the Google Cloud Platform. The function fetches 200 questions from “LeetCode.com” (third-party source) and populates the question database with the retrieved questions. Periodic updates are run on the application (1 month schedule) in which the serverless function requests 200 questions again. However, when running updates, the function will either update the question (if it already exists in the database) or add the new question to the questions database.

The serverless function starts off by connecting to the question database. Afterward, a GraphQL query fetches 200 questions from LeetCode’s questions API. Next, we verify that the fetched question has a nonempty title, nonempty description, nonempty difficulty, and nonempty category fields. If all of these fields are nonempty, then we add them to our question database. If one of those fields is missing, we skip the question. During each iteration, there is a check to determine if the current question exists. If it does, we simply update the existing entry in the database, else we create a new entry in the database. Once finished, we return a response of `200` if no errors were encountered, else if there is an error, we return a response object of `500` with the appropriate error message.

6. Suggestions for Improvements and Enhancements

6.1 Scalability

One possible enhancement to PeerPrep is scalability. While not the primary concern for the current iteration of the project, the support for scalability was a factor in choosing the stack for PeerPrep. For example, Socket.IO provides support for horizontal scaling with multiple servers, while the NGINX API gateway supports load balancing configuration for horizontal scaling. MongoDB Atlas, used for database support in the cloud deployment also supports scalability with Atlas cluster autoscaling. In particular, the decision to deploy on the Google Kubernetes Engine (GKE) was in part to leverage scaling in future extensions, with support such as the GKE cluster autoscaler.

6.2 Communication Enhancements [Collaborative Space]

Improvements could also be made to the communication feature in the collaborative space. Given the nature of PeerPrep as a technical interview preparation platform, adding a video call or voice call feature could enhance the effectiveness of interview preparation with PeerPrep by providing an environment that better emulates a technical interview, allowing users to also practice their verbal communication, through speech, and non-verbal communication, through body language.

6.3 Code Execution

The collaborative space can also be enhanced with a mechanism for code execution. Code in the code editor could be extracted and executed in an appropriate environment, potentially with a set of test cases as input, and the results presented to the users in the collaborative workspace. This would allow users to test for the correctness of their code, as would be the case when presented with a question and being expected to come up with some solution in a standard technical interview, and would remove the need to copy the code elsewhere for testing.

7. Reflections and Learning Points

7.1 Development Process

The development process was one of the most important learning points of this project, where focus extended beyond technical aspects. One important factor to learn from was effective communication when working in short sprints, namely, that a team could only operate effectively if there was active communication between its members to keep each member updated on the status of their tasks and any issues faced. This helps to create a tight knit development team that is able to coordinate well when working on tasks in parallel, and integrate the work of each member effectively into the main project.

7.2 CI

CI introduced the team to the usage of GitHub actions for workflow automation. The use of `.yaml` configuration files to configure workflows was new, and setting up CI with GitHub Actions workflows defined in `.yaml` files led to more familiarity with the nuance and structure of such files. Some of the key features of such files are shown below.

Event Triggers	Workflows can be triggered by specific events, such as pushes to certain branches.
Jobs	Each workflow consists of one or more jobs that run concurrently.
Steps	Jobs are composed of steps, which define individual tasks to be executed.
Actions	Steps can include predefined or custom actions, encapsulating specific functionality.

7.3 New Technologies Learned

Developing PeerPrep led the team to embrace new technologies, gaining more in-depth knowledge about the functionality and use of each. Notable examples include the Angular framework, Socket.IO, NGINX, and deployment with Vercel and the Google Kubernetes Engine (GKE). Learning more about these technologies served to enhance the adaptability and technical proficiency of each member, and led to much growth in software engineering proficiencies.

8. Appendix

8.1 Deployment Instructions

8.1.1 Local Deployment - Docker Desktop

Local deployment for development and testing purposes was primarily done with Docker Desktop.

