NUS
National University of Singapore

Final Report for CS3219

Software Engineering Principles and Patterns

Group 3

Galvin Leow Wen Yuan	A0200204J
Tan Yu Li, James	A0196491R
Muhammad Niaaz Wahab	A0200161E
Yao Yuan	A0205128N

Contents Page

1. Background and Purpose

1.1 Purpose of Document

This is a report detailing the features, implementation and difficulties faced when developing the project PeerPrep.

1.2 Project Scope

PeerPrep will facilitate collaborative coding on technical questions between two users on an online code editor. Users will be randomly matched based on their choice of difficulty level. Furthermore, the application allows paired users to communicate with each other through video and voice chat, allowing them to practice active learning. This open and free communication between peers would allow the two users to more readily learn from each other.

The project is hosted on Google Cloud Platform (GCP). It allows any two users from anywhere around the world to connect with each other. Subsequently, the users will both view the same coding question and proceed to code out a solution together.

Lastly, the PeerPrep platform supports compiling of code to allow users to view the state of their code at any one time, also allowing users to run their own test cases to evaluate the correctness of their code.

1.3 Accessing the project

To access the final project, please follow the instructions carefully under the README of our code repository here.

2. Individual Contributions

Name	Contributions
James Tan	Report ● Created architecture diagrams for Overall Architecture, Kubernetes Setup, Frontend, Video, and Editor ● [4.3.2] Microservices Overall Architecture ● [4.3.3] Kubernetes Setup ● [4.3.4] Layered Architecture: Horizontal Slicing ● [4.3.5] Frontend Architecture ● [4.3.6] Editor Microservice ● [4.3.9] Video Microservice ● [4.3.10] Compiler Microservice
	Code ● Frontend ○ Developed Home and Room pages, Editor, Video and Terminal components ○ Developed AJAX connections to backend microservices to update state correctly ● Backend ○ Improved upon video backend code initially developed by Niaaz ○ Developed Editor and Compiler microservices ● Kubernetes ○ Created deployment YAML files for compiler, editor, frontend, and video microservices ○ Created service YAML files for compiler, editor, frontend, and video microservices
Galvin Leow	Report ● [4.2] Design Decision ○ [4.2.1] Technology Stack ○ [4.2.2] Setup and Configuration ● [4.3.5] User Management Microservice Architecture ● [5.1] Security ○ [5.1.1] Authentication

	○ [5.1.2] Kubernetes Deployment ● [6] Future Plans
	Code ● [FR1.1] Login API ● [FR 1.3] Update Password API ● [FR 1.4] Get Single User API ● [NFR 2.2] Sign Up API ● Utilise Redis Cache to improve authentication middleware performance ● Docker ○ Dockerfile ○ docker-compose ○ Push to docker hub ● Kubernetes deployment file ○ Secret & ConfigMap ○ Service ○ Deployment ● Github Action ○ Running test when pushing to branch ● Mocha and Chai test case
Muhammad Niaaz Wahab	Report ● [1] Background and Process ○ [1.1] Purpose ○ [1.2] Project scope ● [4.1] Software Development Process ○ [4.1.2] AGILE ○ [4.1.3] Software Development Life Cycle (UI / UX) ● [4.3.6] Matchmaking Microservice Architecture ● [5.1] Security ○ [5.2.3] Matchmaking service ● [7] Reflections ○ 7.1 Difficulties Faced ■ 7.1.1 Designing frontend ■ 7.1.2 Designing backend ■ 7.1.3 Connecting backend microservices to frontend

	Code ● Backend ○ Developed the backend video microservice to enable video and voice chat (which was further integrated by James) ○ Developed matchmaking service that was used for matching of users ● Kubernetes ○ Created Docker files for all frontend and backend services ○ Dockerized and uploaded services to Docker Hub, and later to Google Container Registry ○ Worked with James to conduct deployment of services initially to AWS, and later to GCP
Yao Yuan	Report ● 3. Project Requirements ○ 3.1 Functional Requirements ○ 3.2 Non-Functional Requirements ● 4.1.1 Milestones & Timeline ● 4.2.1 Technology stack ● 4.3.1 User flow ● 4.3.6 User Management Microservice Architecture ● 5.1 Security ○ 5.1.1 Authentication and Authorization ○ 5.2.1 Kubernetes Deployment ● 7.2 Takeaways
	Code ● Frontend ○ Developed Login page, Signup page and Profile Page ○ Created public and private route with JWT token ○ Connected frontend and backend for user management microservice ● Kubernetes deployment file for user management ○ Secret & ConfigMap ○ Service ○ Deployment

3. Project Requirements

3.1 Functional Requirements

FR 1	Account component (authentication)
FR 2	Matching
FR 3	Room component
FR 4	Editor component
FR 5	Voice / Video Component

FR 1	**Account component (authentication)**		
FR 1.1	The application should allow users to be able to login with valid credentials	High	Completed
FR 1.2	The application should allow users to be able to log out from account	High	Completed
FR 1.3	The application should allow users to be able to reset password	Medium	Completed
FR 1.4	The application should allow users to be able to view account details under the profile page	Medium	Completed
FR 2	**Matching**		

FR 2.1	The application is able to provide filters for difficulty level, algorithm category and coding language	High	The current implementation only allows users to select the difficulty of the coding question as we only stored the difficulty level for each question. By storing more properties to describe the questions, we can easily extend the feature to allow users to choose based on more identifiers, such as data structure.
FR 2.2	The application is able to select the corresponding question based on the filter	High	Completed
FR 2.3	The application is able to match 2 users based on the same filter	High	Completed
FR 2.3.1	A room will be created for the matching users with the selected question	High	Completed
FR 2.3.2	The application is able to handle unsuccessful matching after 30 seconds	Medium	Completed. The current implementation will timeout upon 5 seconds of unsuccessful matching. This is to reduce the waiting time for our users.
FR 3	**Room component**		
FR 3.1	The application allows two matched users to collaborate on the editor.	High	Completed
FR 3.2	The application allows matched users to use both video and voice for communication	High	Completed

FR 3.3	The application allows either user to end the room session	High	Completed
FR 3.4	The application allows either user to reload to a new question	High	We have not implemented this feature as after re-consideration, we think this is a *LOW* priority requirement. We want to encourage users to re-match for new partners upon completion of a question, to gain different insights and learning tips from different peers. In addition, if they would really like to change the question, they can do so by leaving the room and re-matching at the homepage.
FR 3.5	The application will return either user to the home page if either user exits the session	High	Completed
FR 4	**Editor component**		
FR 4.1	The application should be able to support Python and Javascript compilation	Medium	The current implementation only supports python compilation, as it is the most popular language for programming interviews.
FR 4.2	The application should be able to run code and display STDOUT	Medium	Completed
FR 4.3	The application should be able to show updates to the text field in near-real time	High	Completed
FR 4.4	The application should be able to accept STDIN value	Low	We plan to implement these low priority features in the future, as it does not affect the use of our

FR 4.5	The application should be able to support multiple themes	Low	application, but is good to have.
FR 4.6	The application should be able to validate test cases	Low	
FR 5	**Voice / Video Component**		
FR 5.1	The application should allow users to provide access to webcam and microphone	High	Completed
FR 5.2	The application should allow users allowed to toggle on/off the microphone	High	Completed
FR 5.3	The application should allow users to toggle on/off the camera	High	Completed
FR 5.4	The application should allow either user in the same room to start a call	High	Completed
FR 5.5	The application should allow either user in the same room to answer a call	High	Completed
FR 5.6	The application should allow users to hear their partner's voice during a call if their partner has toggled on their microphone	High	Completed
FR 5.7	The application should allow users to see their partner's camera feed during a call if their partner has toggled on their webcam	High	Completed

3.2 Non-Functional Requirements

NFR 1	**Performance**		
NFR 1.1	The application should respond within less than 1 second when the user navigates through major features	Medium	Completed
NFR 1.2	The application should reflect changes in real-time when a user modifies the text field in the editor	High	Completed
NFR 1.3	The editor in the application will only compile and run code for at most 5 seconds	High	We decided to scrap this feature because we want users to have a real sense of how long their code takes to run. We also want users to be able to see their output for relatively inefficient code that takes more than 5 seconds.
NFR 2	**Security**		
NFR 2.1	The application should only accept strong passwords when a user registers for a new account	High	Completed
NFR 2.2	The application should store user's passwords in a hashed and salted form	Medium	Completed
NFR 2.3	The application should prevent a user from logging in for 5 minutes after four failed login attempts within the timespan of 1 minute	Low	Since we have implemented the requirements above for security consideration, logging out a suspicious user might not be as essential.

