CS3219 Software Engineering Principles and Patterns
AY22/23 Semester 1

Final Project Report

Group 33

Team Members	Matriculation Number
Ho Zong Han	A0218388R
Kwan Hao Wei	A0218213R
Chua Wei Jie Timothy	A0218271J
Goh Zheng Teck	A0218285Y

Tutor: Koh Rui Ling

Table of Contents

Glossary of Terms

Term	Description
ETCD	A disk-based key-value store technology.
STUN	A technique for performing NAT Hole-Punching for reversed traversal on NAT-based routers.
TURN	A proxy for bridging UDP traffic between 2 clients.
Code Snapshot	A frozen record of the contents of the code workspace.
Attempt History	A record of a code snapshot for a particular question.
SOLID	A series of 5 software design principles, the Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle, Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP)
CRDT	Conflict-Free Replicated Data Type is a data structure that can be replicated across a distributed system.
Elo Rating	A formula for estimating the skill level of a particular user.
CRUD	Create, Read, Update, and Delete.
Workspace / Editor	A text editor space where code collaboration takes place between users.
Session / Room	The terms are used interchangeably to refer to a collaboration room that contains at most 2 users.

1. Introduction

This report details the process of the development of a web application known as PeerPrep. The overall idea is to provide a collaborative workspace for solving coding challenges, similar to platforms like Kattis and Leetcode, but with the additional quirk of allowing 2 users to simultaneously work on the same piece of code.

1.1. Links to Project

The following are links to the project's repository and deployments. The same information can be found on GitHub under environments.

Repository	https://github.com/CS3219-AY2223S1/cs3219-project-ay2223s1-g33
Staging	http://ec2-18-138-225-25.ap-southeast-1.compute.amazonaws.com/
Production	https://peerprep.launchsite.com.sg/

1.2. Product Vision

The product vision, which encapsulates the team's high-level ideals for the application, was formalised and refined before any planning or development work began. This vision served as a guide for subsequent product design and requirements definition. The vision statement that the team reached is as follows.

PeerPrep is a web application for collaborating on technical interview questions and challenges. With an intuitive and simplistic user interface (UI), users can practise with other like-minded users on coding challenges and take each other to new heights through peer learning. The collaborative nature of the application breaks the monotonous grind of solving technical obstacles alone and provides questions tailored to the user's capabilities and preferences for the optimal experience.

1.3. Scope

The team also defined the target audience and the related high-level functional goals for the product, that is, what the product should do for its end-users. These definitions formed the scope document, which detailed the boundaries within which the product should be designed. The scope document is as follows.

Target User(s)	● People who have some prior coding experience ● Students / Professionals who are preparing for coding interviews ● Professionals who are looking to refresh their technical skills ● Professionals who want to learn modern coding patterns ● Beginners who are trying to learn algorithms and problem-solving ● People looking for coding buddies
High-Level Functionality	The application should ● Provide a collaborative workspace that ○ Allows both users to see the question ○ Allows both users to work on the same code ● Provide questions that are separated by difficulty levels ● Show test cases and expected outcomes, from which correctness of a given solution can be determined

1.3.1. Envisioned Usage

This section is meant to elaborate on the high-level functionality defined in the scope section, which in turns helps scope the requirements for the product. This process flow only captures the absolute minimal functionality expected and does not cover failure cases or unexpected events.

A typical user would use the product in the following sequence.

1. User opens his/her web browser and navigates to the application
2. User registers for an account if it does not already exist, and logs into the account
3. User enters the matching queue and looks for a partner to code with
4. User finds a match, and enters the collaborative workspace environment
5. A question is presented to all users in the same workspace, and the users solve the question together
6. The users may choose to mark the question as complete or leave the session

1.4. Quality Attributes

Quality attributes, which would guide the development of requirements, were also determined by the team, and ranked according to their perceived importance in the product design. In particular, the team separated the list of quality attributes into 4 categories, with 3 priority levels and the last category being “irrelevant”. The following table summarises the ranking of these quality attributes.

High Priority	Medium Priority	Low Priority
Usability	Installability	Availability
Modifiability	Performance	
Scalability	Robustness	
Security		

Table 1 – Prioritisation of Quality Attributes

Ultimately, the team decided to focus on usability, scalability, and modifiability, with security as a nice-to-have. The other quality attributes may factor into the requirements specifications directly or indirectly but are not the primary focus when drafting and prioritizing the requirements. The following sections detail the reasons that these quality attributes were selected.

1.4.1. Usability

From the vision statement, the team intends for the application to be simplistic, where the actions that a user can do are immediately apparent without any further explanation. This streamlines the user experience and makes it so that both beginners and seasoned users can utilise it to the fullest extent. This is indirectly related to installability, where the goal is for the system to be used in the same way across all desktop machines, as long as a web browser is present, without requiring the user to take additional installation or setup steps.

1.4.2. Scalability

While the system may not be deployed at scale in the context of the project, the team decided that it should still incorporate design decisions that enable simple capacity scaling similar to that of production systems. This elastic scaling capability is useful in that more resources can be added when necessary, and torn down when the load eases off, making small-scale local testing and large-scale deployments simple. It also provides flexibility in the development and deployment process, with the ideal goal being to efficiently allocate resources to parts of the system that need it the most. Scalability is also closely related to availability and robustness, both of which can be transiently achieved to some extent with redundancy from scaling.

1.4.3. Modifiability

Due to the speed at which development will proceed under this project, it is important that the overall system is receptive to changes as new features are introduced. This includes design decisions that encapsulate the spirit of the Open-Closed Principle (OCP) and Single Responsibility Principle (SRP), where new features and components plug in seamlessly into the existing system, while modifications to existing components should not affect the other components. This allows for independent parallel development efforts, which is critical in the context of this fast-paced project.

1.4.4. Security

Since the product is a web application that involves network communications, security is a key consideration, involving secure data transport, identity management and attack resiliency. However, since the only sensitive piece of information handled by the application is the users' passwords, security is not of utmost importance. However, it is still beneficial to adhere to best security practices and security is therefore included as a nice-to-have quality attribute.

2. Requirements

This section summarises the key ideas behind the development of the requirements and some notable decisions that the team had to make. The full list of requirements can be found in Appendix A.

2.1. Requirements Organization

Broadly, the requirements can be separated into functional and non-functional requirements. Within each category, they are further subdivided to make requirements management easier. A hierarchical requirement coding is used to help identify the requirement category, which differs for functional and non-functional requirements. The coding used is as follows.

```
<F(unctional) / N(on-Functional)><Category> - <Running Number>
```

2.2. Functional Requirements (FRs) Organization

Functional requirements are categorised in phases, which will be further detailed in the development schedule section. Generally, all requirements in a phase need to be completed before the next phase can begin. However, exceptions for the lowest priority in a phase can be made, which may be a nice-to-have, and would have inconsequential effects going into the next phase. The prioritisation of the requirements is defined globally, and generally decreases with the phase. It is intended as requirements defined in later product phases naturally have a lower priority in the bigger picture.

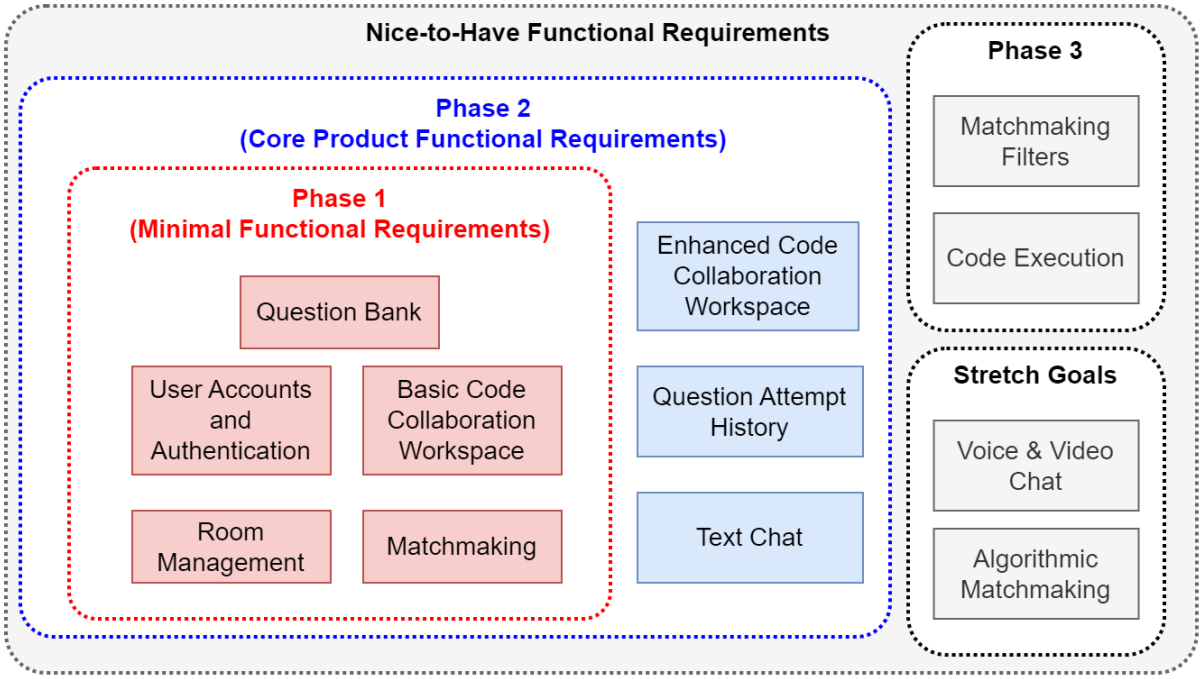


Figure 2 - Global View of Requirements

2.3. FR Categories

The following table summarises the high-level intents of each FR category. The exact FRs in Appendix A1 breaks down these high-level goals into more granular and refined requirements. These categories are broadly derived from the envisioned usage set out earlier.

FR Category		FR Category Intent
A	User Accounts and Authentication	The system should incorporate some way of identifying between different users
B	Matchmaking and Queueing	The system should be capable of pairing users up for a room
C	Rooms Management	The system should manage the lifecycle of collaboration sessions
D	Code Workspace	The system should provide an interactive workspace that users can collaborate in
E	Question Generation and History	The system should maintain a database of questions and each user's attempts for them
F	Text Chat	The system should allow users to communicate in textual format during collaboration
G	Code Execution	The system should provide some way for users to test their code through execution
H	Voice Chat	The system should allow users to communicate using voice during collaboration

Table 3 – FR Categories

2.4. FR Prioritization

The backend requirements aim to focus on the essential features required for achieving peer coding functionality, such as enabling users to work on the same code at the same time. To achieve this goal, priority was placed on functionality that is core to the product, which helps achieve the product vision. These FRs primarily fall under the categories A to E and are defined in Appendix A1.1 (Minimum Viable Product FRs).

These core minimal FRs are augmented by additional FRs in Appendix A1.2 (Phase 2 FRs), which include FRs mainly from the same categories A to E and additionally category F (Text Chat). These FRs incrementally increase the usability of the product, particularly on the user-facing frontend. They focus primarily on components that the user interacts with, guided by the vision statement that the product should be simple and intuitive. Therefore, instead of introducing many features which might convolute the product, the requirements focus on the minimal set of features that would satisfactorily deliver the intended functionality.

FRs detailed in Appendix A1.3 (Phase 3 FRs) complete the experience by including functionality that would make the product a complete experience for the end user. These FRs include the category G (Code Execution) and tweaks to the matchmaking system in category B. The code executor allows the user to verify that the code submitted is working correctly. This functionality is seen as essential in completing the user experience for collaboration but does not affect the product's core vision if omitted. Therefore, it has lower prioritisation even though it provides a more complete product that users can use without necessitating external dependencies.

Other FRs detailed in Appendix A1.4 (Stretch Goals) such as voice chat functionality were considered, but are very low in priority as these functionalities will have marginal impacts on the peer-coding component, with minimal benefits to the overall peer-coding experience. While such functionalities are nice to have, the limited time frame of the project warrants a conservative approach to defining FRs, with a focus placed on FRs that are essential to the product's vision.

2.5. Non-Functional Requirements (NFRs) Organization

Non-functional requirements (NFRs) employ a different grouping system since they should hold in all phases of the product life cycle. Instead, they are categorised based on how they affect the user experience, specifically, whether they are:

- Product-Centric NFRs (Appendix A2.1) which directly impact user experience, or,
- Deployment-Related NFRs (Appendix A2.2) that indirectly affect the user

2.6. NFR Trade-Offs

NFRs that relate to the categories defined in the quality attributes section are the primary focus, and therefore have higher priorities amongst all the NFRs.

The team decided to focus heavily on the scalability of the backend, which would afford the application elasticity in its capacity. This has implications for lowering operational costs and increasing peak loading capabilities, both of which would be beneficial to the user experience in terms of latency, as well as reducing costs for the upkeep of the system.

Modifiability is also another key focus in the NFRs, which concretely define what it means for the system to be easily modifiable. These requirements work together to allow new components to be added easily at the expense of performance, which are modestly defined in the NFRs. Since performance was not a selected quality attribute, performance-related NFRs are generally medium or low priority, except for key bottlenecks where low performance starts to negatively impact user experience.

The product-centric NFRs focus heavily on intuitiveness and usability. The overall goal of the product-centric NFRs is to ensure that the users have a painless experience when using the application across the devices used and user skill levels, along with some minimal security considerations, which is a nice-to-have quality attribute.

3. Software Development Life Cycle

The team employed an iterative AGILE software development methodology, producing complete and deliverable products at fixed milestones. Based on the SCRUM framework, the team followed a process that bears a close resemblance to the idea of having weekly sprints. However, there were several key modifications made to the framework that the team deemed necessary to maintain the pace of development.

Firstly, the team eliminated the daily stand-ups that SCRUM describes due to the difficulty of arranging common timeslots to meet up. Moreover, the team believes in a self-driven workflow, where members have the flexibility of contributing when they most prefer, as long as targets are achieved weekly. Following this approach, the only meetings that the team held were weekly sprint meetings, in which pending work items in the sprint and product backlogs were reviewed.

Another modification made to SCRUM is the introduction of a higher-level idea of a phase, on top of the weekly sprints. A phase is described at the product backlog level, where groups of work items in the product backlog are tagged to a phase based on their corresponding FRs. Generally speaking, the highest 2 priorities of work items for a phase should be completed before the next phase can begin. A weekly sprint may select any work items from the current active phase, helping the team better define product iterations on top of the weekly Sprints.

3.1. Modified Workflow

Therefore, the development workflow roughly follows the diagram shown below. The product backlog is attached in Appendix B, while the sprint backlogs are managed on GitHub issues.

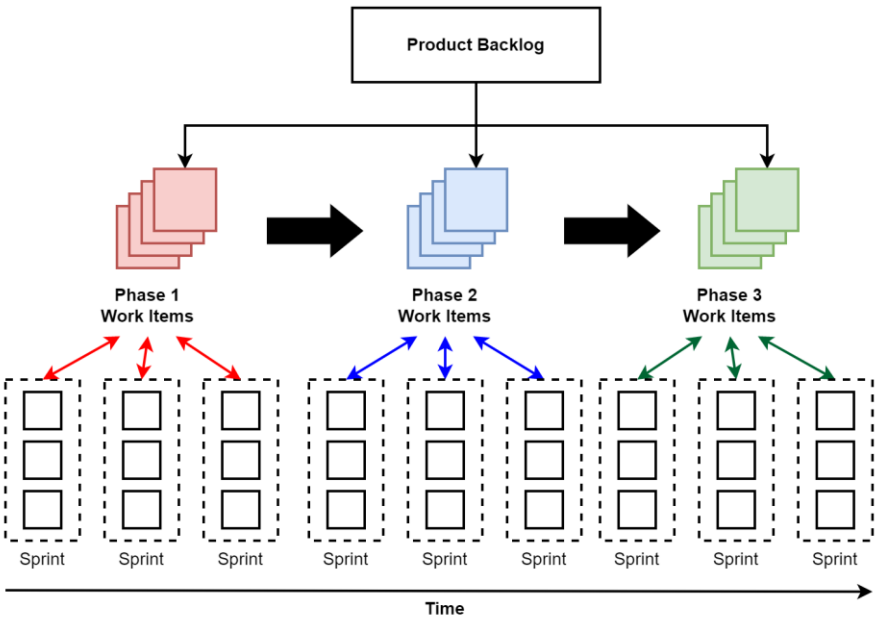


Figure 4 - SDLC Workflow

3.2. Schedule

In order to more concretely timebox the phases in development, the team used an approximate guide of 3 weeks for a single phase, including the recess week. The timeline below shows the scheduling plan, with descriptions of the deliverables of each milestone (separate from the module's official milestones).

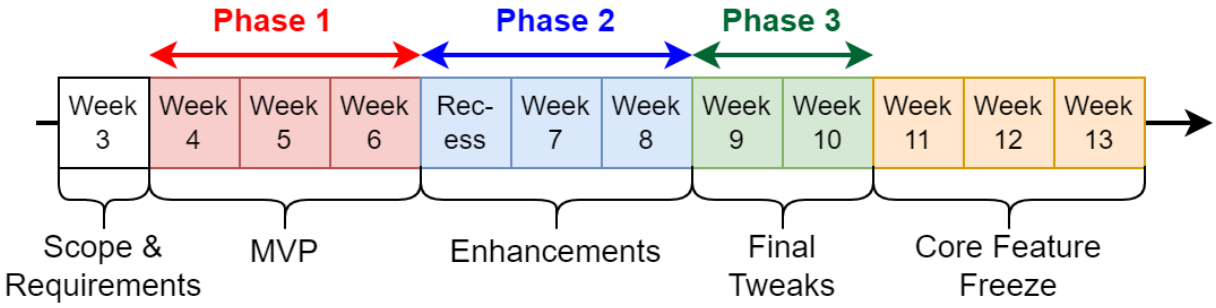


Figure 5 - Development Timeline

Phase	Deliverable
Scope and Requirements	Clear definition of the overall product and its associated requirements.
Minimum Viable Product (MVP)	The minimum viable product which contains the minimal subset of functionalities required to achieve the product's vision and goals. This includes the set of FRs in Appendix A1.1.
Enhancements	A refined version of the MVP, which contains quality of life improvements and improvements to the core product. This includes the set of FRs in Appendix A1.2.
Final Tweaks	A final refinement of the feature set in the core product, which involves small changes to the existing functionalities that further improve on the user experience. The features implemented in this phase are mainly nice-to-haves. This includes the set of FRs in Appendix A1.3, and if time permits, Appendix A1.4.
Core Feature Freeze	The feature freeze is a hard lock on the feature set of the core product, that is, any additional features that may be developed are non-essential, considered as nice-to-haves. Any feature developed during this phase must plug in with minimal changes to the core product.

4. Product Ownership and Responsibilities

The team operates on the belief that it is often more efficient to have a single source of truth for information, planning and management. Oversight from this central entity eases coordination and communication, eliminating confusion that may arise from misalignments within the team.

4.1. Product Ownership

In order to better manage the product's feature set within the team, a notion of "ownership" is introduced. Team members own a particular subset of features, for which they are the primary point of contact for any issues that might arise from it. The owner of a feature is meant to be the subject-matter expert on the area, including all its related components and the data flows between them to achieve the feature's functionalities. This ownership system helps ensure that every part of the product is being taken care of, helping delegate responsibilities at a high level.

It should be noted that a feature's owner is only responsible for oversight and does not restrict other members from contributing to that feature should the need arise.

Zong Han	**Hao Wei**
Authentication and Matchmaking	User Experience and Synchronisation
Timothy	**Zheng Teck**
Attempt History and Collaboration	Questions, Deployment and User Experience

Table 6 – Feature Owners

4.2. Team Member Responsibilities

With the agreement of all members, a hierarchy is established internally with a clear leadership structure. The key appointment holders are responsible for the oversight of a subsystem within the product, providing technical oversight in implementation as opposed to the functionality oversight of feature owners. The team structure is shown below.

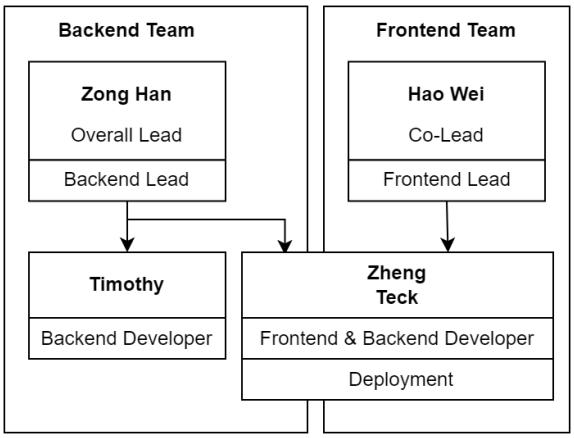


Figure 7 - Team Structure

4.3. Team Member Contributions

The team strongly believes that every member should know, at least at the surface level, every part of the system. This facilitates code reviewing and debugging, where everyone knows roughly where to look when issues surface.

As such, the table below lists the areas that each member has contributed the most to, and is by no means exhaustive since everyone has dabbled in every part of the system. **The team believes that all members have played their part in contributing to the project.**

Zong Han	● Backend Architecture Design
	● Design and Maintain API Server Framework for Node services
	● Develop and test all high performance Golang components
	  ○ Gateway
	  ○ Session Service
	  ○ Matchmaker
	● Develop and Test User Service
	● Develop and Test Matching Service
	● Setup Kubernetes Deployment
	● Setup ProtoBuf Workflow
	● Setup CI Workflow

Hao Wei	● Frontend UX and Architecture Design ● Primary Developer for Frontend ● Enhance and Upkeep Frontend Look-and-Feel ● Develop Frontend-Backend Networking ● Develop and Integrate Multi-Client Synchronisation ● Overall Team Administration
Timothy	● Develop and Test Collaboration Service ● Develop and Test History Service ● Develop and Integrate Execution Service ● Develop Frontend-Backend Networking ● Integrate Inter-Service Messaging Systems Across Services ● Setup Docker Compose Deployment
Zheng Teck	● Develop and Test Question Service ● Database Schema Design ● ORM Layer Implementation ● Assist in Frontend Development ● Setup Database ● Setup AWS Deployment ● Setup CD Workflow

Table 8 – Team Contributions

5. Architecture

The high-level architectural pattern employed is the microservices pattern, which was selected for its scalability and strong adherence to the Separation of Concerns (SoC) principle. Due to the fragmented nature of the pattern, groups of closely related components can be developed and maintained independently and in parallel, allowing feature ownership to be used as a mechanism for delegating work. Unlike monolithic systems, this structure also enables each component to act and scale independently, contributing to the overall scalability of the system.

Communication between different services is performed over a small set of well-defined interfaces, which takes the form of API calls over the network. This reduces the coupling between microservices and contributes to modifiability (NFR NMM-1 and NMM-2). New components added to the system can plug into the system and tap on functionality provided by existing services, while the internal implementations for each service can change frequently as long as the contracts defined in the external facing interfaces are satisfied.

The system architecture is shown below. Note that the configuration of the Gateway differs based on deployments, depending on whether an NGINX server sits in front of the gateway. This variance will be discussed in the deployment section.

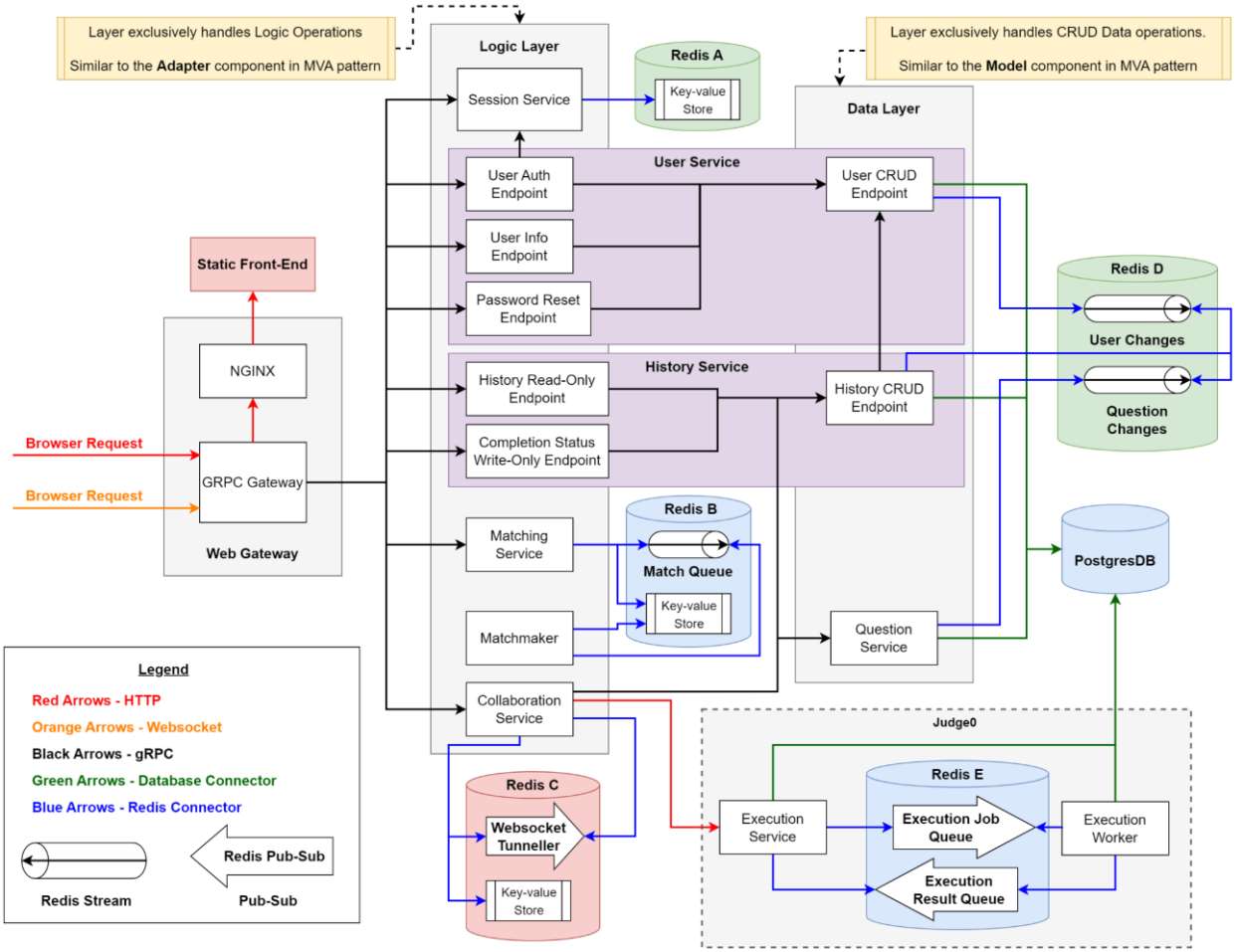


Figure 9 - Overall System Architecture

5.1. Architectural Pattern Design Considerations

There are 4 key ideas that drive the design of the system's architecture, namely,

- Horizontal Scalability
- Fault Tolerance
- Service Interoperability
- Layered Abstraction

The former 2 ideas relate to scalability (NFR NSS-1 to NSS-5), while the latter 2 relate to modifiability (NFR NMM-1 to NMM-3).

5.1.1. Horizontal Scalability

Horizontal scalability (NSS1 to NSS-3) is the ability for services to scale up and down in an elastic manner by simply spawning more instances of itself, a scaling strategy that is facilitated by the use of the microservices pattern. This approach to scalability dictates several architectural design choices. Specifically, each service should be designed and implemented in a stateless logic-only manner, providing operations on data but not storing them internally. Stateful components like persistent data or queues need to be offloaded to centralised repositories such as caches or databases, effectively maintaining a single globally consistent state across all services and their multiple duplicated instances.

This design consideration can be clearly seen in the architecture of the system, where Redis and the database are extensively used to store all the system's state. As a result, each service is stateless and can be duplicated and de-duplicated seamlessly.