Requirements:

- Install Docker Desktop
- Configure the `JWT_SECRET` for use in local deployment in `/backend/user-service/config/auth.config.js` if desired. The default value used is `"the-inner-machinations-of-my-mind-are-an-enigma"`

The instructions for local deployment are as follows:

1. Ensure that a Docker daemon is running.
2. Navigate to the root of the project.
3. In a terminal, run the following command:
   I. `docker-compose up` , or,
   `docker-compose up -d` , to run in detached mode

The local deployment of PeerPrep may be accessed at the URL **`http://127.0.0.1:8000`**.

To elevate a user to an `admin` role in the local deployment:

1. Execute the following commands in the integrated terminal of the running `user-db` container:
   (On Docker Desktop, go to "Containers" > "`user-db`" > "Exec")
   I. `mongosh`
   II. `use user_db`
   III. `var adminid = db.roles.findOne({name:'admin'}, {_id:1})._id`
   IV. `db.users.updateOne({ username: '<CREATED_USERNAME>' }, { $set: { roles: [adminid] } })`

(*where `<CREATED_USERNAME>` is the username of an existing user to be elevated to an `admin` role*)

To tear down the project:

1. In a terminal, run the following command:
   I. `docker-compose down`

8.1.2 Cloud Deployment - Vercel, Google Kubernetes Engine (GKE)

Cloud deployment was done with Vercel for deployment of the Angular frontend, and the Google Kubernetes Engine (GKE) for backend deployment, consisting of the API gateway and the microservices.

Requirements:

- Set up a Vercel account
- Set up a Google Cloud project
- Set up a cluster in the project
- Set up a repository in the Artifact Registry
- Set up a service account with sufficient permissions
  - *For reference, PeerPrep uses a service account with the following roles:*
    - Artifact Registry Reader
    - Artifact Registry Repository Administrator
    - Artifact Registry Service Agent
    - Artifact Registry Writer
    - Editor
    - Kubernetes Engine Service Agent
    - Security Reviewer
    - Service Account Token Creator
    - Service Usage Admin
- Generate a service account key for the service account
- Configure the following GitHub repository secrets ("Settings" > "Security" > "Secrets and variables" > "Actions"):
  - `GKE_PROJECT`: the ID of the Google Cloud Project set up
  - `GKE_SA_KEY`: the generated service account key for the service account with sufficient permissions
- Set up a MongoDB Atlas account
- Under "Secrets & ConfigMaps" on GKE, configure the following for the created cluster:
  - A `db-admin` secret with `db-username` and `db-password` matching the username and password of the created MongoDB Atlas account
  - An `auth-jwt` secret with `JWT_SECRET` set to the desired secret for cloud deployment
- Provision TLS certification for the domain name used by the backend
- Reserve a global static IP on Google Cloud
- Configure `./frontend/src/environments/environment.ts` with the certified domain name `<BACKEND_DOMAIN_NAME>` as follows:
  - `BACKEND_API: <BACKEND_DOMAIN_NAME>`

The instructions for cloud deployment of the frontend with Vercel are as follows:

1. Log in to Vercel, and add a new project from the homepage.

2. Under “Import Git Repository”, press “Import” on your GitHub repository containing the project. If the repository is not listed as an option, you can add it by clicking on “Adjust GitHub App Permissions →”.
3. At the “Configure Project” page, give the project a name. Choose “Angular”, and select `./frontend` as the root directory.
4. Click deploy. Vercel will now build the deployment and update it whenever a change is pushed to the repository.

The instructions for cloud deployment of the backend, i.e., the API gateway and microservices, on the Google Kubernetes Engine (GKE), are as follows:

1. In each `deploy-___.yml` file of the `/.github/workflows` directory, configure each of the `GKE_CLUTER`, `GKE_ZONE`, `REPOSITORY_NAME` as per the setup in "Requirements"
2. Configure the image in each of the `.yaml` files under `./gke` to point to the appropriate image in the repository on the Artifact Registry
3. Configure the `global-static-ip-name` annotation in `api-gateway-ingress.yaml` to point to the global static IP you reserved
4. Manually trigger the deployment workflows for each component under "Actions", or have them automatically trigger on changes to the relevant files

8.1.3 Serverless Function Deployment - Google Cloud Platform (GCP)

Configure the GitHub repository secrets `GKE_PROJECT`, the Google Cloud Project ID, and `GKE_SA_KEY`, the service account key for a service account with sufficient permissions.

