



NUS School of Computing
CS3219 Software Engineering Principles & Patterns

Prepared by:
G16

Name	Matriculation Number
Roy Chan Chuan Zhi	A0197135Y
Pang Wai Kye	A0197283R
Jerryl Chong Junjie	A0206131Y
Chua Yong Yang, Jay	A0199464L

Introduction

Background

When applying for software engineering roles, applicants are often required to go through rigorous technical interviews. These interviews are often challenging and common problems for applicants include being unable to articulate their thought process or being unable to formulate a solution for the given problem due to the stressful circumstances. It can also be tedious and monotonous to continuously practice coding questions alone.

Purpose

PeerPrep is a web application that provides users with a peer learning system. PeerPrep helps users find a partner to do coding questions with as we believe that performing during coding interviews is a skill that can be mastered with practice.

Users can adopt the roles of an interviewer and an interviewee, allowing them to practice articulating their thought process. Also, users can help each other when either struggles to formulate a solution.

Product Requirements

We have adopted a microservice architecture and specified the product requirements accordingly. More details on the architecture can be viewed at the architecture section.

Name	
Frontend [Must have] and Fancy UI [Nice to have]	
Description	
The frontend provides an web interface for users to interact with PeerPrep	
Functional Requirements	
<p>FR1.1. The system should allow users to log in or sign up with text inputs [High]</p> <p>FR1.2. The system should allow users to change their password with a button and text input [High]</p> <p>FR1.3. The system should allow users to delete their account by clicking the DELETE and CONFIRM button [High]</p> <p>FR1.4. The system should allow users to select a question difficulty level with EASY, MEDIUM and HARD button [High]</p> <p>FR1.5. The system should allow users to edit code simultaneously on a shared editor [High]</p> <p>FR1.6. The system should allow users to view a server defined timeout countdown spinner while waiting for a match [Medium]</p> <p>FR1.7. The system should allow users to run code and display the result [Medium]</p> <p>FR1.8. The system should allow users to chat with each other via a chat box [Medium]</p> <p>FR1.9. The system should allow login to be persistent [Low]</p>	
Interface	
Pages	Components
<ul style="list-style-type: none">• Login Page• Home Page• Room Page• Account Page	<p>Login page:</p> <ul style="list-style-type: none">• SignIn Component• SignUp Component <p>Home page:</p> <ul style="list-style-type: none">• Difficulty Selector• Match Modal• Timer <p>Room page:</p> <ul style="list-style-type: none">• Question Container• CodeMirror Editor• Code Results• Chat Box

	<ul style="list-style-type: none"> • Language Menu • Language Modal <p>Account Page</p> <ul style="list-style-type: none"> • Change Password Modal • Delete Account Modal • History List • History Modal
Non-functional requirements	
NFR1.1.	Works with all mainstream desktop browsers (Chrome, Safari, Firefox)

Name		
User service [Must have]		
Description		
The user service provides a RESTful HTTP API for maintaining information about users of PeerPrep		
Functional requirements		
FR2.1. The system should allow users to create an account with username and password [High] FR2.2. The system should ensure that every account created has a unique username [High] FR2.3. The system should allow users to log into their accounts by entering their username and password [High] FR2.4. The system should allow users to change their password [Medium] FR2.5. The system should allow users to delete their account [Medium] FR2.6. The system should allow users to view their question history [Medium]		
Interface		
Commands	Queries	Events Consumed
<ul style="list-style-type: none"> • createUser • loginUser • deleteUser • changePassword 	<ul style="list-style-type: none"> • getUserHistory 	<ul style="list-style-type: none"> • Messages from “user-question” kafka topic
Non-functional requirements		
NFR2.1. Users' passwords should be hashed and salted before storing in the DB NFR2.2. deleteUser, changePassword and getUserHistory api routes should be protected		

Name		
Matching service [Must have]		
Description		
The matching service provides an API for matching users based on difficulty level of questions		
Functional requirements		
FR3.1. The system should allow users to select the difficulty of questions they would like to practice [High] FR3.2. The system should be able to match two waiting users with the same selected difficulty [High] FR3.3. If there is a valid match, the system should match users within 30s [Medium] FR3.4. The system should inform users that no match is available if a match cannot be found within 30 seconds [Medium] FR3.5. The system should provide a means for the user to leave a room once matched [Medium]		
Interface		
Commands	Queries	Events Published
<ul style="list-style-type: none"> requestNewMatch leaveMatch 	N/A	<ul style="list-style-type: none"> New match requested Match success Match failed Match started User left match User disconnected
Non-functional requirements		
N/A		