5.1.2. Fault Tolerance

Fault tolerance is closely tied to horizontal scalability, and is in fact achieved through horizontal scaling, but has a different focus. A fault tolerant system should be able to seamlessly failover services to other instances with minimal impact on the integrity of the entire system (NSS-4). Using duplicated instances of a single service, the system's architecture accommodates such a fault tolerant deployment owing to the stateless nature of the services. All traffic will be redirected to the healthy instances, with the rest of the system not even knowing a failure has occurred.

The system is also designed to avoid cascading failures, with failure boundaries clearly defined and isolated to each service (NSS-4). If a downstream service is not available, the upstream services will gracefully handle the problem instead of themselves crashing.

5.1.3. Service Interoperability

Interoperability between different services is a design consideration that heavily influenced the design choices related to inter-service communication. The high-level goal is to achieve system-level modifiability through service-level interoperability by standardising all communications between services to a common, language agnostic protocol (NMM-1). This allows services to be developed in different languages and abstracts the interface between services from their implementations. New services can be developed in any language as long as they implement the common protocol, introducing flexibility into how the system can be extended (NMM-2).

This consideration can be clearly seen in the architecture, where all services in the backend employ the gRPC protocol for communication with each other, with the exception of the execution service. This exception is due to the use of an external library, Judge0, that only supports API calls over HTTP, a limitation that the team has no control over.

5.1.4. Layered Abstraction

The architecture follows the Model-View-Adapter (MVA) pattern at a high level, with the intent of logically segregating the responsibilities of each service or module of a service (NMM-2, NMM-3). The data layer corresponds to the model, the logic layer corresponds to the adapter, and the gateway corresponds to the view. The MVA design follows the principle of least knowledge (Demeter's Law), which states that an actor should not communicate with the children of its children, through abstracting the model away from the view.

In this case, all communication from the gateway must pass through some component in the logic layer before reaching the data layer. It can never directly communicate with the data layer and does not know how data is handled and stored in the data layer, effectively decoupling the model and view components. Should the data layer change, only the logic layer needs to change, minimising the impact of changes, increasing modifiability (NMM-2).

5.2. Service Responsibilities

The table below summarises the responsibilities of each service or component, and the functional requirement categories that it corresponds to.

Service	Responsibility	FR Category
Gateway	Aggregate user-facing APIs for the front-end and proxies requests between external requests and the internal services.	None
Session Service	Manages session and refresh JWT token issuance and maintenance.	**A** - Authentication
User Service	Manages user accounts and its related operations, including authentication.	**A** - User Accounts and Authentication
Matching Service	Handles requests related to the matching queue, including joining, leaving and status checks.	**B** - Matchmaking
Matchmaker	Handles actual matching of users in the matching queue.	**B** - Matchmaking
Question Service	Manages the creation, retrieval, updating and deletion of questions.	**E** - Question Bank
History Service	Manages the historical records of questions attempted by users.	**E** - Question Attempt History
Collaboration Service	Handles the room session and all code workspace collaboration functionality.	**C** - Room Session Management **D** - Synchronisation Relaying
Execution Service	Handles functionality related to code execution in the workspace.	**G** - Code Execution
Frontend	Component that the user interacts with and handles the actual code workspace synchronisation across clients.	**D** - Workspace Synchronisation **F** - Text Chat

Table 10 – Service Responsibilities

5.2.1. Service Boundary Considerations

An architecture design where the User service and History service are each separated into a logic-only service and Create-Read-Update-Delete (CRUD) service was considered. This design would adhere more strongly to the Single Responsibility Principle (SRP), where business logic and data handling are separated into 2 distinct services. However, such an architecture would induce an exceptionally chatty channel between the 2 services, since most operations on the logic service will involve the data handling service, and was deemed as a suboptimal design. Therefore, the final architecture merges the 2 components into a single service, but employs a clear modular separation to mark the boundaries of the logic and data layers.

5.2.2. Matchmaker Isolation and Scaling

Since the matchmaking process should be atomic across the cluster, the matchmaker should not scale. With this consideration, it was separated out from the matching service to allow both components to scale independently. However, the matchmaker still allows for redundant instances, but one of the instances will automatically negotiate for a master position, and the other matchmaker instances will sit idle. Should the master matchmaker instance fail, one of the idling instances will take over as master, giving the component some level of fault tolerance.

5.3. Observability in the Microservices Pattern

The microservices architecture also facilitates observability through the use of logging and metrics. Since each service is only responsible for a small subset of operations, the logs it produces will be closely related with a predictable scope. This eases the tracing of requests and the identification of problems as the logs can detail how a request flows through services and narrow down the area where the problem occurs.

Aggregated logs provided by the deployment environment, either Docker Compose, Kubernetes or Amazon CloudWatch, are used intensively for tracing and debugging during development and deployment. An example of such a trace across multiple services is shown below.

```
session-service_1     | 2022/11/07 18:24:55 Issued token for thomas@gmail.com
user-service_1        | [INFO][11/7/2022, 6:24:55 PM] Login from: thomas@gmail.com
gateway_1             | 2022/11/07 18:24:55 Authed thomas@gmail.com
session-service_1     | 2022/11/07 18:24:59 Issued token for johnny@gmail.com
user-service_1        | [INFO][11/7/2022, 6:24:59 PM] Login from: johnny@gmail.com
gateway_1             | 2022/11/07 18:24:59 Authed johnny@gmail.com
gateway_1             | 2022/11/07 18:25:02 Authed johnny@gmail.com
matching-service_1    | [INFO][11/7/2022, 6:25:02 PM] Joined Queue: johnny@gmail.com
gateway_1             | 2022/11/07 18:25:03 Authed johnny@gmail.com
gateway_1             | 2022/11/07 18:25:03 Authed thomas@gmail.com
matching-service_1    | [INFO][11/7/2022, 6:25:03 PM] Joined Queue: thomas@gmail.com
matchmaker_1          | 2022/11/07 18:25:03 Matched (johnny@gmail.com, thomas@gmail.com): 516c37cc-defb-4d02-a30e-44fdda79d1d4
gateway_1             | 2022/11/07 18:25:04 Authed johnny@gmail.com
gateway_1             | 2022/11/07 18:25:04 Authed johnny@gmail.com
gateway_1             | 2022/11/07 18:25:04 Starting WS Tunnel for johnny@gmail.com
collab-service_1      | [INFO][11/7/2022, 6:25:04 PM] Topic 516c37cc-defb-4d02-a30e-44fdda79d1d4 registered: johnny@gmail.com
collab-service_1      | [INFO][11/7/2022, 6:25:04 PM] Joined Room: 516c37cc-defb-4d02-a30e-44fdda79d1d4 , johnny@gmail.com
gateway_1             | 2022/11/07 18:25:04 Authed thomas@gmail.com
gateway_1             | 2022/11/07 18:25:04 Authed thomas@gmail.com
gateway_1             | 2022/11/07 18:25:04 Starting WS Tunnel for thomas@gmail.com
collab-service_1      | [INFO][11/7/2022, 6:25:04 PM] Topic 516c37cc-defb-4d02-a30e-44fdda79d1d4 registered: thomas@gmail.com
collab-service_1      | [INFO][11/7/2022, 6:25:04 PM] Joined Room: 516c37cc-defb-4d02-a30e-44fdda79d1d4 , thomas@gmail.com
gateway_1             | 2022/11/07 18:25:08 Authed thomas@gmail.com
gateway_1             | 2022/11/07 18:25:08 Authed johnny@gmail.com
```

Figure 11 - Docker Compose Aggregated Log Trace from Room Joining

6. Frontend-Backend Integration

This section details the design decisions for the integration of the frontend with the backend.

6.1. Peer-to-Peer vs Client-Server Model

One key design decision is the adoption of a client-server model over a peer-to-peer model for collaboration networking. The following diagrams illustrate the 2 architectures that were under consideration in terms of network architecture.

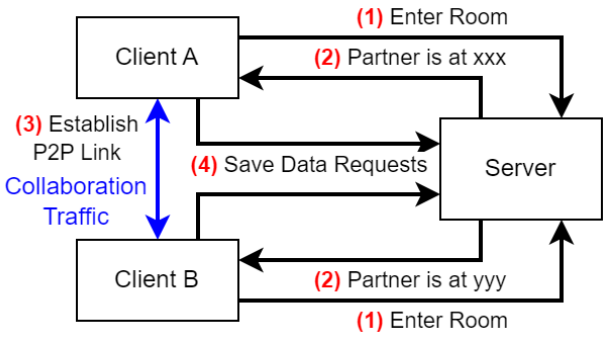


Figure 12 – Peer-to-Peer Model

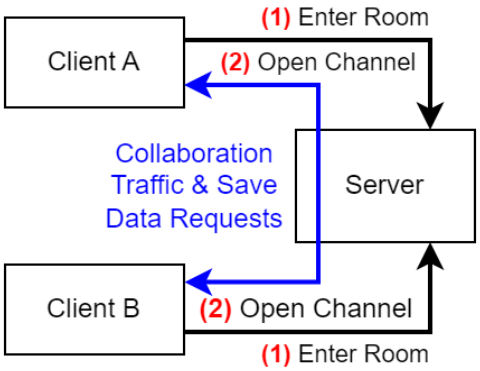


Figure 13 – Client-Server Model

The peer-to-peer model would drastically reduce the capacity required for the collaboration service, only handling requests that involve database access, while the majority of the traffic for collaboration is routed between the clients directly over a peer-to-peer link. However, this architecture has the drawback of introducing more complexity and points of failure into the session establishment flow. NAT traversal is a well-known and difficult-to-solve problem in this networking model, requiring the server to act as a Session Traversal of the User Datagram Protocol (STUN) server for the clients. Moreover, if the NAT traversal process fails for some reason, the backend will still need to act as a Traversal Using Relay NAT (TURN) server to proxy traffic between the 2 clients.

Due to the nature of peer-to-peer communication, the model also lacks observability from the server's standpoint, and data exchanged between the 2 clients cannot be easily intercepted. Any data received from the save data requests therefore cannot be validated, potentially resulting in inconsistencies when saving information.

Considering these drawbacks, the client-server model was chosen at the expense of having to allocate resources for handling the proxying of client streams because it is seen as a more reliable architecture.

6.2. Websocket Stream-Based Communication

Websockets provides a two-way asynchronous communication channel between a user and a server. It enables a full-duplex event-driven messaging channel, where data from the server can be pushed directly to the client. As a result, websockets are ideal when the user is in a session because data will be sent and received in rapid succession. This full-duplex channel provides a more responsive experience compared to a polling-based communication approach, where changes may be sent in chunks with longer response times.

In Peerprep, the websocket connection is initiated by the client to the gateway, which is then proxied onwards to the collaboration service that handles inter-client communications. The gateway performs translation from websocket to gRPC, which will be further described in the API gateway section.

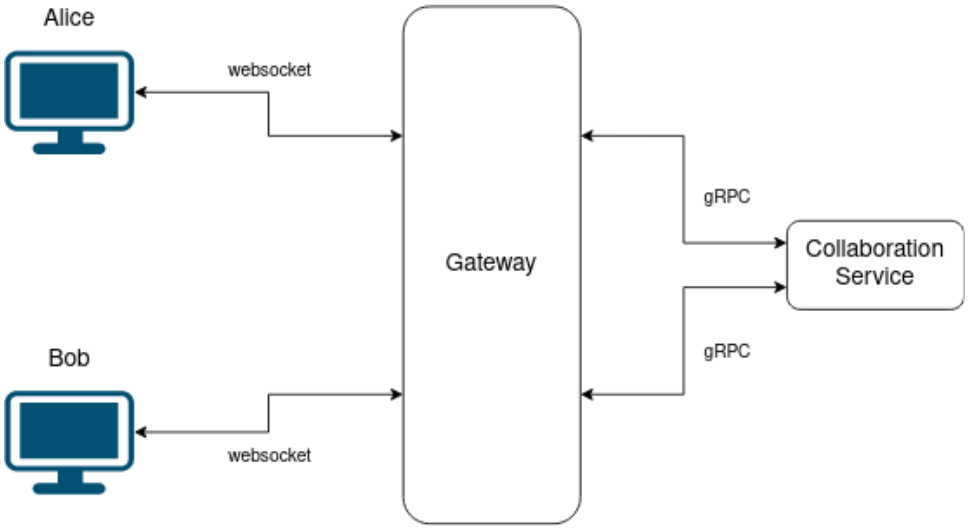


Figure 14 - Websocket Communication Architecture

Since the collaboration service needs to behave differently depending on the data sent from the client, there is a need to inspect the messages sent on the websocket. The websocket message format is designed in such a way that costly full message inspection is not required, with the operation code (opcode) located at the start of the message being sufficient for deciding what the collaboration service needs to do. This approach reduces the computational cost and reduces latency induced from message processing.

6.3. Polling-Based Communication

Polling-Based communication is another way to achieve duplex communication by periodically sending synchronous status check requests. This avoids long blocking requests and allows the client to perform other tasks while waiting for the request to complete, only polling when it is ready to handle the reply. The approach is lightweight as it eliminates the persistent connection of websockets at the expense of longer latencies. As such, it is useful when the data sent from the server to client is latency-insensitive.

An application of polling-based communication is used in the queueing and matching process, where the user may need to wait up to 30 seconds to be matched. Polling is achieved by first sending the long-running join queue request, followed by shorter status check polling requests sent in 1-second intervals. Below is an illustration of the polling communication workflow.

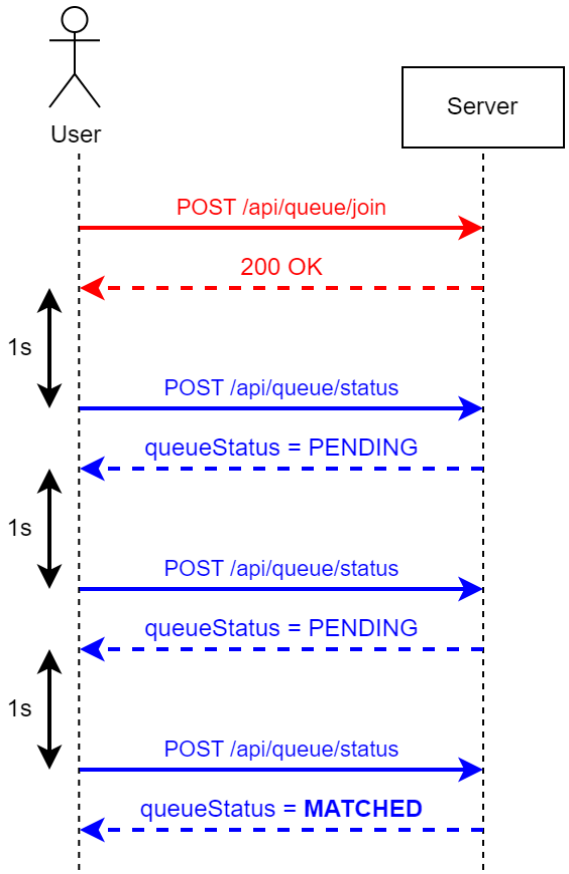


Figure 15 - Queue Status Polling

6.4. Request-Reply-Based Communication

For all other API calls that do not fall under the abovementioned protocols, the classic Request-Reply communication pattern is used, with HTTP as the synchronous transport protocol as it is the standard and most widely adopted protocol for web applications. Generally, clients performing API calls require the response from the backend to continue performing some action, an inherently blocking process that matches the request-reply pattern.

7. Inter-Service Communication

Due to the nature of the microservices architecture, services need to communicate over the network for the exchange of information. This can be achieved through various means, including synchronous and asynchronous methods of data transfer. This section discusses the design considerations for how different types of communication are used throughout the architecture.

The general guiding principle used in selecting the type of communication between services is based on latency. Specifically, if an asynchronous transport is used, it must be able to handle long delays between sending and receiving the message, upwards of a few seconds.

7.1. Service Discovery

The system uses the Domain Name Service (DNS) as the primary form of service discovery. All services contact other services through a well-known name, resolved to a specific IP address using the DNS name resolution provided by the deployment environment, be it Kubernetes or routing through a load-balancer. This is consistent regardless of the networking architecture used in deployment, which will be further discussed in the deployment section. Such a service discovery mechanism obfuscates the management of services and eliminates the need for service registration code in each service, simplifying the codebase.

An example flow of user-service contacting session-service is shown in the diagram below.

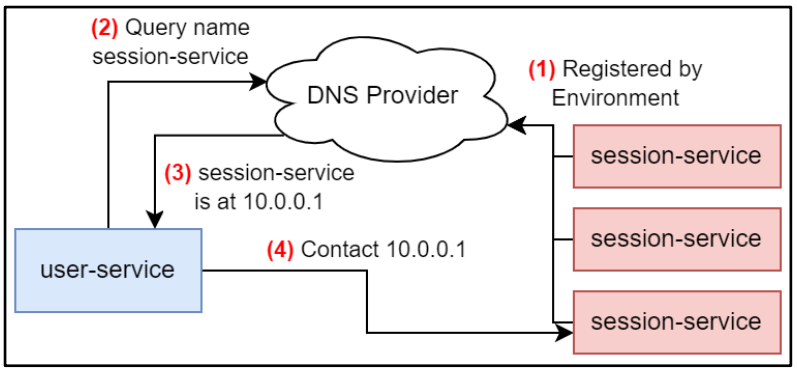


Figure 16 - Service Discovery Flow

7.2. On-Wire Data Format Definition - Protocol Buffers

Protocol Buffers (ProtoBuf) are used as a common declarative language for the structures of data that will be exchanged between microservices. This declaration is language-neutral and implementation agnostic, acting solely as a means of defining common data formats and the associated specifications for serialising the format for on-wire transport of these data units. This definition is shared across all microservice, ensuring consistency through a single source of truth for data formats. This also allows for the design of data messages and objects to be independent of serialization concerns, allowing the focus to be shifted to the business logic aspects of communication.

7.3. Synchronous Communication

For the majority of inter-service communication, synchronous communication was chosen. This is due to the latency requirement for API calls that were previously defined in the NFRs (NPP-8). Given the tight deadline, synchronous communication is preferable as it enables a quicker round-trip to and from downstream services. Synchronous calls also reduce the complexity of the protocol design as it is naturally transaction-oriented, employing a Request-Reply pattern which allows each request to be tightly encapsulated within its context.

However, synchronous communication comes with the drawback of tight coupling across the microservices. This was seen as an acceptable trade-off given the latency implications and the increased complexity of request handling that asynchronous communication would bring. Moveover, at the small scale of deployment, the benefits of asynchronous communication may not even be immediately apparent.

7.3.1. gRPC for Synchronous Communication

The protocol selected for synchronous communication between services is gRPC, which uses the serialisation format defined by ProtoBuf for its data, transported over HTTP/2. gRPC is as a result speedier and less network intensive than vanilla HTTP because the ProtoBuf data being transferred is more tightly compacted, resulting in smaller request and reply payloads. Unlike the older HTTP/1, HTTP/2 also features streaming and multiplexing functionalities, giving gRPC the ability to stream multiple requests and/or replies in a single connection, resulting in better latencies and throughput.

7.3.2. Protocol Selection - gRPC vs HTTP/1

While gRPC is overall a more efficient network protocol owing to its newer transport and dense data representation, it does have its drawbacks. Immediately apparent is the lack of universal support, especially for front-end clients. A full end-to-end gRPC-based network communication stack is therefore not feasible due to the low-level access required by gRPC clients for writing the data in a serialised format, a functionality that has patchy support across major browsers. This also necessitates a translation layer between gRPC-based services and HTTP-based clients, which is an added overhead.

However, the team decided that the interoperability benefits that come with the consistent use of gRPC and ProtoBufs outweighed the disadvantage of requiring a translation layer, which can be easily implemented at the API gateway. While the team moved forward with gRPC, most backend services still employed a dual-stack approach as a safety net, where all user-facing services were available over both gRPC and HTTP/1 with JSON. This allowed falling back to HTTP/1 should the need arise and made it easier for testing since clients for issuing HTTP/1 requests are more widespread than its gRPC equivalents.

7.3.3. ProtoBuf and gRPC Workflow

Using the ProtoBuf and gRPC stack provides a streamlined workflow due to the automatic stub generation capabilities that ProtoBuf provides. ProtoBuf compilers automatically translate the ProtoBuf definitions and gRPC method signatures to code, generating the class structures, method calls and serialisation routines for the declared types and gRPC calls. Since this can be done with a single command, there is no need for handling low-level details when the ProtoBuf declarations change, enabling changes to be incorporated into code rapidly.

7.4. Persistent Messaging Communication

Some inter-service communications are not latency sensitive and can afford delayed receipt. However, receipt of the transmitted data must be guaranteed, even if it is perpetually delayed. For these applications, a persistent messaging system is used to facilitate communications because it reduces the coupling between the services and enables broadcasting to multiple targets using a Publisher-Subscriber (Pub-Sub) messaging pattern. It also allows the target to poll for the message when it is ready, instead of requiring the target to be active when the message is sent.

For instance, persistent messaging is used to propagate mutation events on the data store to other services listening for the event, so that they can themselves synchronise their data stores according to the updates. This specifically involves the different create, read, update and delete (CRUD) services. Additionally, these update messages need to be reliable in order to maintain an accurate and consistent data state across the entire system. Due to these reasons, a persistent Pub-Sub messaging pattern is used.

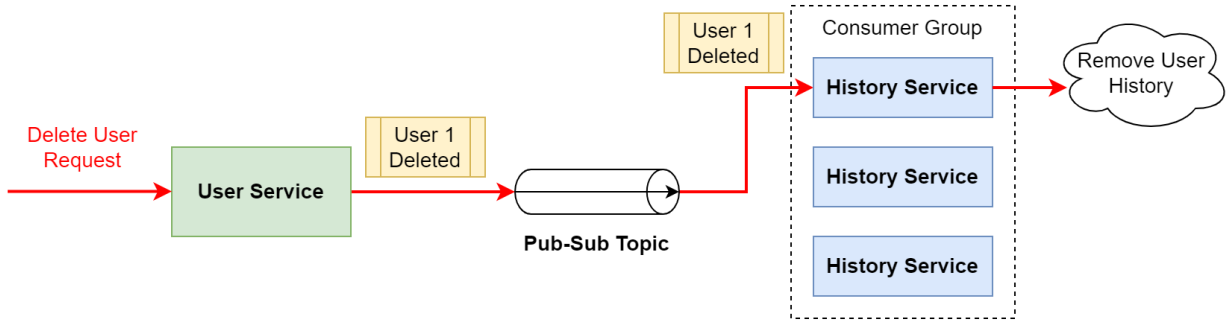


Figure 17 - Persistent Messaging for Delete User Event

7.4.1. Selected Persistent Messaging System - Redis Streams

Redis streams is used as the persistent messaging system because of its high performance and lightweight nature. It supports the Pub-Sub communication pattern, while persisting messages that have not been acknowledged indefinitely, satisfying the requirements for the use case. The support for consumer groups also makes it compatible with the horizontally scalable design of the architecture, where only an instance of a particular service will receive a message.

Compared to other persistent messaging systems like Kafka and RabbitMQ, Redis streams is significantly lighter in both its deployment requirements and protocol, featuring only the minimal set of features required for a Pub-Sub topic. Its simplicity and in-memory nature make it ideal for the intended use case, which is not expected to see high volumes or require complex routing logic. However, Redis does not guarantee durability in persistence, and messages may be lost in the event of Redis failures. This is seen as an acceptable risk as the data being exchanged is not critical (NIT-1). The implications of not receiving a message are simply extra records in the database that do not impact the overall system integrity or consistency.

7.5. Transient Messaging Communication

Transient messaging is used in latency-insensitive and loss-tolerant communication between services. This form of messaging is used in the system only for intra-service communication, where the horizontally scalable nature of the application necessitates data transfer between multiple instances of the same service. Specifically, this is used for the collaboration service, where data between 2 connected clients are exchanged over a point-to-point messaging channel. Since the data transferred is not critical to the system, they can be lost without any major side-effects.

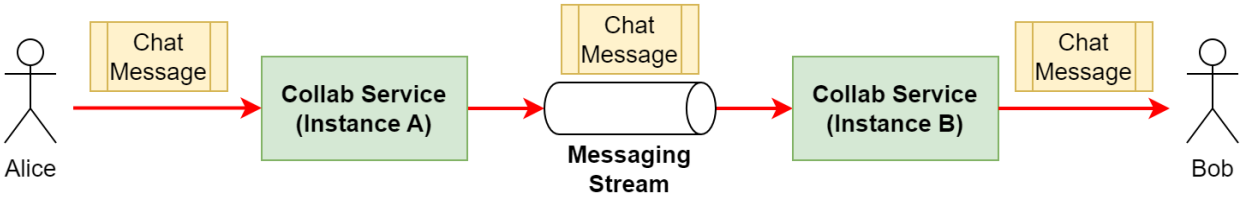


Figure 18 - Collab Service Transient Messaging

7.5.1. Selected Transient Messaging System - Redis Pub/Sub

While low latency is not strictly required in this application of transient messaging, it would be ideal if it can be kept reasonably low since it affects how fast changes propagate between users. Since persistence and durability are not necessary, Redis Pub/Sub is ideal as its in-memory nature makes it highly performant. Moreover, since the messages can be lost, there is no concern in using the push-based relay mechanism employed by Redis Pub/Sub, where messages are lost if the recipient is not ready at the time of sending.

Compared to other transient messaging systems like ActiveMQ and RabbitMQ, Redis cannot scale or replicate itself across a cluster reliably. However, with a message handling capacity of roughly 1 million messages, a single Redis node acting as a Pub/Sub relay is likely more than sufficient for this use case and does not necessitate horizontal scaling.

8. Backend

The backend encapsulates all the microservices that are shown in the architecture. They can be classified into 2 distinct categories, namely operation-oriented microservices and data-oriented microservices. This classification drives the internal design of the microservice, and how sub-components within a service are laid out.

8.1. Operation-Oriented Microservices

The primary concern of Operation-oriented microservices is performing transactions that result in certain side-effects. These transactions are typically singular operations, submitted through an API call. This group of services includes the User Service, Question Service, History Service, Matching Service and Session Service.

8.1.1. API Server Framework Pattern

Except for Session Service which only supports gRPC as it is only used internally, all the other services follow a common architectural pattern, colloquially known as the API server framework, since they handle requests in a similar manner on a dual protocol stack (gRPC and HTTP). This is a custom framework designed for Typescript services and aims to abstract business logic away from the networking and implementation aspects of the microservice. The architecture of the API server framework is illustrated below.