NFR 3	**Usability**		
NFR 3.1	The application should be supported on Chrome, Safari, and Firefox	High	Completed. On these browsers, you need to allow insecure access to our application in order to access the video
NFR 3.2	With the help of a user guide, the application should be easy for users of any technical skill to use	High	The application adopts a minimalistic design, and only includes necessary features to reduce complexity for users. Thus, we don't see the need for a user guide, as our application should be easy and intuitive enough for anyone with a programming background to use.
NFR 4	**Availability**		
NFR 4.1	The application should be available for use 98% of the time	High	Completed
NFR 4.2	The application should allow for at most 1000 users to sign up	Medium	Completed
NFR 4.3	The application should support up to 100 simultaneous users	High	Completed
NFR 5	**User Interface**		
NFR 5.1	The application should allow users to enter Matchmaking within 1 click from the home screen	High	Completed
NFR 5.2	The visual interface should be dynamic during matchmaking	Medium	Completed
NFR 5.3	The user interface should have minimal options	Low	Completed

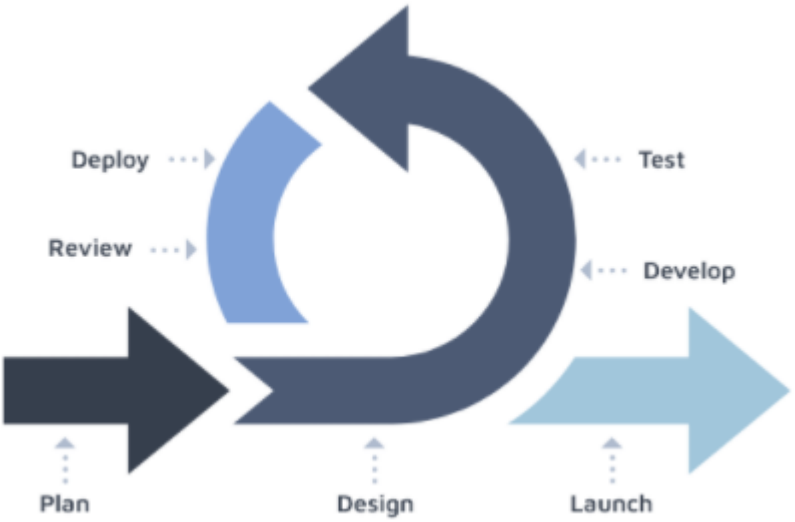
4. Developer Documentation

4.1 Software Development Process

4.1.1 Milestones & Timeline

Milestones	Tasks	Timeline
1	● Decide on functional and non-functional requirements ● Research and decide on the technology stacks, followed by learning those technology stacks ● Design for frontend ● Design for microservice architectures ● Set up frontend and microservices backend & database	Week 5, 6, 7
2	● Complete all microservices, including: ○ Account component ○ Matching component ○ Room component ○ Editor component ○ Video / audio component	Week 8, 9, 10, 11
3	● Deployment ● Testing ● Documentation	Week 12, 13

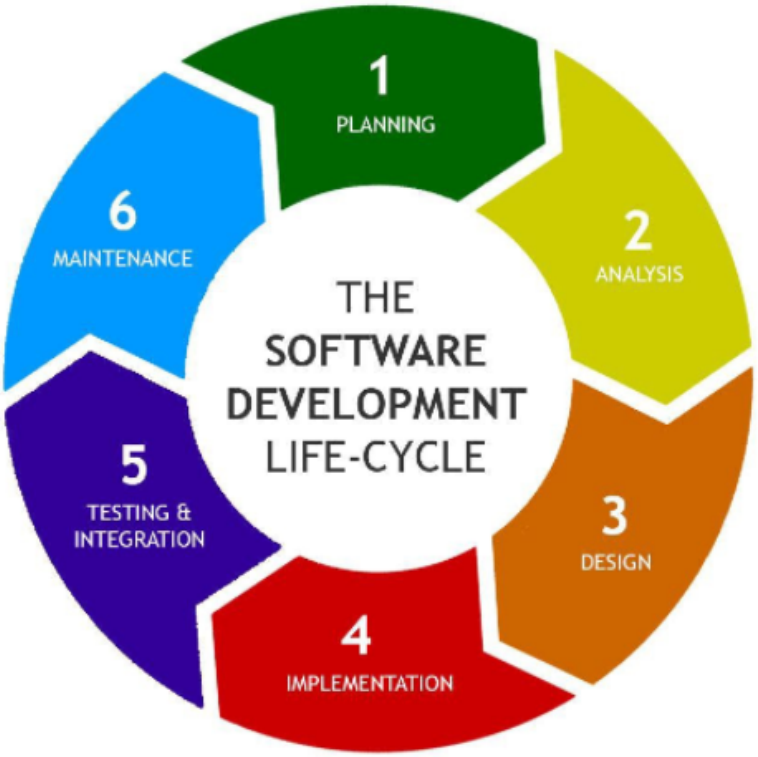
4.1.2 AGILE



The development process of AGILE was employed for PeerPrep. AGILE is based on iterative development, whereby each iteration introduces incremental changes to the product but remains a viable product at every stage. By applying AGILE, we remained disciplined about maintaining usable software in every iteration. This ensured that we stayed on track in every iteration, and was able to constantly produce working code.

Additionally, we based developing PeerPrep around weekly Sprints. At the start of each Sprint (or iteration), we would decide on features to work on for that week and check in frequently with each other to ensure we are all on track. We could not adopt the concept of daily SCRUM meetings due to our collective time constraints, but touched bases with each other twice a week to ensure we kept on track with our tasks. If someone were stuck on an error, we would try to help out that person or push back the tasks to subsequent Sprints.

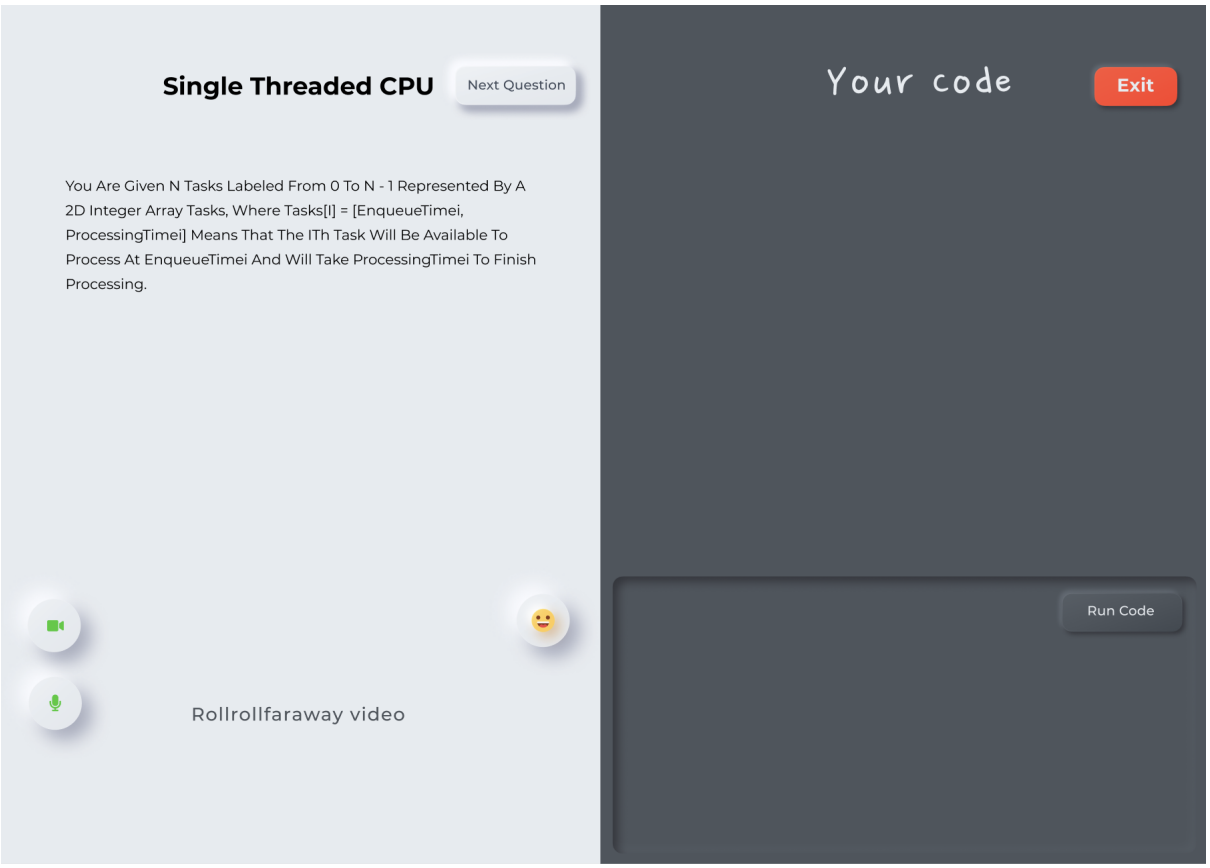
4.1.3 Software Development Life Cycle (UI / UX)