Name
Question service [Must have]
Description
The question service provides a provides a RESTful HTTP API for clients to retrieve questions
Functional requirements
FR4.1. The system should allow users to get a random question with a selected difficulty [High]

FR4.2. The system should allow users to fetch details of a question from its question id [Medium]		
Interface		
Commands	Queries	Events Published
N/A	<ul style="list-style-type: none"> getRandomQnByDifficulty getQuestionById 	<ul style="list-style-type: none"> "user-question" kafka topic
Non-functional requirements		
NFR3.1. Queries should respond within <= 100ms		

Name		
Collaboration service [Must have]		
Description		
The collaboration service provides an API for synchronizing the code editor between users in the same room		
Functional requirements		
FR5.1. The system should allow users to edit code in the code editor and view each other's changes in real time [High]		
Interface		
Commands	Queries	Events Published
N/A	N/A	<ul style="list-style-type: none"> Code changed Language changed
Non-functional requirements		
N/A		

Name		
Code runner service [Nice to have]		
Description		
The code runner service provides a RESTful HTTP API for running code		
Functional requirements		

FR6.1. The system should allow users to run their code and view its output [High]		
Interface		
Commands	Queries	Events Published
<ul style="list-style-type: none"> runCode 	N/A	N/A
Non-functional requirements		
NFR6.1.	The service will timeout after 10s if the code takes too long to run	
NFR6.2.	The service will support C++, Java, Javascript, and Python	

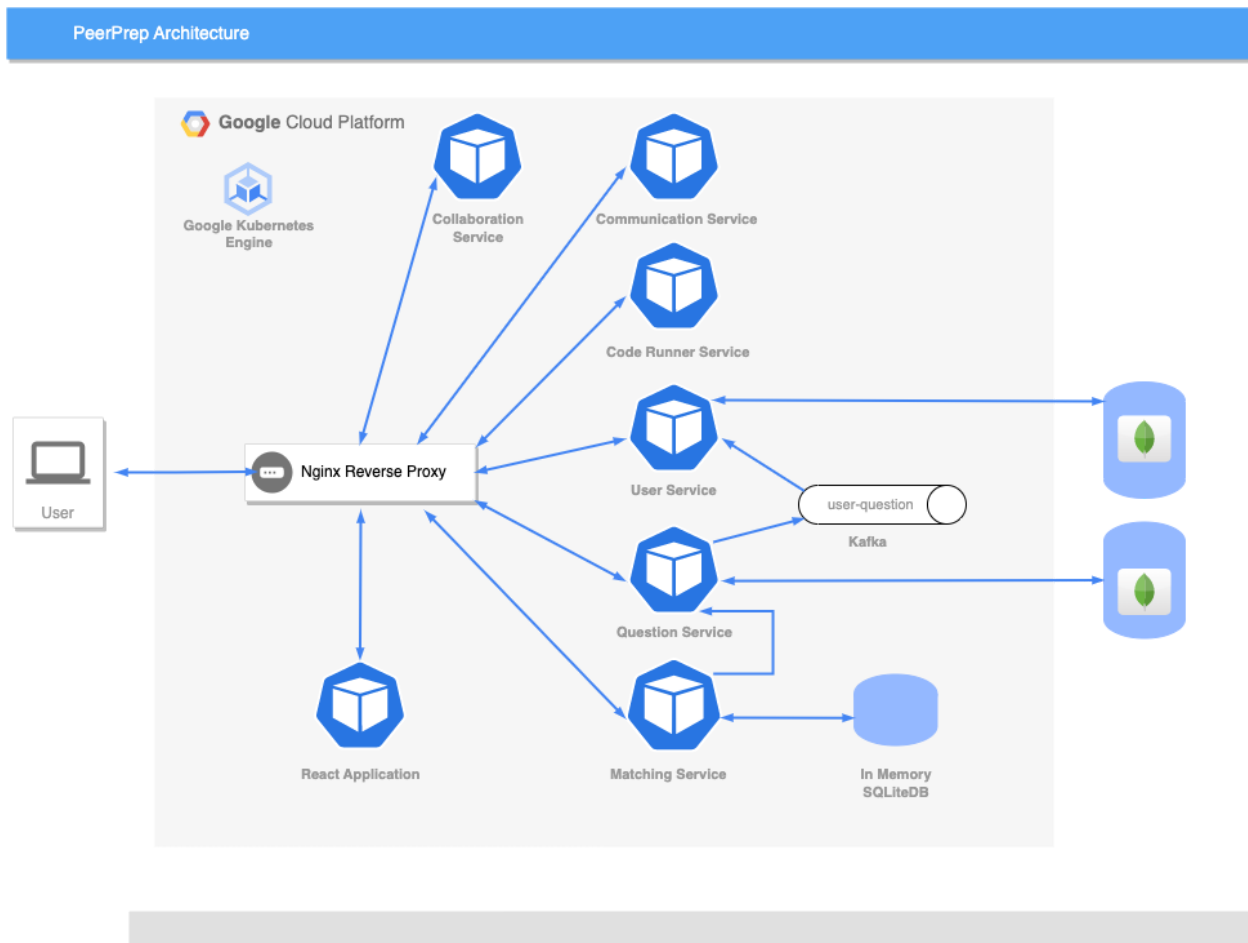
Name		
Communication service [Nice to have]		
Description		
The communication service provides an API for allowing users to chat with each other over text		
Functional requirements		
FR7.1. The system should allow users in the same room to send text messages to each other [High]		
FR7.2. The system should show if users are typing [Medium]		
Interface		
Commands	Queries	Events Published
N/A	N/A	<ul style="list-style-type: none"> Chat created User left chat Message received User typing
Non-functional requirements		
N/A		