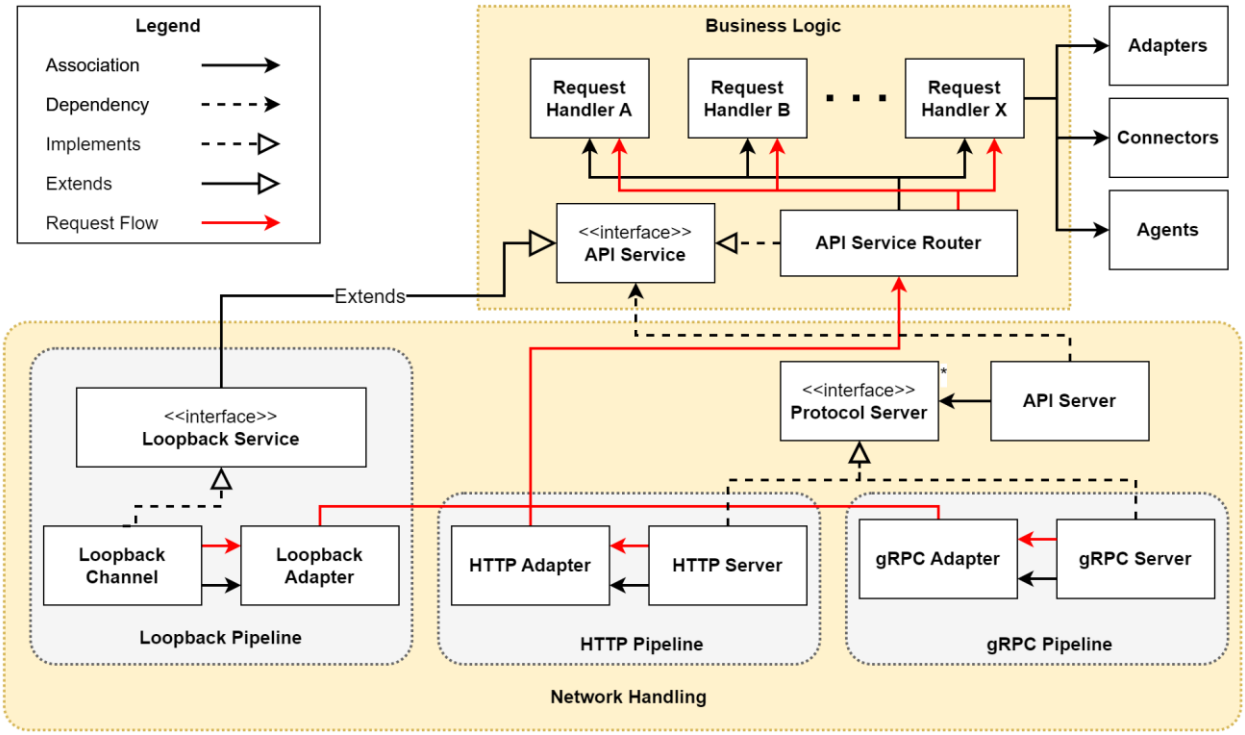


Figure 19 - API Server Framework

It is immediately noticeable that the business logic is encapsulated into a single submodule, while the lower levels of protocol handling and networking are hidden behind the API Server abstraction, decreasing the coupling between the 2 parts. This clear segregation that separates out common low-level network handling logic enables reuse of the framework across all the services. It also adheres to the Single Responsibility Principle (SRP) where network-related changes only affect the networking components, while business logic changes only affect the handler components (NMM-4).

Further abstraction is introduced for each of the networking protocols using the API Service abstraction. It serves as a common interface that each network protocol pipeline needs to adapt requests to and adapt responses from. This enables extensibility, where new protocols can be added to the service using adapters that translate to and from this common interface.

8.1.2. Service-Level CQRS Pattern

The session service follows the Command and Query Responsibility Segregation (CQRS) pattern in its blacklist design due to the unique handling requirements. Adding to a specific blacklist directly writes to the data store for that blacklist but querying the validity of a particular token uses a pipeline that runs logic across multiple blacklists. As such, the logic for writing to the blacklists is distinctly different from the logic that queries from them and can be segregated.

The decision to apply a CQRS pattern was made for extensibility, where more blacklist filters can be added to the query pipeline in the future without major changes to the query process. While writing to a blacklist depends on the type of checks carried out, querying the blacklist simply returns a Boolean, which is easily abstractable into the query pipeline.

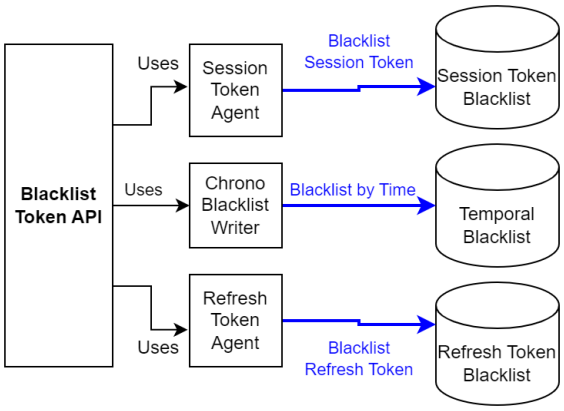


Figure 20 – Insert into Blacklist Command

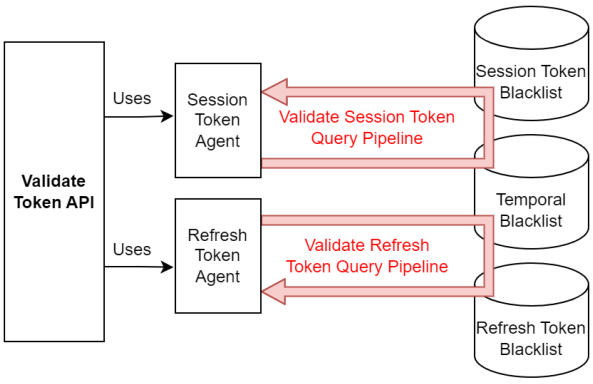


Figure 21 – Validate Token Query

8.2. Data-Oriented Microservices

Data-oriented microservices follow a range of architectures based on the data handling requirements. There is no common architecture across the services, and this section will discuss the variants that are seen in the system.

8.2.1. Pipeline Data Flow

The gateway follows a pipelined flow for streamlined processing of HTTP requests. It closely resembles the pipe-and-filter architecture, but additionally introduces routing logic into the processing flow. A similar pattern can be seen in the matchmaker for the pipeline processing of matching requests. For both services that employ this architecture, the design is driven primarily by how data flows through the service and the logic for handling it.

The activity diagram below shows how the gateway handles incoming HTTP requests.

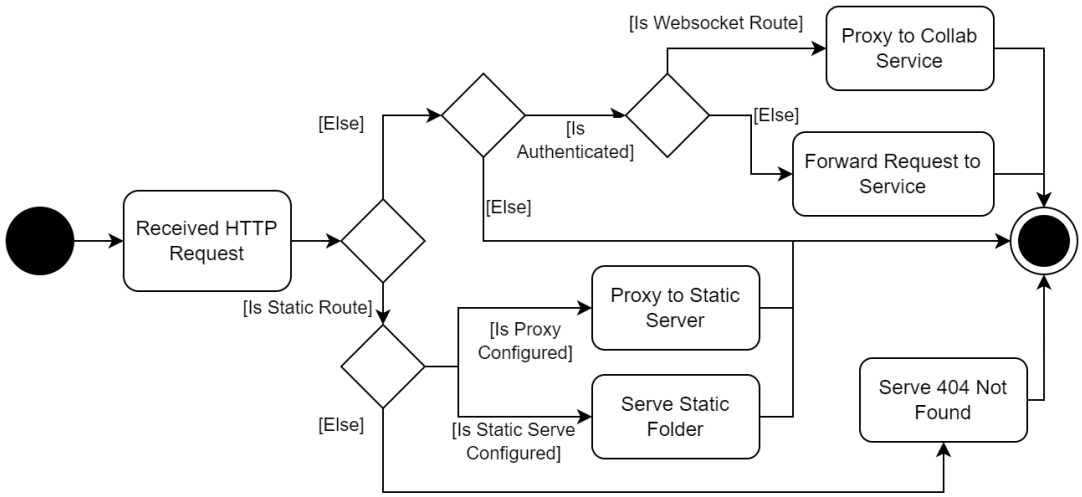


Figure 22 - Gateway Request Processing Flow

This directly translates into the architecture design used in gateway, with decisions being abstracted into submodules or filters in the processing flow.

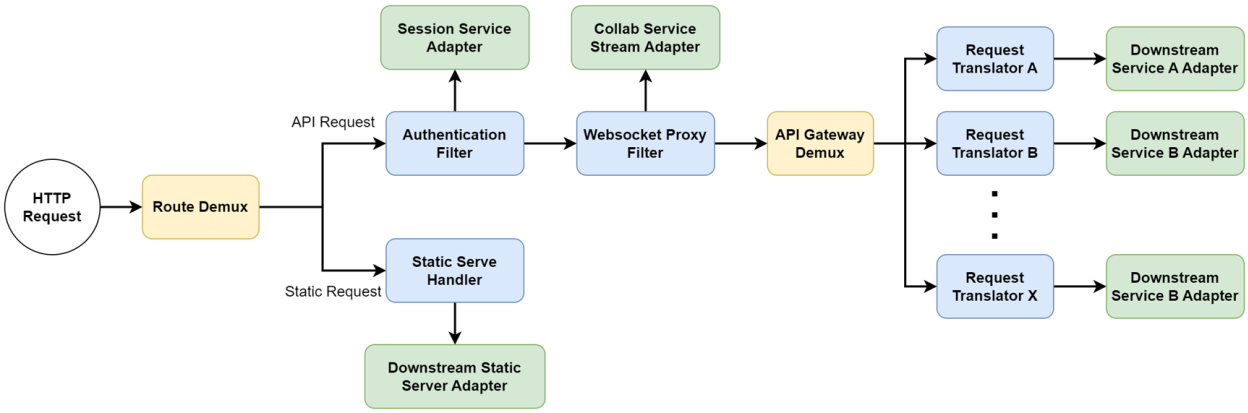


Figure 23 - Gateway Architecture

8.2.2. Mediator-Based Data Flow

When handling communication for collaboration between 2 clients, the architecture adopted is similar to the Mediator Pattern, but at a higher architectural level. The collaboration service acts as the mediator for all communications between the clients and downstream services. This architecture differs from the pipeline data flow in that it is handling multiple streams of data, acting as a broker between them.

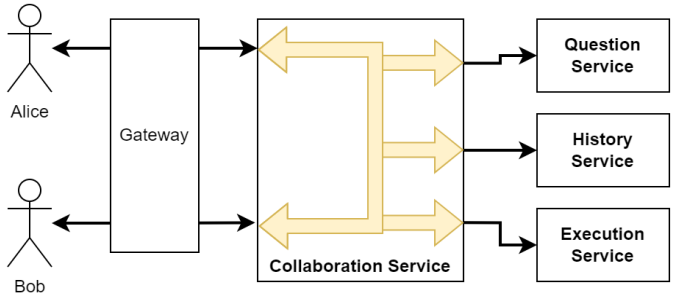


Figure 15 - Collaboration Service High-Level Block Diagram

The mediator pattern is used to centralise control over a room session to a single point, orchestrating how clients and downstream services communicate and cooperate with each other to achieve the intended functionality. With all participants communicating only with a single entity, collaboration service simplifies the protocol design between participants, and adds a layer of abstraction between the clients and downstream services. Clients will never interact directly with internal services such as the question service, decoupling them entirely.

The architecture diagram below shows the details of how the collaboration service brokers varying incoming messages between clients and services in order to serve the different functionalities a collaboration room provides to the clients. Note that the bridge in the diagram is **not** the bridge pattern, and it is actually the core mediator component.

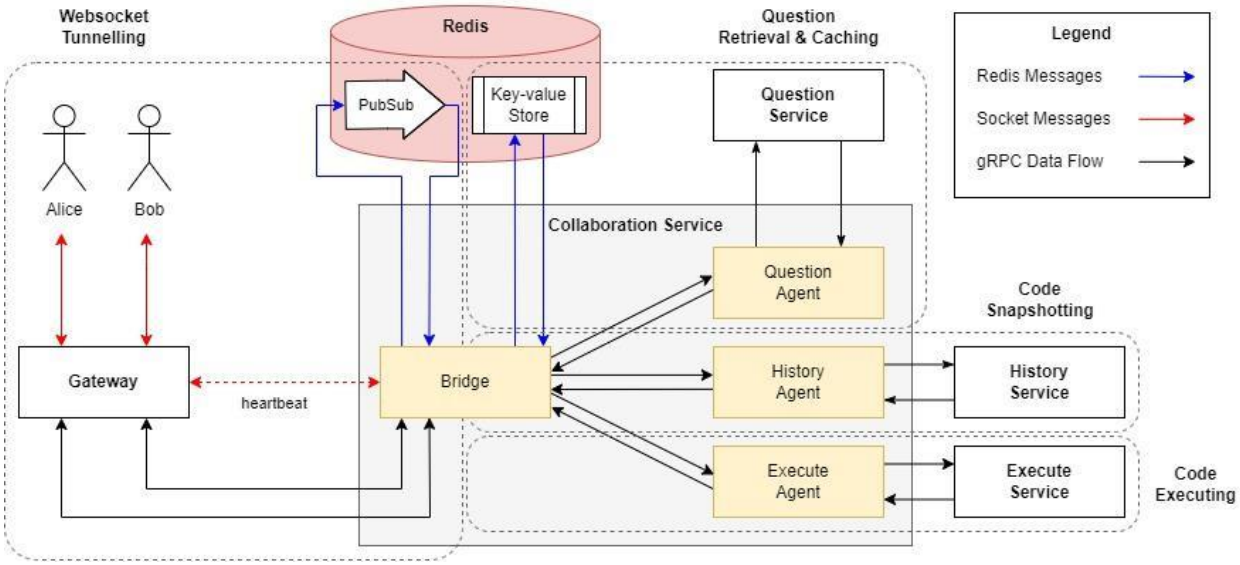


Figure 24 - Collaboration Service Detailed Architecture

8.3. Backend Tech Stack

This section details the technologies used in the backend, as well as the decisions that led to them being selected.

8.3.1. Persistent Data Storage - Postgres

One of the first design decisions that the team made was that the persistent data store should be some form of a relational database rather than a noSQL-based data store. This was founded on the consideration that relational databases are capable of higher-complexity queries should the need arise, including efficient data aggregation, grouping and association queries. These operations would be more difficult to implement, although not impossible, in a noSQL database. Aggregation and filtering at the database reduce the amount of traffic between the database and services, making the database connector less chatty.

Within relational databases, Postgres was selected due to its widespread support and ability to handle extensions to conventional SQL, including useful constructs such as the RETURNING clause. This enables more complex operations to be performed in a single query, greatly reducing the number of costly high-latency calls to the database.

The downside to relational databases is the increased computation required at the database, shifting compute demand from microservices to the database. However, considering the fact that query operations occur much more frequently than write operations, the master-slave replication approach to database scaling can be employed should the need arise.

8.3.2. Transient Cache - Redis

Redis was selected as the transient caching technology because of its nature as an in-memory database, as opposed to a disk-based cache like ETCD. This allows high-performance writes as well as reads, which is important since cache access will be frequent given the dependence on them for the global system state. Moreover, Redis also provides data structures which are useful in structuring the stored state in a streamlined manner, enabling queries to be performed at Redis and reduces the chattiness between Redis and the client services.

The lack of horizontal scalability for Redis was considered during the selection process. However, due to the small scope of each microservice, cached data can be easily sharded across multiple Redis servers or zones. This form of scaling allows for multiple low-performance Redis servers to be used in a way comparable to horizontal scaling, but with clearly segregated data responsibilities.

8.3.3. Programming Languages

Typescript and Golang were selected as the 2 programming languages used in the backend. Both languages notably offer compile-time validation, which is an important requirement agreed upon within the team to streamline development. Typescript offers a platform agnostic and mature language for development, allowing services to be created quickly. However, as it ultimately compiles into Javascript and runs on the NodeJS engine, it is not particularly performant due to NodeJS's single-threaded event-based architecture and just-in-time compilation. The NodeJS engine is also extremely bloated and has relatively demanding runtime requirements. While this lack of performance is acceptable for services that mostly wait on blocking operations, it is not performant enough for services that handle heavy loads such as the Gateway and Session Service.

As such, Golang was also used as an alternative language. Golang is not platform agnostic and requires compilation specific to the runtime it will be targeted to. Moreover, Golang is a procedural language rather than object-oriented, making it difficult to be expressive in object relations and business logic processing. However, Golang makes up for this with its extremely powerful concurrency model based on Goroutines. It allows for efficient concurrent execution with minimal resource usage, and its nature as a compiled language means that it is highly performant. This makes it perfect for the Gateway and Session service, which require high performance and a high level of concurrency for handling simultaneous parallel streams of requests. Its lightweight nature also simplifies the horizontal scaling of services.

8.4. API Gateway

An API gateway is used as a Facade to hide the complexities of the microservice backend from the requesting user. It aggregates user-facing endpoints into a single location for the frontend to query and routes incoming requests to a specific subset of downstream services. This hiding allows for the backend architecture and services to change in an opaque manner, where the client would not be aware of changes that do not affect the requests and responses that it uses, decreasing coupling between the frontend and backend. It also serves as a layer of security, where the user can only access the endpoints that are exposed.

8.4.1. Gateway Capabilities

The gateway provides 2 key services on top of acting as a proxy, namely authentication and protocol translation.

Internally within the backend, all services (except for execution service) use the gRPC protocol for interoperability and consistency. However, as described in the inter-service communication section, gRPC cannot be used on the client. Therefore, as the first touch point for clients, the Gateway needs to translate the HTTP requests received from clients into gRPC requests for the microservices and vice-versa for the response. This functionality is provided transparently and allows the gateway to control the gRPC endpoints that it is exposing to the client. The same translation is also done for Websockets to gRPC since gRPC streaming mode allows for stream-based communication in a way similar to Websockets. Websocket is used between the frontend and backend as HTTP/1 does not support stream-based communication.

Being the single choke point for incoming requests, the gateway is also the perfect location for performing request authentication. This is done by contacting the session service, checking the requests for a valid token, and blocking unauthenticated requests from continuing downstream. The internal microservices are therefore guarded with the guarantee that all forwarded requests, except for login, register and reset password, are from authenticated users whose identities will be tagged as metadata in the forwarded gRPC requests.

8.4.2. Gateway Design Considerations

Due to the gateway's nature as the single entry point into the system, it must be both performant and available. The performance aspect comes from using Golang, a performant and highly concurrent language as well as in its streamlined architectural design. It is also augmented by its horizontal scaling capability, which allows more instances to be spawned when the load is high. Horizontal scalability also enables high-availability deployments, ensuring that the gateway is resilient to failures.

8.5. Data Schema

The database schema is generated from models defined in Typescript using an Object-Relational Mapping (ORM) layer, specifically TypeORM. The ORM layer provides mappings from the relational database tables and columns to Typescript objects and vice-versa, abstracting the database adapter away from the code that handles business logic.

The schema-per-service pattern is adopted in the architecture, where each microservice's data is private to itself, and does not depend on the schema of other services. All operations that the service performs only involve its own data table(s), effectively isolating the modifications to the database that a service can make. This enables a database-per-service approach, giving each service the flexibility to use different types of data storage. However, since the preference is for relational databases, all the services will employ Postgres as the database server.

The schemas used for data-layer services are shown in the ER diagram below, with a detailed version included in Appendix C.

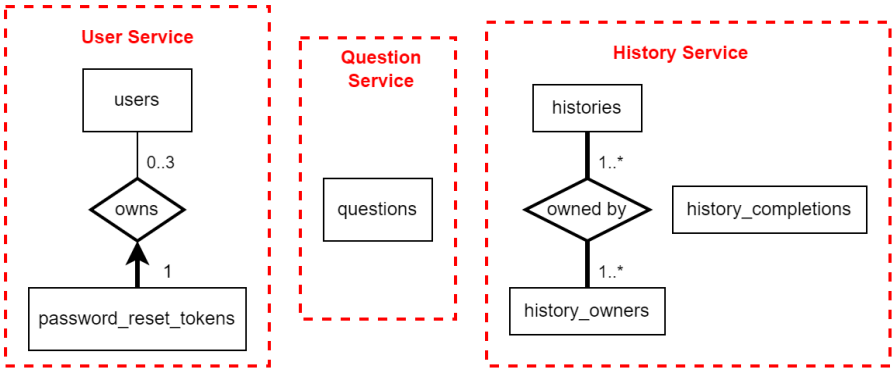


Figure 25 - Simplified ER diagram

While multiple databases can be used from an architectural standpoint, the actual deployment uses a single shared database for all services, but with a logical separation of data using distinct groups of schemas for each service. Nevertheless, the architecture allows the database to be scaled up when required by introducing multiple database servers, effectively sharding the schemas for different services across them. These database servers handle the load independently and allow for better scalability in the data layer.

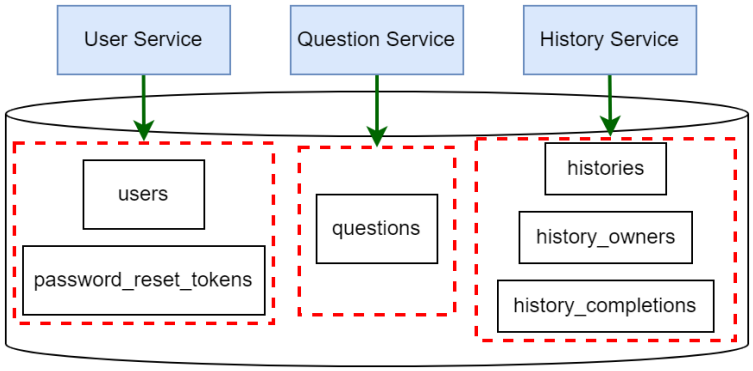


Figure 26 - Database Deployment

9. Frontend

The frontend is designed as a Single Page Application (SPA) and dynamically renders the page based on the user's actions and data received from the backend.

9.1. Frontend Tech Stack

As the frontend's architecture is specific to the framework and tech stack used, it is helpful to first discuss the technologies selected for the frontend and the decisions that led to their use.

9.1.1 ReactJS

ReactJS is a frontend framework for building user interfaces (UIs) centred around the paradigm of reusable units called components. In React, a complete UI is achieved by recursively composing multiple smaller isolated components into a more complex component, except for the lowest layer which contains only trivial components. React also features a virtual Document Object Model (vDOM), which handles state-dependent re-rendering. Business logic is encapsulated in reusable and extensible React hooks, which serve to mutate the state and allow re-rendering without requiring changes to the component itself.

React was selected as it is the single most popular frontend framework deployed in production, with a mature codebase and extensive community support. The vast amount of open-sourced React components available makes it easy to create complex applications by simply reusing components that have already been created by others. It also encourages a modular approach to developing the frontend, which corresponds to the modifiability quality attribute.

9.1.2 Redux

Redux is a centralised state container that simplifies state handling amongst the numerous components that make up a React UI. This is achieved by splitting the state store into smaller slices based on the specific functional domains of the application. Changes to the state are sent to the Redux dispatcher, which dispatches an action to the central store with the necessary information. The store in turn updates the state in the slice.

Redux is used for streamlining state updates, relying on the characteristic that dispatched actions immutably change the state of the store. This eliminates accidental mutations that can otherwise lead to unintended behaviours in the frontend. Furthermore, Redux reduces coupling between components as data is referenced directly from the central Redux store instead of depending on intermediate components for propagating the state inwards. This eliminates unnecessary passing of properties (known as prop drilling) into components, making the components more reusable.

9.1.3 Yjs

Yjs is an open-source conflict-free replicated data types (CRDT) implementation that handles synchronisation of shared data across distributed systems. It automatically generates network protocol agnostic delta update messages for changes to the shared data, allowing it to handle large shared documents, in this case many lines of code, with minimal network traffic.

The use of Yjs eliminates the need for the backend to be the single source of truth for the code workspace state shared between 2 clients. Instead, the clients themselves track and synchronise changes, shifting the responsibility for state management and deconfliction from the backend to the clients.

9.2. Frontend Architecture

The frontend is hierarchically structured, made up of multiple layers. Subsequent sections will detail the frontend architecture and how each component interfaces with one another. The frontend architecture is illustrated below.

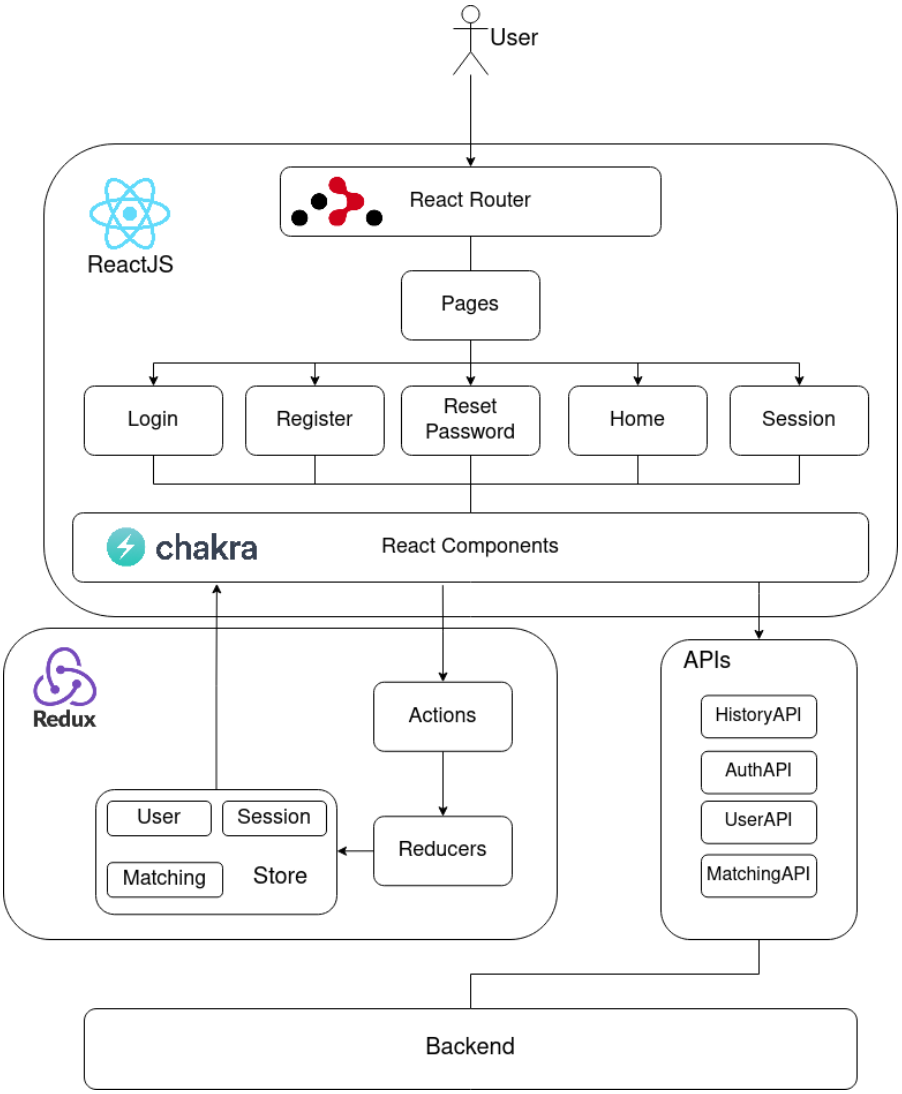


Figure 27 - Frontend Component Architecture

9.2.1. Frontend Render Hierarchy

When a user loads the frontend, it first passes through the React Router, which acts as a demultiplexer and determines the page to render. These pages then contain React components, which are rendered based on the state information provided by the Redux store.

9.2.2. Model-View-ViewModel Architecture

The frontend adopts a Model-View-ViewModel (MVVM) architecture, which separates the user interface (view) from the business logic and data (model). A view-model is introduced as a link between the 2 components to handle the presentation of data in the model to the view. The view-model provides bindings for data to the view and performs logic requested from the view. Updates in the data are then pushed by the view-model down to the views, resulting in a unidirectional flow of data which makes the view components predictable.

This architecture is clear in the HistoryTable component which holds the logic and data for the attempts that the user has submitted. At the HistoryTable, it fetches the data from the Redux store or the History API and exposes pagination details for the child components to consume.

