PeerPrep

Code Repo URL:
https://github.com/CS3219-SE-Principles-and-Patterns/cs3219-project-ay2122-2122-s1-g9

CS3219 Software Engineering Principles and Patterns

AY21/22 Semester 1

Project Report

Group 9

Team Members	Student No.	Email
Tan Jun Han Ivan	A0196518R	e0389004@u.nus.edu
How Yu Hin Chester	A0196501H	e0388987@u.nus.edu
Arthur Lee Ying Kiu	A0182447Y	e0309242@u.nus.edu
Azeem Arshad Vasanwala	A0200395M	e0407376@u.nus.edu

Background and purpose of project

We have decided to do Option 2, PeerPrep. PeerPrep is a web application that aims to help students prepare for challenging technical interviews when applying for jobs. It does so by using a peer learning system, in which two students are matched with one another, and are presented with a technical question that they will attempt to solve together.

As PeerPrep is based on an existing project done in a previous semester, we aimed to differentiate ourselves by having a strong focus on usability. This means having features that close the loop in the user flow. Specifically, after discussing a specific leetcode question with your teammate, the natural action would be to try it out on leetcode itself. Therefore we introduced features such as copying the text editor code and a button that links to the specific leetcode question. This facilitates copying the code discussed and trying it out on the Leetcode question itself. Our focus on usability also led to an emphasis on having clear confirmation and recovery actions for the user.

Individual contributions to the project

Name	Technical Contributions	Non-Technical Contributions
Azeem	● Add user to difficulty queue ● Add Cloud Function to detect users in queue and create session ● Add Cloud Function to handle change question request ● Remove user from queue on timeout using Google Cloud Tasks ● Github Actions setup	Non-technical work was spread across the team. Every week the team would meet to determine the work for the week ahead, account for overflow and distribute both technical work and non-technical work such as report/documentation writing. Each person wrote the section in the report related to the feature they owned and implemented.
Arthur Lee	● Frontend work ○ Sign in flow ○ Chat functionality ○ Display question details with hints ○ Change question flow in the frontend ○ Top editor and bottom toolbar functionality, including copy, finish session ● Made figma mock used as reference for the application	
Ivan	● Setup backend repository and testing infrastructure ● Scrape Leetcode questions and write scripts to insert questions into Firestore ● Add Cloud Function to retrieve question ● Add Cloud Function to check if user is in current session ● Add basic tests for critical Cloud Functions ● Add lint and build CI	
Chester	● Frontend work ○ Configure linter and auto formatter ○ Configure Redux and client-side routing ○ Build real-time collaborative editor ○ Matching flow, from the home page to the queue page to the collaborate page ● Setup frontend deployment on Netlify	

Project Requirements

Functional Requirements

Legend: Higher priority levels are more important, i.e 1 is more important than 2, etc.
Priority 1: Must Have for product to meet minimum spec
Priority 2: Good to have to improve the product experience

As seen in our development plan, we prioritised functional requirements that are priority 1 first over priority 2. This is due to us feeling that we needed to meet the minimum spec first and also because most of the time, the priority 1 FRs block the priority 2 FRs.

Label	Description	Priority	Completion
F1. User Management			
F1.1	The application should allow users to register for an account via Google or Facebook.	1	☑
F1.2	The application should allow users to sign in to their account via Google or Facebook.	1	☑
F1.3	The application should allow users to sign out of their account.	1	☑
F2. Matching based on difficulty			
F2.1	The application should match users based on the difficulty level selected.	1	☑
F2.2	The application should allow the user to select the difficulty level before matching.	1	☑
F2.3	The application should keep the user in the queue for at most 30 seconds before informing the user that it is unable to find a match.	1	☑
F2.4	The application should show 2 matched users the same question of the desired difficulty after matching them	1	☑
F3. Real-time collaboration			
F3.1	The application should allow 2 users to concurrently write text on the text editor.	1	☑
F3.2	The application should allow a user to chat with the other user on the same space.	1	☑
F3.3	The application should allow any of the 2 users to end the session.	1	☑
F3.4	The application should allow any of the 2 users to change the question they are attempting.	2	☑

F4. Sharing discussion notes			
F4.1	The application's text editor should allow any of the 2 users to go to the corresponding Leetcode question link.	1	☑
F4.2	The application's text editor should allow any of the 2 users to copy its text.	2	☑
F4.3	The application should keep a record of the text editor contents a session had for both users.	2	☒
F4.4	The application should allow users to access the text editor contents from their previous sessions as well as the question they attempted	2	☒

Some of the priority 2 FRs were not developed due to time constraints.

Non-Functional Requirements

Our application aims to help the user practice for technical interviews. The interface of the app shall act as a backdrop for the user so that he can forget about the UI and focus on the task at hand. This means that the UI should not be too jarring such that it will distract the user from the main task. The app should also provide a seamless experience during operation so that learning/collaborating could be more effective. A seamless experience encompasses the point about UI above and that the usage of the app should generate as little friction for the user as possible. This means making the app smooth and reducing latency for synchronising editor and chat windows. For instance, it would not be good if the app takes a long time to show a user's code to his partner. Moreover, some users may not be able to accept the latency and will stop using the app. In a nutshell, the crafted NFRs will reflect the focus on developing a low-friction and seamless platform for preparing technical interviews.

Performance requirements

For performance requirements, we will mainly focus on the responsiveness of the app and the time required to synchronize editor and chat windows. This is because those points are crucial to the user experience of the app. Listed below are the requirements addressing those needs.

Label	Description	Completion
N1. Performance		
N1.1	System should respond to users' input in under a second.	☑
N1.2	From N1.1, visual feedback for users' interaction should be provided in under a second of receiving user's input. So that the users will know that their interactions are registered.	☑
N1.3	System should authenticate users in under 3 seconds.	☑
N1.4	System should perform real time updates for chat in under 1 second.	☑
N1.5	System should perform real time updates to the text editor in under 1 second.	☑
N1.6	System should perform real time updates to the cursors of both users in under 1 second.	☑
N1.7	System should perform transitions of pages (e.g. switching of login screen to home screen) in under 2 seconds.	☑
N1.8	System should complete API calls for fetching data in under 1 second.	☒
N1.9	From N1.8, fetched data should be re-rendered on the user's display in under 1 second.	☑

Security requirements

The below security requirements were chosen because they represent the best practices of mainstream web applications.

Label	Description	Completion
N2. Security		
N2.1	System should persist user sessions for 24 hours before they have to re-login.	☑
N2.2	System should remove user information from the browser on logout if applicable (e.g. cookies containing user information should be cleared if the user logs out).	☑
N2.3	System should ensure that sessions cannot be joined by anyone beyond the 2 authenticated users involved.	☑

Software Quality attributes

Usability

The below usability requirements were chosen because they support our focus on providing a seamless user experience on the platform. Users should not be aware of the UI when using the platform so that they can have better focus on preparing for technical interviews.

Label	Description	Completion
N3. Usability		
N3.1	System should render on screen widths starting at 1280px onwards.	☑
N3.2	System should keep the user in the queue for at most 30 seconds before informing the user that it is unable to find a match.	☑
N3.3	System should use a readable font, namely the sans-serif font Rubik	☑
N3.4	System should use a font size large enough so that users can easily read it.	☑
N3.5	System should use a font colour that has a high contrast with the background of the page.	☑
N3.6	System should show a confirmation prompt whenever the user performs a destructive action (e.g. leaving a session).	☑
N3.7	System should show clear visual indicators of the severity of the actions that the user will perform.	☑

N3.8	The system's user interface should feel unified by using the same design system, resulting in consistency in terms of color choices and typography.	☑

Portability

The portability requirements below were chosen because of our focus on providing a low-friction and seamless experience for our users. To allow users to easily access the platform, we decided to develop our app for the web. This means that regardless of the operating system that the user is on, they should be able to access our platform as long as they have access to a modern web browser. Compared to a desktop app, a web app would have a wider audience as the app does not have to be installed on the computer to be used. It would also be easier for the development team as they can work on one codebase instead of multiple ones for different OSes (provided they are using the native language of the OS).