Design Details

Architecture

Architecture

We have chosen to use a microservice architecture with each major functionality of the application split into its own independent parts. The requirements of each microservice is specified in the requirements section.



This architecture allows us to:

Pros

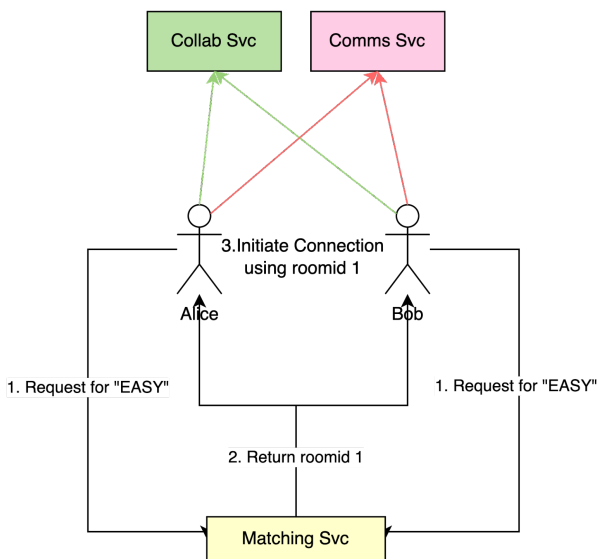
- Work efficiently as we now can work on each service independently
- Allows our services to be deployed and scaled up independently
- Allows us to use the right tools for the different services such as the choice of programming language and database
- Better segregation of responsibilities with loose coupling

Cons

- Increased complexity of application as we have to deal with communication between microservices (Synchronous vs Asynchronous communication)
- Increased complexity of application as we have to deal with distributed data since each microservice encapsulates its own data
- Resiliency of the system has to be considered when network calls fail between nodes in a microservice architecture

Socket.IO Synchronization

One of the major communication methods we use is socket.IO and because we are using microservices and not using a monolithic application, we have to deal with coordination and synchronization of multiple services instead of just one. To combat this, the roomId generated from Matching Services to identify a unique pair of entities.



Although slightly more complex to implement, this allows our application as a whole to be resilient as the failure of one service does not bring down the application.

Notable External Dependencies

Frontend

1. UI library: Chakra UI
2. Client state management: Zustand
3. Form management: React hook form
4. Code editor: Code mirror
5. Event communication: Socket.IO
6. Build tool: Vite

Backend

1. Event communication: Socket.IO

2. Messaging middleware: Kafka

Deployment [Including deployment tech stack]

Deployment Patterns

We chose to use the Service instance per container pattern which runs each service instance on its own container. We can reap the benefits of containerization such as portability, security, efficiency, and it helps to decouple our microservices from each other.

Technologies

Based on the deployment pattern, we chose to deploy our application on Kubernetes as it both supports it and provides powerful features such as service discovery, and self-healing nature. This ensures close to zero downtime for our application.

Due to the large number of microservices, applying each individual manifest (both service and deployments) to the kubernetes cluster would be very cumbersome and messy. In addition, updating each manifest individually and keeping track of changes becomes increasingly difficult as more services are added. Therefore we chose to use Helm to manage our application as a form of Infrastructure-as-code (IaC). This allows us to use the powerful features of git to track changes made to our application specifications. In addition, Helm allows us to package our application into a chart which allows for easy deployment across different clusters and can be used by other developers.