The serverless function is deployed to Google Cloud Platform as a Cloud Function. It will automatically deploy to the Google Cloud Platform when a push to the `master` branch is performed. This is due to automation included using GitHub actions workflows configured with the `deploy-serverless.yml` file.

The instructions to manually deploy the serverless function are as follows:

1. Ensure that `gcloud` is installed. Run the following command in terminal
    I. `npm i gcloud`

2. Log in to Google Cloud Platform. Run the following command in terminal
    I. `gcloud auth login`

    II. Login using your Google Cloud Platform account

3. Next, `cd` into the function directory:
    I. `cd question-fetcher`

4. Deploy the application using the following command:
    I. `gcloud app deploy`

8.2 Sprint Logs

[Week 4] (Sprint 1) 11/9	Assignee
Question Service ● REST API Development ● MongoDB integration ● CRUD of Questions	**Chia Yu Hong**
Client-side SPA CRUD ● Develop a Single Page Application ● Client-side storage with local data persistence ● CRUD in Js ● Simple frontend using HTML & CSS	**Chinthakayala Jyothika Siva Sai**
Authentication ● Authentication state management with JWT tokens ● Definition of User & Admin roles ● Front-end development for registration & login	**Justin Widodo**
User Service ● User Login system, integrated with Authentication ● Create & Read of Users	
[Week 5] (Sprint 2) 20/9	**Assignee**
Containerization ● Docker containerization of question service ● Docker containerization of user service	**Chia Yu Hong**
Database ● Database schema design	**Tan Yuan Zheng**
Serverless ● Serverless Function design	**Xu Richard Chengji**
[Week 6] (Sprint 3) 27/9	**Assignee**
Frontend ● Frontend development for CRUD of questions ● Frontend development for matching service ● Integration of frontend with backend	**Chinthakayala Jyothika Siva Sai**
Matching Service ● Develop matching service backend	**Justin Widodo**
Serverless ● Developed serverless function to fetch questions from Leetcode ● Deployed the function using AWS Lambda to verify that the function is working correctly	**Xu Richard Chengji**
[Week 7] (Sprint 4) 4/10	**Assignee**
Collaboration Service ● Develop collaboration space between users ● Frontend development for collaboration space	**Chia Yu Hong**
Enhancements to Authentication/User Service ● Develop Delete and Update of users	**Justin Widodo**

• Frontend development for deleting & updating users • Enhance session management to cover new services & resources	
[Week 8] (Sprint 5) 11/10	**Assignee**
Syntax Highlighting • Develop syntax highlighting feature in collaboration space	**Chia Yu Hong**
Communication • Develop communication via chat in the collaboration space	
History Service • Develop user history for each attempt • Frontend development for the history service	**Tan Yuan Zheng**
[Week 9] (Sprint 6) 18/10	**Assignee**
API Gateway • Implement an API Gateway with Nginx as a reverse proxy between frontend and backend • Authenticate requests using `auth_request`	**Chia Yu Hong** **Justin Widodo**
Question Tagging • Implement question tagging for the backend • Retrieving questions on the fly during a session initiation	**Chinthakayala Jyothika Siva Sai**
[Week 10] (Sprint 7) 25/10	**Assignee**
Frontend • Refine frontend for additional services	**Chinthakayala Jyothika Siva Sai**
CI • Writing unit tests for all functions • Writing frontend tests for components • Develop workflows for GitHub actions	**Justin Widodo** **Xu Richard Chengji**
[Week 11] (Sprint 8) 1/11	**Assignee**
Code Review • General bug fixes • Improvements to error handling • Improvements to UI	**Justin Widodo** **Chinthakayala Jyothika Siva Sai**
CI • Fixing integration bugs	**Xu Richard Chengji**
[Week 12] (Sprint 9) 8/11	
Cloud Deployment •	**Tan Yuan Zheng**
CI • Integrate CI into main repository	**Xu Richard Chengji**
Serverless • Deploying serverless function fo Google Cloud Platform	**Xu Richard Chengji**