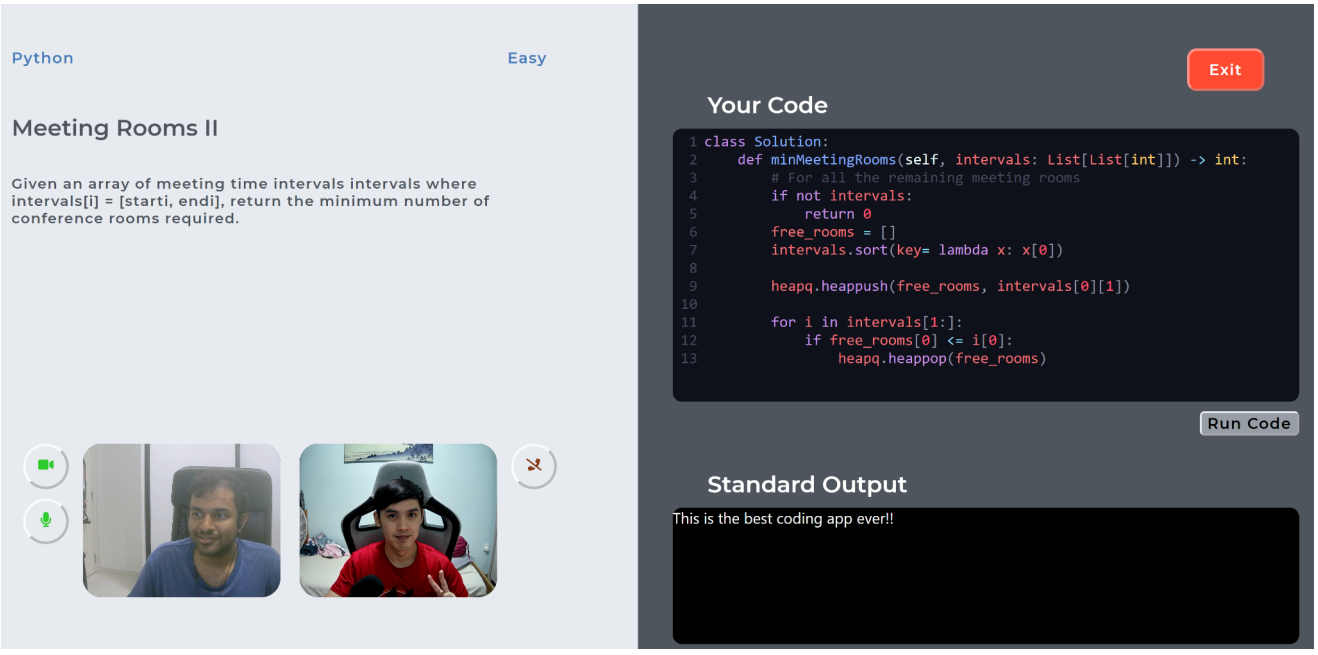
We employed the full software development life cycle for the frontend of PeerPrep. The frontend was first prototyped with Figma to test appearance and User Experience. We also conducted user testing for PeerPrep to ensure that the design was usable and enjoyable. The feedback gathered from the user testing allowed us to refine our UI (User Interface) and streamline it to include more useful features without sacrificing usability of PeerPrep.

Only after this lengthy process did we embark on coding out the frontend for PeerPrep. This allowed us to save significant time as all our design changes were changed quickly on Figma, allowing us to save plenty of valuable developer time in the actual coding of the frontend. We then had to implement simple testing before integrating with PeerPrep's backend services.

Here are some examples of both our Figma designs and the actual PeerPrep design.



Initial Figma design



Final PeerPrep design

4.2 Design Decisions

In the process of translating the requirements of the application into a working application, there are many decisions that have to be made, it includes decisions on the technology stack and specific configuration of the tools.

4.2.1 Technology stack

The technology stack decisions are usually based on the tools that members of the team have experience in using, however, there are a few other factors that we have considered. For example, for frontend, we choose to use react instead of other JavaScript libraries due to the availability of well-documented libraries and the support of handy tools. In addition, the reusable components reduce the need to implement additional components and help to make our app easier to develop and maintain. React also improves performance using virtual DOM as it only changes individual DOM elements instead of reloading the complete DOM every time.

We also used Github Issues as the tool to manage our project due to the consensus that using lesser types of software tools will benefit the project due to the short timeline. Although we are aware of the tools that companies use like Jira, this would not be suitable for this project. Jira has many features that are available like timeline tracking, issue tracking, cost monitoring and many more. However, this project would not need such a comprehensive tool to keep track of our tasks, hence we selected Github Issues. Github Issues have the ability to assign issues to users and also ties the pull request to the issue. This is beneficial for us as the automation of tasks being marked as done when pull requests are accepted reduces clutter.

Another decision that we have made was the use of MongoDB. MongoDB is a non-relational database and we decided to use one after considering the data that needed to be stored and retrieved. The reason for using a non-relational database is due to the simplicity of design and better horizontal scaling. There are few data points in this application hence having a simple design would be sufficient, the user-management database is the only possible permanent database that would handle massive amounts of data. The usage of data in the user-management database is simple as the endpoint would require all the user details, hence storing in a non-relational database would mean that the JSON is readily available for the endpoint.