Process of Deployment

We first need to provision a Kubernetes cluster on Google Kubernetes Engine and set up the Google Artifact Repository to contain our application helm charts which allows us to keep track of different versions of our application.

We would then build the different microservices' Docker image and update our helm chart to use these images.

We can package the helm chart and push it to the Artifact Registry, which can then be pulled by the Kubernetes cluster and installed or updated via helm.

Design Principles and Patterns

Application

Because we have adopted the microservice architecture, we have abided closely to the Single Responsibility Principle where each component serves a single purpose (e.g. Matching Service only deals with matching users).

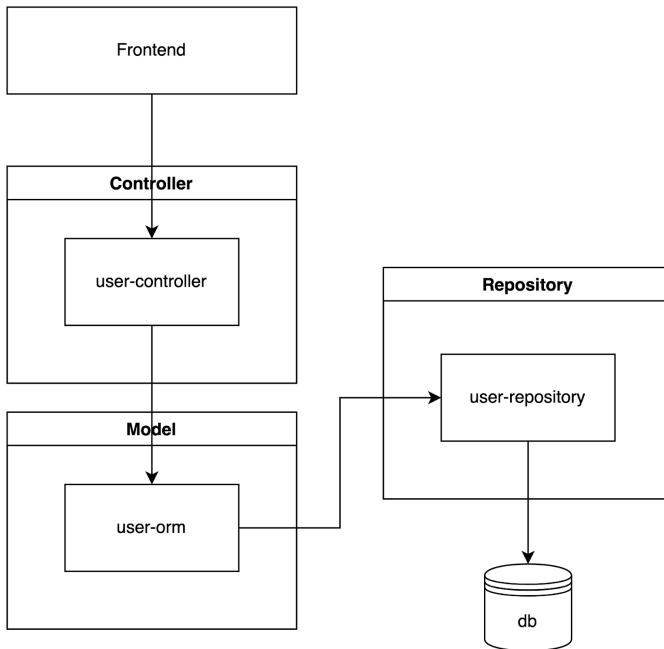
We also adopted a database per service pattern. This isolation allows us to use a type of database that is appropriate for the respective service. For example, the matching service uses an in-memory sqlite database for better performance while the question service uses a NOSQL MongoDB database since the data is non-relational and we can take advantage of flexible schemas.

In terms of microservice communication, we have implemented both synchronous communication and asynchronous communication. Synchronous communication is carried out when the matching service calls the question service after a successful match to query for a question and send it to both users. Asynchronous communication is carried out between the question service and user service when the question service publishes a message which is consumed by the user service which subscribes to the corresponding topic. These data flows are illustrated in the architecture diagram shown in page 8.

Additionally, we have also implemented a reverse proxy which sits in front of all our microservices and encapsulates the internal architecture. The reverse proxy provides a unified interface for the frontend to call and handles routing, instead of the frontend having to call each of the services directly. This adheres to the facade design pattern by providing a single entry point, making the entire microservice subsystem easier to use. Furthermore, by using kubernetes as our container orchestration tool, its service discovery capabilities abstracts away failures from the frontend. For example, if any of the backend service pods goes down, kubernetes automatically restarts a new pod with new network configuration. This failure is not known to the frontend who can continue using the reverse proxy to hit the new pod.

User Service

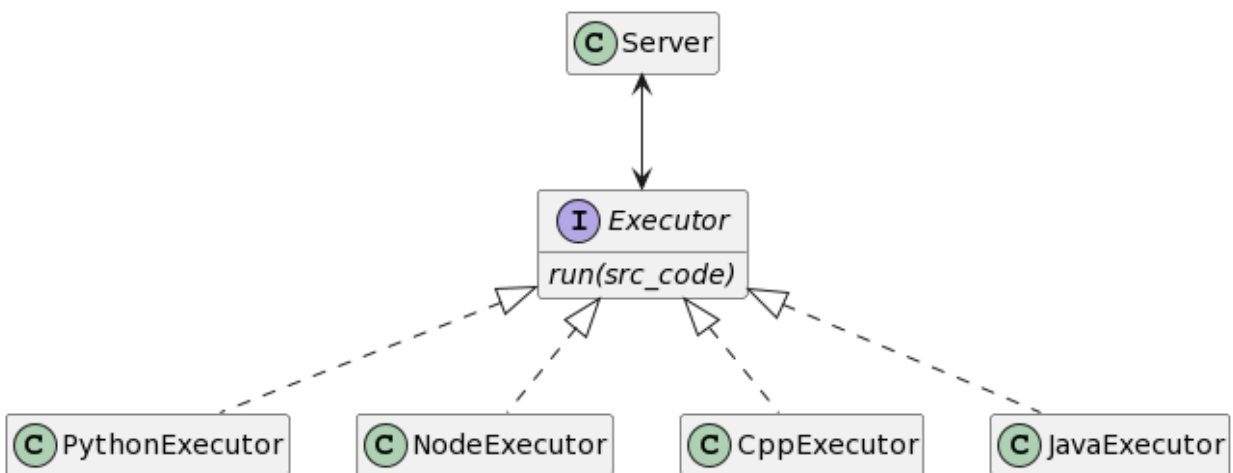
Taking inspiration from MVC models and Onion architecture, we came up with a streamlined architecture that has the modularity and extensibility of MVC models as well as the inversion of control of Onion architecture which dependencies only go downwards to lower level components. This allows our components to be more reusable and have a more lean responsibility.