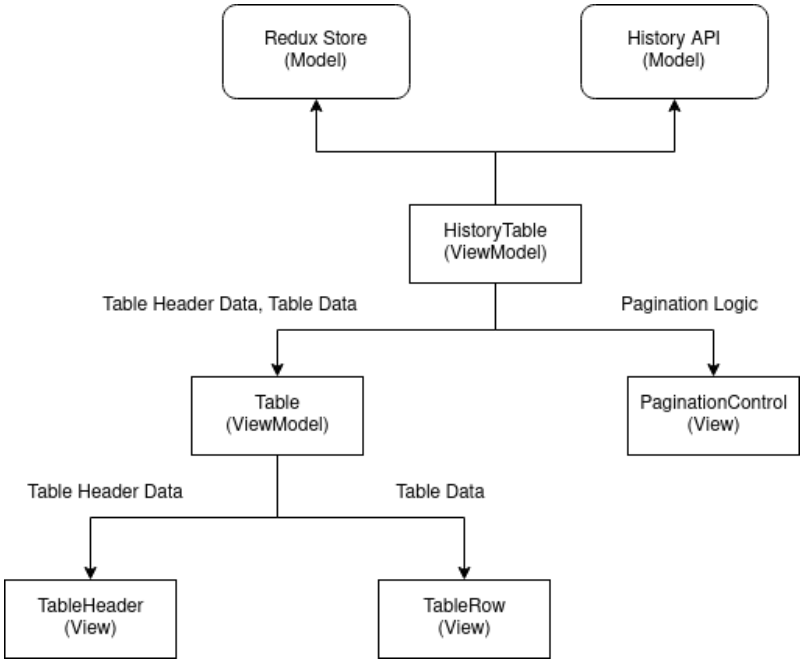


Figure 28 - MVVM Frontend Architecture

Since the model is isolated from the view, coupling is decreased, and the separation of concerns is increased. This decreased coupling allows the child view components to be reusable, as any similar view-model component can simply pass the logic down to the client for consumption without any further modification.

9.2.3. Redux Singleton Pattern

The Singleton pattern ensures that there is only a single instance of a particular class, and all references to that class's objects are to the same instance. This pattern provides a single source of truth which is accessible to all the objects within the application. As a result, data consistency can be guaranteed.

Redux is an implementation of the Singleton pattern, where there is only 1 global Redux store. Although Redux allows components to be loosely coupled by reducing prop drilling, it introduces additional complexity due to the strong coupling between the singleton and the classes accessing it. Despite this, the benefits of having a global single source of state in the complex mesh of components outweighs the introduction of this form of coupling.

However, Singletons are prone to side effects since any object can access it and change its state. Hence, various defensive measures must be taken to prevent such side effects from happening in Redux. The flux pattern is used to ensure a unidirectional data flow, and changes made to the singleton are predictable since it must first route through the dispatcher.

9.2.4. Redux Flux Pattern

Redux implements the Flux pattern, which is meant to streamline data flow in the frontend. The pattern consists of 3 key components, the dispatcher, store(s) and view. The dispatcher serves as the sole entry point for receiving actions and propagates the updates requested by the actions to the appropriate stores, which in turn apply business logic to update their contents. The updated contents are then propagated to the view, which finally updates the visible components.

In PeerPrep, there are 3 stores, called slices in Redux, used - User, Matching and Session. The User slice handles user authentication and provides useful information to the application. The Matching slice handles the details relating to queueing and matching such as the difficulties selected and if the user is in the queue. The Session slice handles details relating to the session such as the selected language and chats messages.

Within each slice, reducers which encapsulate the business logic to transform actions into state updates are implemented. These reducers are run when actions are received from the Redux dispatcher. Components then bind to the state for each slice and re-render their contents according to the updated states.

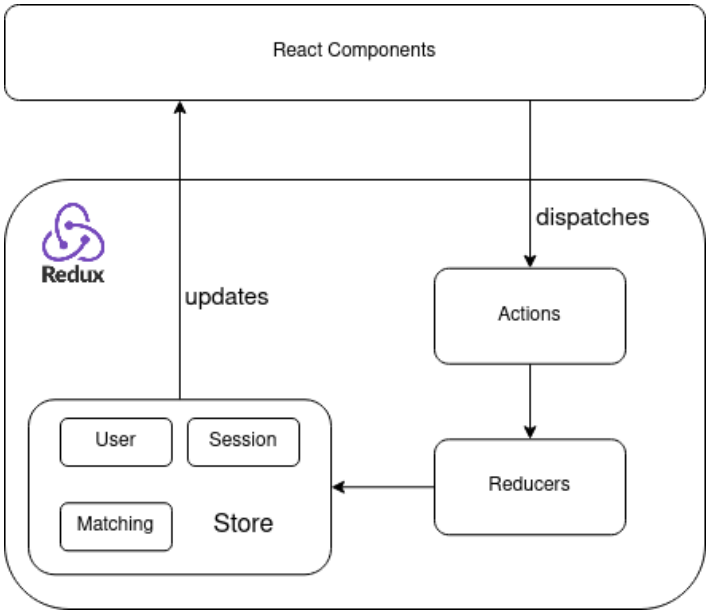


Figure 29 - Redux Flux Architecture

It can be seen that in the entire update process, data flow is unidirectional, which greatly simplifies state updates. Since state management is offloaded to Redux, the pattern also avoids heavy top-level components for state management, increasing render performance of the components.

A concrete application of the Flux pattern can be observed on the Session page. Since the page involves many components to support the multitude of features, it is particularly complex, and data flow to each component within the page can become messy, leading to unexpected behaviours when data updates. Flux's unidirectional data flow simplifies the updates, and components only listen to the state in the Redux store.

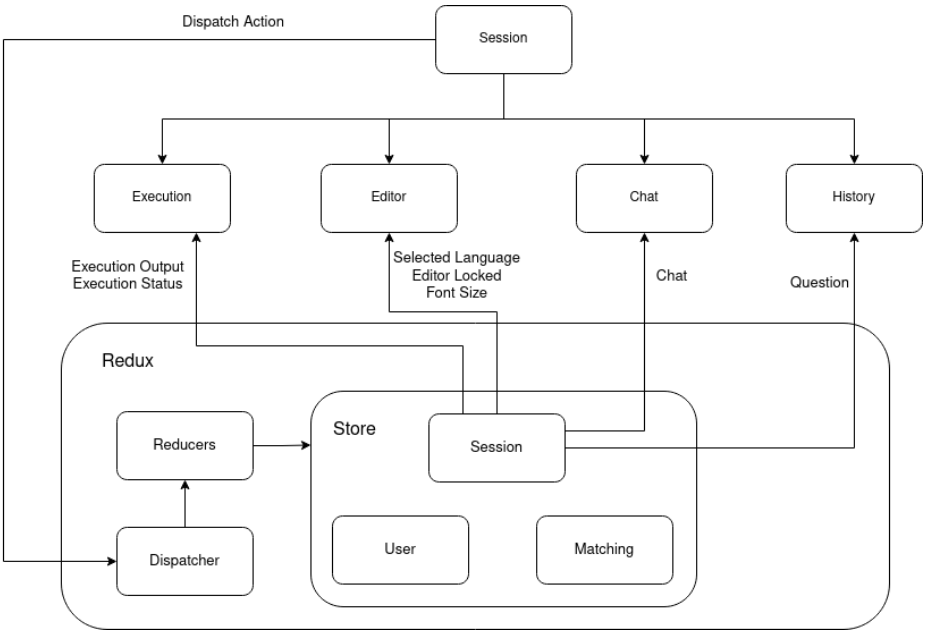


Figure 30 - Flux Pattern in Redux

9.3. CodeMirror and Yjs Integration

CodeMirror is a component editor for the web. It is able to provide a similar experience to desktop editors by providing features such as syntax highlighting and auto-indentation for various programming languages. Furthermore, Yjs supports a binding for CodeMirror to allow collaboration over multiple clients. Through this integration, whenever a user makes a change in CodeMirror, Yjs broadcasts a message to other clients. Additionally, Yjs adopts an observer pattern which determines if it should update the editor or broadcast its changes to other clients to ensure that the other listening clients are properly synchronised.

As the websocket provider for Yjs follows the observer pattern, it can be extended by adding more observable events without breaking the existing implementation. The goal of extending the observable events is to have all communication related to the room session over a single Websocket.

To achieve the extension, the websocket provider package was forked and new observable events specific to Peerprep were added to the provider. This includes events such as language change and code snapshotting. These changes allow the frontend to send interactive session-related messages using the stream-based channel offered by the websocket that was already created by Yjs. This approach improves responsiveness as compared to the polling-based communication approach that would otherwise be required.

10. Process Flow of Key Operations

This section details some of the processes for key operations in the system.

10.1. Authentication and Token Issuance

The authentication system employs a session-with-refresh token system, modelled after the OAuth 2.0 specification. This involves the server issuing 2 JSON WebTokens (JWTs), a short-lived session token and a long-lived refresh token through cookies. The session token is visible to Javascript on the browser, while the refresh token is stored in the browser's secure enclave and is inaccessible to any scripts running on websites, with the browser transparently attaching the cookie to all requests towards the backend.

From a security standpoint, this scheme prevents Javascript from being the attack vector for acquiring the long-lived refresh token, while limiting the amount of time an adversary has to use the short-lived session token. In the overall threat model, transport layer security (TLS / SSL) mitigates Man-in-the-Middle attacks and ensures that the refresh token is not visible to adversaries during transport.

Verification of the authenticity of a token relies on the hash embedded in the JWT. A valid hash indicates that the token is authentic and was previously issued by the server. The JWT also embeds timestamps for token expiry, token issuance and user-specific information, which can be directly used if the token is verified to be authentic and not expired.

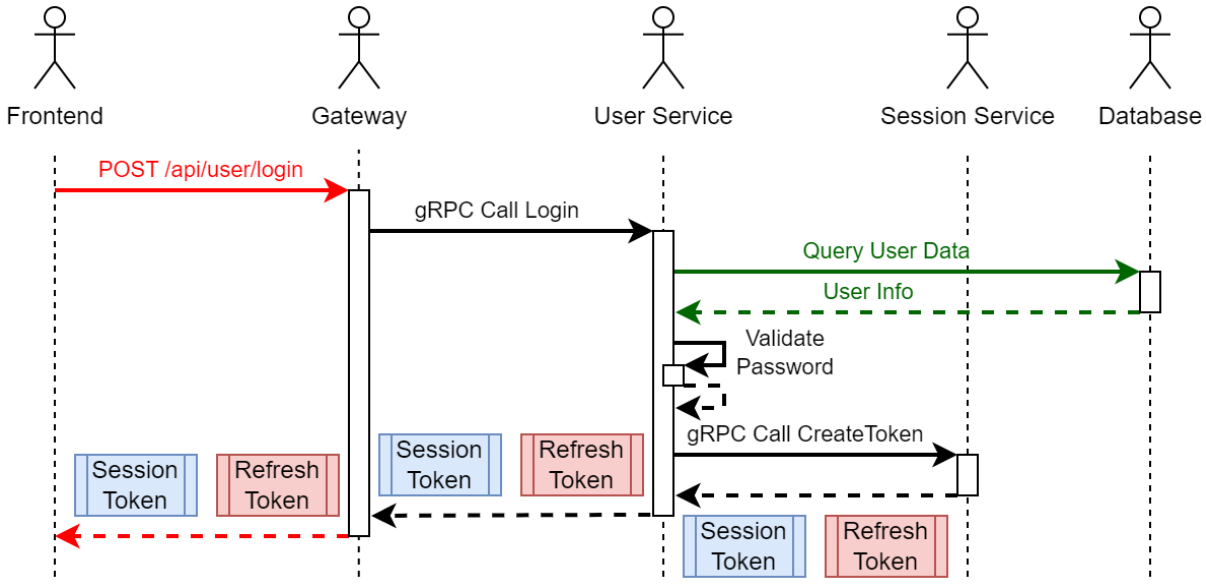


Figure 31 - Login Flow

Internally, the Gateway contacts the session service for token validation, and attaches a metadata tag should the authentication succeed. This flow is shown in the sequence diagram below.

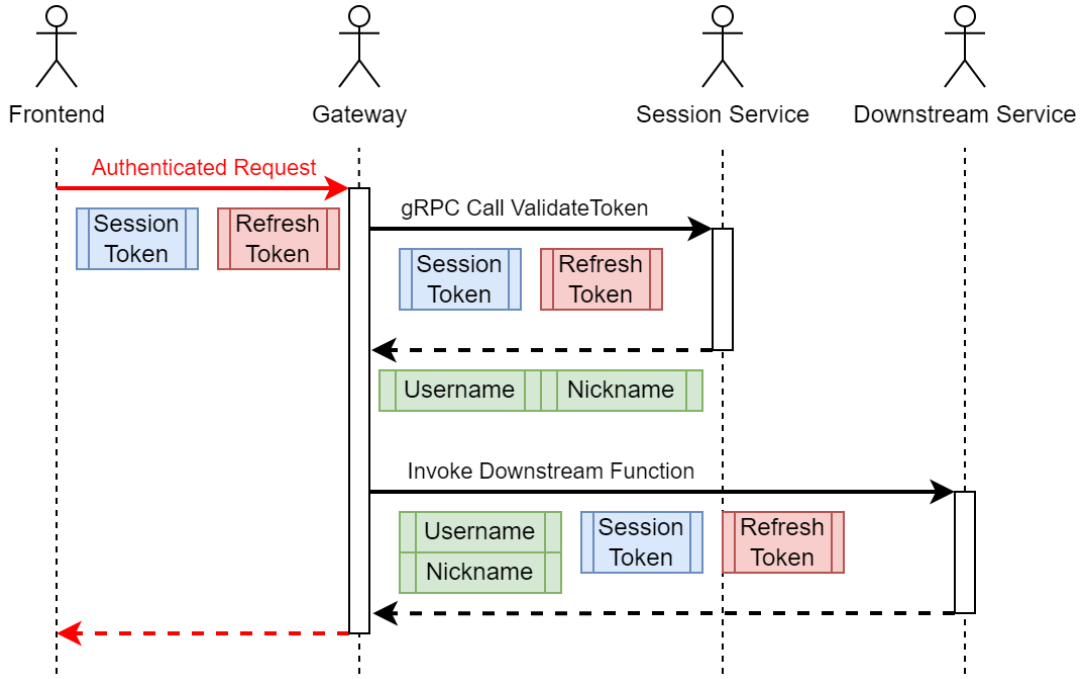


Figure 32 - Gateway Token Authentication Flow

During Logout, the User service contacts the session service to blacklist the token, ensuring that it cannot be used in the future. Furthermore, it issues a clear cookie command to the client.

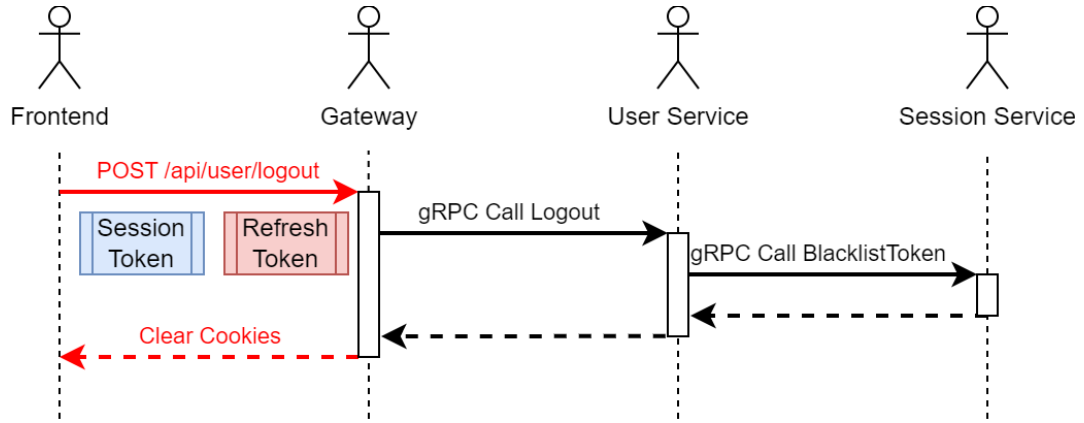


Figure 33 - Logout Flow

10.2. Token Maintenance and Refresh

When the session token expires, the refresh token is validated by the backend and a new session token is issued transparently. The entire token issuance and refresh process does not require any action on the frontend and is solely backend-controlled. If the refresh token is no longer valid, the user is forced to log in again.

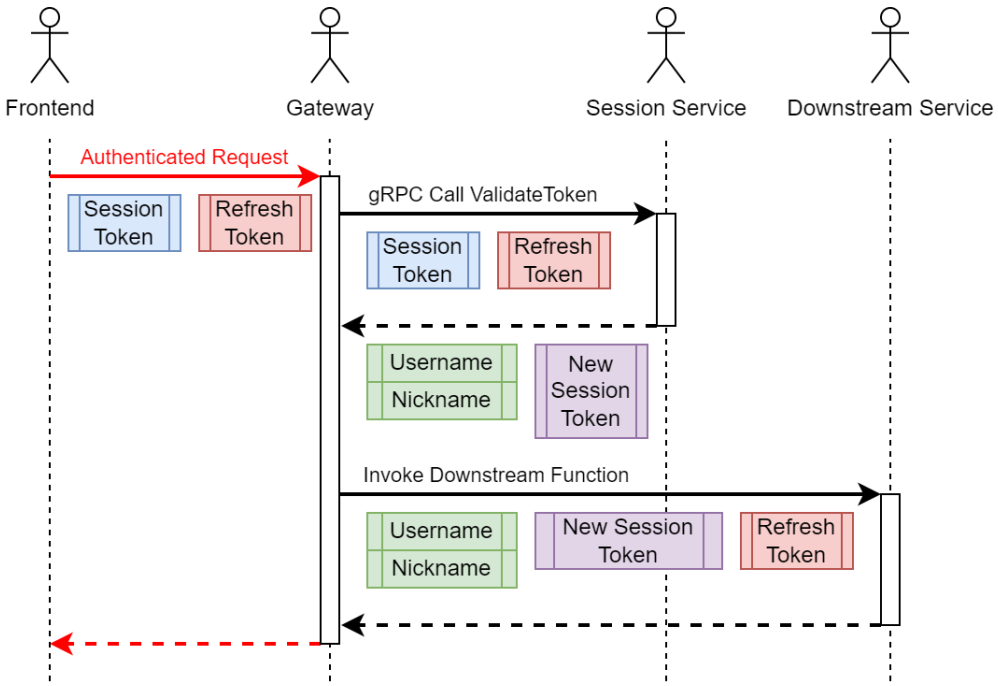


Figure 34 - Token Refresh Flow

The following diagram shows the flow where the refresh token is expired. It can be seen that the request is blocked at the gateway and not forwarded downstream.

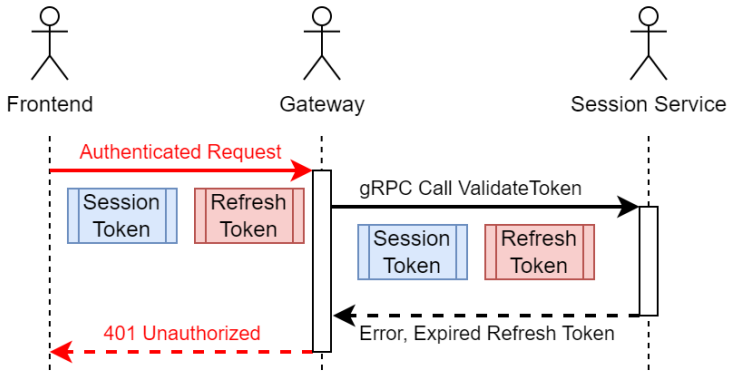


Figure 35 - Refresh Token Expired

10.3. Joining a Session

The process of joining a session is split into two key phases - Matching and Room Creation. Subsequent sections will detail the operations and services involved in each phase and how the information is handed off to other services for further processing.

10.3.1 Matching

Users are matched on a first-come-first-serve basis. The first 2 users with at least 1 common difficulty level will be matched together. If 2 users are matched with multiple difficulties, the heuristic used to resolve is selecting the highest common difficulty. Furthermore, if the user is not matched within the stipulated period, in this case, 30 seconds, the user will be removed from the queue, which is achieved using the Time-to-Live (TTL) mechanism built into Redis and a similar TTL mechanism implemented in the matchmaker.

A match between 2 users is identified using a unique room ID, which is sent to the matching service. A JWT token is issued by the matching service based on this room ID and is used as a mechanism to handover users to the collaboration service, allowing information to be transferred without a direct communication between the 2 services. The integrity of the information is maintained by the hash embedded in the JWT. While this introduces data coupling, it totally eliminates communication between the services, and is seen as a looser coupling between the services.

The diagram below shows the client-backend interaction, but omits the Gateway since it only acts as a relay in every operation.

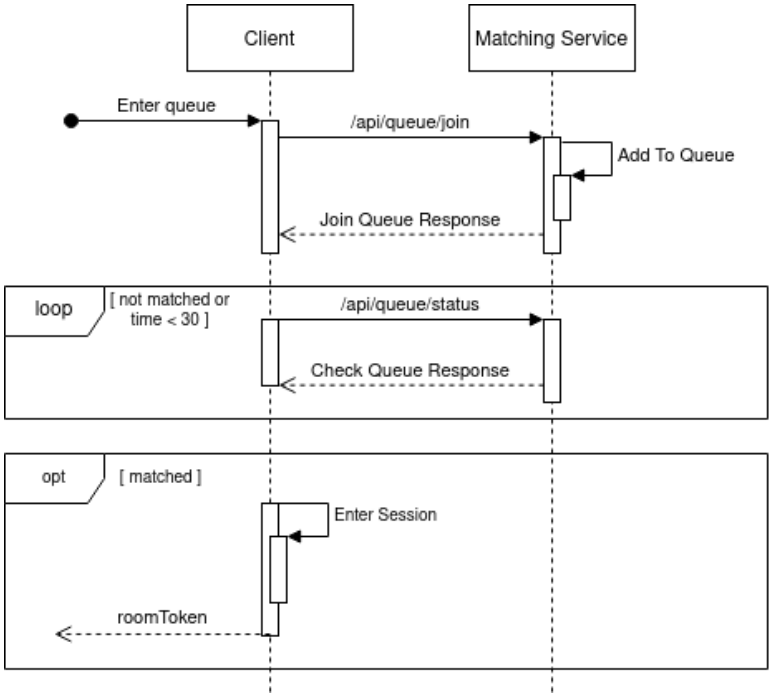


Figure 36 - Matching Sequence Diagram

The sequence diagram below shows the full matching sequence between 2 users.

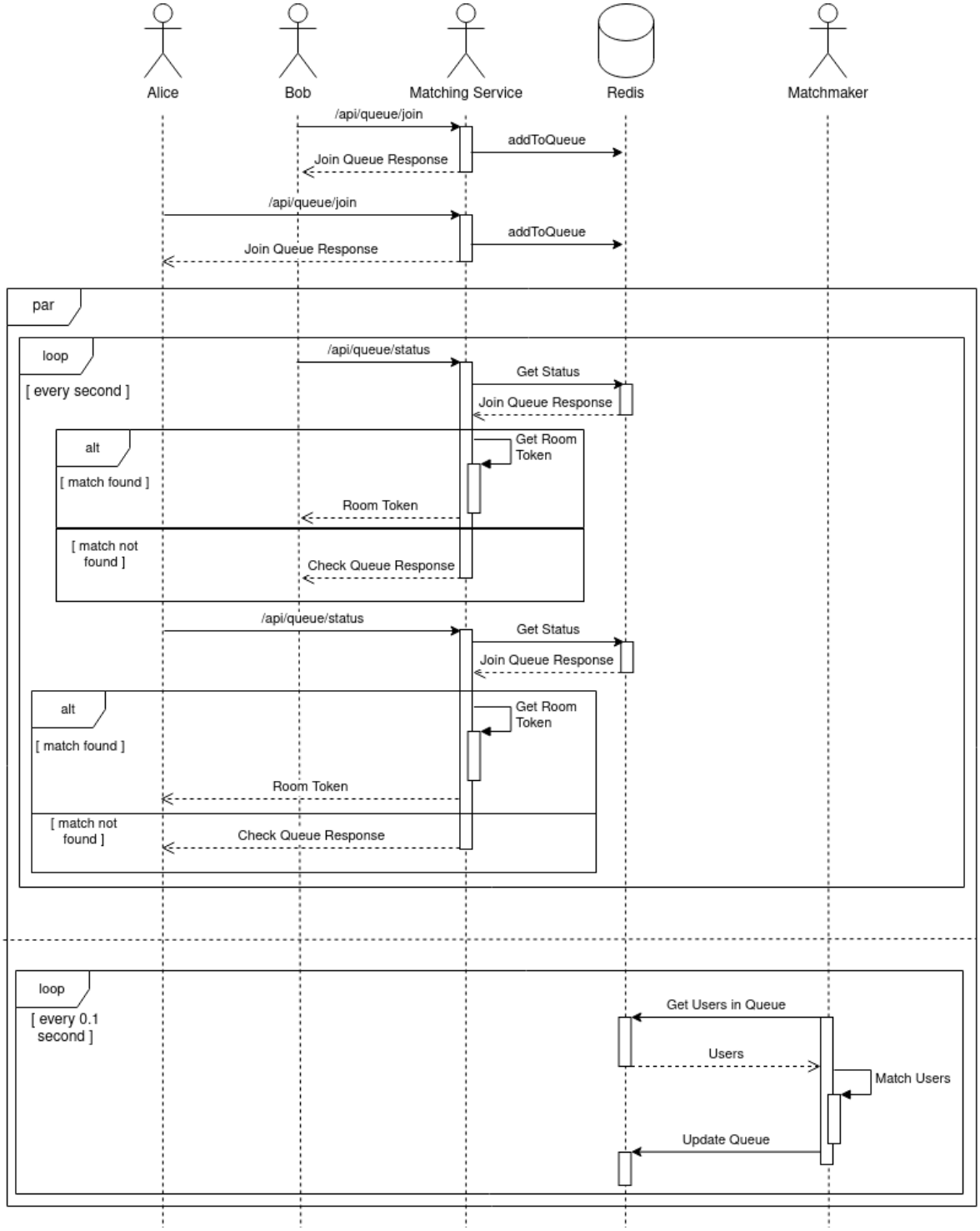


Figure 37 - Sequence Diagram for Matching Process

10.3.2 Room Creation

After successfully matching with another user, users connect to the collaboration service using the room JWT issued earlier. The token contains the room ID and the difficulty matched, which is used by the collaboration service to bootstrap the room session. The overall flow of the room bootstrapping is as follows.

1. User connects to the collaboration service via Websocket and presents room JWT.
2. Collaboration service verifies the room JWT, and disconnects the user if it is invalid.
3. Collaboration service registers a new topic on Redis Pub/Sub, which is used as a point-to-point messaging channel identified by the room ID, and subscribes to it.
4. Upon subscription, the collaboration service will broadcast on Pub/Sub that this user has joined the room, and the broadcasted message will be received by the other user if they have already joined.
5. Collaboration service then retrieves a random question from the question service based on the difficulty and caches the question in Redis, using room ID as the key. This question is shared across all users in the room.
6. Collaboration service contacts history service for the question completion status, and forwards it to the user.

This process is illustrated in the sequence diagram below.