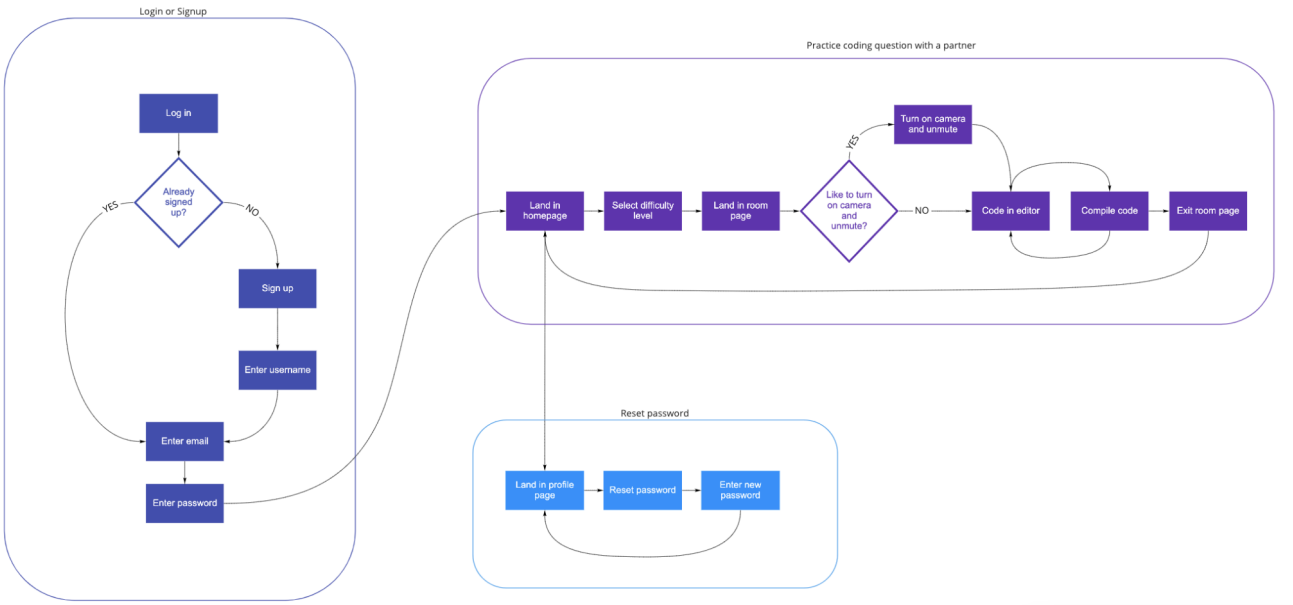
4.2.2 Setup and Configuration

There are a few configurations that we would have to consider when developing the application. One of the examples is using Redis in the user management microservice to achieve better performance. Redis cache is not necessary for the usability of the application but we have decided to set it up for a better user experience. The Redis cache key-value pair would also have to be decided so as to utilise effectively and we settled with having the key as the JWT token and the values as the users' data. This would help improve the performance significantly when there are large numbers of users. Every time there is a call to the backend endpoints that required authentication, the Redis cache would be the first to act, increasing the performance. There are other related configurations that were considered under this use case, like the token expiry duration to the time to live data in the Redis.

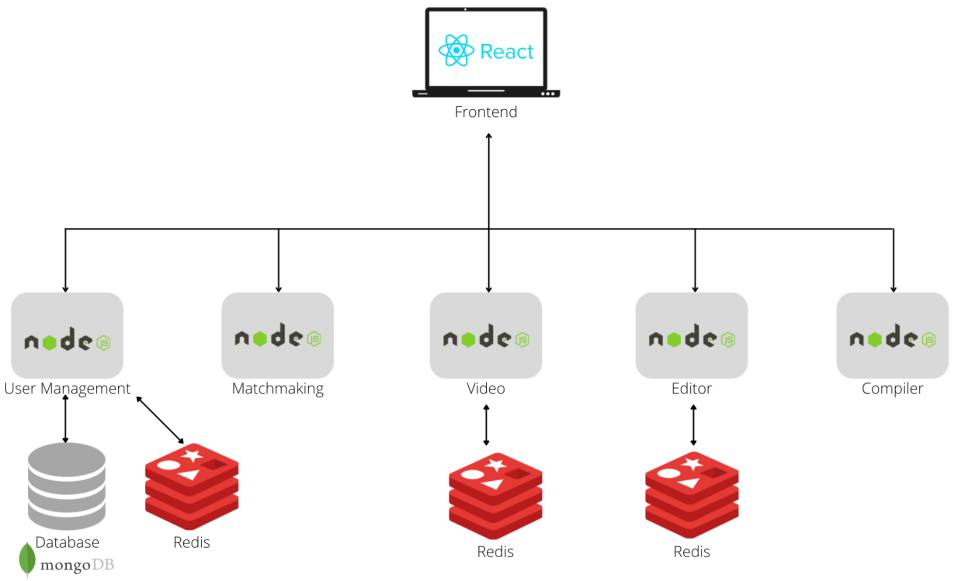
Another example of configuration decision was the scaling, since we are using GCP Google Kubernetes Engine, scaling the pods would be fairly simple. There are 2 ways that we could scale, horizontal or vertical, and we chose to scale horizontally. The factors that were considered when making this decision would be the availability of the website and the traffic flow of the website. Since the flow of traffic would be minimal, there was no need for an excessive amount of CPU and memory and hence, no need for vertical scaling. Horizontal scaling is preferred as computation power was not the cause of concern in this application. Horizontal scaling will solve the issue of the availability of the application because the replica will take over if the main pod is down.

4.3 Software Architecture and Design Principles applied

4.3.1 User flow



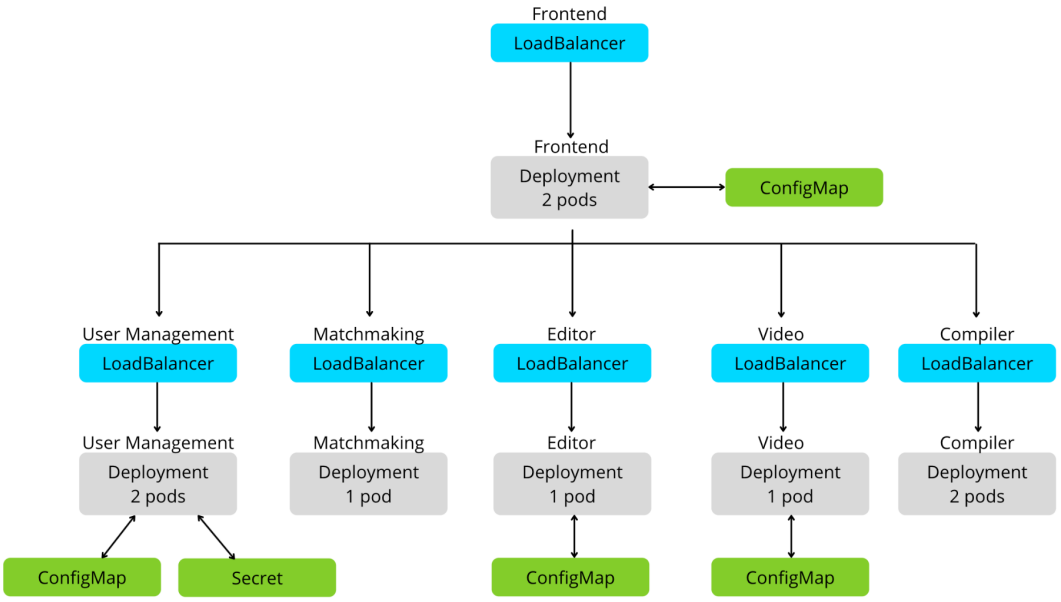
4.3.2 Microservices Overall Architecture



This is a high-level overview of the architecture of our deployed web application, Peerprep. We used a microservice architecture because it ensures **loose coupling** and **modularity** between the different compute components. The architecture of individual microservices and the Kubernetes setup will be discussed in more detail later.

The front end will be sending API requests to the relevant microservices in order to achieve different functionalities.

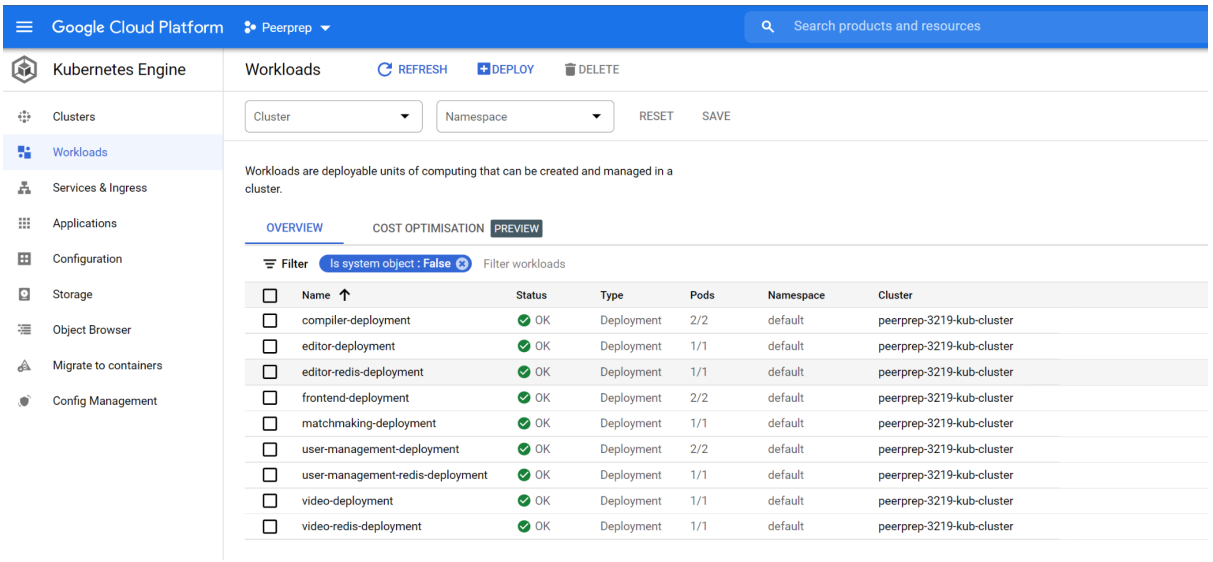
4.3.3 Kubernetes setup



Consideration for number of pods

For compute-intensive microservices such as User Management and Compiler, they are deployed with 2 pods so that incoming requests can be balanced among the 2 pods.