Code Runner Service

In our code runner service, we have utilized the command pattern to decouple our server from each individual language executor. Usually in command patterns there are 4 key terms, *command*, *receiver*, *invoker* and *client*. The command object is the interface that hides the underlying logic which interacts with some receiver object to output the result. The invoker is the one that activates the command but does not know about the actual implementation. Finally, the client is the orchestrater

Since our service is small, our Server object plays the role of a client, invoker and receiver. Our Executor interface is our Command.



By adopting this command pattern, we are able to decouple our application as well as better segregate the responsibilities of each class which allows us to easily add support for new languages in the future.

Design Decisions

*For all decision decisions, the underlined one is the implemented one.

Question service

Language choice

	<u>Go</u>	NodeJS
Advantages	<ul style="list-style-type: none">- Faster due to being a compiled language- Scalable concurrency model with goroutines	<ul style="list-style-type: none">- Greater ease of implementation
Disadvantages	<ul style="list-style-type: none">- Can be harder to learn and implement compared to NodeJS	<ul style="list-style-type: none">- Slower than Go- Less elegant single threaded concurrency model

Indexing

The question service uses MongoDB as its database. From an end user perspective, end users only query the question service to retrieve questions and from an admin perspective, admins usually write questions to the question database once. Thus, the access pattern for the question service database is more **read-heavy than write-heavy**. **Latency in writes can also be more tolerated than reads** since end users are only reading and writes are done by admins. Thus, the use of indexes is ideal in this situation to improve query performance while trading off write performance. As seen in the figure below, questions are indexed by `_id`, difficulty, title and topics. This indexing design decision helps to achieve our non-functional requirement of having response times of $\leq 100\text{ms}$ for queries to the question service.

question-svc-dev.questions

4

4

DOCUMENTSINDEXES

Documents

Aggregations

Schema

Explain Plan

Indexes

Validation

Refresh

Create Index

Name and Definition	Type	Size	Usage	Properties
> _id_	REGULAR ⓘ	36.9 KB	0	UNIQUE ⓘ
> difficulty_1	REGULAR ⓘ	36.9 KB	0	
> title_1	REGULAR ⓘ	36.9 KB	0	UNIQUE ⓘ
> topics_text	TEXT ⓘ	36.9 KB	0	

History feature

When implementing the history feature (F.R. 2.6.), there were several design decisions to be made:

Deciding a suitable user schema which encapsulates history information

The user service database needs to store history information for the PeerPrep web client to query and display. There are several options in designing the schema for the history information.

	<u>List[{id, title}]</u> *Store id and title	List[id] *Store only id	List[Question] *Store entire question
Description	The user schema contains a list of objects storing the id and title of the question.	The user schema contains a list of question ids.	The user schema contains a list of questions.
Advantages	<ul style="list-style-type: none">- Best compromise between the other 2 options- Only 1 api call from client needed to get history information in the form of the question title	<ul style="list-style-type: none">- No data redundancy	<ul style="list-style-type: none">- Can get all information in a single api call
Disadvantages	<ul style="list-style-type: none">- Need to make an additional network call to get all of a question's details- Has data redundancy in one field, thus there is risk of showing stale question title if the question's title is updated (though updates are	<ul style="list-style-type: none">- Poor performance: when hitting the user service endpoint to get the user's history, an additional n synchronous api calls to question service are needed to do a "join" (where n is the length of the history).	<ul style="list-style-type: none">- Large data redundancy since the entire question is copied

	infrequent)		
--	-------------	--	--

Deciding between asynchronous vs synchronous communication between question service and user service

When the question service endpoint is hit to retrieve a question after a match, the question service needs to send over the entire history information to the user service for user service to store persistently. There are 2 main options for this inter microservice communication: asynchronous messaging and synchronous call.