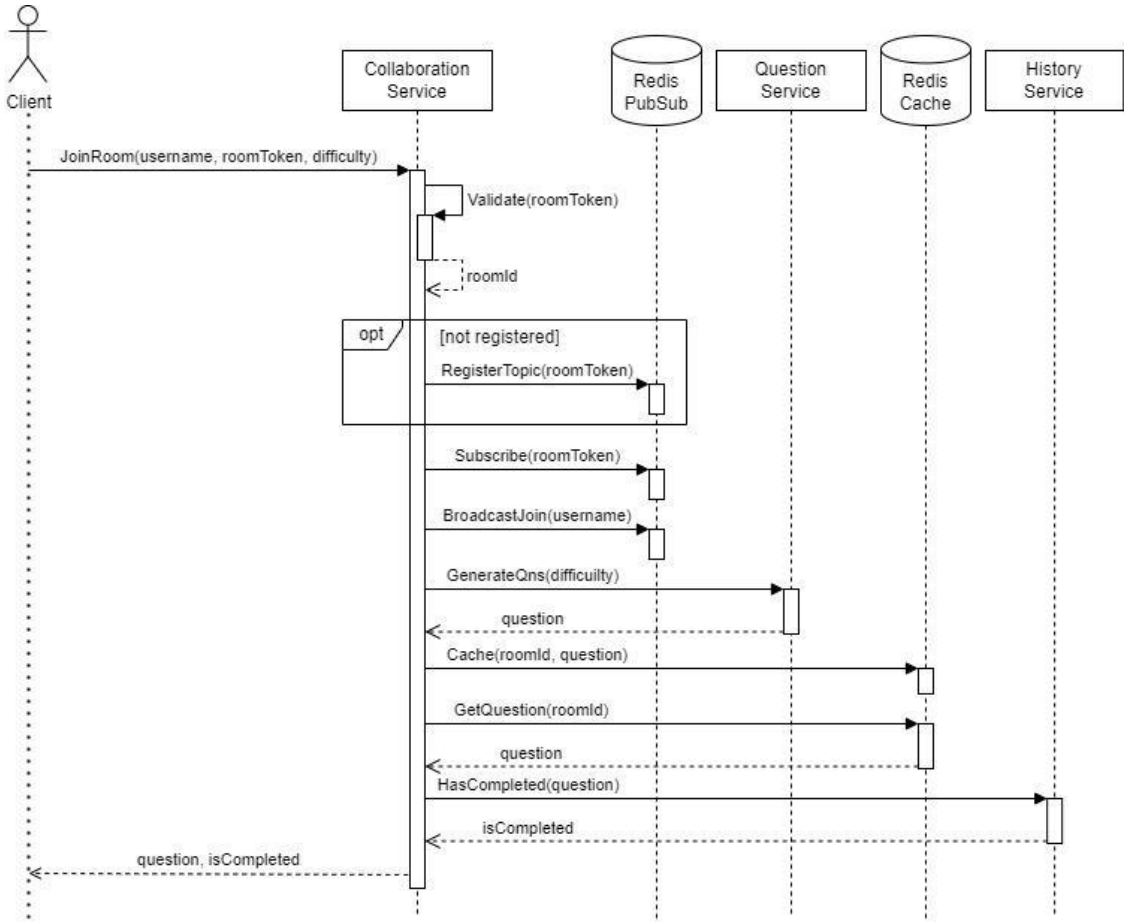


Figure 38 - Sequence Diagram of Room Creation

10.4. Code Snapshotting

Code snapshotting allows users in a session to save their current progress. This information saved allows users to revisit their attempts outside of the session. However, since the state is synchronised across 2 users, there is a need to first disable actions from both of them to preserve a common static state that can be saved. Due to the distributed nature of the collaboration workspace state maintained by the frontend, this is a challenging task for the backend and relies on a protocol that is modelled after the Address Resolution Protocol (ARP) discovery process.

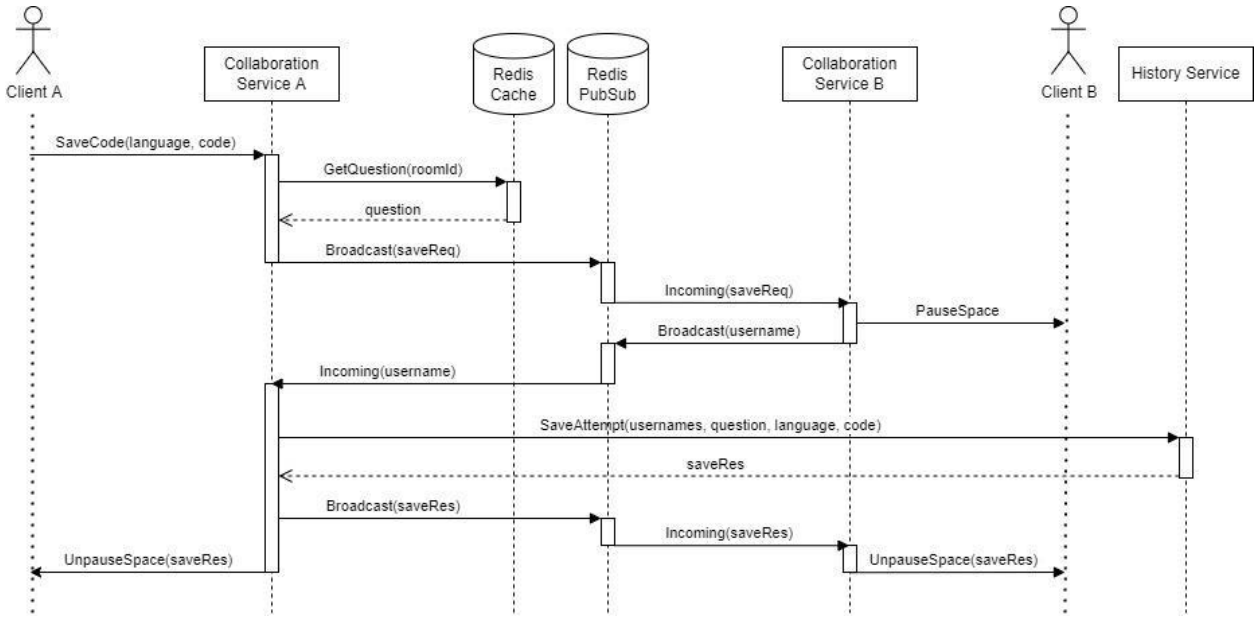


Figure 39 - Sequence Diagram of Code Snapshotting

The process is as follows:

1. User sends a save snapshot request to the collaboration service containing the details of the workspace since the backend is not aware of them.
2. Collaboration service broadcasts the intent to save a snapshot and waits for all workspaces to be locked.
3. Collaboration service creates a snapshot and saves it to history service.
4. On completion or timeout, the collaboration service will broadcast a message to unlock the code workspace of both users. The timeout mechanism acts as a failsafe to ensure that a deadlock for clients will never occur.

11. Object-Level Design Patterns

This section details various object-level design patterns used when implementing the features of Peerprep, and the rationale behind applying them. These patterns mostly relate to the SOLID principles, and aims to increase modifiability through decreased coupling (NMM-4) at the object level.

11.1 Observer

The Observer pattern is implemented in the frontend's Yjs Websocket provider package. This is meant to reduce the coupling between the networking logic from the React frontend logic using the event driven interface, provided by the Observer pattern, for communication between the 2 modules. This pattern provides an event-driven implementation where different events can be emitted to specific listeners which runs some logic as defined on the listener function that is **`Observable#on`**.

Note that the interface Observer and its implementing ConcreteObservers are lambda functions in code and is not strictly an interface in the object-oriented sense.

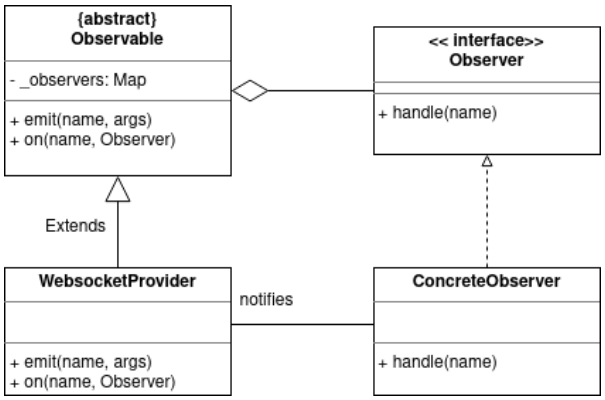


Figure 40 - Class Diagram of Observer Pattern

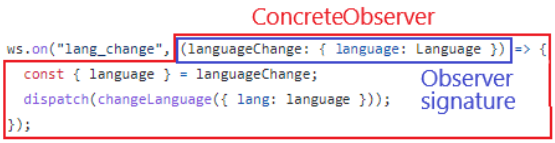


Figure 41 - Observer Implementation

While this eliminates the control coupling, it introduces data coupling as the observer and observable needs to communicate using a common data format. The primary benefit of using the Observer pattern is the extensibility introduced by the event abstraction, allowing the event emitter, the network module, to evolve and change without impacting the React module as long as the existing events do not change. New network events can be easily added by simply defining new event names, and the React frontend can choose whether it wants to listen on this new event.

11.2 Facade

The Facade pattern introduces an additional layer of abstraction that simplifies and hides the intricacies of a complex system or web of modules. It exposes a simplified and aggregated interface to clients, which is used by both the clients and the system behind the facade as a contract. This pattern is widely used throughout the system to reduce coupling, with an example usage in the session service, where the TokenAgent hides the complex JWT validation and blacklist checking logic behind a single simple interface.

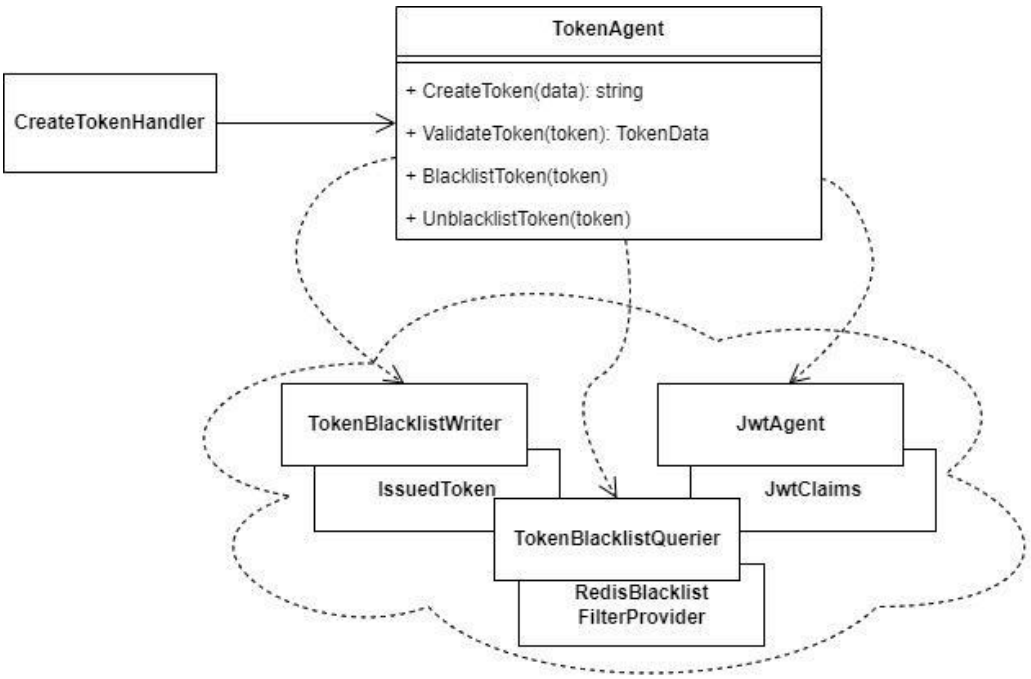


Figure 42 - Facade Pattern in Session Service

The shared common contract adheres to the Dependency Inversion Principle (DIP) and decreases the coupling between the complex system and its clients. Instead of the clients implementing the logic to contact each subcomponent, and in the process introducing many dependencies, it only contacts the TokenAgent facade, which calls the dependencies to perform token operations.

By hiding behind the facade, the JWT token validation logic and blacklist checking pipeline can change independently of clients (the handlers) as long as the facade's contract is satisfied. In this case, the validation process only needs to produce a TokenData struct, and how it does that is not important and not known to clients.

11.3 Chain of Responsibility

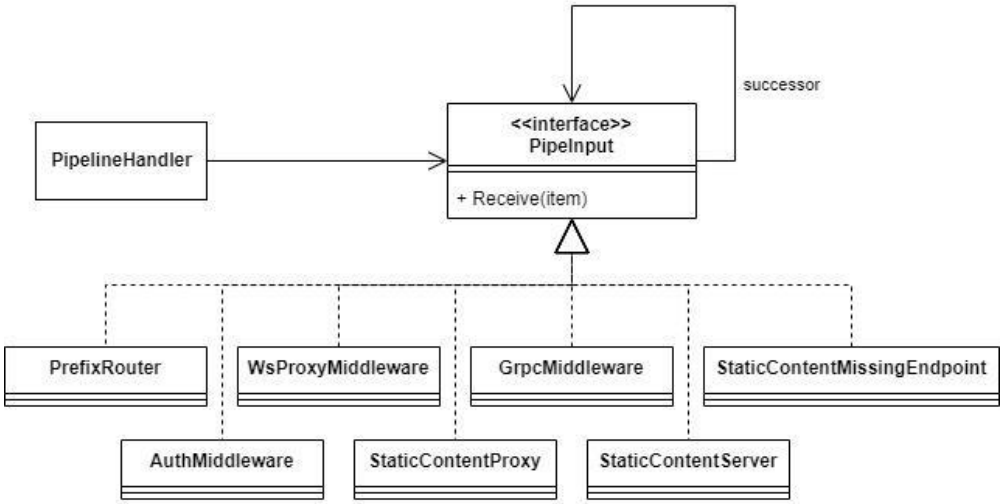


Figure 43 - Chain of Responsibility in Gateway

The Chain of Responsibility object-level pattern is present in the Gateway's request processing pipeline, in which every middleware implements the interface PipeInput and may call another middleware that implements the same interface. This allows multiple middlewares to be lined up in a chain in any arbitrary order, establishing a clear but flexible order in which processing is done. The ordering employed in Gateway follows the decision flow required for request processing.

This pattern is used as it helps decouple and isolate the logic for each middleware to distinctly separate components. Each middleware only needs to change when its own logic is affected and is unaware of other middlewares in the chain. It also allows for extensibility, where a new request middleware in the chain can be added arbitrarily in any section and would not affect the other middlewares present in the chain.

11.4 Command

The command pattern is employed in the API server framework, a use case which has commands in the form of API calls. Each API call has a handler, which implements a common generic interface that provides a function for handling requests. This handler is invoked by a driver that receives requests. An example application is shown below for the Question Service.

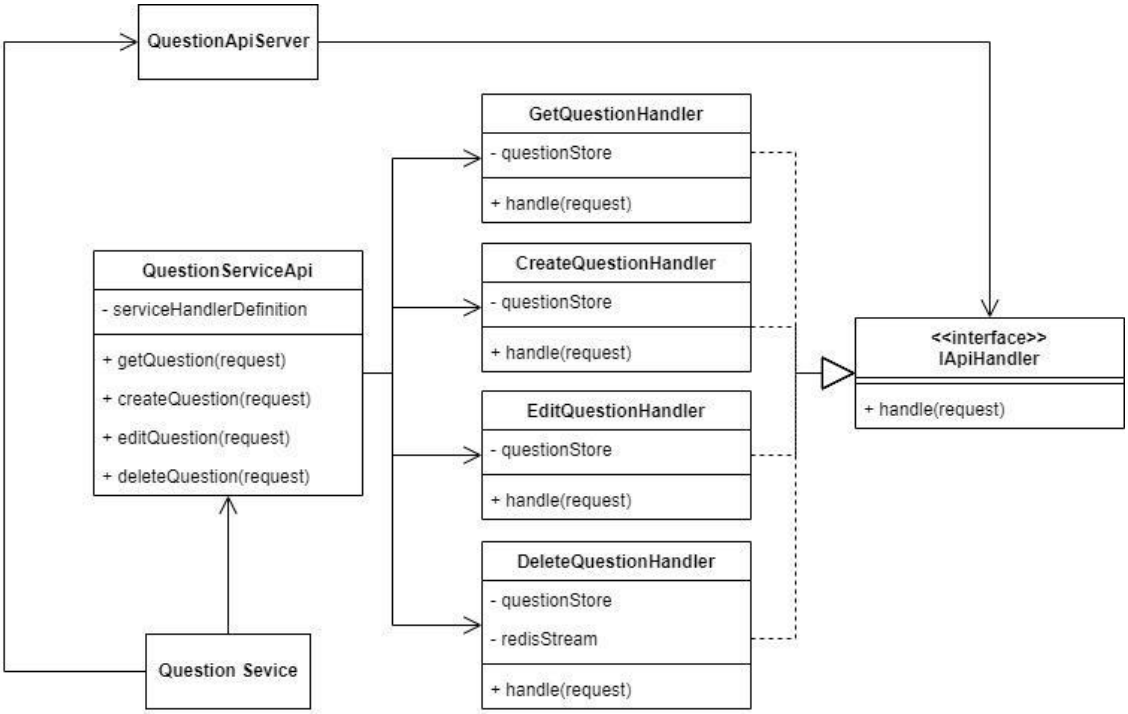


Figure 44 - Command Pattern

This pattern is used due to its strong adherence to the Single Responsibility Principle (SRP) and Open-Closed Principle (OCP). The logic for each handler is independently encapsulated into singular classes, while new handlers can be added easily to the service by simply adding new handler classes and registering it with the invoker. Common generic request objects are used to store the information that the handlers will act on, while common generic response objects are used to return this information back to the invoker. Such a pattern allows for rapid changes in the service's API and corresponding handlers once the base of the pattern has been implemented.

11.5 Strategy

The strategy pattern is employed in the session service's blacklist pipeline, where different blacklists have different processing logic. Ultimately, a blacklist's sole purpose is to maintain a ledger and check for membership in the ledger, operations that can be abstracted into an interface (RedisBlacklistClient). Each type of blacklist then implements its own logic for adding to the ledger and determining if a token is contained in the blacklist.

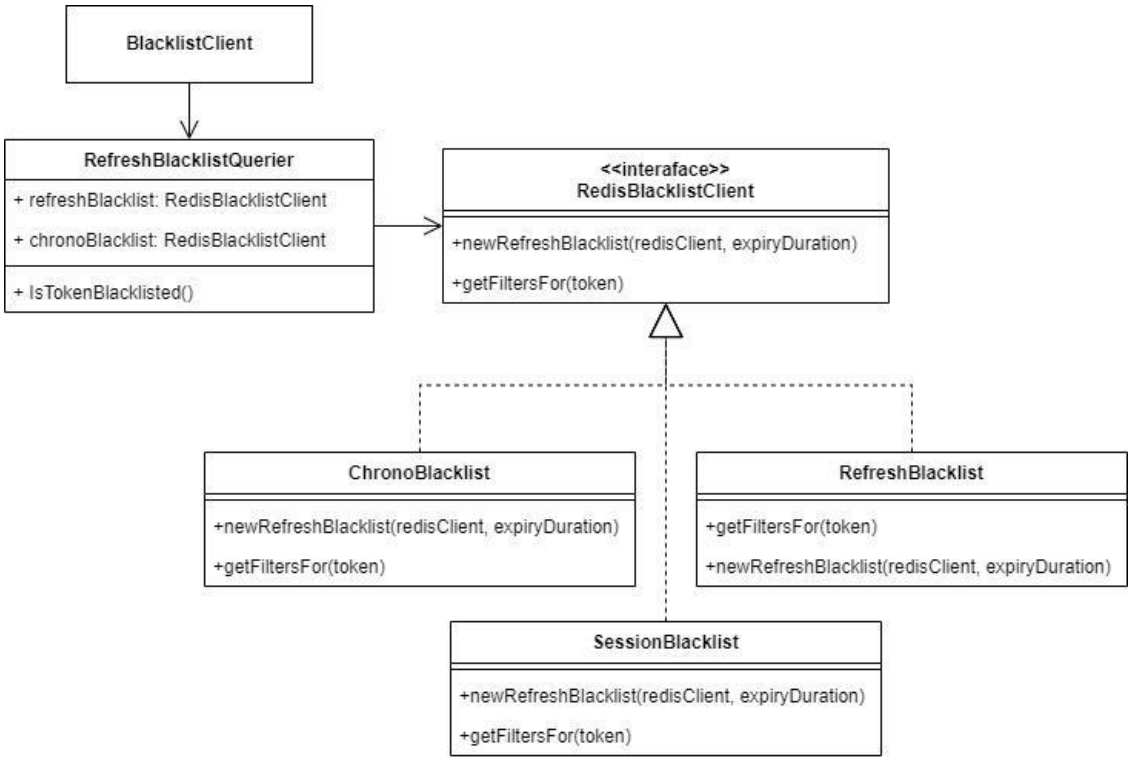


Figure 45 - Strategy Pattern in Session service

In the entire process, the upstream users do not need to know how this logic is implemented, and a query only results in a Boolean result. The strategy pattern achieves this abstraction and separates the concerns of the calling class from the implementing blacklist. It adheres to the Open-Closed Principle (OCP) as new blacklists can be easily added by implementing the interface, and upstream users only need to know that the blacklist can satisfy the contract of the 2 operations. It also exhibits the Dependency Inversion Principle (DIP), where both the caller and implementer depend on a common contract, with no direct coupling between them.

11.6 Adapter

The Adapter pattern enables components that are otherwise incompatible to interact with each other through an intermediate layer. This intermediate layer performs the translation between the incompatible formats and abstracts the 2 sides of the adapter from one another. This decouples both components and allows one to change without affecting the other.

This pattern is heavily employed throughout the services, especially when contacting external dependencies like other services, Redis or the database. One example is the HistoryAgent of the collaboration service. Using this adapter, the collaboration service is able to interact with the history CRUD service and utilise its history operations rather than having the collaboration service implement another method to communicate directly with the database. Note that while the pattern closely resembles the bridge pattern, its intent is behavioural to perform data conversion rather than a structural isolation of components.

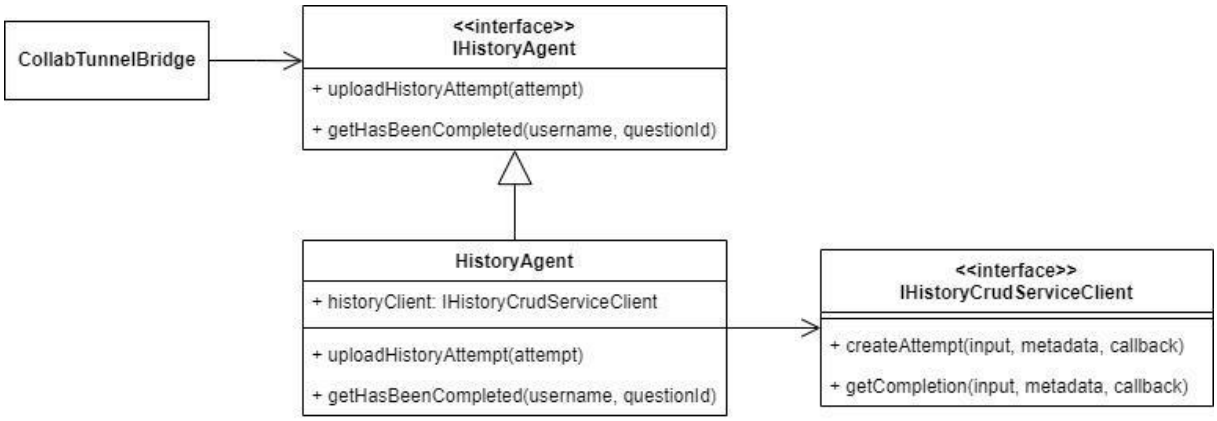


Figure 46 - Adapter Pattern in Collaboration service

The adapter pattern abstracts the networking aspect away from the core business logic and allows the business logic to remain untouched even if a change is made to the external dependency. This also opens the option of changing the dependency entirely, such as from Postgres to MySQL for data storage, without affecting the business logic, as long as the adapter is able to satisfy the contract defined by the interface.

12. Development Process

To create a streamlined development and release process, ideas from DevOps were incorporated into the development process to create a continuous feedback loop between the development and deployment aspects of the project. This involved Continuous Integration (CI) for maintaining a minimum quality in the code base and Continuous Delivery (CD) which automates the release process to the staging environment once a product iteration is ready for release. The process of deploying to production is manually invoked, so the process falls short of continuous deployment. The diagram below shows the overall workflow.

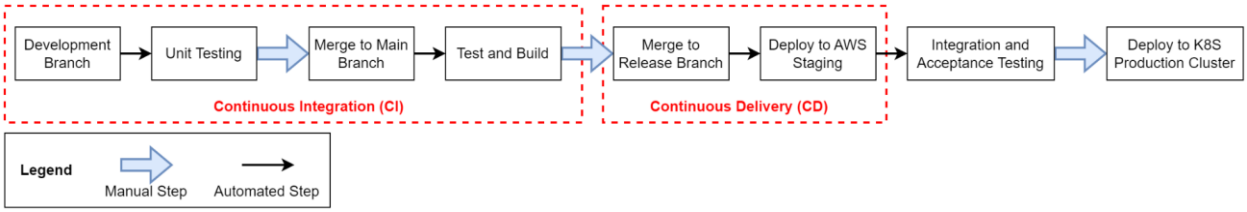


Figure 47 - Development Workflow

Any problems found during the CI phase must be fixed before it can be merged into the release branch for CD, and likewise for CD which blocks deployment to production in the workflow. Both the CI and CD pipelines are implemented using GitHub Actions.

12.1 Continuous Integration (CI)

The development workflow emphasises on the correctness of the codebase, and that all code on the main branch should be compilable, and satisfy a minimum level of quality. This quality metric is made up of 2 parts, code quality in the form of code style and code correctness defined by unit testing.

The CI workflow checks for both of these qualities for all pushes and pull requests. Additionally, the CI for pull requests also checks that the codebase can be compiled into a production version. Finally, the CI for the main branch checks that a docker image can be built correctly, serving as a safeguard for the CD process. Testing and Linting are done in the **`ci-tests`** workflow, compilation is done in the **`ci-build`** workflow and the **`cd-deployment-dry-run`** workflow performs the building of docker images.

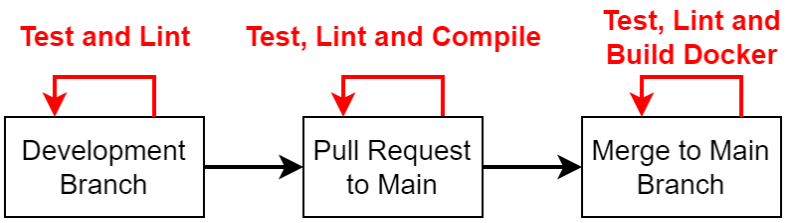


Figure 48 - CI Workflow

12.1.1. Linting

The linting standard used for the Typescript components is the AirBnB standard, while the standard used for Golang is the built-in Golang language standard. Linting ensures that the codebase is written in a uniform coding style and makes it understandable without convoluted constructs or blocks of code. This helps spot issues with code such as circular dependencies and unused variables during development and eases debugging, ensuring that the code that ultimately merges into main is readable and consistent.

12.1.2. Unit Testing

Automated unit testing is employed for each service in the system, with Jest as the testing framework for NodeJS services and Golang's built-in testing framework for Golang services. Unit testing ensures that the correctness of the service is always maintained, picking up any regressions that may occur during development.

12.1.3. Compiling and Building Docker Images

As the programming languages used feature compile-time checks, it is easy to perform type checking and ensure that the codebase will at least compile and run, eliminating CD runs that fail from simple programming mistakes. This is more applicable for Golang, which compiles down to binaries that may have different behaviours from the debug version used in testing.

Building the Docker images that can be used for CD serves as another layer of checks to prevent CD failures and pick up on bad code. This step is targeted towards testing the steps used for building the Docker images and verifies that they can be successfully created.

12.1.4. Code Review

To further enhance code quality, code reviews are made necessary for Pull Requests (PRs), and all code that merges into the protected main branch must have been reviewed in some PR. At least 1 approver who is not the author(s) of the PR must scan through the code and approve the changes made. Doing this step places an additional set of eyes to scrutinise the changes and hopefully pick up mistakes that the author might have missed. This safeguard ensures that the code in the main branch is of high quality.

12.2 Continuous Delivery (CD)

Continuous Delivery is implemented in the **`cd-deployment`** workflow, which automatically deploys changes on the release branch to the staging environment on Amazon Web Services (AWS). Note that this should not be confused with Continuous Deployment, as the deployment is only to staging and not production. A strict rule is imposed that restricts the merging source to only the main branch, and that all PRs for merging to the release branch must be approved by 2 other team members. This is to ensure that the majority of members are aware of the release.

The CD workflow builds each service into docker images and uploads them to Amazon Elastic Container Registry (ECR). The images are tagged according to the service name and commit hash, allowing easy identification and rollback of image versions where necessary.