Label	Description	Completion
N4 Portability		
N4.1	System should work on Google Chrome Version 93 and above.	☑
N4.2	System should work on Firefox Version 91 and above.	☑
N4.3	System should work on Safari Version 14 and above.	☑

Availability

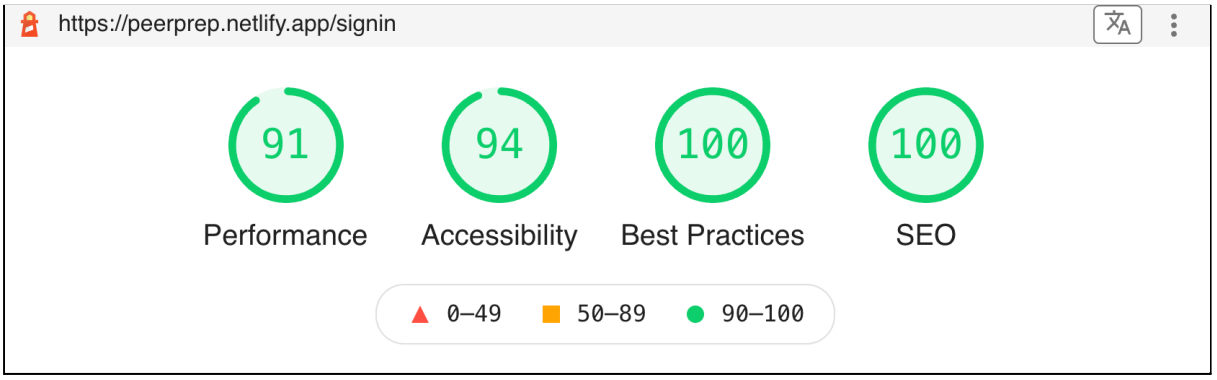
The below availability requirements were chosen because of our focus on providing a low-friction and seamless experience for our users. A low-friction app should allow users to practise for interviews at any time they want.

Label	Description	Completion
N5. Availability		
N5.1	System should be accessible (i.e. no down time) for 99.9% of the time.	☑

Meeting the NFRs

Many of our NFRs were met by nature of the SLA provided by the service. For example, Firebase guarantees 99.95% uptime, which our system inherits by virtue of using their platform. Similarly, Netlify guarantees a 99.99% uptime for the paid tier, but we did not utilise that as we are not paying for Netlify's service.

To measure accessibility and usability, we audited the web application using Lighthouse which gave the system scores above 90 for performance and accessibility. Our usability NFRs were met through conscious design choices to provide confirmation prompts and other such UI elements.

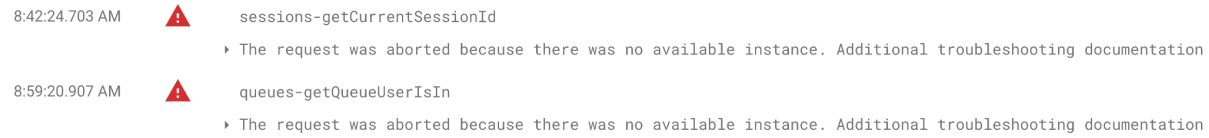


Functions are stateless and their execution environment has to be initialised from the beginning once triggered. This is known as a cold start. If a Cloud Function is constantly called, Firebase will cache the execution environment leading to better response times for subsequent requests.

Some of our performance NFRs were not hit because of cold starts. Another reason is our functions were deployed in the US region. Our APIs experienced a higher response time due to the geographical distance needed to travel. However, once the function had warmed up and was cached, the response times did drop as can be seen in the second call to sessions-updateAndGetWriter.

Name	Status	Protocol	Type	Initiator	Size	Time
verifyAssertion?key=AIzaSyCb5oCWPkECCcsJB...	200	h3	xhr	auth.esm.js:221	2.4 kB	595 ms
getAccountInfo?key=AIzaSyCb5oCWPkECCcsJB...	200	h3	xhr	auth.esm.js:221	465 B	203 ms
queues-getQueueUserIsIn	200	h3	fetch	service.ts:210	76 B	3.81 s
sessions-getCurrentSessionId	200	h3	fetch	service.ts:210	72 B	3.60 s
queues-addUserToQueue	200	h3	fetch	service.ts:210	72 B	2.01 s
questions-getQuestion	200	h3	fetch	service.ts:210	1.6 kB	3.61 s
sessions-updateAndGetWriter	200	h3	fetch	service.ts:210	100 B	3.18 s
sessions-updateAndGetWriter	200	h3	fetch	service.ts:210	100 B	521 ms

We tried to shift the location of Cloud Functions to Singapore. Logically, it should improve the response time since it is nearer to us. However, Cloud Functions deployed in Singapore have weird behaviours and even longer cold start times than the ones deployed in the US. An example of weird behaviour is shown in the screenshot below. Firebase was not able to run the called Function at all.



We ended up shifting the Cloud Function back to the US. We did not experiment further with other regions as it took quite a bit of effort to change the functions' location.

Development Process

The overarching development process of the project followed a waterfall approach. Requirements gathering was done in the first half of the semester before the recess week, including an outline of all the features we aimed to deliver as well as some stretch goals. The general architecture of the solution was also planned out at this stage and we sought approval from the teaching team to implement our solution.

Timeline

Once the project was approved, the development was broken down into 5 week-long sprints from weeks 7 to 11. We also provisioned a one week buffer in week 12. The entire development aimed to finish by the end of week 11. Week 12 was allocated for testing and documentation, but with the option to compress this timeline if needed. A copy of the development plan is provided in the Development Plan section.

Project Management

The team set aside an hour every Monday from 11am to 12pm to meet and take stock of the work that was done the previous week and what needed to be done next. We followed the schedule laid out in our intermediate report and stuck to it for the most part, with 1-2 day spillovers occurring once or twice.

DevOps

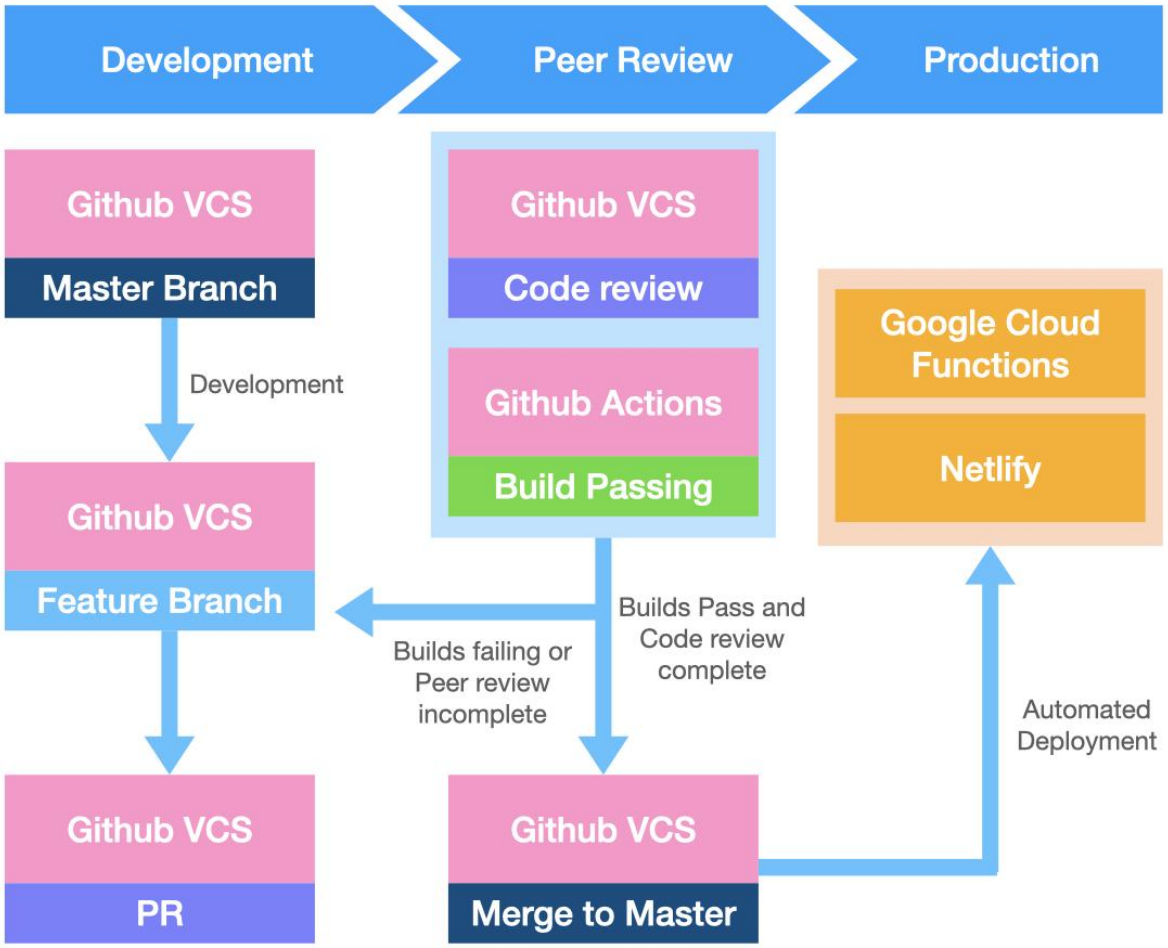
We made use of continuous deployment to assist in the development process. As all our services were on Google's Cloud Platform (GCP) and we had a decoupled front-end, it was imperative that our serverless functions could quickly and easily be deployed so that the front-end's development process was not obstructed. This will enable quicker development for the frontend since they do not need to concern themselves with the details of emulation of functions on their local environments. Thus we needed a solution that allowed for quick deployment of our cloud functions once development was complete. Similarly for the front-end, we needed a quick and easy way to update the production website when PRs were merged to master. To solve this issue, we made use of CD with GitHub Actions.