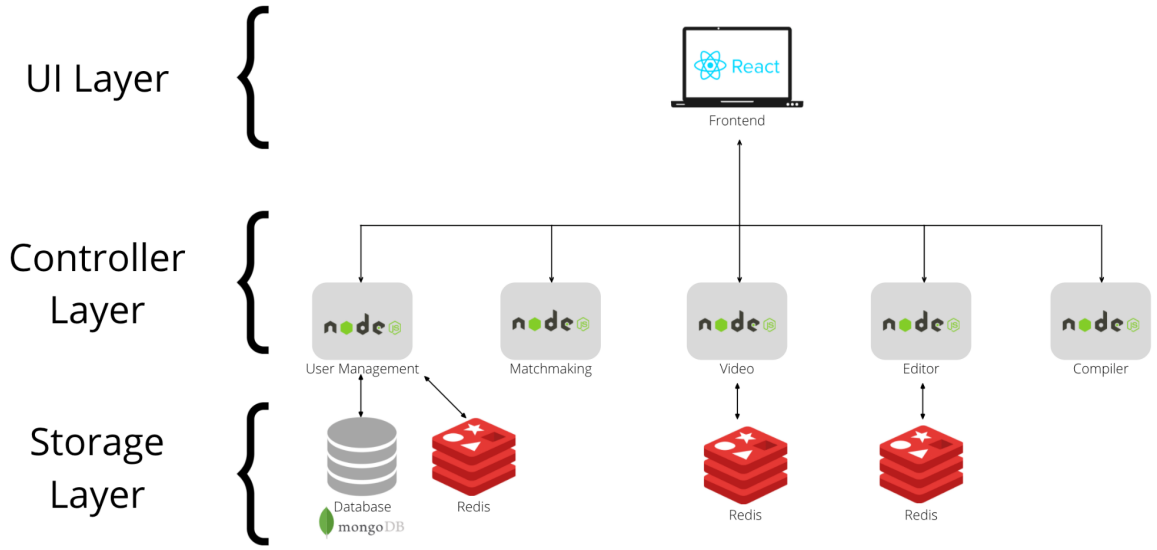
For microservices where it is important to have a consistent internal state among all pods (such as Matchmaking, Editor and Video), only 1 pod was deployed. For example, for the Editor microservice, two clients must be connected to the **same server** in order to successfully push events and subscribe to events from their partner. If there are more than 1 pod, we realised that sometimes, these events are routed to different pods, resulting in the changes not received by either party.



Deployed Kubernetes pods

4.3.4 Layered Architecture: Horizontal slicing

We designed our microservices architecture in a way such that it is segregated into different horizontal layers, as shown below.



Each layer has a distinct and specific responsibility as shown in the below table.

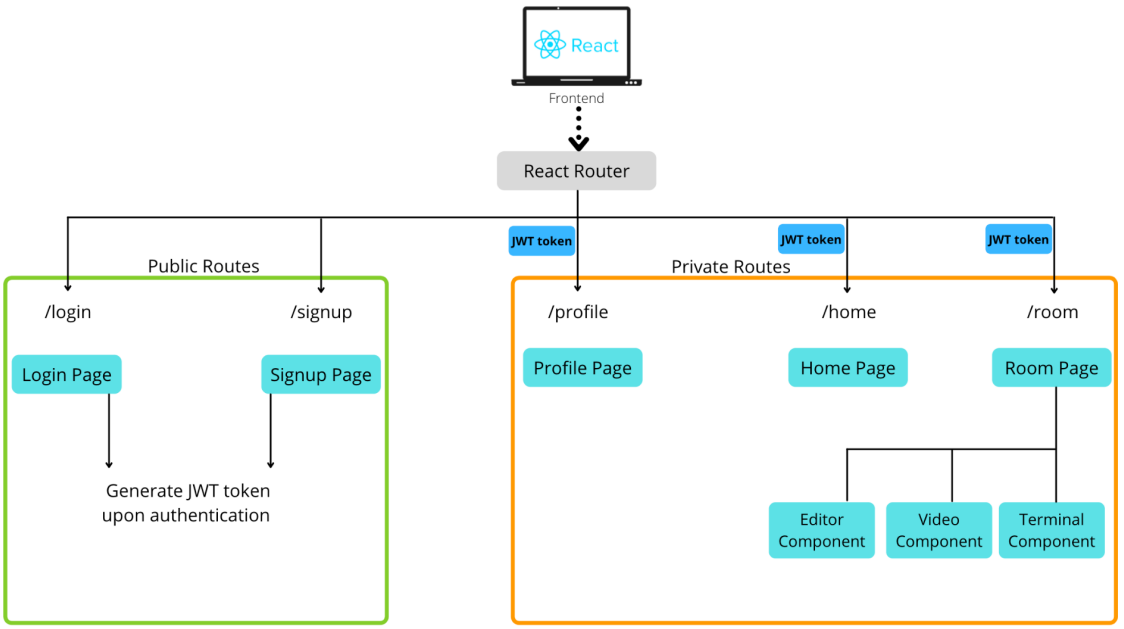
Layer	**Responsibility**
UI	Respond to user interactions Show visibility of the system state

	Make API calls to the appropriate microservices in the Controller layer
Controller	Accept incoming requests from clients Perform their relevant compute-intensive tasks Send response back to clients
Storage	Accept connections from microservices in the controller layer Ensure data persistence Allow microservice to read and write data

This separation of concerns allows us to independently develop and evolve the layers for this project.

4.3.5 Frontend Architecture

We used React to develop our frontend application, as it allows us to modularise our application into discrete components that can be reused. Below is the architecture of our frontend application.