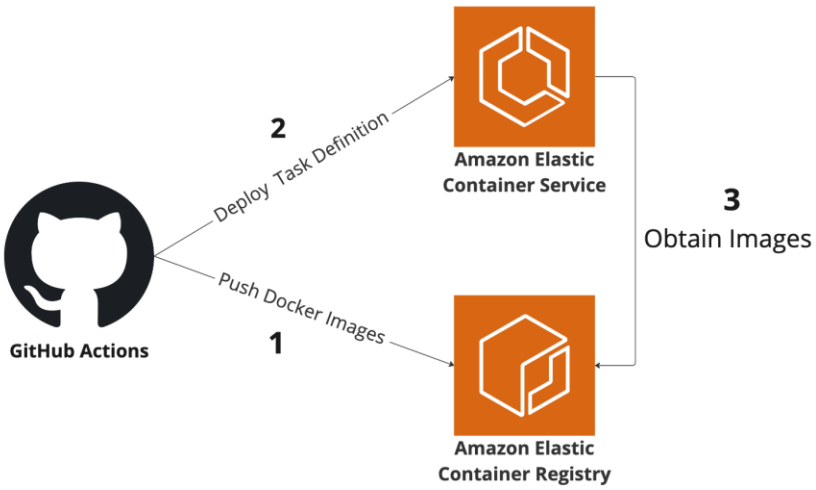


Figure 49 - Continuous Delivery Workflow

Thereafter, the CD workflow will add the new ECR image links to the predefined task definition for Amazon Elastic Container Service (ECS) from the GitHub Repository. The environment variables are also configured at this stage, which are pulled from the encrypted GitHub repository secrets. A deployment command is then issued to ECS to roll out this new task definition, which will update the services on the staging environment.

12.3. Deployment to Production

Deployment to the production Kubernetes (K8S) cluster must be manually invoked, but the deployment process itself is automated using the **`production-deployment`** GitHub actions workflow. The workflow builds and deploys the images to the K8S cluster, and also allows for scaling actions on the cluster. The workflow is restricted to only run on the release branch when deploying code.

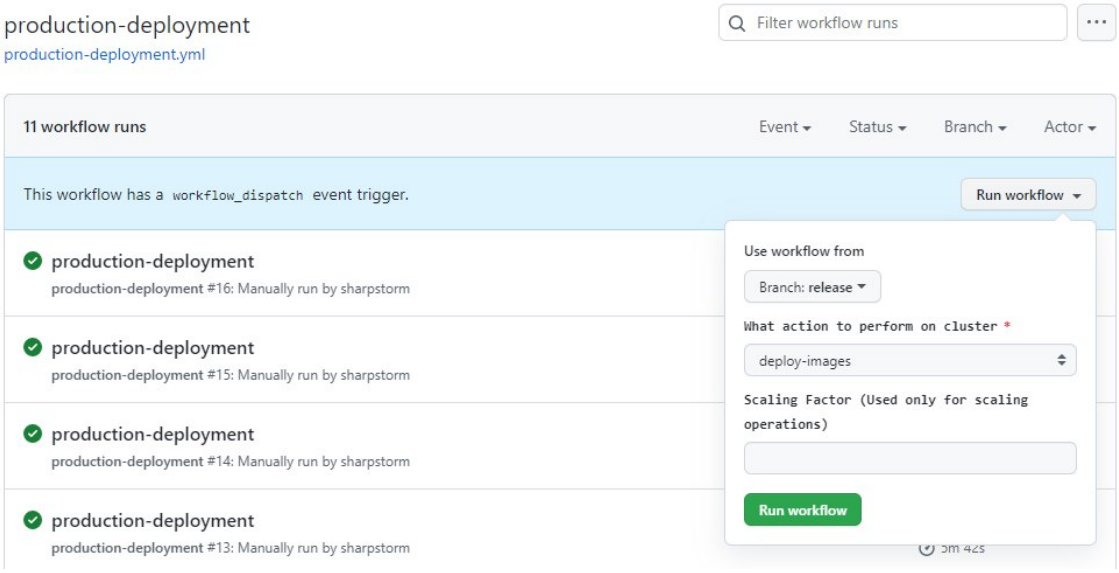


Figure 50 - Production Deployment Workflow

13. Deployment Architecture

This section will describe the different deployment architectures considered and employed.

13.1. Staging Environment - AWS Deployment

An Amazon Elastic Container Service (ECS) cluster-based deployment is used as a staging environment for the system. Amazon ECS is a fully managed containerised orchestration service that closely resembles K8S, but abstracts away the low level details of K8S. ECS uses a cluster of EC2 instances to run “tasks”, which correspond to the different services deployed on the ECS environment. A task definition is used to specify the deployment information for the microservices, which includes the number of containers for the task, their respective allocated resources like compute and memory limits, environment variables and network links.

Since minimal scaling is required in the staging environment, ECS is preferable because the abstraction of orchestration greatly reduces the resource overheads introduced in a K8S deployment. This simplifies cluster orchestration and allows it to use much less cloud resources, while being almost an exact replica of the production K8S deployment setup.

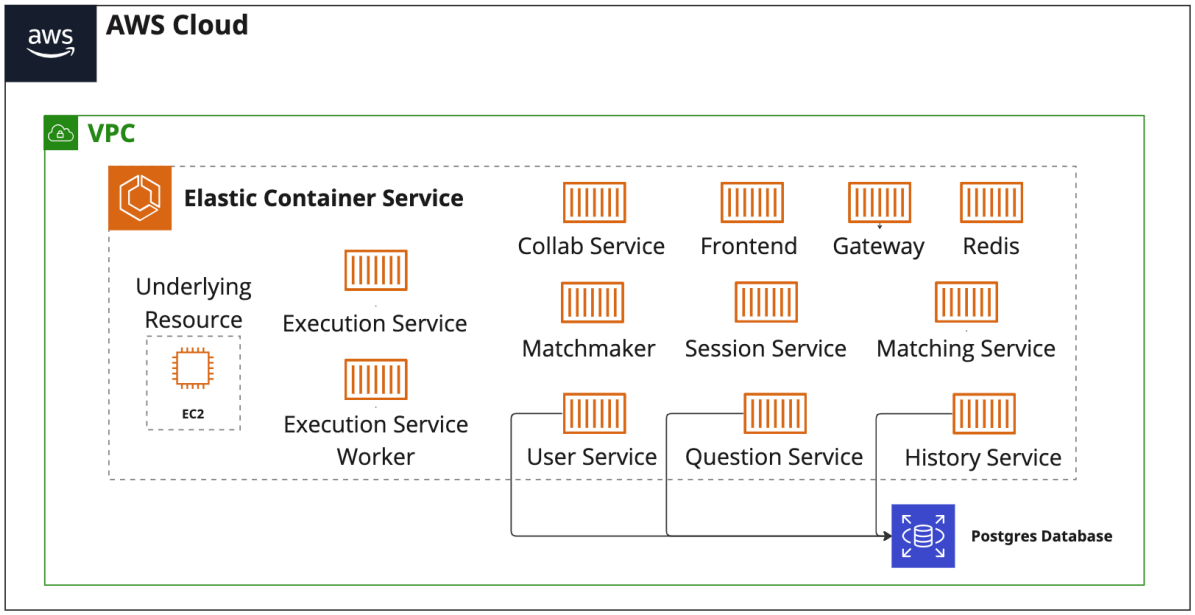


Figure 51 - ECS Deployment

Note that services within the ECS cluster can reach each other without the need for complex networking setups. The staging environment does not feature HTTPS, which is reserved for the production deployment.

13.2. Scalable Production Environment

The team did explore using AWS for a scalable deployment, specifically using networking services provided by AWS that are supposed to integrate with ECS seamlessly. However, none of the approaches worked properly, and the team resorted to a K8S deployment for production.

The following sections will discuss methods explored for a scalable AWS deployment.

13.2.1. Deployment Networking Patterns

The networking aspect of a scalable deployment is fairly complex as there is a need to route traffic destined for a service into one of the running instances. This section serves to provide some context to the available architectures for this routing.

The table below shows the high-level structure of the commonly used patterns.

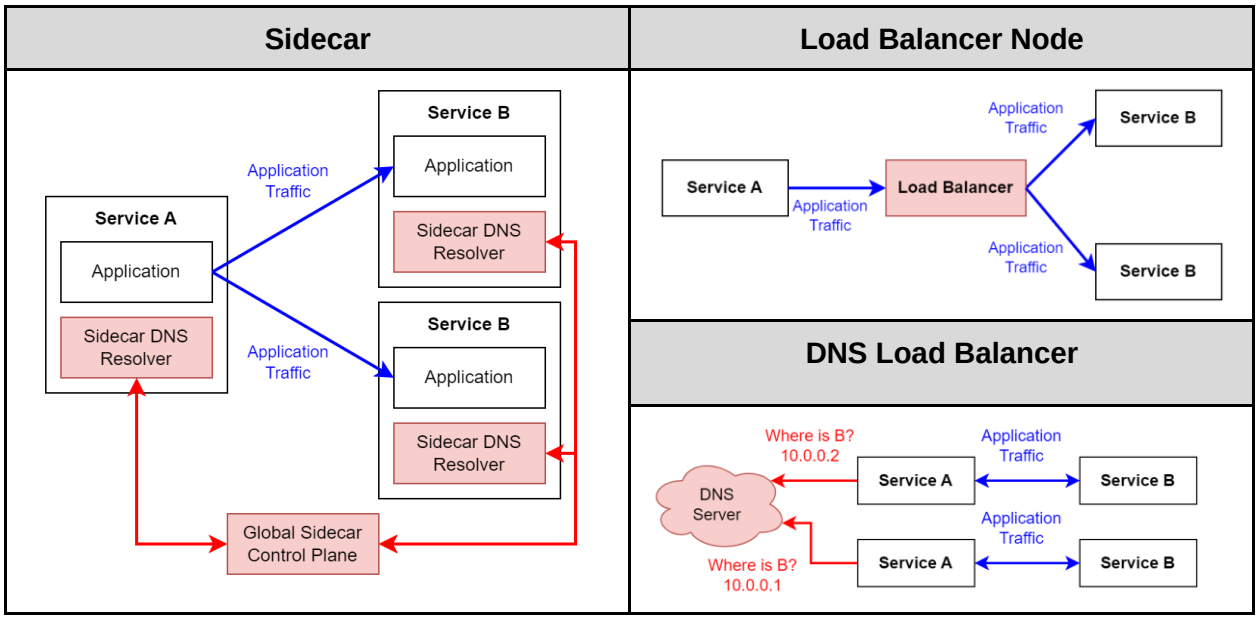


Table 52 – Commonly Used Deployment Networking Patterns

The sidecar pattern uses an additional companion module (known as the sidecar) to perform DNS resolution at the replica level, where each replica has its own independent DNS resolver. A global sidecar control plane which provides updates to each resolver as the cluster changes. This architecture allows for a single hop directly from one service to another, resulting in low latency. However, the distributed nature of the pattern means that each service has the added complexity of maintaining its sidecar module and requires additional resources to run it.

The load balancer node pattern on the other hand implements DNS resolution opaquely from the microservices. All services are hidden behind a load balancer node, and which chooses the replica of the service to contact before forwarding traffic to it. This means that the load balancer is the single point of ingress for a particular service. This centralised approach enables the load balancer logic to change independently, and reduces the need for complex sidecar modules, but introduces additional latency into the request-reply flow.

The DNS load balancer is a variant of the load balancer node approach, but the node is eliminated, and load balancing is done at the DNS level. This restores the single-hop low latency but shifts the load to the DNS server since all services now depend on it for routing.

13.2.2 AWS AppMesh

AWS AppMesh implements the sidecar pattern for networking. It provides a dedicated infrastructure layer (control plane) for governing microservices and facilitates service-to-service communications using an additional module tagged onto each service. This network routing enables scaling up to multiple replicas of a single service with a low-latency single-hop architecture.

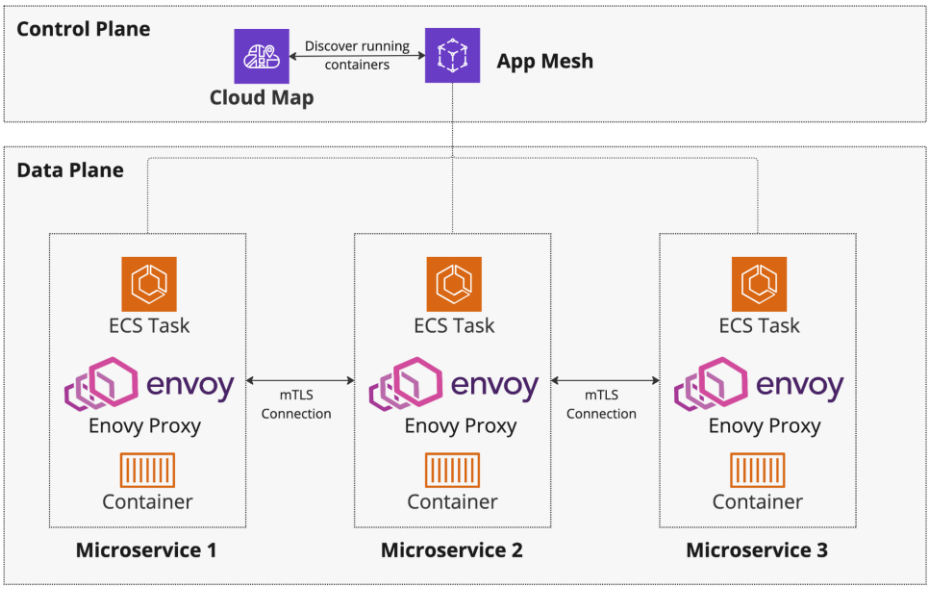


Figure 53 - AWS AppMesh Architecture[1]

In the control plane, the AppMesh and Cloud Map act as the global sidecar control plane to allow service discovery over the Domain Name Server (DNS). It manages all the launched instances of individual services and provides a control panel from which this can be monitored. In the data plane, the envoy proxy of each microservice plays the role of the sidecar to communicate with the control plane. The envoy proxy automatically intercepts traffic from the services and redirects it based on the control plane's mapping.

However, this approach was not successful as the team could not get the AppMesh envoy image to connect to the control plane despite the team's best efforts. The image was overhauled recently, but the official documentation was out of date and referred to deprecated code, showing instructions for the older versions. Since there was no accurate documentation available in the public domain for properly configuring the AppMesh, in particular the environment variables for the envoy proxy, the team decided to drop this approach.

1 - https://aws.amazon.com/blogs/architecture/deploying-service-mesh-based-architectures-using-aws-app-mesh-and-amazon-ecs/

12.2.2 AWS Load Balancer Deployment

The load balancer approach is similar to the non-scalable ECS deployment and implements the load balancer node pattern. The key difference is that each microservice will have its own load application balancer node to route traffic to the instances. This enables services to be spread across multiple EC2 instances or even ECS clusters.

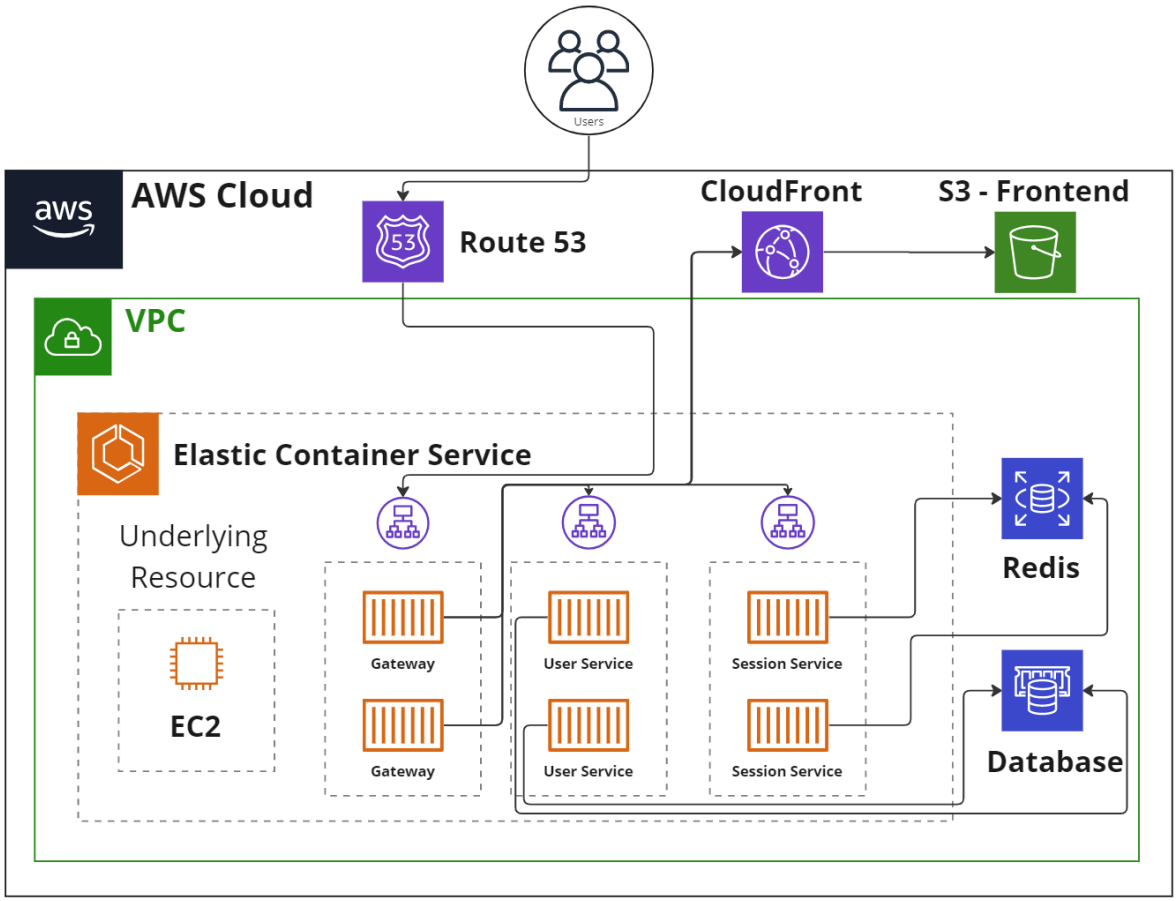


Figure 54 - Load Balancer Deployment

The planned architecture uses Amazon Route 53 to use its load balancing capabilities for ingress into the deployed AWS ecosystem. Each service is hidden behind an internal local balancer, which manages routing to the spawned instances. It also moves the frontend to an AWS S3 bucket for it to be delivered to the end user via AWS CloudFront, a content delivery network (CDN). These modifications serve to improve the load handling characteristics of the deployment.

While this approach is also a plausible deployment architecture, it is extremely expensive to upkeep since each AWS service and load balancer are billed separately. Moreover, the increased complexity and heavy tie-in to the AWS ecosystem makes it less appealing and increases the difficulty in switching over to another cloud provider should it not work out. As such, the team also decided that this approach was not feasible in the context of this project.

13.3. Kubernetes Deployment

Finally, the team arrived at a Kubernetes (K8S) based deployment on Google Cloud Platform (GCP). Since K8S is a widely used and open-sourced orchestration service common across cloud vendors, there is significantly more flexibility than AWS ECS. Moreover, the open-sourced and well-documented nature of K8S makes it easy to deploy on. However, it does come at the added cost of high overheads, which the team felt was unavoidable if a scalable deployment was to be pursued.

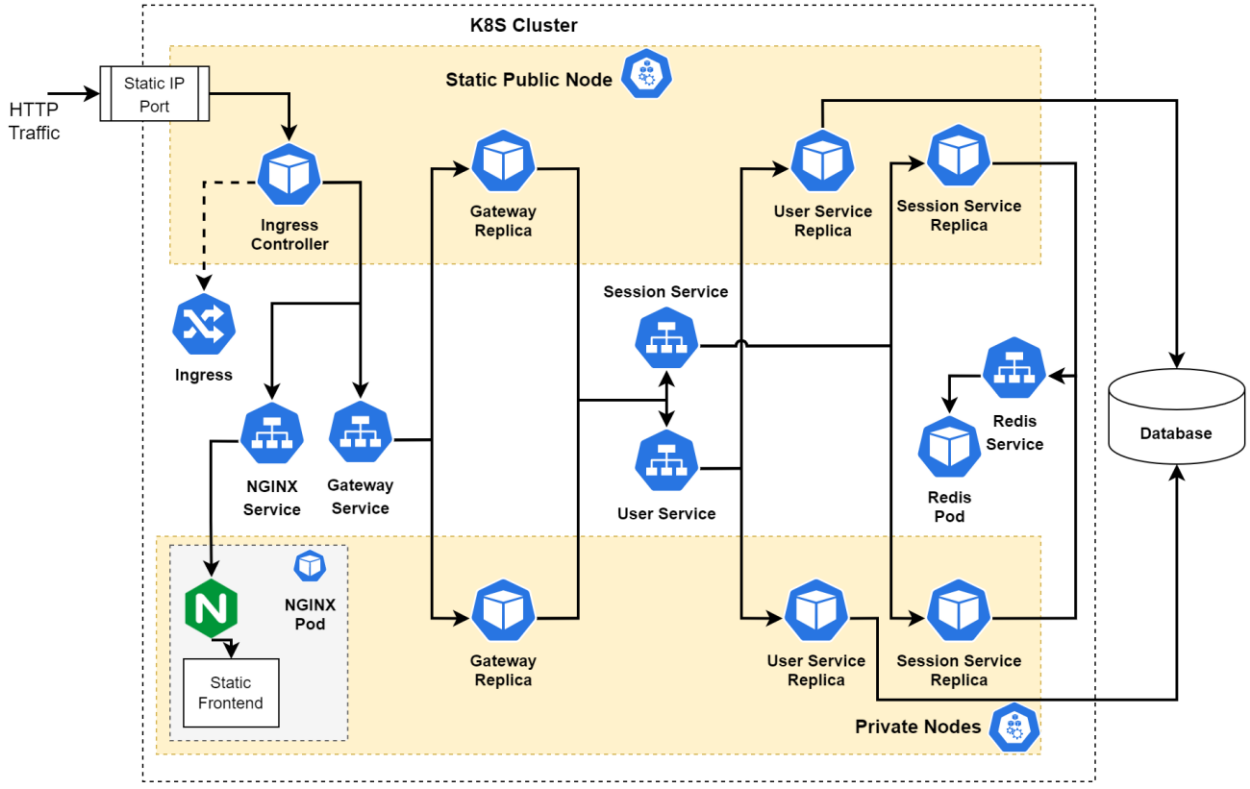


Figure 55 - Final Production Environment

Note that this deployment closely resembles the planned AWS Load Balancer deployment but offloads the load balancing to K8S's internal service discovery and routing mechanisms. K8S implements a mix of sidecar and DNS load balancer patterns, where each K8S node contains a sidecar module (kube-proxy) for multiple pods and these pods resolve routing through the node's own DNS server, similar to how routing is done in a DNS load balancer pattern.

The K8S deployment used a modified architecture when compared to AWS due to the presence of an ingress controller, which enabled Layer 7 routing logic at the control plane. This allows the ingress controller to directly proxy to an NGINX server to serve static pages without having to go through the API gateway. In this deployment, the gateway is said to sit behind an NGINX server, since requests for static resources never reach it. This helps shift the load away from the Gateway and reduces the amount of traffic that it handles.

13.3.1. Transport Layer Security in Production

In addition to GCP, Cloudflare is used as a reverse proxy to configure the domain name and Secure Sockets Layer (SSL) of the production environment. It serves as a jumping off point to present clients with a valid SSL certificate without having to configure it in GCP, allowing both the domain and its associated SSL certificates to be managed at the Cloudflare level. It should also be noted that both the user-cloudflare and cloudflare-GCP connections operate in HTTPS, with GCP using a self-signed certificate.

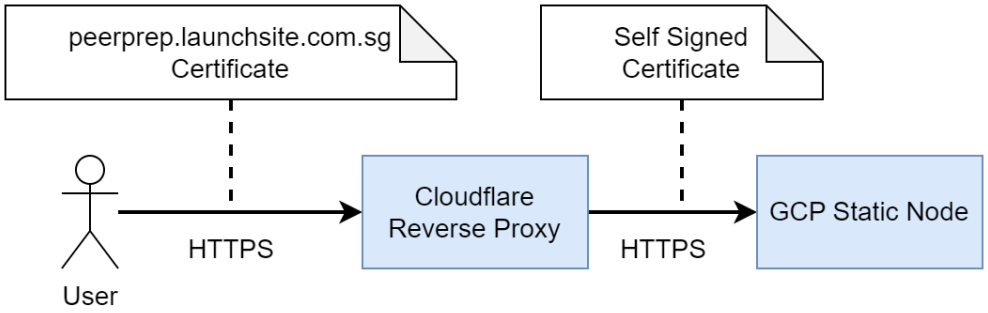


Figure 56 - Cloudflare Reverse Proxy

13.3.2. Automatic Horizontal Scaling

While the K8S deployment only supports manual scaling in its current configuration and not automatic horizontal scaling, it can be easily added by creating HorizontalPodAutoscaler objects in the K8S cluster. These horizontal scalers may tap into the existing metric server on the K8S cluster and use the CPU consumption to determine if services can be scaled up or down, satisfying the scalability NFR NSS-3.

13.3.3. Production Deployment Latency

While the deployment works for all intents and purposes, there was high network latency for the K8S cluster deployment. In fact, the same HTTP requests were much slower than equivalent requests in the AWS staging deployment. The team did not fully understand the issue as it could not be seen in both local testing and staging environments and could not be reliably reproduced.

However, the team noticed that this issue typically occurs after the K8S deployment has been left untouched for a while and hypothesise that it could be due to latencies involved in contacting the Database, since they are now hosted on different cloud providers. This issue could possibly be mitigated by shifting the database onto GCP, but in the interest of time, the team did not do this. Ultimately, the team did not manage to resolve the latency issues that were present.

14. Improvements to Product

While the final product did achieve the team's vision and goals set out at the start of the project, there were several ideas and features the team had to drop due to time constraints. Given more time, the team would have liked to make the following improvements to the end-product.

14.1 Voice Chat

In the stretch goals that were initially laid out, the team had intended to incorporate multimedia communications in the form of Voice chat for the code workspace. However, this was a low priority feature for the team due to the complexity and the relatively little benefits it would bring.

Nevertheless, it is a feature that enhances the product and makes it more well-rounded since one of the main quirks of the product is the ability to collaborate with others. Communicating over voice would be more convenient than text and create a more collaborative and complete workspace, eliminating the need for users to resort to external services for communication.

An implementation of this feature could integrate an external service such as Agora or use the WebRTC standard to achieve this goal.

14.2 Improved Matchmaking

In the delivered product, the matchmaking process is a rudimentary First-Come-First-Serve (FCFS) system, where the earliest 2 users with the same difficulties would be matched. While this serves the purpose, a smarter matchmaking algorithm would improve the user experience since it will better tailor matches to the users based on certain conditions.

An implementation of such an improved algorithm, as suggested by the FRs developed (FB7 to FB10), is to introduce filters for questions that the user has completed before and to match users based on a perceived Elo rating. Both of these would improve the matching accuracy and reliability for users, giving them an overall better experience in the long run as their attempt records build up. Such a system would engage users for a longer time period by adjusting the difficulty level of the question and capabilities of their partner to be closer to their skill level.

14.3 Question Selection and Single-User Sessions

Currently, users must match another user in order to begin tackling a question. Moreover, questions issued to users are randomised, and users are also unable to specifically select a question from the question bank to tackle. An improvement would therefore be to allow single users to enter "solo sessions", where they are alone and they have full control over the workspace, including the question selected for the workspace. While this circumvents the collaborative nature of the platform, it provides users with more options and allows those who prefer it to use the platform without interacting with others.

15. Reflection

Over the course of the project, the team gained invaluable experience in working with enterprise tooling, standards, and deployment platforms. This stems from the team's efforts in attempting to create a system that would be compliant with standards used in the industry, and adopting practices employed in production systems where feasible.