We made use of GitHub Actions to deploy our serverless functions to Firebase when our PRs were merged. The actions additionally ran unit tests and blocked the deployment if any of the tests failed. This allowed us to catch issues that cropped up during the introduction of new features that may have inadvertently broken existing ones.

Similarly, we made use of GitHub Actions to automatically deploy the front-end project. The CD pipeline generated a preview version of our front-end when a PR was created or updated, and deployed on Netlify when a PR was merged to the master branch.

The CD pipeline made it easy to ensure the cloud resources were in sync with our code, and allowed for quick testing of changes made by the team on both the front and backend.

An image detailing our process can be found below on the next page.



Development Plan

		W7	W8	W9	W10	W11	W12
	F1. User Management						
F1.1	The application should allow users to register for an account via Google or Facebook.						
F1.2	The application should allow users to sign in to their account via Google or Facebook.						
F1.3	The application should allow users to sign out of their account.						
	F2. Matching based on difficulty						
F2.1	The application should match users based on the difficulty level selected.						
F2.2	The application should allow the user to select the difficulty level before matching.						
F2.3	The application should keep the user in the queue for at most 30 seconds before informing the user that it is unable to find a match.						
F2.4	The application should show 2 matched users the same question of the desired difficulty after matching them						
	F3. Real-time collaboration						
F3.1	The application should allow 2 users to concurrently write text on the text editor.						
F3.2	The application should allow a user to chat with the other user on the same space.						
F3.3	The application should allow any of the 2 users to end the session.						
F3.4	The application must allow any of the 2 users to change the question they are attempting.						
	F4. Sharing discussion notes						
F4.1	The application's text editor must allow any of the 2 users to go to the corresponding Leetcode question link.						
F4.2	The application's text editor must allow any of the 2 users to copy its text.						
F4.3	The application should keep a record of the text editor contents a session had for both users.						
F4.4	The application should allow users to access the text editor contents from their previous sessions as well as the question they attempted						
	Devops						
	Provision Firebase Firestore						
	Provision Firebase Real-time Database						
	Provision API Gateway						
	Setup Continuous Deployment for frontend and lambdas						
	Setup staging and production env						

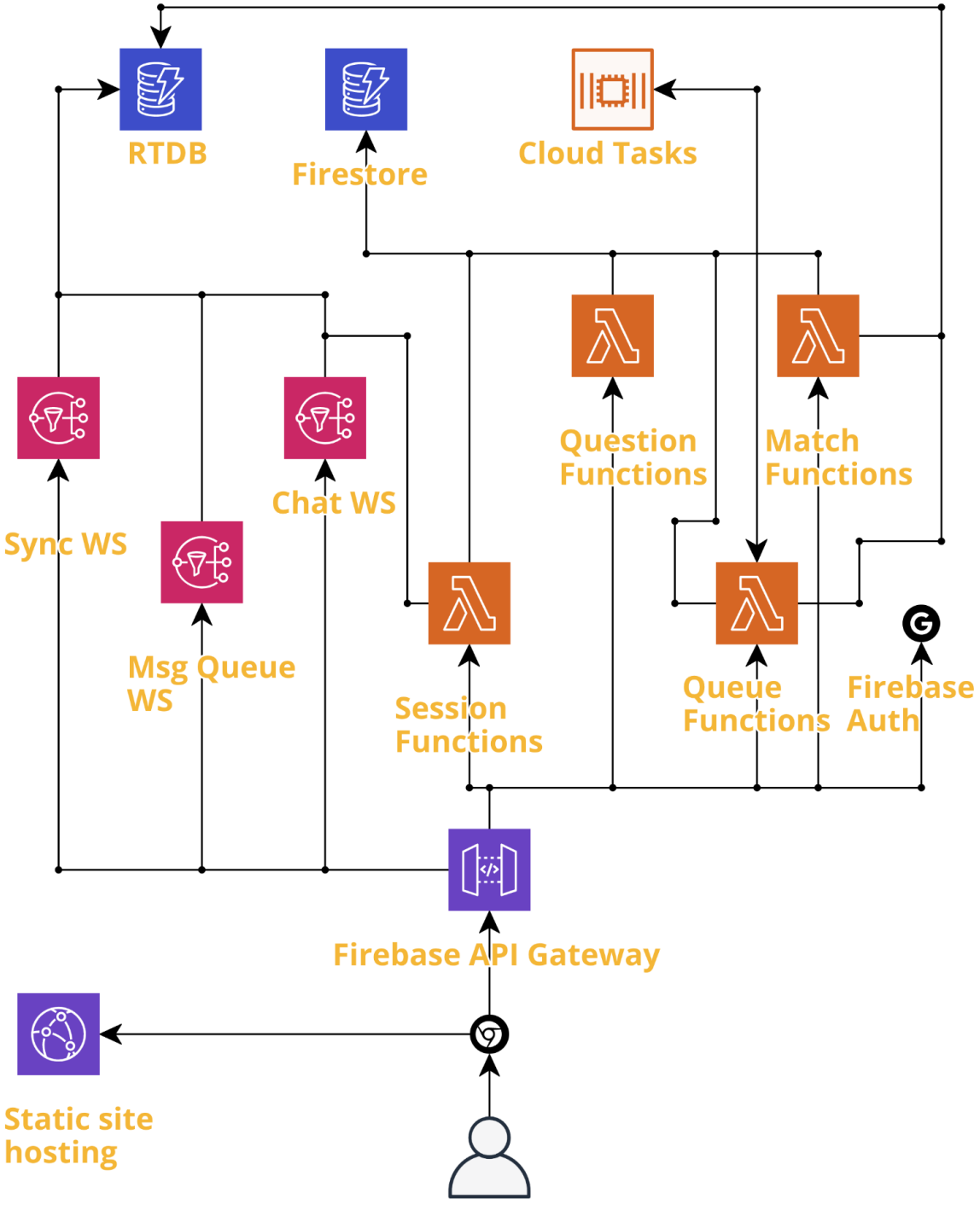
Developer Documentation

Tech stack

Components	**Technologies**
Backend	Firebase Cloud Functions (Serverless Architecture)
Frontend	React SPA
Database	Firebase Firestore (noSQL)
Pub/Sub	Firebase Realtime Database (noSQL)
Scheduler	Google Cloud Tasks
CI/CD	Github Actions

Architecture

This section details the architecture employed by the project and how each portion of the system works. To understand the rationale behind these choices, please see [Developer Documentation]



Note: While this diagram was created using AWS interfaces, the project is deployed on Firebase with identical infrastructural offerings.
Legend: WS: WebSocket

Frontend

The Frontend will be a Single Page Application (SPA) developed with React, a Javascript library for building user interfaces. The Frontend will interact directly with the RTDB websockets for better synchronisation speed. The Frontend will retrieve data from the Firestore through the Cloud Functions. This creates a Facade layer between the Frontend and data storage, making it easier for us to change the data format if needed.

Data Handlers

Each handler represents a set of Cloud Functions to interact with the related data. For instance, the Question Handler will manage Cloud Functions that deal with questions.

There are two main types of Cloud Functions. The first type will only be run in response to actions from Firebase services. These are called background functions. The second type is more common and can be directly triggered through HTTP requests. They are the ones mostly used in the backend. Both types of Cloud Functions will live together in their respective handlers. Hypothetically, we should be able to perform any operations that a normal backend will be able to do with Cloud Functions.

We detail the handlers used in the backend below.

Question Handler

Questions are scraped from Leetcode, sorted by difficulty levels and stored into the database. This will reduce the time needed for question retrieval. The question handler then returns the stored questions based on the selected difficulty and displays them to the user.

Queue Handler

The queue handler will write users looking to be matched to the real-time NoSQL database under their desired difficulty level. These difficulty levels can be thought of as topics in a message queue.

Match Handler

The match handler will have a background Cloud Function monitoring all the queues for the 3 difficulty levels. When users are written to a specific difficulty level, it will be triggered. The Cloud Function will then check if there are 2 such users who mutually are in the same queue, and if there are, it will create a session for them and remove them from the RTDB.