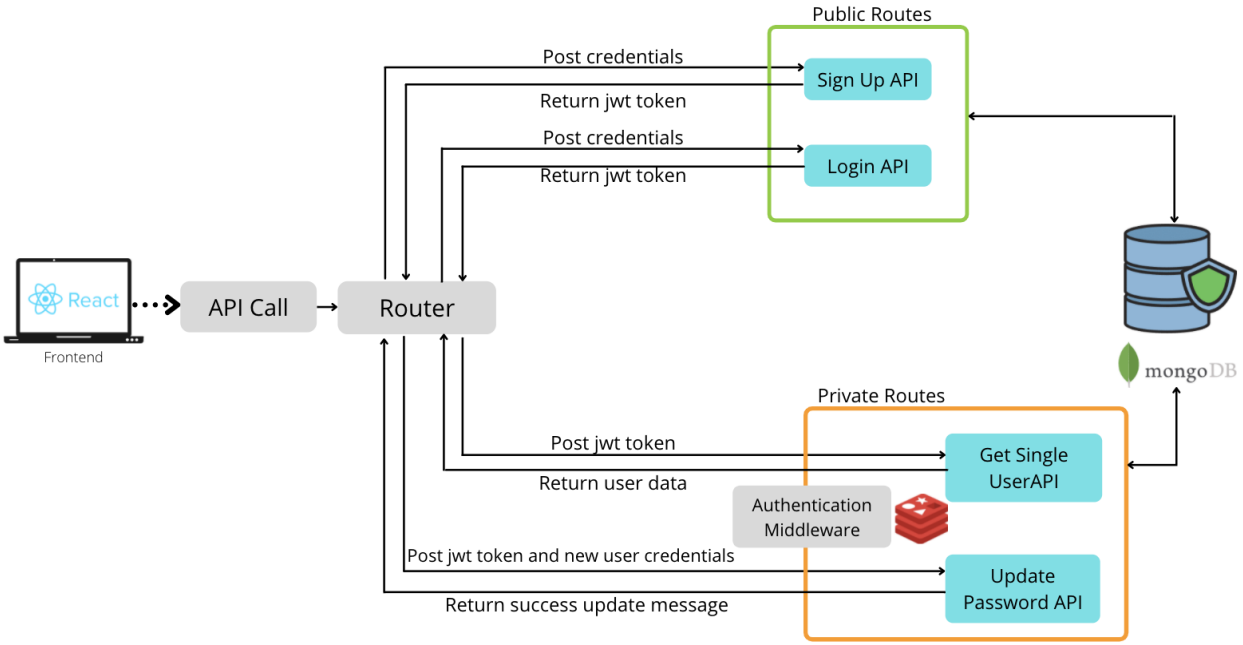
Routing:

We used react-router to do client-side routing as it results in a faster loading time when navigating between pages. Under react-router, we

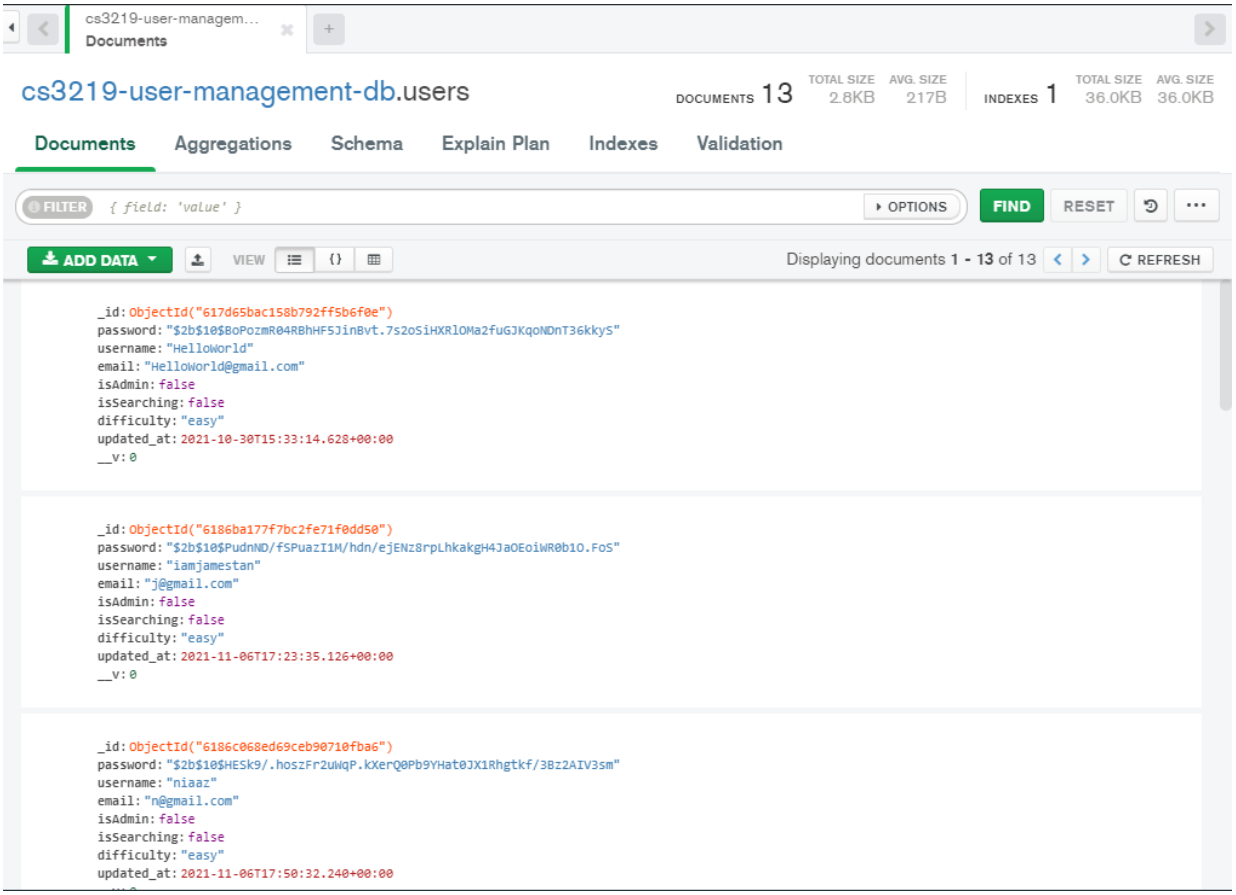
- Mapped URL routes to distinct pages
- Differentiated between public and private routes. Public routes can be accessed by anyone regardless of whether they are authenticated or not, whereas private routes can only be accessed by clients who are authorised with an appropriate JWT token.

4.3.6 User Management Microservice Architecture

For the user management microservice, routes are splitted into public routes and private routes. Login and signup pages are accessed through public routes for unauthenticated users, while profile pages are only accessed for authenticated users. For user login and sign up, user credentials (email & password) are required for user authentication. For retrieving user data and resetting password in profile page, JWT token is required for authentication. The authenticity of the user will be verified by the authenticated middleware which uses Redis cache to improve its performance. Redis cache using the JWT token as its key so that it would be able to retrieve the user information efficiently if it exists. Upon successful authentication, the server will add, update, or retrieve data from the database.



MongoDB is a non-relational database, hence it stores each users' data as a document. The password that is stored in the database is salted and hashed using bcrypt so that in the situation that the database is being accessed by the attacker, they still wouldn't be able to get the users' password. MongoDB is hosted in AWS using the MongoDB Atlas, to help with availability and disaster recovery there are 3 instances created.