	<u>Asynchronous messaging</u>	Synchronous call
Description	<p>Kafka serves as a message queue for asynchronous communication between the 2 services.</p> <p>When the <i>getRandomQnDifficulty/{difficulty}/{userOne}/{userTwo}</i> endpoint of the question service is hit, the question service sends a message to the kafka broker at topic 'user-question'. Once this is complete, the client receives a success response from the question service.</p> <p>Separately, this message is consumed by the user service who then writes the information into the user service database.</p>	<p>When the <i>getRandomQnDifficulty/{difficulty}/{userOne}/{userTwo}</i> endpoint of the question service is hit, the question service calls a RESTful POST endpoint provided by the user service to write the history information into the user service database. Once this is complete, the client receives a success response from the question service.</p>
Advantages	<ul style="list-style-type: none"> - Highly decoupled where question service can simply fire and forget - Failures in each of the services are isolated from one another - Better performance instead of performing a synchronous call to user service and waiting for a response 	<ul style="list-style-type: none"> - Simpler to implement
Disadvantages	<ul style="list-style-type: none"> - Additional implementation complexity in introducing a new component 	<ul style="list-style-type: none"> - Highly coupled - Poorer performance as question service has to make a HTTP call to user service and wait for user service to finish writing to the database

		before receiving a response
--	--	-----------------------------

Matching service

When deciding on the type of database to use for storing matches between users, we weighed the pros and cons of an in-memory database against a separate database. We stuck with an in-memory database because of a few reasons.

Firstly, our priority is matching users quickly, this is better achieved with an in-memory database.

Secondly, persistence is not of vital importance in this context; Matches are ephemeral and even in the event of the service going down, at most we will lose the user's work for the past 30 mins max. Hence the added overhead of managing yet another database is not worth for a object that has such a short lifespan.

Lastly, even though some correctness may be compromised if we were to perform horizontal scaling, e.g. 2 users looking for matches in 2 separate pods, this side effect is negligible. This issue will only arise in the case that we do not have enough traffic to match the users which contradicts the need for horizontal scaling. Hence if we scale horizontally, we would have enough traffic to mitigate this small issue.

Therefore we have decided to go with an in-memory database for matching service.

	<u>In-memory Database</u>	Separate Database (e.g Postgres)
Advantages	<ul style="list-style-type: none"> - Fast, Matches Instantly - Simple to Use 	<ul style="list-style-type: none"> - Persistence, allowing for resilience - Ability to scale horizontally
Disadvantages	<ul style="list-style-type: none"> - No Persistence - May be problematic when scaling horizontally 	<ul style="list-style-type: none"> - Slower - One more dependency to manage

Collaboration service

While deciding how to implement the collaboration service, we noticed that the text editor library we were using (Codemirror) had support for collaborative editing. We also found a method that involves using socket.io.

	<u>Socket.io</u>	Codemirror Collaborative Editing
Advantages	<ul style="list-style-type: none"> - Simpler to implement 	<ul style="list-style-type: none"> - Custom data structures to handle updates to the editor - Users' changes will not be lost
Disadvantages	<ul style="list-style-type: none"> - Potential for users to overwrite each other 	<ul style="list-style-type: none"> - More complicated to implement - Overhead when user makes changes

We found that, for our use case, one user is an interviewer and the other is an interviewee. Usually, only the interviewee types the code and the interviewer only needs to see the changes. Hence, most of the time, it is expected that only one user is typing, so the likelihood of there being users overwriting each other is low. Thus, we chose the simpler implementation of socket.io over using Codemirror's collaborative editing feature.

Communication service

For our communication service, we were deciding whether we should use a database to store the chat history when sending messages (FR7.1).

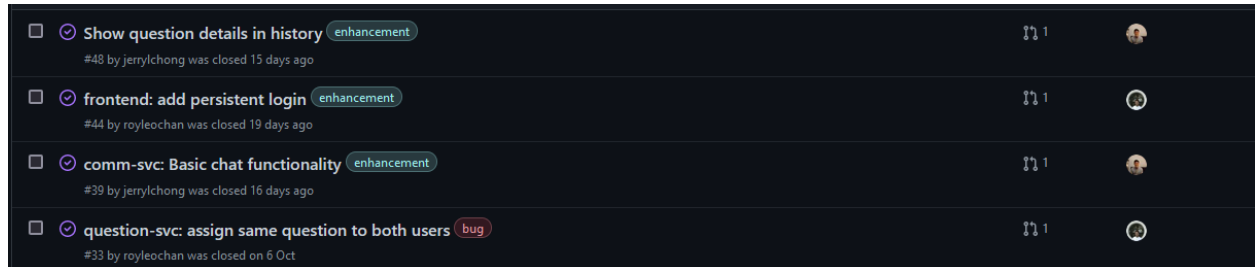
	With database	<u>Without database</u>
Advantages	<ul style="list-style-type: none"> - Persistent chat history - Able to restore chat - Messages sent before user joined can be retrieved 	<ul style="list-style-type: none"> - Sending messages is less expensive - Simpler implementation makes it easier to read and maintain
Disadvantages	<ul style="list-style-type: none"> - Additional overhead when sending messages 	<ul style="list-style-type: none"> - Chat history is lost when connection is closed