Session Handler

The session handler will allow the storage and retrieval of session related information. The frontend will also use this handler to retrieve information about ongoing sessions.

Databases

Realtime Database (RTDB)

The RTDB is mainly used for its pub/sub capabilities. RTDB stores data as a single giant JSON tree. Clients can then choose to observe just a particular section of the JSON tree.

Firestore

Firestore will store all other information that is required to run the application but is not suitable to be stored in the RTDB. Firestore works similarly to NoSQL databases like MongoDb. The two main concepts in Firestore are collection and documents. You can only store collections at the root level. A collection can contain multiple documents. A document can contain complex nested objects and subcollections. For Firestore, we can only retrieve documents so special attention needs to be given on the structure of data inserted into the Firestore.

Subscriptions

Session

The frontend will create a subscription to the Firebase realtime database (RTDB) for synchronization of session state. The session state consists of chat messages and text editor state. The pub/sub pattern will allow updates to the realtime databases to be synchronized across clients who subscribed to it. The subscriptions will be keyed by session identifier so the frontend only receives updates for the information it needs.

Message Queue

The frontend will create a subscription to the RTDB to receive command messages from the backend. The path to subscribe to will be keyed by user ID so each user will have his own message queue. These command messages will tell the frontend what to do next, for example, to move the user to the editor page after a session has been found.

Message Format

All the messages have a consistent format to make it simple for the frontend to parse. The structure of a message is:

Attribute	Description
type	Tell frontend what to do with that message
data	Contain the necessary information for frontend to process the message
createdAt	A timestamp for when the message was created

Cloud Tasks

The user needs to be removed from the queue and informed if there is not a match after 30 seconds. To implement the feature, we create a new Cloud Task when a user joins the queue. The Cloud Task runs after 30 seconds and will trigger a Cloud Function to check if the user is still in queue. If the user is still there, the user will be removed and a message will be sent to notify the frontend.

Routing and Access Control

API Gateway

Firebase has its own implementation of API Gateway to direct functions calls from the frontend to the correct Cloud Function in Firebase. It is essentially a mapping of Cloud Function names to the deployed Cloud Function. This will provide a uniform routing mechanism for services to interact with the Cloud Functions. The gateway will also pass authentication tokens into the Cloud Functions for further verification and usage.

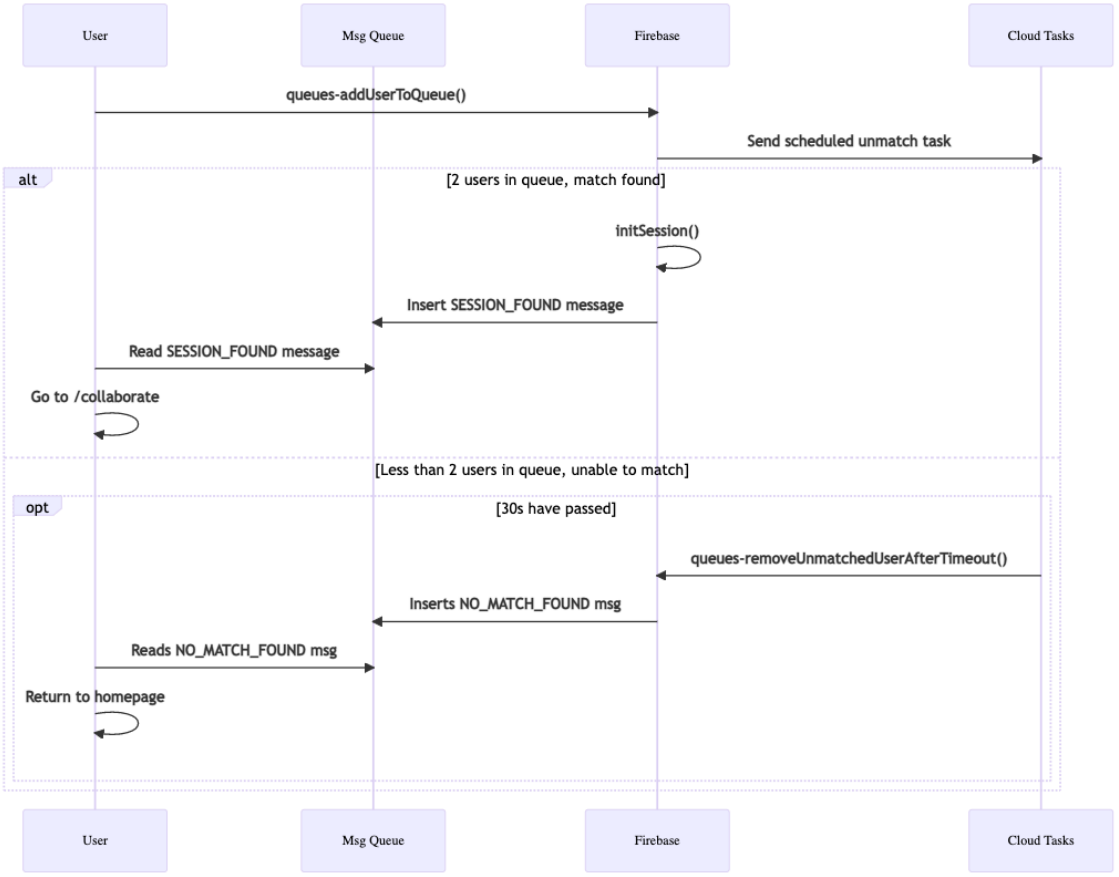
Authentication

Firebase Authentication will be used as the aggregated authentication provider, acting as the bridge between our application and other OAuth providers such as Google, Facebook, etc. It also helps us to store and retrieve user-related information.

Sample Interactions

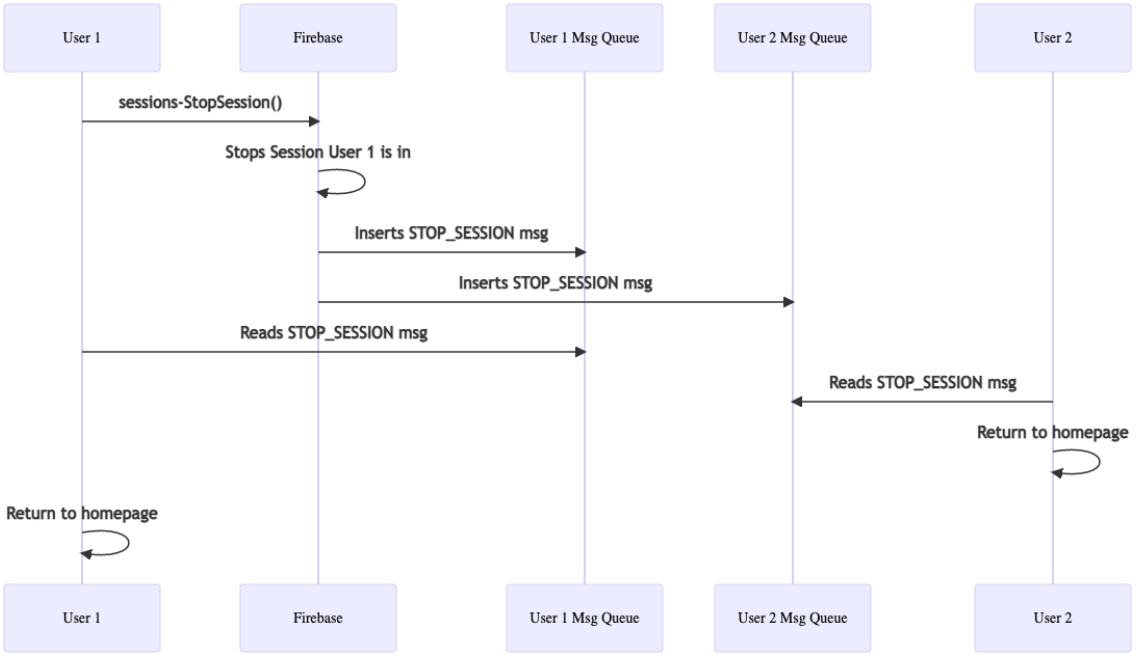
This section details some sample interactions within the system. They are provided so that the reader can have a feel of the general interactions between the frontend and the backend. We have provided links to full resolution images within the repository in case the images are too small to be viewed.