4.3.7 Matchmaking Microservice Architecture

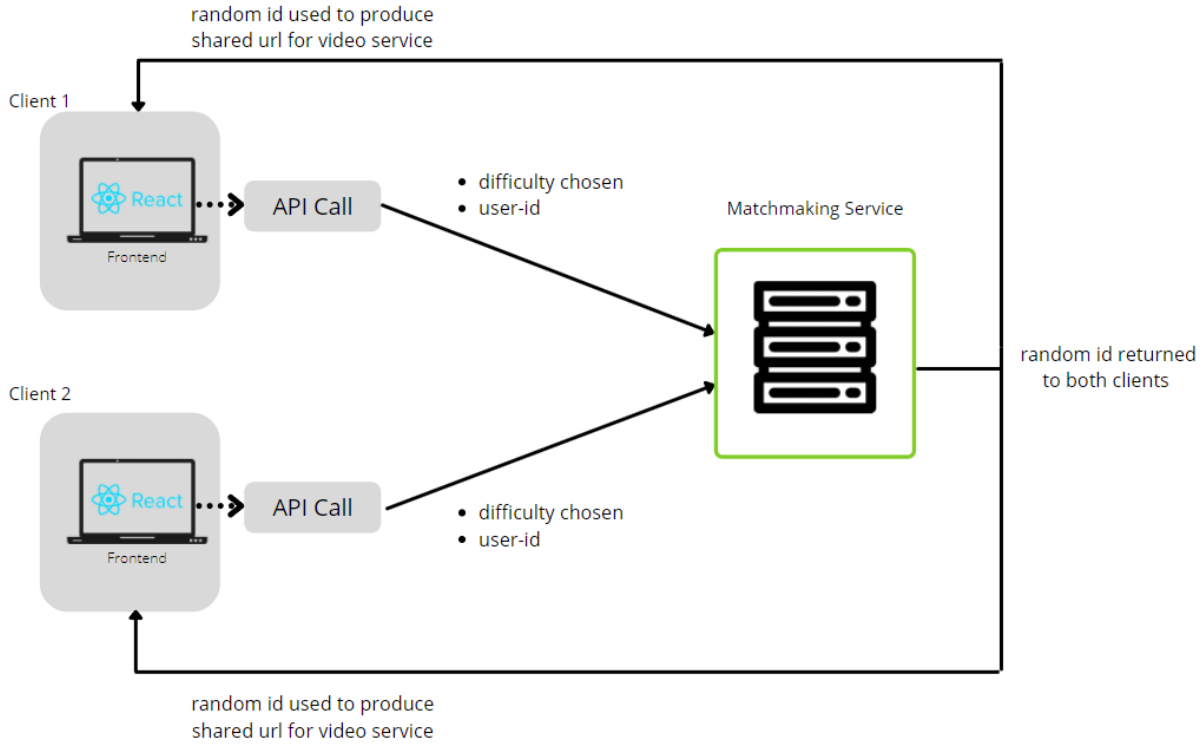
4.3.7.1 Purpose of Matchmaking

The use of a matchmaking system in PeerPrep was paramount to ensuring that users get paired randomly and fairly according to their preferences. The matchmaking system should allow a user to be matched with another user randomly if they have both selected the same difficulty levels. Users can thus match based on the difficulty they are looking for, which ensures a smoother experience for participants.

4.3.7.2 Design of Matchmaking service

The matchmaking service was designed to be a service that could be queried from the frontend and not incur any additional overhead in the way of a database to reduce the overall footprint of PeerPrep. To tackle that problem, we designed the matchmaking service to take in one user of each difficulty into a “queue” of sorts. This “queue” can only hold one user (User A), and thus when a User B tries to join the “queue”, it would be full. But since we only need to match 2 users, we would simply pop User A off the queue to be matched with User B.

This “queue” structure ensures that we minimise processing overhead as well as reduce space taken up by this service. This helps to keep our space complexity as O(1) even as the system scales up, as only 2 users are kept in memory per difficulty at any one time.



4.3.7.3 Procedure of matchmaking

The process of matchmaking is done in such a way as to ensure that every user needs only to input a randomly generated id and difficulty, and when matched, will be returned a unique id to be used to connect to the video service.

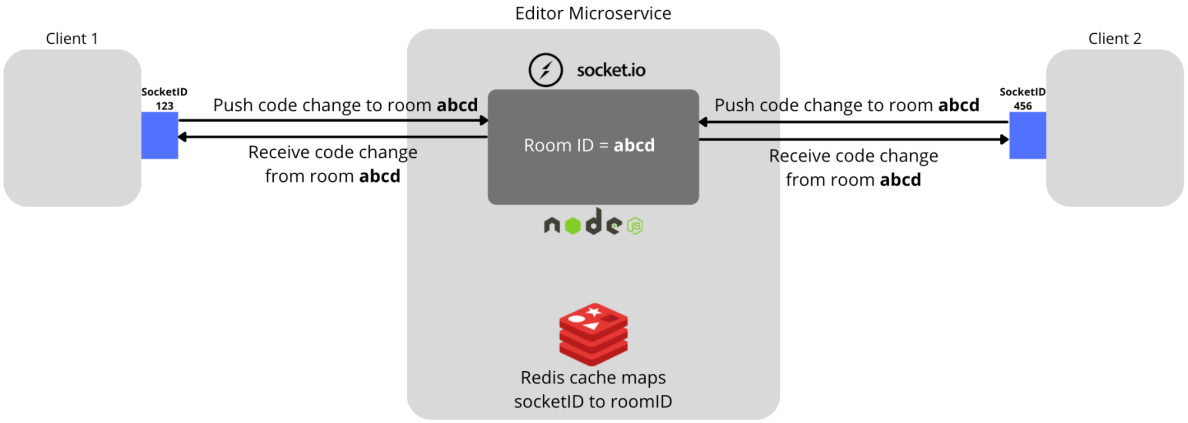
When User B joins User A in the “queue”, the system then combines the two (randomly generated) user ids’ in lexicographic order, and finds a SHA of this value, which is returned to both users. In this fashion, no value needs to be “shared” between users in the Matchmaking service itself, only needing the values provided by users, thus not needing any additional state.

This method of matchmaking has allowed us to both keep the system lightweight but also highly responsive and reliable.

4.3.8 Editor Microservice Architecture

For the editor microservice, the most important functional requirement was that a user should be able to see near real-time updates to their editor code when their partner makes changes to it (FR 4.3).

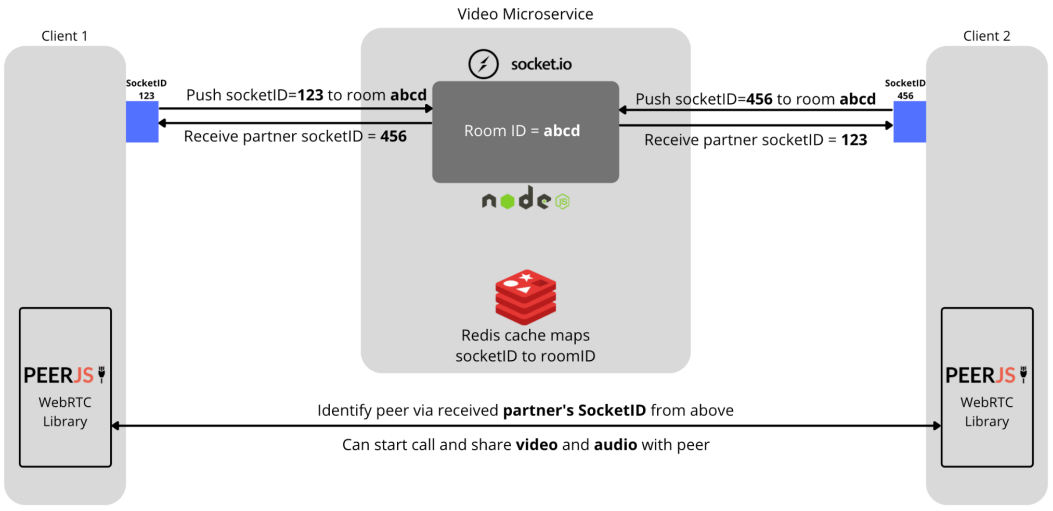
Hence, we have decided to use an **implicit invocation architecture** with the help of Socket.io. When two partners are matched, they share a common room ID, and they both register an interest to be notified about any event that is broadcasted to that room.



When either partner makes changes to the code, these events will be pushed to the editor microservice under that same room ID, and the editor microservice will then broadcast these changes to the other partner in the room.

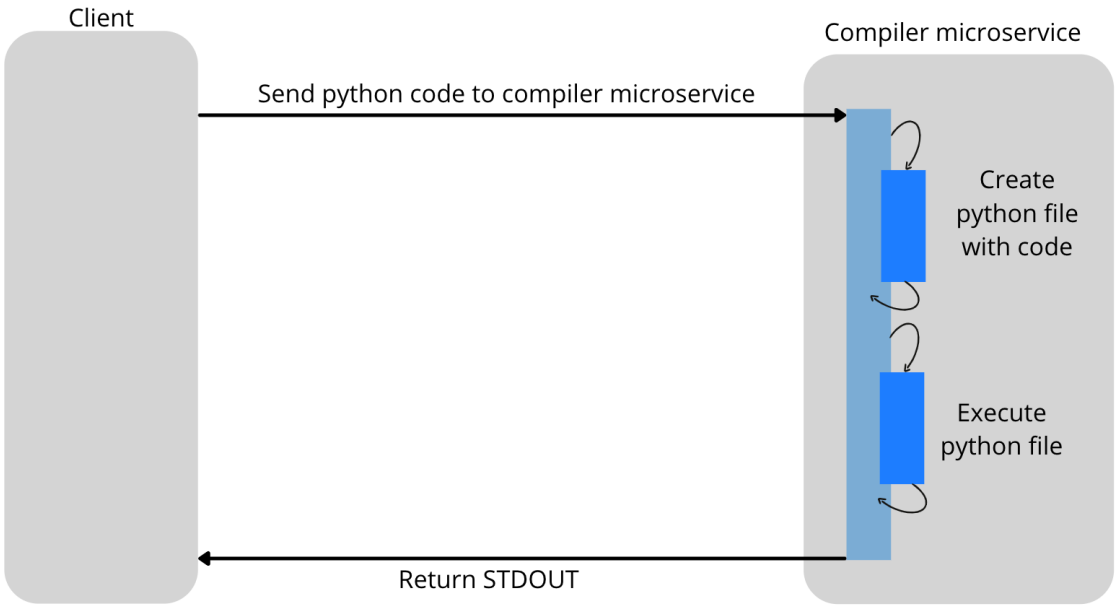
4.3.9 Video Microservice Architecture

For the video microservice, we have also decided to use **implicit invocation architecture** but for the purpose of exchanging socket IDs with each other. Once the exchanging of socket IDs are done, the clients can now call each other with the help of PeerJS.