For our app, we do not plan on showing the chat history once the session has ended, and we do not allow users to rejoin the session. Furthermore, the time between each user joining the connection is very small, so it is unlikely for messages to be sent during this time. Hence, it would be more beneficial for us to not use any database.

Thus, we only use socket.io to transmit messages from one user to another.

Development Details

Issue tracking

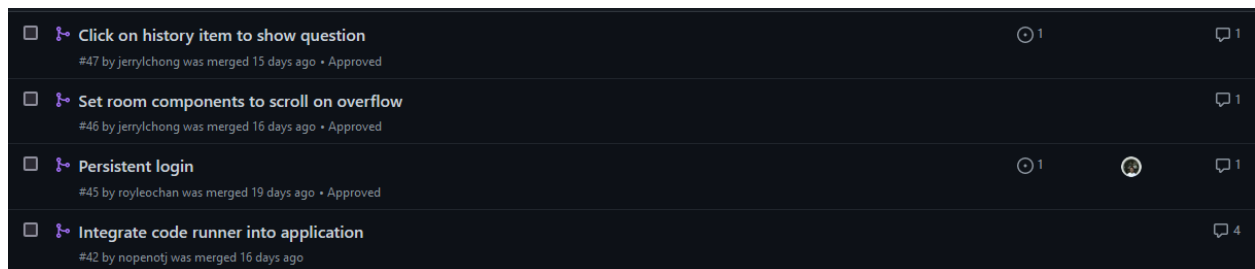


<input type="checkbox"/>	Show question details in history enhancement	1	
#48 by jerrylichong was closed 15 days ago			
<input type="checkbox"/>	frontend: add persistent login enhancement	1	
#44 by royleochan was closed 19 days ago			
<input type="checkbox"/>	comm-svc: Basic chat functionality enhancement	1	
#39 by jerrylichong was closed 16 days ago			
<input type="checkbox"/>	question-svc: assign same question to both users bug	1	
#33 by royleochan was closed on 6 Oct			

Used Github Issues to keep track of task deadlines and task assignments.

This provides us with a place to consolidate all our tasks and makes it easier to keep track of our responsibilities and also our deadlines.

Pull requests and code review



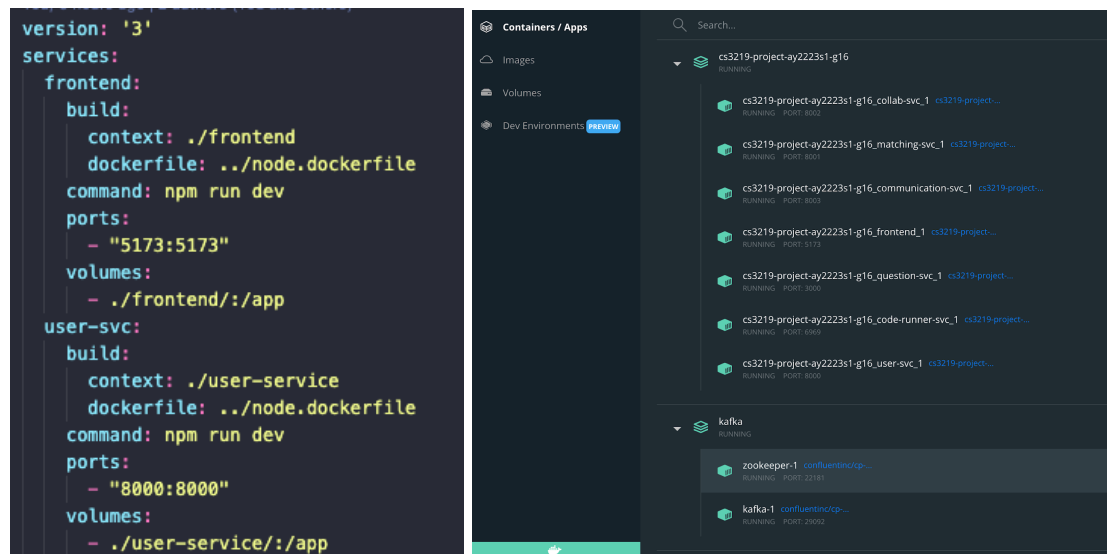
<input type="checkbox"/>	Click on history item to show question	1		1
#47 by jerrylichong was merged 15 days ago • Approved				
<input type="checkbox"/>	Set room components to scroll on overflow			1
#46 by jerrylichong was merged 16 days ago • Approved				
<input type="checkbox"/>	Persistent login	1		1
#45 by royleochan was merged 19 days ago • Approved				
<input type="checkbox"/>	Integrate code runner into application			4
#42 by nopenotj was merged 16 days ago				