User joins the queue



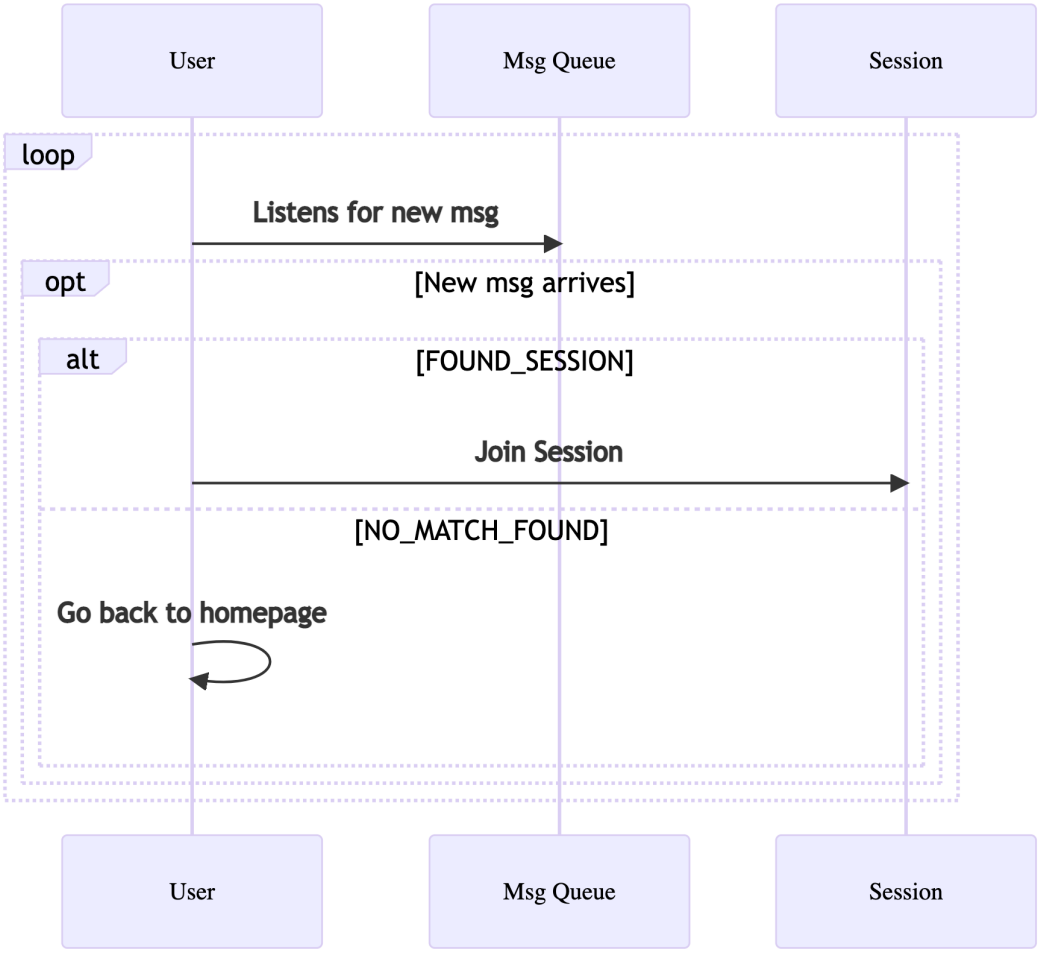
Reference file: documentation/images/user-join-queue.png

User stops the session



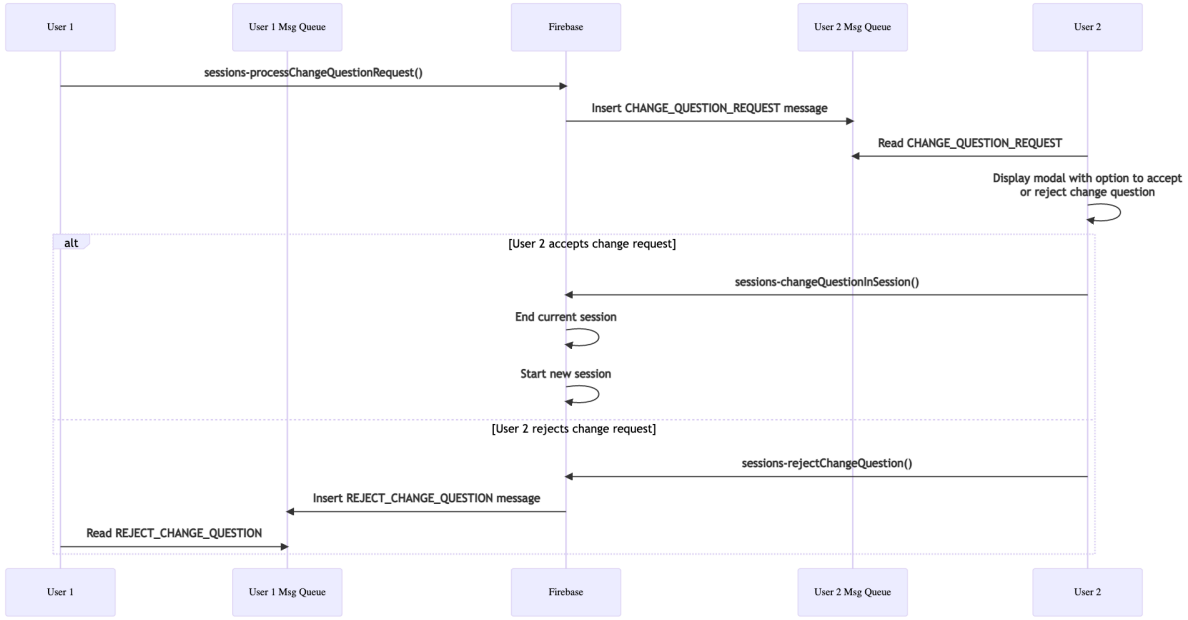
Reference file: documentation/images/user-stop-session.png

User listens to message queue



Reference file: documentation/images/user-listen-msg-queue.png

User changes the session



Reference file: documentation/images/user-request-change-qn.png

**Note that the "end current session" and "start new session" flows are defined under "User stops the session" and "User joins the queue > match found". We are unable to use reference frames to depict this as the service we are using for the diagrams lacks this capability*

Keep in mind that the sequence diagram above does not show all the possible message types in the application.

Project Design Decisions & Patterns Employed

Questions Ingestion

Background

The questions in PeerPrep are Leetcode questions. We wanted to scrape and insert the questions into the database in an automated manner so that we could easily do it for a large number of questions. Rather than seeding just a few questions, we recognised the importance of providing a large number of questions as the questions are chosen randomly for each difficulty level, as we did not want users to get repeated questions after using the app just a few times. We looked into scraping Leetcode questions but that was difficult as most of the content is rendered using Javascript. The content has to be first fetched from Leetcode's backend server before it is displayed on the site. This delay makes it difficult for typical scraping setup to work as most tools purely scrape the HTML file initially fetched from the site. More sophisticated solutions like Selenium have to be used to emulate the browser environment and account for the delay before starting the scrapping.

Since we know that the content of Leetcode questions are fetched from Leetcode's backend server, we looked into interfacing with those APIs ourselves. We found a repo on Github that does exactly that. Leetcode-cli brings Leetcode into the command line. It allows various interactions with Leetcode on the CLI through Leetcode's API. The CLI tool uses your Leetcode credentials, performs authentication against Leetcode's auth server and uses the returned tokens for future API requests.

Process

The repo has not been updated for quite a while so we had to upgrade the dependencies to make it usable. There is one particular feature of this CLI tool which makes it helpful to retrieve Leetcode questions in our use case: the tool caches the retrieved questions as JSON files. We know that questions are identified using an auto-incrementing ID. This was deduced by inspecting the responses from the API through the browser's network tab. We also know the maximum number of questions in Leetcode by exploring the site using the web UI. We can then write a script to use the CLI tool to retrieve each individual question using the question ID. We now have a set of approximately 2300 JSON files, each containing the contents of a single Leetcode question.

We also noticed that Leetcode uses GraphQL for its APIs. The CLI tool only retrieves a small set of question data that it needs. We wanted our UI to have similar information to Leetcode as our users are quite familiar with the platform. Hence, we needed to collect the same set of data that is used to render the questions on Leetcode's frontend. Luckily, Leetcode does not use multiple APIs to provide different amounts of question information. The same API question endpoint is used in the CLI tool and on Leetcode's frontend. To get the same information as Leetcode's frontend, all we had to do was ensure that the same GraphQL types used in the leetcode-cli's API request are the same as the ones on Leetcode's frontend. Since we have access to the code of leetcode-cli, we can make the changes ourselves.

Insertion of data to Firestore was trivial. We wrote another script to iterate over all these files and create a document on Firebase for each question. The scripts are stored in the qns-ingestion folder in our project repository.

Frontend

Real-time Collaborative Editor

One of the key features of the application was the real-time collaborative editor. This was actually two features in one, which involves a code editor as well as real-time collaboration support. Developing both from scratch, while not impossible, was unrealistic given our project requirements and timeline.