4.3.10 Compiler Microservice Architecture

For the compiler microservice, it is a relatively simple architecture where the server just accepts requests, generates a file, and then executes the file and returns the output to the client.



5. Other considerations

5.1 Security

5.1.1 Authentication and Authorization

The user management service ensures the authentication of users, which are shown in the following features. Firstly, during user signup, a password checker will check the strength of the passwords that the user keys in. The application also allows users to reset passwords. All these measures are to protect user credentials from hackers and brute force attacks. In addition, to ensure authorisation, we differentiate public routes and private routes in the frontend to determine what resources a user can access. In particular, only login and signup pages are visible to the public, while home page, room page and profile page are only accessible to authenticated users.

The backend of user management microservice has few security considerations that would have to be addressed. Firstly, the data that will be used to generate the JWT token have to be non-sensitive, before generating the token the removal of all sensitive information is

essential. Next, the selection of a secret key for the generation of the token, this has to be kept secret as the attacker would be able to generate a valid token with this information. The secret key would need to be something complex and long increase the entropy for better security. Lastly, the password that the backend receives is in plaintext, however, we should not save the password as a plaintext due to security issues. In an unfortunate case that the database leaks information, the attacker will still not be able to know the users' password if they are salted and hashed.

5.2.3 Matchmaking service

Key to ensuring the video chat rooms are secure is ensuring that outsiders are unable to join the chat rooms.. Initially, our service simply created a link based off the SHA of the users' names. However, this would be vulnerable to an attack as an attacker could hash two names of users ahead of time and hijack the video call.

To keep the chat rooms more secure, we instead generate a random number from the user upon joining a video chat room. This would eliminate the chance of a replay attack from a malicious user trying to hijack the call.

5.2.2 Kubernetes Deployment

There are fields in the configuration file of user management microservice that are sensitive information and should not be stored in clear text. In kubernetes, there is a mechanism known as secrets that will be used to store such information in a safe and reliable way. There are 2 fields that require the use of secrets, the MongoDB url that contains the username and password and the secret that is used to generate the JWT token. The rest of the variable in the environment file will be stored using configMap.

6. Future Plans

In every application, there are always things that could be improved and PeerPrep is no exception.

There are a few features that we have not managed to implement in the current version but would value-add to the current application. Firstly, it would be an enhancement of the matching feature. Currently, the matching feature would only accept the selection criteria by difficulty of the test questions. It would be beneficial to add selection criteria by algorithm category and even coding language. The users of this application might want to practice a specific type of algorithm question, but they are unable to do so currently.

Next, we could enable the editor to compile and run different code languages. Currently, it only supports python, which is a popular coding language but it would be better to expand to more types of language as in the real coding interview or technical test there won't be a restriction to the language that can be used. Hence, in order to provide an experience that is as close as the real thing, it is best to support more languages.

Lastly, the project could use the CI/CD tools better by increasing the number of Github workflows. The workflows will help us better automate the flows that are repetitive and hence give us more time for things that are more important. The automation of tests by the workflow will also help ensure that the code pushed to the main branch would be of a certain standard. However, we should also consider the time needed to set the workflows up so as to manage our time effectively.

7. Reflections

7.1 Difficulties Faced

7.1.1 Designing frontend

We initially found it very challenging to move from Figma design to code. It was difficult to break down the UI into different frontend components (e.g. breaking the room component into video component, editor component etc). This was particularly challenging as we did not know how big or small each component should be. We would have too many components if it’s too small. On the other hand, we might end up having giant complex components if it’s too big.

Finding a middle ground between these two was challenging, however, after plenty of trial and error as well as research, we managed to strike this balance and design components which we felt were a good size.

7.1.2 Designing backend

Designing backend services while decidedly less complex individually, turned out to be a nightmare. Running each service on its own took a decent amount of time to do. However, a bulk of the time was spent dockerizing and making sure each container could be accessed. Versioning of certain tools introduced an unforeseen element of difficulty regarding compatibility which caused us many sleepless nights.

Finally, the process behind uploading images to Docker Hub was very involved and we also faced many issues with the images throwing obscure errors and our versioning being incorrect. Over time and with plenty of Googling, we managed to resolve these issues as well.

7.1.2 Connecting backend microservices to frontend

By far the most challenging portion was deployment. Hosting of containers, orchestration with Kubernetes, and finally connecting the frontend to the backend endpoints. Performing these actions in tandem and in sequence was a hassle to say the least and the tools provided by AWS and GCP do have a bit of a learning curve.

We initially hosted all our containers on Docker Hub and pulled them to AWS from there. However, we faced issues with connecting our containers together, and made the decision to transition to GCP. With GCP, we hosted our containers in Google's container registry, and pulled the images from there. GCP was significantly more user friendly than AWS, however we did notice that AWS was a lot quicker in deployment of nodes and clusters than GCP.

Overall, while we faced more issues with AWS, we believe AWS is the better service to choose as its speed and range of services does indeed make it more useful than GCP. A lesson learnt for us is to get more familiar with cloud services, but especially so for AWS, which we believe is still the better service.

7.2 Takeaways

Although we faced many difficulties during the process, all the challenges and obstacles have made us think more carefully when designing for software applications, and also provided us with a deeper understanding towards both software architecture and software development cycle. To build a complete and functional software application, coding is just a small part of it, the planning and testing are as important as the code itself, as they work together to make the application work as expected. Another takeaway is that, when it comes to designing implementation details or resolving a bug, we should always explore alternatives and weigh trade-offs. There is no easy way to find the perfect solution, and we should not stop at the easiest solution, but always choose the most suitable one. During the process, we also realised the importance of teamwork and communication. It is important that all members are clear about what they are responsible for and deliver their parts on time. The key for our success is that all of our group members take full responsibility for their task and even take the initiatives to help each other. Besides, communication is the

fundamental of teamwork. Our team has biweekly meetings with all group members and more frequent meetings among small teams (frontend team, each microservice team etc). Thus we managed to work together and resolve the obstacles along the way. In spite of the difficulties we faced, we believe that the lessons learned will come in handy in the future and help us build greater software products.

8. References

We would like to give credits to the resources below that have been used to guide us for our project.

8.1 User Management Microservice

https://www.youtube.com/watch?v=mbsmsi7l3r4&t=577s

8.2 Video Microservice

https://www.youtube.com/watch?v=gnM3Ld6_upE&list=LL&index=15&t=902s
https://www.youtube.com/watch?v=7a_vgmEZrhE&list=LL&index=14
https://www.youtube.com/watch?v=0fWN_q4zAqs&list=LL&index=13&t=335s

8.3 Editor Microservice

https://www.youtube.com/watch?v=gq4dDiZNXCI&list=LL&index=19&t=14s

8.4 Compiler Microservice

https://www.youtube.com/watch?v=RZ66yGyEKFg&list=LL&index=20&t=407s

8.5 Matchmaking Microservice

https://github.com/anze-k/matchmaking
https://www.reddit.com/r/gamedev/comments/k1wdm7/simple_matchmaking_service_nodejs_websockets/
https://github.com/jduyon/matchmaking