15.1. Standards and Technologies

The focus on standards meant that the team had to learn new technologies such as Protocol Buffers, gRPC and Redis amongst others used in the tech stack. This led to the adoption of more efficient workflows and processes that greatly simplified the development process, knowledge that can be carried over to future projects. Protocol Buffers in particular helped to standardise ideas and formats across the entire system, which is invaluable when dealing with the numerous services that were being developed in parallel.

During the project, the team also explored and considered various frameworks to solve the issues that arose during development. While these frameworks may not ultimately be used in the project, exposure to them will definitely be useful as it can factor into decisions made for future software engineering projects.

15.2. Deployments

This project was also the team's first venture into application deployments, which led to learning the techniques in which deployment can be done and the frustrations that come along with it. Different cloud vendors were explored, with experimental deployments done on different solutions and cloud architectures. Through this process, the team was able to learn how the different toolchains that cloud providers offered manage scaling and networking differently, despite having the same exact codebase to deploy. While the team did not dive into the details, exposure gained from troubleshooting problematic deployments enriched the team's knowledge of scalable applications, especially the microservices architecture, which the team believes will be useful in future endeavours.

In deploying and operating the system, the team also learnt the importance of certain DevOps practices, specifically CI and logging, over the course of the project. While CI was adopted in the early stages, it did not test the code rigorously enough and led to issues in the staging environment. Implementing logging into each service helped debug issues, and ultimately showed the team the importance of a properly managed CI process.

15.3. Design and Architectural Patterns

Furthermore, the team is able to apply some of the various architectural, communication and object-level patterns taught in lectures to the project and incorporate these into the system's design. This allowed the team to better understand and appreciate the patterns, including the benefits and potential problems of applying them. Application of these patterns helped the team achieve a cleaner and higher quality architecture design at all levels.

15.4. Team Processes

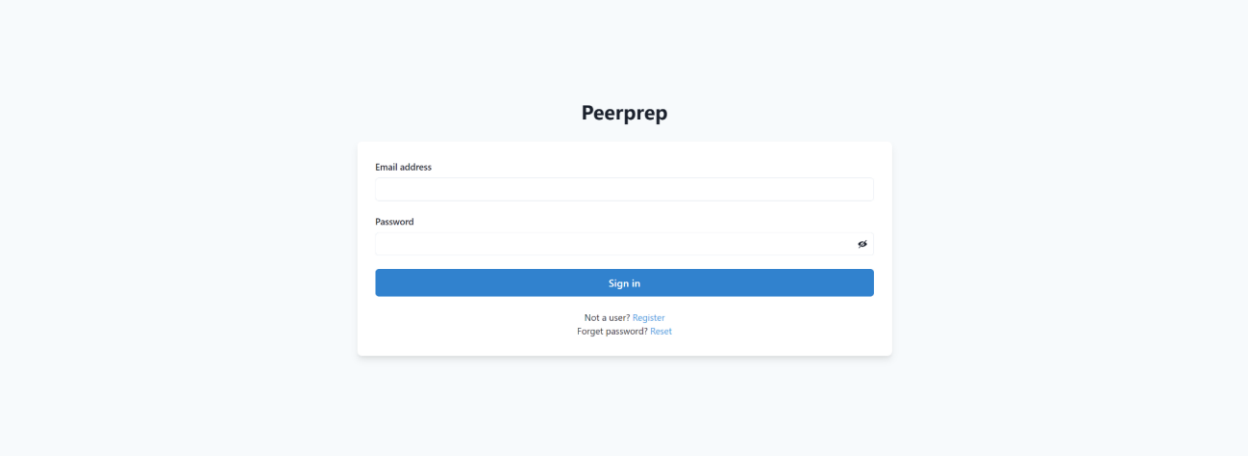
As each individual team member takes ownership of certain components of the application, a practice similar to how a software engineering team is typically structured, the team was able to understand the development processes and challenges in greater detail. Given the complexities of the product, the team undoubtedly faced bumps along the way when members who owned different portions of the system were not aligned on certain issues. This led to extra effort in fixing the misaligned components, and highlighted the importance of sprint meetings and open communication, important pillars of the SCRUM framework.

16. Product Screenshots

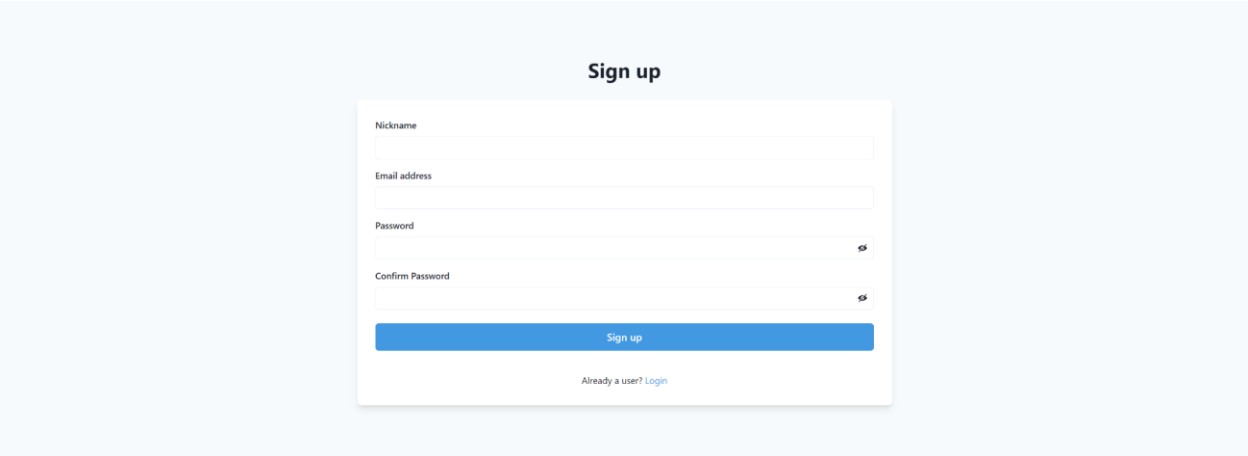
This section includes screenshots of the final product, deployed on the deployment environment.

16.1 Account Pages

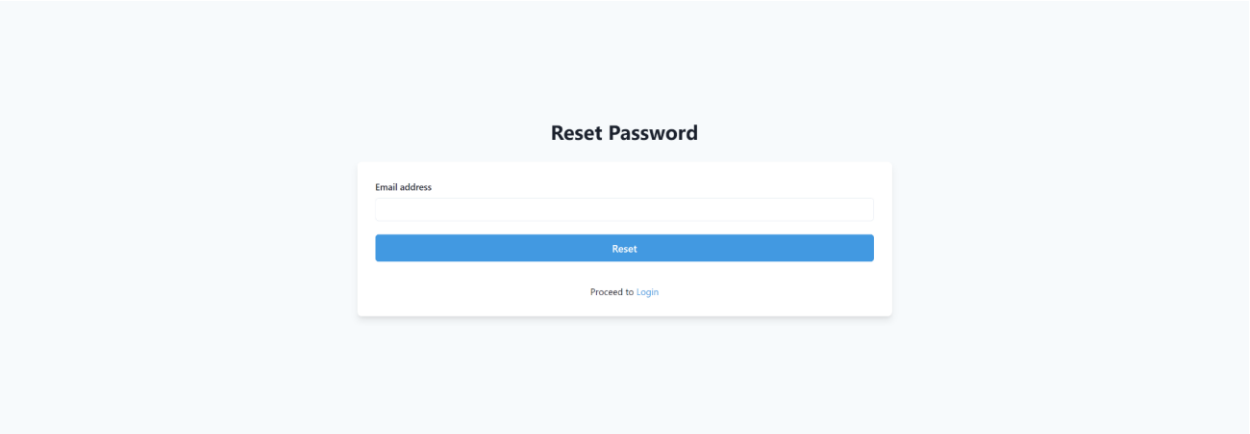
16.1.1 Login Page



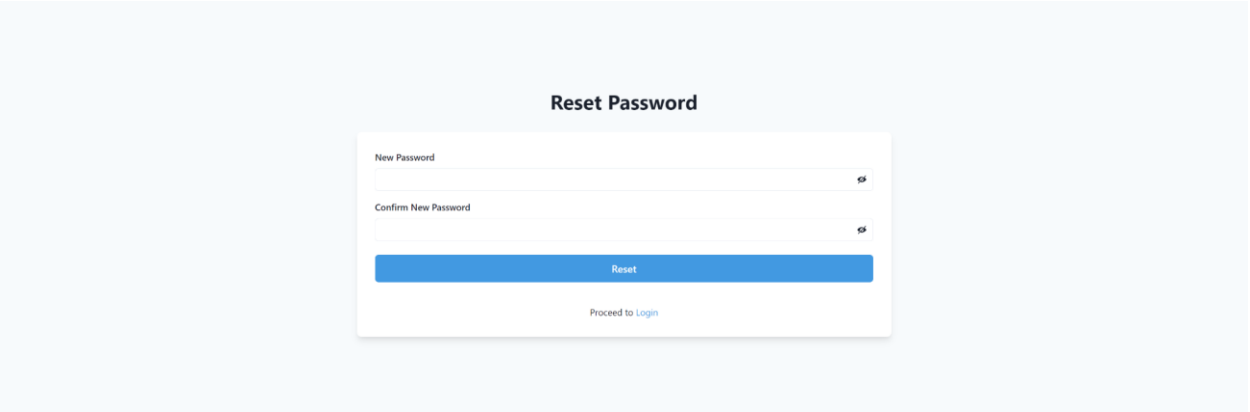
16.1.2 Sign Up Page



16.1.3 Reset Password Page

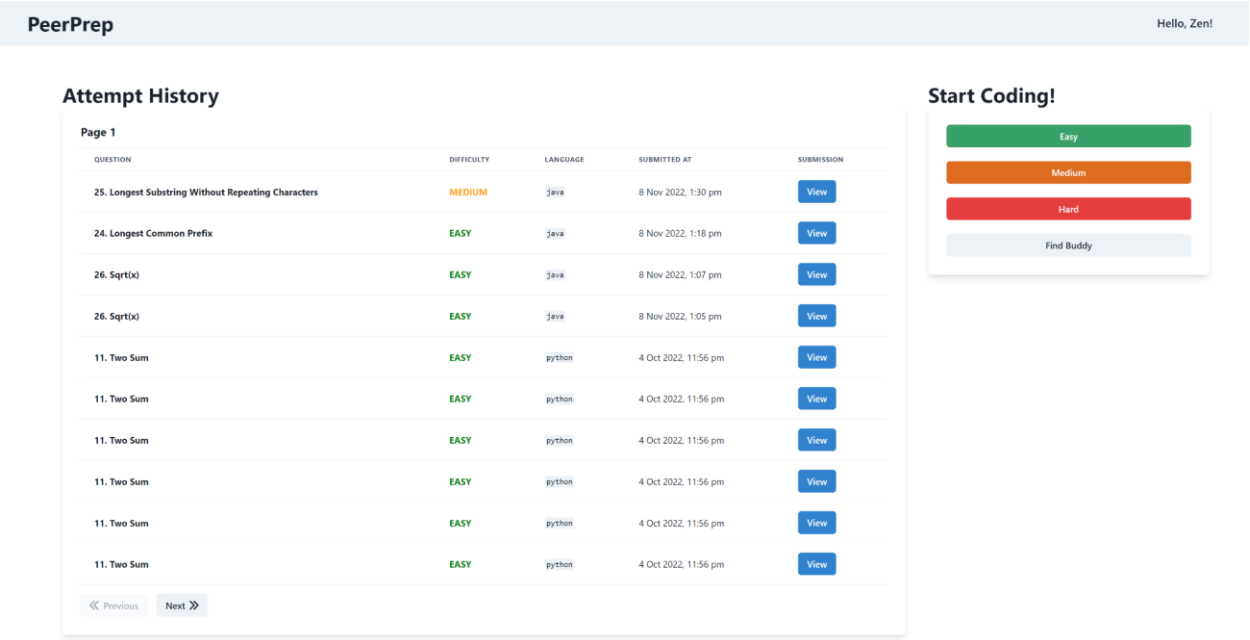


16.1.4 Set New Password Page

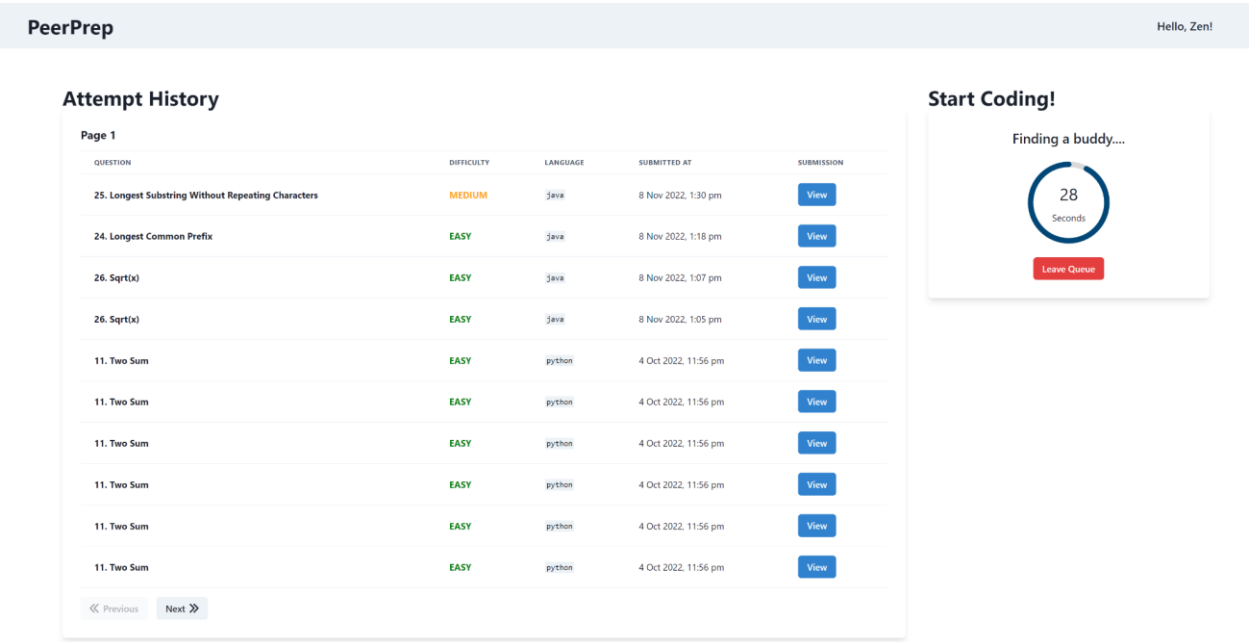


16.2 Portal Page

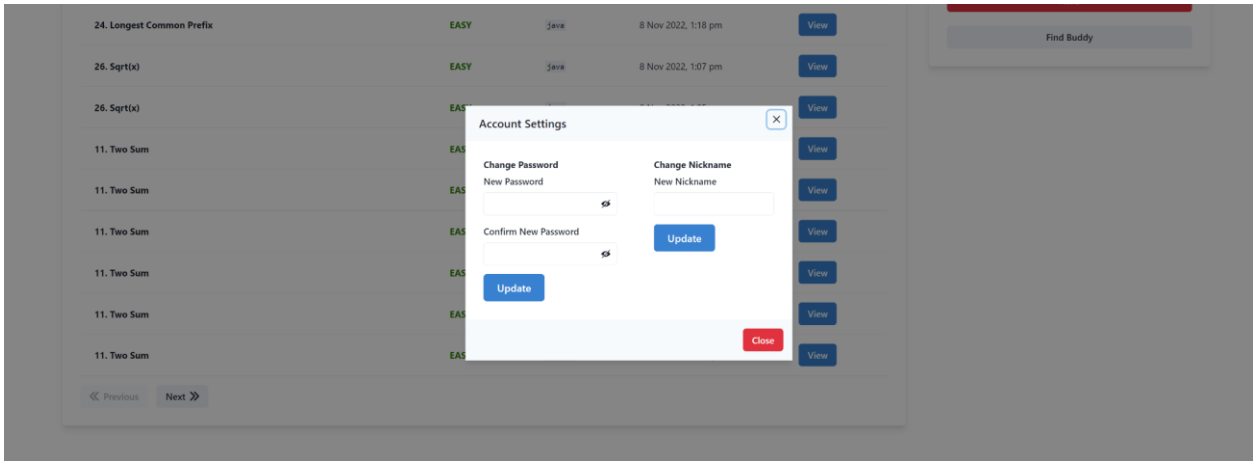
16.2.1 Logged-in Landing Page



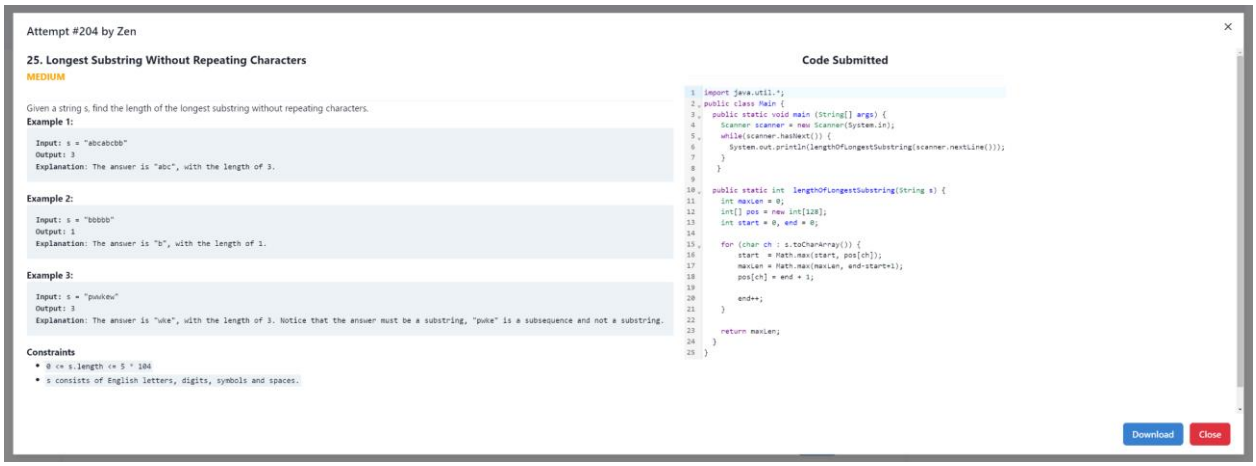
16.2.2 Joining a Queue



16.2.3 Edit Account Settings

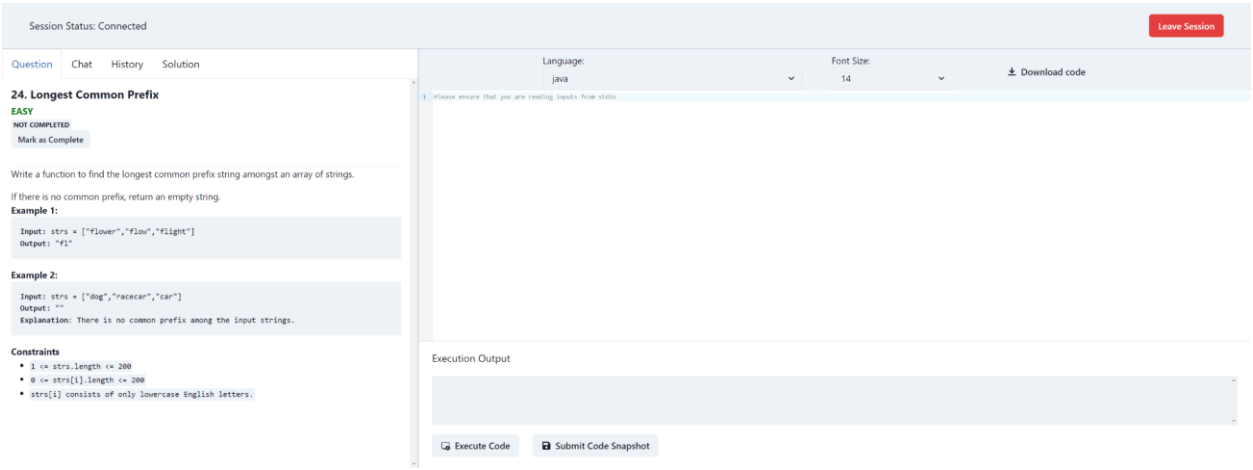


16.2.4 Viewing History Attempt Details

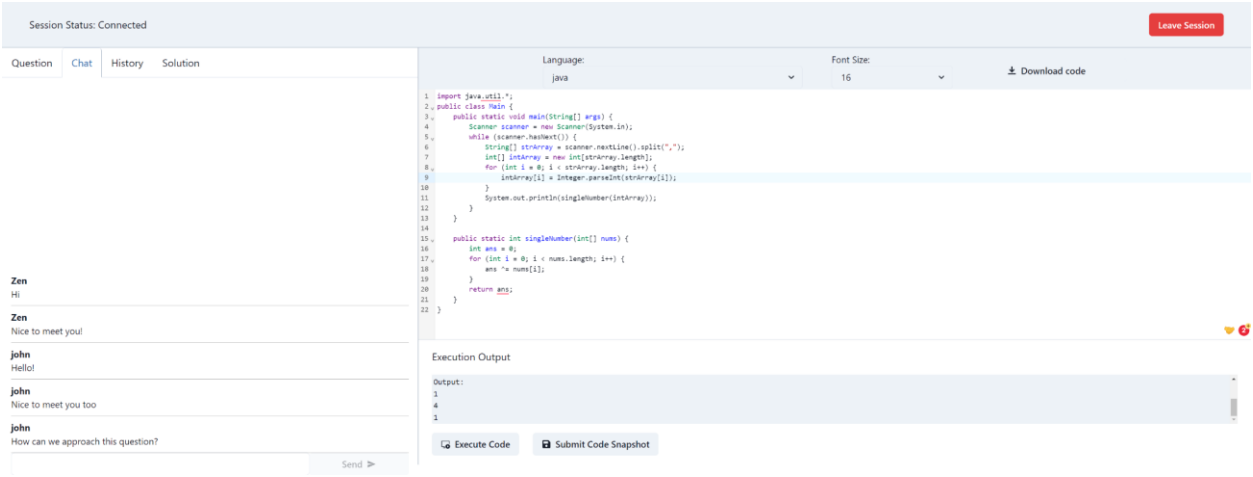


16.3 Coding Session Page

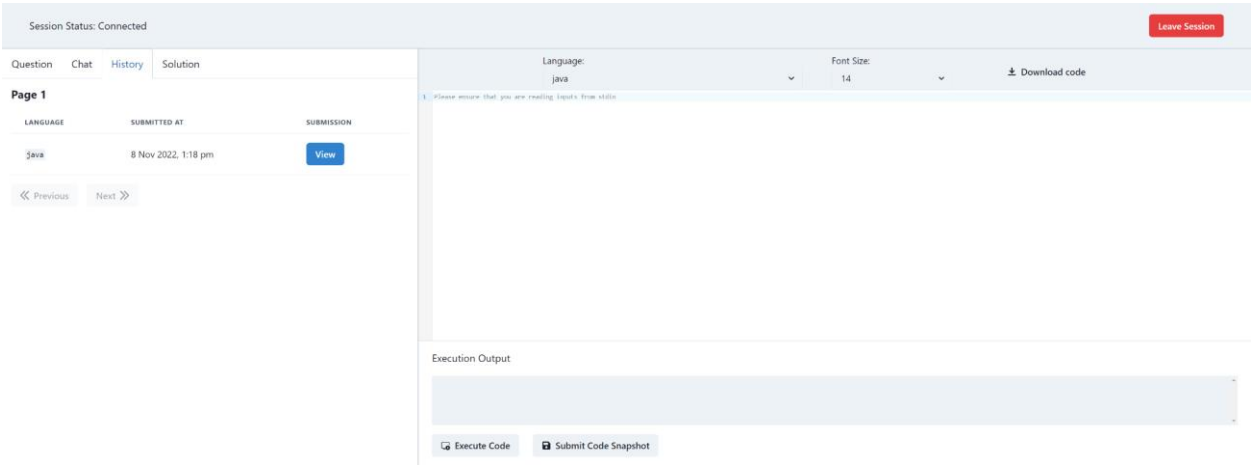
16.3.1 Session Page with Question Tab



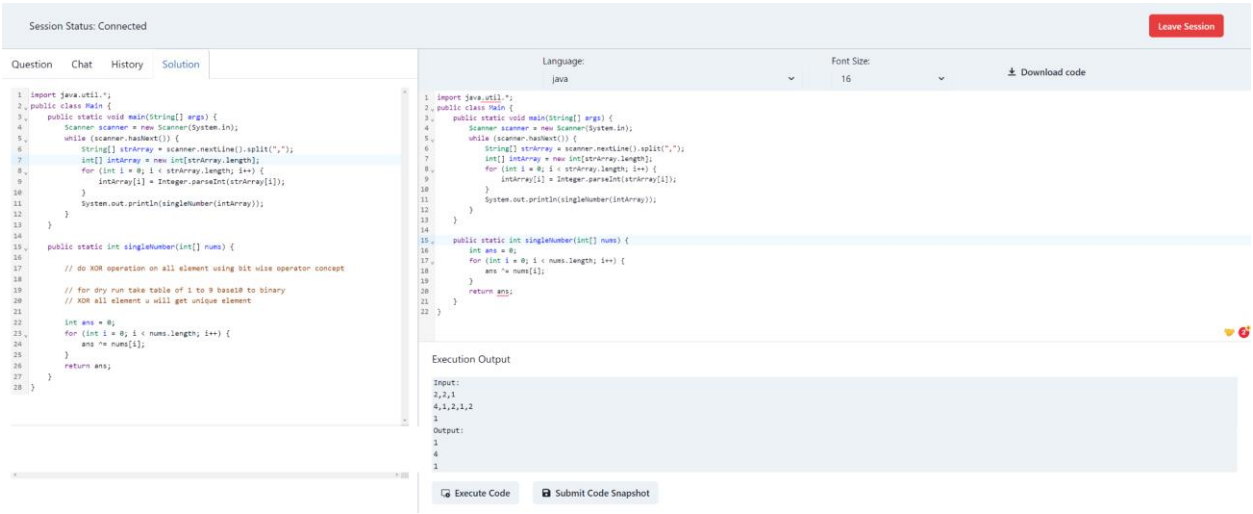
16.3.2 Session Page with Chat Tab



16.3.3 Session Page with History Tab



16.3.4 Session Page with Solution Tab



Appendix A - Requirements Document

A0. ID Notational Meaning

Requirements are coded in a hierarchical manner, where the exact nature of the requirement can be deduced from the requirement ID.

Format	`[Requirement Type][Subcategory]-[ID]`
Requirement Type	● **F** - Functional Requirement ● **N** - Non-Functional Requirement
Subcategory	Refer to tables within each requirement type
ID	Running number for each requirement

A1. Functional Requirements (FRs)

The FRs are broadly organised based on product phases, followed by FRs of closely-related functionality that work together to deliver a feature. This is meant to further add a layer of prioritisation and introduce a prerequisite ordering to the FRs, that is, the FRs of preceding product iterations should be mostly met before the next iteration can begin.

The priority is defined globally as a **5-valued** system and the highest priority in one phase will not be allowed to appear in the next phase. This reflects the prerequisite idea for requirements.

Subcategory	Component
A	User accounts and Authentication
B	Matchmaking and queueing
C	Rooms management
D	Code workspace
E	Question generation and history
F	Text chat functionality
G	Code execution
H	Voice chat functionality

A1.1. Phase 1 - Minimum Viable Product (MVP)

This product iteration defines the functional requirements for the bare minimum system that satisfies the product vision and scope.