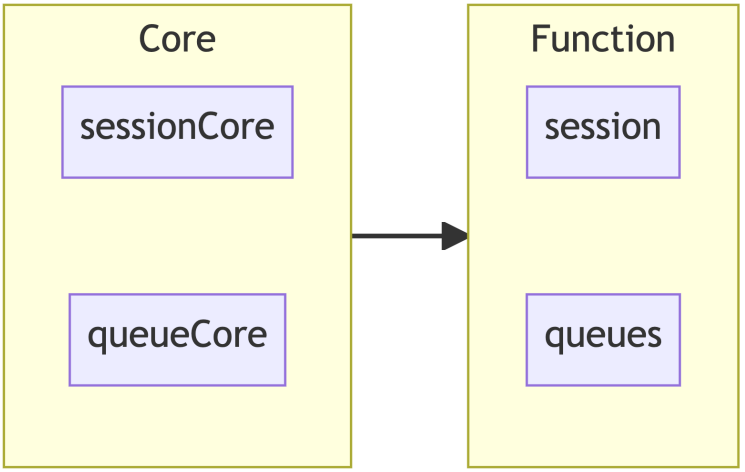
Thus we decided to utilise some of the existing libraries in place to tackle these features. For the code editor, we adopted Monaco Editor, which was a library maintained by Microsoft and is also the code editor used in VS Code. It provides syntax highlighting and supports the same keyboard shortcuts as VS Code. This choice aligned well with our usability requirement as our editor now feels and works just like VS Code, which many are already familiar with. Hence, there should be little to no learning curve and the editor is immediately usable.

For the real-time collaboration component, there are two popular algorithms that are widely used, Operational Transform (OT) and Conflict-free Replicated Data Type (CRDT). However, the team was not familiar with either of these, and we knew that if we naively relied on simple “on change” events to update the source, we would very easily run into conflicts. Likewise, we decided against implementing our own algorithm as it was not feasible considering the scope and timeline of the project. We then turned to Firepad, a collaborative code editor as it worked with Firebase Real-time Database, which was what we were using on the backend to store our editor’s contents. Unfortunately, we ran into a few problems. Firstly the Firepad library was no longer maintained and has not been updated in a while. Secondly, Firepad did not support Monaco, the code editor we had adopted earlier.

Thankfully, the team at Hackerrank did a rewrite of the Firepad library, and their version of Firepad supports the Monaco editor. This new library uses OT for real-time collaboration and implemented the Adapter Pattern in their codebase so as to allow for future integration between other databases and editors. We adopted this library, integrated it with our editor and our application now provides an excellent real-time collaborative editing experience.

Backend

Software Architecture



Code Structure

The main bulk of our backend consists of code for Cloud Functions. We decided to divide the code into two main layers, Core and Functions. The Core layer will contain the main logic and the Functions layer will wrap these core logic and expose them to the frontend. The reason for this arrangement is that it was non-trivial to call other Cloud Functions from a Cloud Function itself. Google does not have built-in support for that. The Functions layer will be as thin as possible to maximise code reuse for the Core layer. Modules in the Core layer are freely able to interact with each other.

Due to how Cloud Function names are given by Firebase, this unique code structure allows us to namespace our functions with the module they are associated with. For example, for the string “queues-addUserToQueue”, the function has the namespace “queues” and the name of “addUserToQueue”. Callers will have to specify the full namespace and function name, “queues-addUserToQueue” to call the correct function in Firebase.

Architecture Styles

The codebase relies on the Shared Repository architecture style heavily. For our case, the repository would be the Firestore and the RTDB. This means that all modules in the Core layer will have access to the databases. We felt that there is not really a point in adding another abstraction layer over the databases since the Firebase APIs to access these databases are already an abstraction layer. Moreover, these Firebase APIs do not change much, reducing the need for a facade layer over the databases.

Cloud Functions inherently uses the Implicit Invocation architecture style. For instance, when an edit is made to the data under a specific RTDB key, Firebase will trigger the Cloud Function registered to execute.

Session Writer Election

Due to the collaborative editor library we are using, the frontend needs to perform the writing of the default code for the programming questions into the RTDB. Please refer here for more details on the libraries used. Since we have two possible writers for each session, the backend will need to make a decision on who to write during the creation of the session. On every programming language (PL) change, the two frontends will call a Cloud Function to check if they are the session writer. If the original session writer is still online, he will remain as the session writer. Otherwise, the new session writer will be the remaining online user. The implementation of this feature will allow default code to be written even if the original chosen writer leaves the session. To implement this, we are utilised RTDB's presence feature found here to know if someone is online.

The alternative approach is to have Firestore detect any changes in the state of any user and trigger a Cloud Function when such a state change takes place. If the user goes offline and he is currently in a session, his session partner will be elected as the writer. While this method will remove the need for a Cloud Function call before every write, it will increase the Cloud Function usage very quickly as any change to any users' states will trigger the writer election Cloud Function. The method described in the first paragraph avoids this issue.

We went with that method because it is more scalable. Furthermore, since regular users will not keep changing the programming language, we accepted the time trade off, i.e the default code may take a few seconds to show up.

Design choice: Firebase as Backend

We chose Firebase as the backend for our application. Compared to other typical monolithic or micro-services backend, Firebase has everything that we need, such as pub-sub messaging, a real-time and NoSQL database, and the ability to separate functionality into smaller modules, similar to microservices.

On the other hand, for other backends, we need to manually implement the pub-sub messaging system needed for the collaborative editor. Furthermore, we must implement an API backend to store and retrieve data needed to run the application. The backend will also have to be connected to a database. Firebase acts as a pub-sub messaging system, API backend and database for us. This reduces the need to implement a lot of things on our own.

Beyond functionality, another key concern is dealing with scalability. A typical backend will consist of multiple services, e.g. a monolithic backend hosting an API will at least have an API layer and a database. We will need an orchestration tool like docker-compose to run and manage these services on a cloud provider. If we used a microservices architecture, we may require container orchestration tools such as Kubernetes to manage the containers. On the other hand, no deployment is required for Firebase databases and its pub/sub system. We only have to deploy the cloud functions and they can be done with a single command. This will reduce the time and effort needed for deployment configuration and management. In addition, Google will handle the scalability of the functions as load increases, as a new instance is spun up and killed per request. This reduces the need to worry about performance under load.

Lastly, Firebase provides tight integration between its services. For example, background Cloud Functions could be triggered when there is new data written to the RTDB. Similarly, user credentials generated through Firebase Authentication will automatically be passed into Cloud Functions if the frontend has the user logged in. Cloud Functions can then perform operations with that user without the developer doing additional work integrating Authentication and Cloud Functions together. This seamless integration between Firebase services makes Firebase a great choice to create our backend on.

In summary, the table below summarises how we evaluated Firebase against other options

Criteria	**Firebase**	**Monolith**	**Microservices**
Presence of real time database	Built in presence, immediate access via Firebase SDK	Developer needs to manually configure and deploy a technology that has similar functionality to RTDB.	
Presence of NoSQL database	Immediate access via Firebase SDK	Developer needs to manually configure and deploy a database instance (e.g. MongoDB).	
Pub-sub messaging	Realtime database can be used	Developer needs to manually configure and deploy a technology that has pub-sub messaging technology (e.g. Kafka).	
API Layer	Functions serve as API endpoints, they're plug and play Typescript functions	May require setting up a framework like Spring or Django and configuring them	
Scalability	Cloud functions automatically scale with load	Will require containerisation to manage all the 3rd party dependencies (DB, MQ, etc). Need to explicitly plan for data models to allow sharding.	Will require containerisation and container orchestration tools to manage instances and load balancers to route traffic
Service integration	All Firebase services accessible through Firebase SDK	Developer needs to write interface/adapter for each 3rd party service and data models to normalise data within the system's internal boundary	

Design choice: RTDB vs Firestore

Pub-sub

When deciding how to separate data between the RTDB and Firestore, we decided to use RTDB for all data that needs to be pushed quickly to the client. The rest of the data will lie in the Firestore.

Both Firestore and RTDB actually have built-in pub/sub capabilities. However, we felt that RTDB is better because its data format is much easier to work with. For Firestore, we can only listen to changes on documents. This means that we have to waste extra effort and time making sure that the data we want to subscribe to is the format that Firestore wants. Another reason is the cost of Firestore to use its pub/sub capability is a lot higher than RTDB. Firestore charges for reads and writes while RTDB only charges for storage used. This means that for Firestore, we get charged whenever the document updates and the client reads that document again. This happens many times in a pub/sub system. Hence, we decided to use RTDB for its pub/sub capabilities

Data Storage

Compared to RTDB, Firestore is just better at being a normal database. Firestore has in-built optimisations to speed up the querying time for documents. Firstly, document retrieval performance scales with the size of the result set, not the underlying data set. This means that the time needed to find 1 document in a collection of 1000 documents is no different than finding 1 document in a collection of 1,000,000 documents. Secondly, when we retrieve a document, Firestore does not return the subcollections nested under the document. Both of these optimisations make Firestore a quicker database than RTDB and hence, we chose to store the majority of our data here.