For each issue, changes are worked on in a new branch.

Once done, we create a pull request from the new branch to the main branch. The other team members then help to review the code changes and suggest changes if required. Only when the pull request has been approved by another member, then can it be merged into the main branch.

This helps to prevent undesirable changes from being pushed into the main branch due to the cross validation by other members. Furthermore, this can also help us learn from comments left during the reviews.

Use of Dockerized Environments for development



In our development, we utilize docker to ensure our application is both portable and platform dependent. This allows us to write our code once and run it anywhere because it mitigates any underlying dependency issues that we might face due to the differences in the development environment. Also, because we have adopted a microservice architecture, having one compose file to spin up all the services helps to reduce administrative overhead.

Future Enhancements

Authorization

Improve security by implementing authorization for our backend services. This authorization logic can be centralized at the API gateway level to avoid code duplication.

Voice chat for communication

Allows users to be able to communicate with one another more easily.

Since the interviewee will be typing out their code, having voice chat will allow them to speak to the interviewer and type at the same time. Similarly, the interviewer will be able to give real-time feedback to the interviewee, and the interviewee would be able to listen and respond while typing.

Show cursor and highlighting for text editor

Shows users each other's cursors and highlighted text in the text editor.

When discussing the code in the text editor, users may find it difficult or time consuming to explain which part of code they are referring to. Hence, by showing each other's cursors and highlighted text, users can highlight or move their cursor to the relevant part. Then, the other user would be able to easily find the relevant code.

User ratings

Gives users the option to rate their partners after the session has ended.

Having a rating system allows users to commend and encourage one another, if they found their partner to be helpful, friendly, etc. This can allow users to identify good partners so that they can maximize their practice session. Moreover, receiving a good rating can make users feel more encouraged and good about themselves.

Similarly, there may be instances where users are matched with trolls. Thus, users can rate trolls poorly. This can help to prevent other users from having the same bad experiences, as users with a low rating can be avoided or punished by a review system.

Support more languages for code runner

Provides users with the language they are most comfortable with.

With a large variety of programming languages in the world, users will have many different preferences for the language they would like to use. Hence, it is important to provide them with as many options as possible to choose from.

Reflection and Contributions

Reflection

This project has been an invaluable experience to allow us to explore and implement an entire microservice architecture. Concepts that we were exposed to include synchronous and asynchronous microservice communication, reverse proxies and container orchestration.

With the lessons from lectures and tutorials in mind, we also paid more attention to tradeoffs in design decisions while considering various design principles and design patterns.

Finally, we also managed to improve our competency in the diverse and modern technology stack used.

Individual contributions

Member	Contributions
Roy	<u>Code contributions:</u> Kafka Frontend User service Question service Matching service <u>Requirements completed:</u> F.R. 1.4. F.R. 1.6. F.R. 1.9. F.R. 2.2. F.R. 2.3. F.R. 2.6. F.R. 4.1. N.F.R. 2.1. N.F.R. 3.1.
Jerryl	<u>Code contributions:</u> Frontend Communication service Question service <u>Requirements completed:</u> F.R. 1.8. F.R. 2.6. F.R. 4.2. F.R. 7.1. F.R. 7.2.

Jay	<u>Code contributions:</u> Matching Service Code Runner Service <u>Requirements completed:</u> F.R. 3.1. F.R. 3.2. F.R. 3.3. F.R. 3.4. F.R. 6.1. N.F.R. 6.1. N.F.R. 6.2.
Wai Kye	<u>Code contributions:</u> Collab Service User Service Kubernetes (Helm) Frontend <u>Requirements completed:</u> F.R. 1.1. F.R. 1.2. F.R. 1.3. F.R. 1.5. F.R. 2.1. F.R. 2.2. F.R. 2.3. F.R. 2.4. F.R. 2.5. F.R. 5.1. N.F.R. 1.1. N.F.R. 2.1. N.F.R. 2.2.