A1.1.1. User Accounts and Authentication

ID	Functional Requirement	Priority
FA-1	The system should allow users to register for an account.	Very High
FA-2	The system should allow users to log in to their accounts.	Very High
FA-3	The system should allow users to log out of their accounts.	Very High

A1.1.2. User Matchmaking

ID	Functional Requirement	Priority
FB-1	The system should allow users to join separate matchmaking queues for each difficulty level.	Very High
FB-2	The system should allow users to join multiple difficulty queues at the same time.	High
FB-3	The system should match users with the same difficulty level.	Very High
FB-4	The system should automatically kick users from the matchmaking queue if no match is found in 30 seconds.	High
FB-5	The system should allow users to leave the matchmaking queue before matchmaking is successful.	Medium
FB-6	The system should prevent users already in the queue from joining the matchmaking queue.	Very High

A1.1.3. Room Session

ID	Functional Requirement	Priority
FC-1	The system should create a room and redirect users into the room after successful matchmaking.	Very High
FC-2	The system should allow users to leave a room.	Very High
FC-3	The system should inform users that their partner has left the room.	Medium
FC-4	The system should handle the disconnection of users from a room.	Medium
FC-5	The system should automatically close a room when there are no users in the room.	Very High

A1.1.4. Code Workspace Collaboration

ID	Functional Requirement	Priority
FD-1	The system should show a common code workspace to all users in the room.	Very High
FD-2	The system should allow users to modify the contents of the code workspace concurrently.	Very High
FD-3	The system should synchronise the code workspace in real-time across all users in the room.	Very High

A1.1.5. Question Bank

ID	Functional Requirement	Priority
FE-1	The system should show a randomly generated question to all users in a room based on the selected question difficulty selected during the matchmaking process.	Very High

A1.2. Phase 2

This iteration focuses on improving the user experience of the code collaboration functionality through quality-of-life improvements to the system.

A1.2.1. Additional User Profile Operations

ID	Functional Requirement	Priority
FA-4	The system should allow users to reset their password.	Medium
FA-5	The system should allow users to change their passwords	Low
FA-6	The system should allow users to change their account information	Low

A1.2.2. Workspace Enhancements

ID	Functional Requirement	Priority
FD-4	The system should allow the users to select the coding language used in the code workspace.	High
FD-5	The system should automatically perform syntax highlighting in the code workspace based on the selected coding language.	High
FD-6	The system should provide automatic indentation formatting in the code workspace.	Low
FD-7	The system should allow users to download and save the contents of the code workspace to a file.	Medium
FD-8	The system should allow for the font size used in the code workspace to be configured by the user.	Low
FD-9	The system should allow commonly used keyboard shortcuts in the workspace.	Low

A1.2.3. Question Attempt History

ID	Functional Requirement	Priority
FE-2	The system should allow the users to mark a question as completed.	Medium
FE-3	The system should allow users to save their attempts for questions.	High
FE-4	The system should save a history of the questions completed for each user.	Medium
FE-5	The system should allow users to list all saved attempts along with the corresponding questions.	High
FE-6	The system should allow users to view the details of a particular saved attempt.	Medium

A1.2.4. Workspace Text Chat

ID	Functional Requirement	Priority
FF-1	The system should allow the users in the same room to send a text message in the code workspace.	Medium
FF-2	The system should allow all users in the same room to see all text messages sent to the room since the creation of the room.	Medium

A1.3. Phase 3

This group of functional requirements mainly focus on further improving user experience through minor tweaks and additions, without drastically changing how the code workspace looks like from phase 2.

A1.3.1. Question Filter

This FR was not achieved.

ID	Functional Requirement	Priority
FB-7	The system should allow the users to filter the question bank to only use questions that have not been marked completed. This filtered list is referred to as the matchmaking question pool.	Very Low
FB-8	The system should only match users with common questions in the matchmaking question pool.	Very Low
FE-7	The system should only select questions from the matchmaking question pool, if requested by the user.	Very Low

A1.3.2. Code Execution

ID	Functional Requirement	Priority
FG-1	The system should allow the users to execute the code that they have written.	Low
FG-2	The system should display the results of the code execution in the code workspace.	Low

A1.4. Additional Optional Functional Requirements

This is the final group of functional requirements that are considered “nice-to-have”, but are not strictly required in the product vision.

A1.4.1. Voice Chat

This FR was not achieved.

ID	Functional Requirement	Priority
FH-1	The system should allow the users in a room to join a voice communication channel.	Very Low
FH-2	The system should allow users to communicate over voice within a room.	Very Low

A1.4.2. Performance-Based Matchmaking

This FR was not achieved.

ID	Functional Requirement	Priority
FB-9	The system should compute a proficiency score for all users based on the questions completed.	Very Low
FB-10	The system should attempt to matchmake users with similar proficiency levels together.	Very Low
FB-11	The system should show the number of matchable users currently in the queued difficulty	Very Low

A2. Non-Functional Requirements (NFRs)

This section lists all the NFRs for the system. Unlike the 5-valued priority system used for FRs, NFRs will employ a **3-valued priority** system.

Subcategory	Aspect
IS	Installability
SE	Security
US	Usability
PP	Performance
AV	Availability
IT	Integrity
SS	Scalability, Robustness
MM	Modifiability

A2.1. Product-Centric NFRs

This section will detail NFRs that are directly related to the system. The prioritisation of NFRs follows the quality attributes defined.

A2.1.1. Installability

While installability is not a selected quality attribute, both relate to the usability of the system in that the user should be able to launch the web application easily and seamlessly.

ID	Functional Requirement	Priority
NIS-1	The web application should work on all major modern browsers, with guaranteed support for: ● Safari (and Apple Webkit based browsers) ● Google Chrome (and Chromium based browsers) ● Mozilla Firefox There will be no support for Internet Explorer 11 because it has officially reached end-of-life since 15 June 2022.	High
NIS-2	The web application should not require additional downloads, plugins or other dependencies before the user can launch it.	High

A2.1.2. Security

While security is a nice-to-have quality attribute, authentication is a key security aspect that should not be neglected as it has the potential to undermine the entire system.

ID	Functional Requirement	Priority
NSE-1	The communication channels between different microservices should be secure.	Medium
NSE-2	The system should only allow authenticated users to access the application, with the exception of registration and login.	High
NSE-3	The system should automatically log the user out if his session has expired.	High
NSE-4	The system should use a rotating session key to limit the time attackers can use information obtained.	Medium
NSE-5	The system should be resilient to Man-in-the-Middle (MITM) attacks between the user and web servers.	Medium
NSE-6	The system should perform input sanitization and rejection to prevent malicious payloads.	Medium
NSE-7	The system should prevent malicious code execution.	Low
NSE-8	The system should store the passwords securely in a hashed manner.	High

A2.1.3. Usability

ID	Functional Requirement	Priority
NUS-1	The system should guarantee support for desktop landscape interfaces.	High
NUS-2	The system should scale correctly on screen resolutions between 1280x720 (720p) and 2560x1440 (1440p).	High
NUS-3	The application should allow registered users to access the code workspace within 5 clicks.	High
NUS-4	A new user should be able to use the basic functionalities, specifically entering a session and using the code workspace, without external help.	High

A2.1.4. Performance

The requirements only cover a partial subset of services, specifically those that play a part in the core product for enabling the code workspace. Most of these performance aspects relate to usability, in that there should be no noticeable slowdowns for the end user.

ID	Functional Requirement	Priority
NPP-1	The system should be capable of supporting a minimum of 50 concurrent users without any noticeable performance degradation.	High
NPP-2	The user service API should support 40 Queries per Second (QPS).	Medium
NPP-3	The question service API should support 50 QPS.	Low
NPP-4	The matching service API should support 30 QPS.	Low
NPP-5	The collaboration service API should support 70 QPS.	Low
NPP-6	The session service API should support 200 Queries per Second (QPS).	Medium
NPP-7	The overall system should support 200 QPS.	Medium
NPP-8	HTTP APIs should not take more than 2 seconds.	Medium
NPP-9	The navigation between different client-side pages should take less than 5 seconds.	Medium

A2.2. Deployment-Centric NFRs

This section will detail NFRs that are related to how the system is architected and deployed.

A2.2.1. Availability

ID	Functional Requirement	Priority
NAV-1	The system should be designed in a way such that it can be expanded to support a High-Availability deployment.	Low

A2.2.2. Integrity

ID	Functional Requirement	Priority
NIT-1	There must be no operationally critical information stored in volatile data storages.	Medium
NIT-2	The matchmaking operation must be atomic across all instances running matchmaking-related code.	High
NIT-3	The code workspace should be consistent, or converge to a consistent state within 5 seconds, across all participating clients.	Medium

A2.2.3. Scalability

ID	Functional Requirement	Priority
NSS-1	The system should be designed in a way such that the capacity of microservices can be scaled up if required.	High
NSS-2	The system should be designed such that microservices can be scaled at a per-service level.	High
NSS-3	The system should support automatic horizontal scaling.	Low
NSS-4	The system should allow microservice instances to go down without major impact to the overall functionality of the system.	Medium
NSS-5	The system should minimise the number of single points of failure.	Low

A2.2.4. Modifiability

ID	Functional Requirement	Priority
NMM-1	The communication protocols used in the system should be standardised, such that it will be compatible with future addition of functionalities and services.	High
NMM-2	The system should have a modular design, such that services can be changed with minimal effects on other services.	High
NMM-3	The system should be componentized so that the individual codebases remain manageable in size (< 10,000 lines).	High
NMM-4	The system should minimise coupling by adhering to SOLID principles where possible.	High

Appendix B - Product Backlog

The priority notation used in this section is as follows.

- VH - Very High
- H - High
- M - Medium
- L - Low
- VL - Very Low

For sprints, the notation is generally W<week>, where W3 stands for week 3 and WR stands for recess week.
If a sprint is marked as **DROP**, then it was decided that the work item cannot be done within the timespan of the project.

F1	**User CRUD Service** **A module of the Node.JS based user service that handles the management of user objects. This is an internal service called by other microservices and the User Service Module.**	**Priority (Phase)**	**Sprint**
F1.1	**User Object Management**		
F1.1.1	The user CRUD service should connect to a persistent Postgres datastore for storing user information.	**VH (1)**	W4
F1.1.2	The user CRUD service should allow the creation, retrieval, and deletion of user objects in the database.	**VH (1)**	W4
F1.1.3	The user CRUD service should allow the updating of user objects in the database.	**H (1)**	W4
F1.2	**User Management APIs**		
F1.2.1	The user CRUD service should expose an API for the creation of user objects.	**VH (1)**	W4
F1.2.2	The user CRUD service should expose an API for the retrieval of user objects.	**VH (1)**	W4
F1.2.3	The user CRUD service should expose an API for the deletion of user objects.	**H (1)**	W4
F1.2.4	The user CRUD service should expose an API for the update of user objects.	**H (1)**	W4
F1.3	**Reset Token Management**		
F1.3.1	The user CRUD service should expose an API for the creation of reset tokens.	**M (2)**	W9
F1.3.2	The user CRUD service should expose an API for the deletion of reset tokens.	**M (2)**	W9
F1.3.3	The user CRUD service should expose an API for the retrieval of reset tokens.	**M (2)**	W9

F2	**User Service** **A module of the Node.JS based user service that aggregates user-related functionality. It handles authentication, user information fetching and session creation using downstream session, user, and history services.** **This is an UI-facing API.**	**Priority (Phase)**	**Sprint**
F2.1	**Login APIs**		
F2.1.1	The user service should expose an API for registering new users.	**VH (1)**	W4
F2.1.2	The user service should expose an API for logging in new users.	**VH (1)**	W4
F2.1.3	The user service should expose an API for logging out users.	**VH (1)**	W4
F2.1.4	The user service should expose an API for the forget password operation.	**M (2)**	W9
F2.1.5	The user service should expose an API for changing their account password.	**L (2)**	W10
F2.1.6	The user service should expose an API for changing their account nickname.	**L (2)**	W10
F2.2	**User Session Management**		
F2.2.1	The user service should create new JWT tokens for users logging in.	**VH (1)**	W4
F2.2.2	The user service should authenticate JWT tokens sent by the users (before being replaced by session management service).	**VH (1)**	W4
F2.2.3	The user service should blacklist JWT tokens for users logging out.	**VH (1)**	W4
F2.2.4	The user service should register blacklisted JWT tokens with the session management service.	**H (1)**	W6

F3	Session Management Service A Golang based microservice that handles session authentication, session refresh and session blacklisting.	Priority (Phase)	Sprint
F3.1	**Session Management APIs**		
F3.1.1	The session management service should expose an API for authenticating JWT tokens.	**VH (1)**	W6
F3.1.2	The session management service should expose an API for registering a JWT token to the blacklist.	**H (1)**	W6
F3.1.3	The session management service should expose an API for removing JWT tokens from the blacklist.	**L (1)**	W6
F3.1.4	The session management service should expose an API for refreshing JWT tokens.	**H (1)**	W6
F3.2	**Blacklist Management**		
F3.2.1	The session management service should store the JWT token blacklist on a Redis server.	**VH (1)**	W6
F3.2.2	The session management service should store blacklisted refresh tokens on a Redis server.	**H (1)**	W6
F3.3	**Refresh Token Management**		
F3.3.1	The session management service should check refresh tokens for validity.	**H (1)**	W6
F3.3.2	The session management service should generate new JWT tokens based on the refresh token.	**H (1)**	W6
F3.4	**Temporal Blacklist**		
F3.4.1	The session management service should blacklist session tokens before a particular timestamp	**M (2)**	W9
F3.4.2	The session management service should blacklist refresh tokens before a particular timestamp	**M (2)**	W9

F4	**User-Related UI** **A React-Based Component of the front-end that is responsible for user related operations.**	**Priority (Phase)**	**Sprint**
F4.1	**User Registration Page**		
F4.1.1	The UI should allow users to register for a new account.	**VH (1)**	W4
F4.1.2	The UI should alert the user if the username is already taken.	**VH (1)**	W4
F4.2	**User Login and Logout**		
F4.2.1	The UI should allow users to login to their account.	**VH (1)**	W4
F4.2.2	The UI should allow users to logout of their account.	**H (1)**	W4
F4.2.3	The UI should allow users to reset their password.	**M (2)**	W9
F4.3	**User Token Management**		
F4.3.1	The UI should store and maintain the user's session token in cookies.	**VH (1)**	W4
F4.3.2	The UI should remove the user's session token in cookies upon logout.	**VH (1)**	W4
F4.3.3	The UI should initiate the token refresh process if the session token is stale.	**M (1)**	WR
F4.4	**User Profile Management**		
F4.4.1	The UI should allow users to change their password.	**L (2)**	W10
F4.4.2	The UI should allow users to change their nickname.	**L (2)**	W10

F5	**Matching Service** **A Node.JS based microservice that handles the pair-matching of users based on selected difficulties. This is an UI-facing API.**	**Priority (Phase)**	**Sprint**
F5.1	**Difficulty-Based Queueing**		
F5.1.1	The service should connect to a group of Redis queues for cluster-wide atomic matching. Each difficulty should have its own logical queue.	**VH (1)**	W5
F5.1.2	The service should poll the existing Redis queue for any users waiting to be matched.	**VH (1)**	W5
F5.1.3	The service should push to the Redis queue if there are no matching users found.	**VH (1)**	W5
F5.1.4	Users should be dequeued from the matching queues once the 30-second timeout is reached.	**VH (1)**	W5
F5.1.5	Users already in the queue should not be allowed to join the queue.	**VH (1)**	W5
F5.2	**Matching API**		
F5.2.1	The service should expose an API for users to join the matchmaking queue.	**VH (1)**	W5
F5.2.2	The service should expose a long-polling API for users to check the queueing status.	**VH (1)**	W5
F5.2.3	The service should indicate to user polls that a match has been found.	**VH (1)**	W5
F5.2.4	The service should indicate to user polls that queueing timeout has been reached.	**VH (1)**	W5
F5.2.5	The service should allow the user to leave the matching queue before the timeout is reached.	**M (1)**	W10

F6	**Matching UI** **A React-Based Component of the front-end that is responsible for the queueing and matching process**	**Priority (Phase)**	**Sprint**
F6.1	**Difficulty Selection UI**		
F6.1.1	The UI should allow users to select the intended difficulty.	**VH (1)**	W5
F6.1.2	The UI should allow users to leave the queue.	**M (1)**	W4
F6.2	**Queueing UI**		
F6.2.1	The UI should show a timer counting down the 30-second timeout.	**VH (1)**	W5

F7	**Collaboration Service** **A Node.JS based microservice that handles the communication and synchronisation of users within a room.** **This is an UI-facing API.**	**Priority (Phase)**	**Sprint**
F7.1	**Code Workspace Synchronisation**		
F7.1.1	The service should expose a persistent API endpoint for bi-directional messaging.	**VH (1)**	W6
F7.1.2	The service should forward messages between users in the same room.	**VH (1)**	W6
F7.1.3	The service should maintain a tunnel between the users to synchronise the code workspace.	**VH (1)**	W6
F7.1.4	The service should monitor the connectivity of users and send a message to users in a room if there has been any disconnection.	**M (2)**	WR
F7.1.5	The service should automatically clean up rooms with 0 users.	**H (1)**	W6
F7.1.6	The service should allow users to submit a code snapshot.	**M (2)**	W8
F7.1.7	The service should proxy submission of code to the execution service.	**L (3)**	W11
F7.2	**Scalability Requirements**		
F7.2.1	The service should use a Pub-Sub system for sending and receiving messages, where the Pub-Sub is atomic across the cluster.	**H (1)**	WR

F8	**Question Service** **A Node.JS based microservice that handles the management of question objects.** **This is an internal service called by other microservices.**	**Priority (Phase)**	**Sprint**
F8.1	**Question Object Management**		
F8.1.1	The question service should connect to a persistent Postgres datastore for storing question information.	**VH (1)**	W6
F8.1.2	The question service should allow the creation, retrieval, and deletion of question objects in the database.	**VH (1)**	W6
F8.1.3	The question service should allow conditional retrieval of question objects from the database.	**VH (1)**	W6
F8.2	**Question Management APIs**		
F8.2.1	The question service should expose an API for the creation of question objects.	**VH (1)**	W6
F8.2.2	The question service should expose an API for the retrieval of question objects.	**VH (1)**	W6
F8.2.3	The question service should expose an API for the deletion of question objects.	**M (1)**	W6
F8.2.4	The question service should expose an API for the update of question objects.	**M (1)**	W6
F8.2.5	The question service should expose an API for the retrieval of a random question object given the difficulty value.	**H (1)**	W6
F8.2.6	The question service should expose an API for the retrieval of random question objects given a set of exclusions.	**VL (3)**	**DROP**

F9	**Collaboration UI** **A React-Based Component of the front-end that is responsible for the queueing and matching process.**	**Priority (Phase)**	**Sprint**
F9.1	**Room Session**		
F9.1.1	The UI should allow users to edit the content in the workspace.	**VH (1)**	W6
F9.1.2	The UI should automatically update the workspace changes by the other user.	**VH (1)**	W6
F9.1.3	The UI should alert the user once the other user has joined.	**H (2)**	W6
F9.1.4	The UI should alert the user if his partner left the room.	**M (2)**	WR
F9.1.5	The UI should allow the user to leave the room.	**VH (1)**	W6
F9.2	**Workspace UI**		
F9.2.1	The UI should allow the user to submit his code.	**H (2)**	W8
F9.2.2	The UI should allow the user to select the coding language.	**M (2)**	W7
F9.2.3	The UI should highlight the syntax for the user depending on the coding language.	**M (2)**	W6
F9.2.4	The UI should allow the user to select the font size in the workspace.	**L (2)**	W12
F9.2.5	The UI should allow commonly used keyboard shortcuts in the workspace.	**L (2)**	W6
F9.2.6	The UI should allow the user to download the contents of the code workspace to a file.	**M (2)**	W7

F9.2.7	The UI should provide automatic indentation formatting in the code workspace.	**L (2)**	W6
F9.3	**Question UI**		
F9.3.1	The UI should show the question number, title, and difficulty.	**VH (1)**	W6
F9.3.2	The UI should show the question content.	**VH (1)**	WR
F9.3.3	The UI should show the question example including the input, output and explanation.	**M (1)**	W6
F9.3.4	The UI should show the bounds of the input variable.	**M (1)**	W6
F9.3.5	The UI should allow the user to toggle the question as completed or not.	**M (2)**	W11
F9.3.6	The UI should show the user if they have completed the question before.	**M (2)**	W11
F9.3.7	The UI should show the solution to the user.	**M (2)**	W12
F9.4	**Synchronisation**		
F9.4.1	The system should automatically resolve workspace update conflicts based on the latest change.	**H (1)**	W6
F9.5	**Execution UI**		
F9.5.1	The UI should show the user if they passed all the test cases.	**L (3)**	**DROP**
F9.5.2	The UI should show the user the sample inputs and actual output.	**L (3)**	W11

F10	**Gateway** **A component that sits between the microservices and the user-facing API.** **It converts the HTTP/1 and websocket connections from users to the internally used fabric protocol.**	**Priority (Phase)**	**Sprint**
F10.1	**HTTP/1 to HTTP/2 Proxy**		
F10.1.1	The gateway should translate HTTP/1 requests to HTTP/2.	**VH (1)**	W4
F10.1.2	The gateway should translate HTTP/1 headers to HTTP/2 metadata.	**H (1)**	W6
F10.1.3	The gateway should translate HTTP/2 metadata to HTTP/1 headers.	**H (1)**	W6
F10.2	**Websocket to HTTP/2 Proxy**		
F10.2.1	The gateway should accept incoming websocket connections.	**VH (1)**	W6
F10.2.2	The gateway should proxy the websocket connections to downstream HTTP/2 streaming endpoints.	**VH (1)**	W6
F10.3	**Authentication**		
F10.3.1	The gateway should authenticate incoming requests using the session token.	**VH (1)**	W6
F10.4	**Static Serving**		
F10.4.1	The gateway should statically serve the frontend	**H (1)**	W4

F11	History Service A Node.JS based microservice that handles the management of history objects. This is an internal service called by other microservices.	Priority (Phase)	Sprint
F11.1	**History Object Management**		
F11.1.1	The history service should connect to a persistent Postgres datastore for storing history information.	**H (2)**	W8
F11.1.2	The history service should allow the creation, retrieval and deletion of history objects in the database.	**H (2)**	W8
F11.1.3	The history service should allow the creation and deletion of completion records in the database	**M (2)**	W11
F11.1.4	The history service should allow the updating of history objects in the database.	**M (2)**	**DROP**
F11.2	**History Management APIs**		
F11.2.1	The history service should expose an API for the creation of history objects.	**H (2)**	W8
F11.2.2	The history service should expose an API for the retrieval of history objects.	**H (2)**	W8
F11.2.3	The history service should expose an API for the deletion of history objects.	**H (2)**	W8
F11.2.4	The history service should expose an API for the update of history objects.	**L (2)**	**DROP**
F11.2.5	The history service should expose an API for the toggling of completion records.	**M (2)**	W11

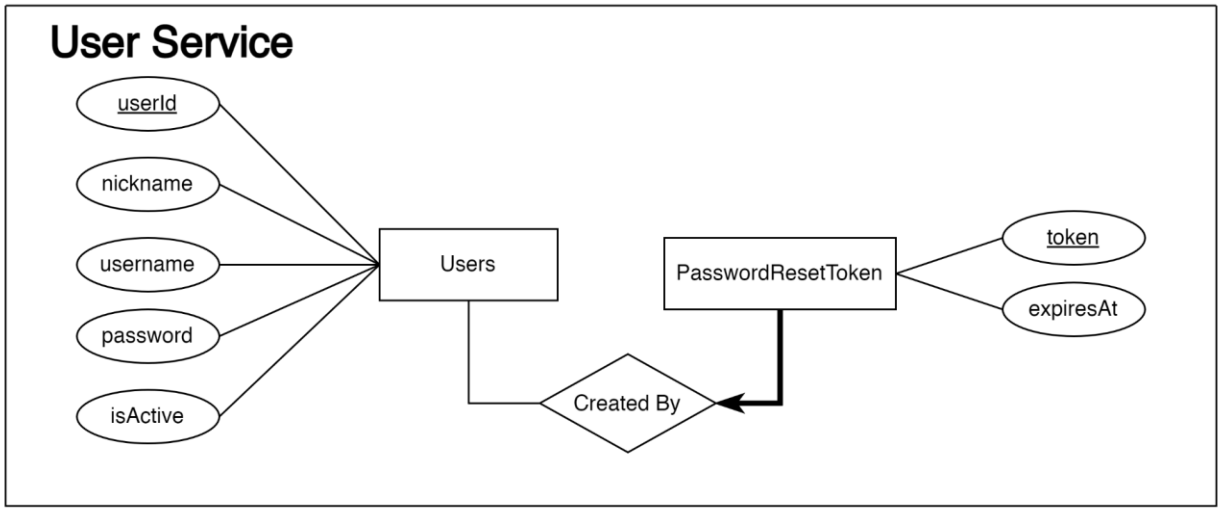
F12	**History UI** **A React-based component of the frontend that is responsible for showing the user his/her previous attempts**	**Priority (Phase)**	**Sprint**
F12.1	**User History UI**		
F12.1.1	The UI should show a summarised list of attempts that the user made.	**H (2)**	W8
F12.1.2	The UI should allow the user to sort the attempts by difficulty and date of attempt.	**L (2)**	**DROP**
F12.1.3	The UI should allow the user to view details of a particular attempt.	**H (2)**	W8
F12.1.4	The UI should allow the user to download their code attempt.	**M (2)**	W8

F13	**Text Chat UI** **A React-based component of the frontend that is responsible for showing the user his/her previous attempts**	**Priority (Phase)**	**Sprint**
F13.1	**Text Chat UI**		
F13.1.1	The UI should display a chat box for users to see messages sent in the room.	**M (2)**	W9
F13.1.2	The UI should allow users to send text messages to the chat box.	**M (2)**	W9
F13.1.3	The UI should automatically update the chat box upon receiving messages.	**M (2)**	W9

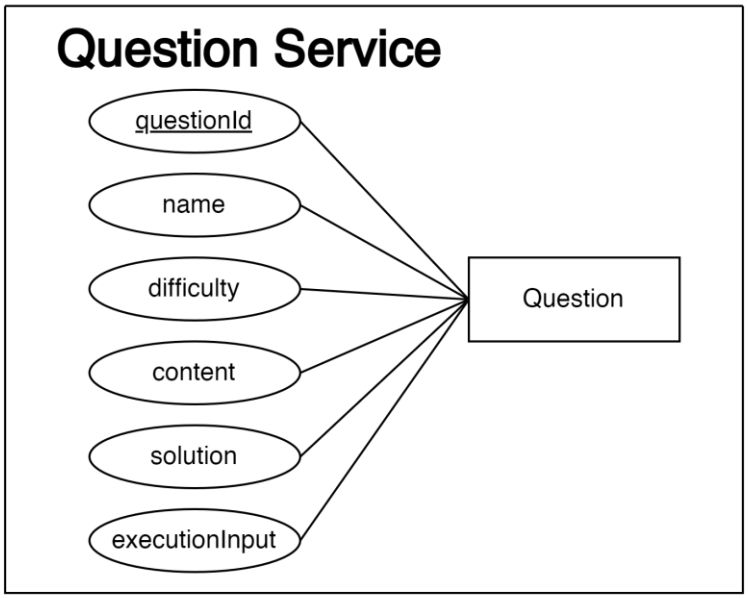
F14	**Execution Service** **A component that allows code to be executed on an execution service**	**Priority (Phase)**	**Sprint**
F14.1	**Execution Queue**		
F14.1.1	The execution service should connect to a queue to submit code execution jobs.	**L (3)**	W11
F14.1.2	The execution service should execute jobs submitted to the queue.	**L (3)**	W11
F14.1.3	The execution service should return the execution results of jobs.	**L (3)**	W11
F14.2	**Execution API**		
F14.2.1	The service should expose an API for code jobs to be queued for execution.	**L (3)**	W11
F14.2.2	The service should expose an API for querying the results of a job execution.	**L (3)**	W11

Appendix C - Database Schema

C1. User Service Schema



C2. Question Service Schema



C3. History Service Schema