In summary, the table below summarises the differences between RTDB and Firestore and why we chose RTDB for pub/sub and Firestore for data storage.

Criteria	**RTDB**	**Firestore**
Pub-sub messaging	Data format is easier to work with. Cost is based on storage	Data format is not convenient. Cost is based on reads & writes
Querying speed	Performance scales with size of underlying data set. All data under a key will be retrieved, resulting in larger pulls, decreasing querying speed.	Performance scales with size of result set. Subcollections need to be manually queried on demand. This results in smaller result sets, thus improving querying speed.

Design choice: Synchronising Client States

To reliably synchronise the states of two clients, we chose to implement a message queue. Whenever a message is received in the message queue, both frontends will react immediately to the message, thus synchronising the states between them. By having a mechanism to synchronise states, we could do things like moving both users into a session at the same time.

There could be other ways to implement this. One other way would be having clients poll the backend at regular intervals to see if there is any new information to act upon. However, this has its own problems too. Polling is not instantaneous and the backend has to manage the relay of information. In addition, this imposes a load on the backend which it needs to manage when the number of clients scales up. On the other hand, pushing a message to the client avoids this issue.

In the end, we went with the message queue because it was simple to implement with the RTDB and the messages sent could be processed instantaneously from the frontend.

The table below summarises the differences between polling and a message queue in the context of our application.

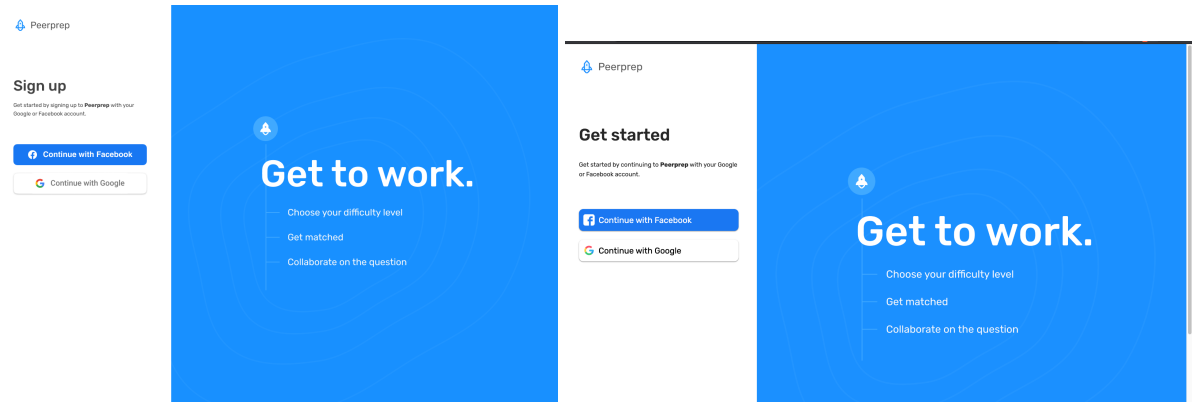
Criteria	**Event-based Message Queue**	**Polling**
Type of retrieval	Asynchronous message retrieval (frontend uses the message delivery as an event to trigger the receiver into action)	Synchronous message retrieval (frontend chooses when it wants to consume the message)
Load on server	Low load on server even with scale	High load on server with scale
Response time	Instantaneous	Will take a while for frontend to be updated depending on poll interval set

Design choice: Match Timeout using Google Cloud Tasks

The user needs to be removed from the queue and informed if there is no match after 30 seconds. A decision has to be made on whether to track the timeout on the frontend or the backend. We chose to track it on the backend. This is so we could store most of the core logic in the backend. We could also better handle the possible edge cases arising from this interaction. For example, if the user drops off (closes his browser) and there is no one available, the user will automatically be removed from the queue. If the frontend tracks the timeout, the backend still has to implement some system to clean up stale users from the queue. On the other hand, a backend implementation would allow the user to be removed from the queue as the state tracking does not rely on the client being online.

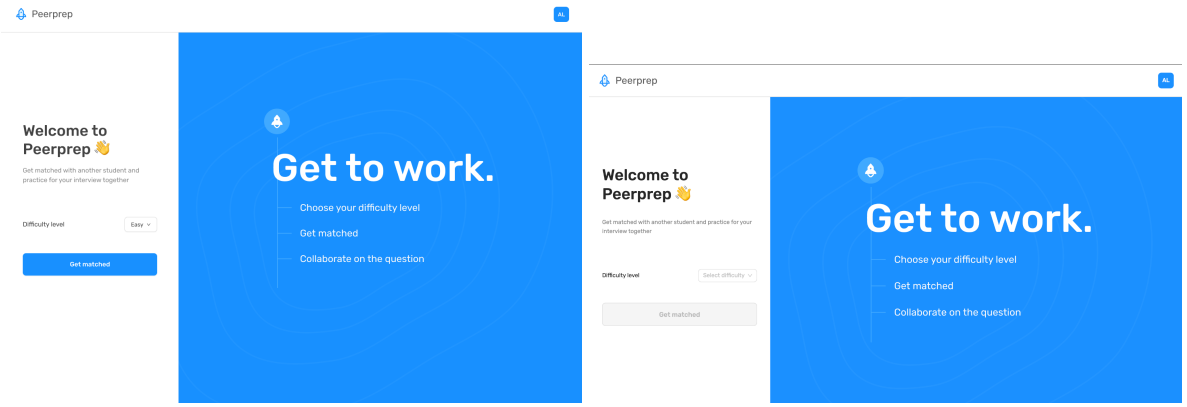
The ideal way to implement this would be to have some sort of scheduling mechanism to trigger a Function to check if the user is still in the queue, and if he is, remove him and send a session not found message to the user's message queue. However, there is no such mechanism for a Cloud Function to dynamically schedule another Function to run after a set duration. Additionally, we cannot sleep in the middle of a Function execution as there is a limit to how long a Function can be executing. Thus we used Cloud Tasks to implement the match timeout, which will trigger a function at a scheduled time. This implementation further decouples the functionality of removing a user from the match queue from the caller and callee as neither are blocked.

Frontend development progress

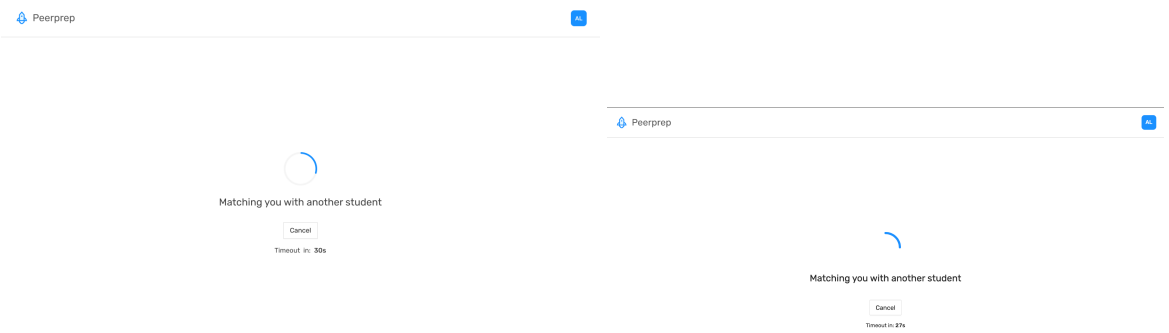


Sign in page with initial figma mock on the left and implemented version on the right

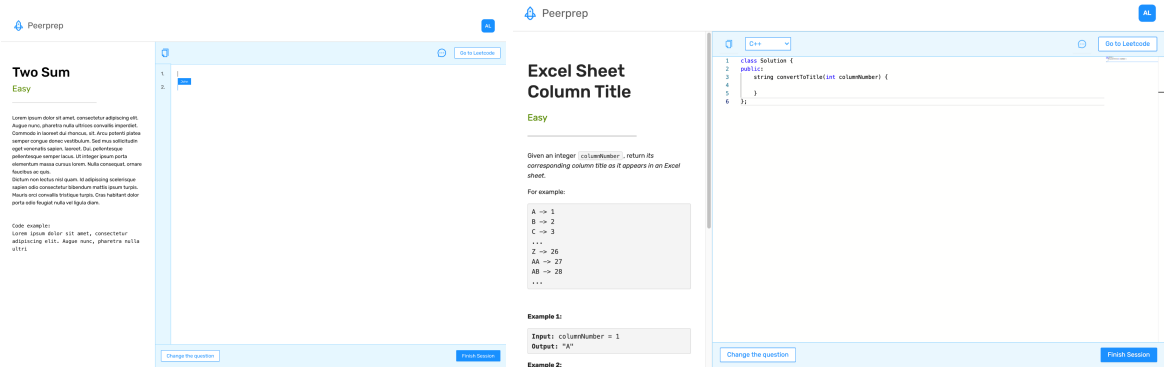
Initially there were separate sign in and sign up options on the figma mock but as the authentication on firebase is the same for both, we decided to change the button to “Continue with Google” and the header to Get Started instead.



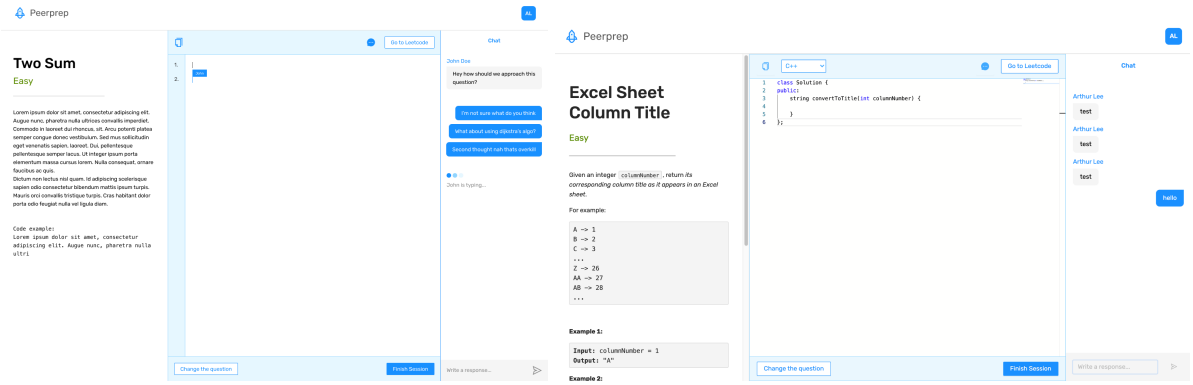
Home page is the same with the figma mock on the left and the implemented screen(right)



Loading screen is the same on figma mock(left) and implemented screen(right)



Editor screen on figma(left) and implementation(right) is the same as well



Editor screen with chat on figma(left), implementation (right)

Some specific features for the chat were not implemented due to complexity and time constraint such as the typing indicator for when the other party is typing.

Testing

Firebase emulators allow us to build and test locally using Firebase services. We are able to create local instances of the Firebase services we are using in our project. Cloud functions are also exposed through a local IP address so that they can be triggered and tested locally. This makes it feasible to start a frontend development server and have the frontend trigger the functions from the emulator. To provide an example of the usefulness of emulation, let us assume that we want to test if a session is created successfully if two users are present in a queue. To do that, we can directly add two users to a queue through the web UI provided by the emulator and check if the session is created using the same UI. Without emulation, we would have to deploy the function, add the two users through the Firebase web UI and check from there. We would also have to deploy the function every single time we change it. This makes it very troublesome to validate if a feature is working.

The same technology could be used for automated testing (i.e. testing Cloud Functions using code). We start a test run through firebase emulators. Firebase has a library called firebase-functions-test that helps with the mocking of Firebase data models and the triggering of the Cloud Functions in the local environment. Essentially, Firebase will wrap the Cloud Function under test and expose its trigger interface. This allows it to be triggered with test code. We also employed the use of other libraries to build up a basic testing infrastructure. Mocha is the test runner, Chai is used for simpler assertions and Sinon for stubbing dependencies of Cloud Functions that are tedious to replicate in the test env.

There are several benefits to using emulators for automated testing. The first is that a new Firebase project does not need to be created since the emulators are able to provide simulated Firebase services. This will help us save on cost and the trouble of maintaining the test project. Second, the emulators will inject environment variables for the function execution environment to the test environment. This is good as we want to have the test environment to be as similar as possible to the production environment, in order to better detect bugs that may arise during actual usage of the app. Third, for test runs with a large number of tests, running them against the emulators will be faster than running against an actual Firebase project. We need to take into account the cumulative network latency that will add up. Finally, emulator tests can be run easily on CI. We felt this was the most important factor that pushed us to use emulator tests. Being able to run emulator tests on CI allows us to perform regression testing. On the other hand, If we test against an actual Firebase project, we would need to deal with authenticating with Firebase via CI, which is not always straightforward.

However, there are also downsides to this approach. As of the time of writing, emulators have to be started and stopped with each test run. There is currently no way of leaving them running. Since the emulators simulate all of the Firebase services, it takes quite a while to start it. We cannot change our code and have near instantaneous feedback on whether it is working. This leads to longer testing times and therefore, longer development time. One way to alleviate this is to have a mix of both manual and automated testing using emulators. Manual testing for development and automated testing to check for regression. This will allow us to have a balance between maintaining the stability of the app and speed of development.

Deployment

Built static assets and pages for the front-end are served through Netlify. Netlify has its own CDN to improve the speed of retrieval for the assets. This led to better load times for the application. Created Cloud Functions are deployed to Firebase through the Firebase CLI.

Suggestions for improvements and enhancements to the delivered application

Session History

A possible enhancement to the delivered application would be session history. Like the name implies, it means preserving the content and history of all the sessions a user has attended. This includes the content in the editor, chat and which question was used for the session.

Some of the groundwork for such a feature has been built already, namely that we actually maintain some session data in Firestore and RTDB and do not explicitly delete it after the session ends. However, this requires quite a bit of work on the frontend side. It would mean designing and implementing a new UI for each user to choose which session to take a look at and thus due to time constraints we did not implement such a feature.

More flexible changing question flow

Currently the change question flow is sufficient in allowing the 2 users in a session to change the question to another random one that is within the same difficulty and with the permission of the other user. However, the feature can be expanded to include options to indicate a different difficulty level or even a specific question category such as dynamic programming.

This feature would actually be a feasible extension to our application due to the way the change question feature is implemented currently. When the change request is confirmed, a new session is created with the same difficulty and users. With this in mind, we could extend the backend API to create a new session based on a specific difficulty and even a category of question. The frontend would also need to be extended to include a select field to indicate such things when sending the request.

Matching by question category

Another possible extension would be to match users based on the question category. This means whether the question is a graph question, dynamic programming question etc based on the categories found in Leetcode.

There was actually some groundwork done for such a feature. Currently the question data scraped from Leetcode includes the category already and is stored as a field in the questions collection on Firestore. On the frontend, it would be trivial to implement as we could just add another select field to choose the category. However, there would be quite a bit of work for the matching logic as we only have 3 queues currently based on the easy, medium and hard difficulty and indicating the category would complicate the logic further.

Reflections and learning points from the project process

Local Emulation of Cloud Functions

We learnt how to test and emulate serverless functions locally. This was quite important as we needed a way of interacting with our cloud functions, and the emulator provided a HTTP endpoint we could use to test whether our functions were behaving in the expected manner.

Usage of Google Cloud Tasks

We learnt how to use Google Cloud Tasks to schedule tasks at a given time. Cloud Tasks work on a “fire and forget” basis, allowing the caller to simply create a task to remove unmatched users when they join the queue. This allowed us to implement the timeout feature asynchronously, as the cloud task would pop at the 30s mark and notify users that have been in the queue for too long to be removed.

Scheduling concerns

We noticed that there were instances when a feature that was slated to be developed in a particular week had a dependency chain that needed resolving. The backend serverless functions needed to be up and tested before the front-end could test the UI for the feature. However, as members in the team worked independently, sometimes the front-end team was left waiting for the backend to complete implementation before they could start work.

To resolve this we started discussing when in the week each person intended to work on the project, and the members coordinated around that to prevent one person from being unnecessarily blocked.

Using unfamiliar libraries

We implemented our real-time collaborative code editor using two different libraries, Monaco for the editor and Firepad for the collaboration. We were new to both of these libraries and our lack of understanding of either of them started to pose a challenge when we encountered a bug with the collaborative editor. As we were unfamiliar with the libraries, debugging proved to be rather difficult. We could not pinpoint if the bug was caused by one of the libraries, or both. To make things worse, the documentation for the Firepad library was lackluster as well. In the end, we had to resort to reading the source code of the libraries in order to understand what was causing the bug. Thankfully, we were able to resolve the bug.

What we took away from this was that we need to be careful when deciding to rely on external libraries for core features of an application. If a core feature is reliant on external libraries, it means

that these libraries can make or break the application. This could potentially be dangerous, and we need to exercise caution when making such decisions.